

IMPERIAL

Imperial College London
Department of Computing

Efficient and Effective Web Application Security Testing via Reinforcement Learning

Myles Luke Foley

Supervised by Associate Professor Dr Sergio Maffei

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of Imperial College London
December 2024

Copyright

The copyright of this thesis rests with the author. Unless otherwise indicated, its contents are licensed under a Creative Commons Attribution-Non Commercial 4.0 International Licence (CC BY-NC).

Under this licence, you may copy and redistribute the material in any medium or format. You may also create and distribute modified versions of the work. This is on the condition that: you credit the author and do not use it, or any derivative works, for a commercial purpose.

When reusing or sharing this work, ensure you make the licence terms clear to others by naming the licence and linking to the licence text. Where a work has been adapted, you should indicate that the work has been changed and describe those changes.

Please seek permission from the copyright holder for uses of this work that are not included in this licence or permitted under UK Copyright Law.

Originality Statement

I hereby declare that this thesis and the work herein were composed and originated by myself, except where appropriately referenced and credited.

Myles Foley
December 2024

Abstract

Web application security is a persistent problem, increasing with the number of internet users and applications. Automated tools and techniques are commonly used to scan, test, or fuzz for bugs and vulnerabilities. However, the use of random search, manual heuristics, and the failure to use feedback makes existing work *ineffective*. Furthermore, the large number of test cases commonly needed to find comparatively small number of vulnerabilities makes it *inefficient*.

With a problem-first approach this thesis examines the ability of Reinforcement Learning (RL) to perform tasks in web application security testing, including the generation and mutation of test cases in black- and grey-box settings. RL is a machine learning approach where an agent take actions in an environment to maximise long-term rewards overtime. Thus, the agent learns to complete a task. RL has achieved better-than-human performance in complex tasks ranging from game playing to protein design. Yet, the application of RL in web security is limited: relying on off-the-shelf RL models, simple heuristics, and domain knowledge to guide learning.

We design bespoke RL agents with rich latent representations. We develop a hierarchical RL approach to cross-site scripting payload generation to constrain the challenge for agents. We then design a single agent that generates SQL injection payloads using latent representations in a changing action space. Furthermore, we evaluate the ability of RL to mutate HTTP test cases to find bugs in REST APIs. Finally, we present the first study to investigate where to perform mutations in JavaScript test cases, designing deep contextual bandit we place inside existing tools improves the *efficiency* of test case validity, coverage, and coverage-per-test case. Our RL agents are *effective*, finding 5 Cross Site Scripting (XSS) CVEs, 6 SQL injection CVEs, and 42 bugs in REST APIs. The agents are also *efficient*, reducing the number of attempts.

Acknowledgements

Friends, Researchers, Cyberists, lend me your eyes, I come not to praise, but to acknowledge. There are many people who contributed (directly or indirectly) to research presented here, it really does take a village, so to everyone not named here I simply say: thank you.

To begin with I should thank Dr Sergio Maffeis, without whom this thesis literally would not exist. For his guidance and support both in (and out of) research, I am incredibly grateful. I'm sure there were times he must of thought I was as mad as a box of frogs, ~~as such~~ I am (again) grateful for his belief, encouragement, and even praise.

The family that have supported me throughout this endeavour (and often reminded me to take a break or that I'm working too hard) I thank you. In hindsight, dragging me away from a laptop for a coffee, walk, or a trip to the pub was worth it. Again, I'm not sure these pages would be here without your support.

The friends I made along the way: Almuthanna Alageel, Abdullah Aldaihan, Fahad Alotaibi, Hazim Hanif, Kate Highnam, Adam Jones, Eman Maali, and (my brother in RL4Sec) Shae McFadden. There isn't enough space to express my gratitude to you guys, but know I am inspired by you all both as researchers and people. And Emily, who for some reason joined me on this journey too (I'm still trying to fathom why): I'm glad you did.

The friends who came along for the ride, often with food, company, the odd drink, or a good board game: Alan, Denisa, Fraser, Harriet, Mike, Omkar, Tanya, and Miles (who gets an honourable mention for his editorial work over the years).

There are also many people who gave me opportunities I didn't know existed, and I'm fortunate to now consider friends. I owe each of you somewhere between a lot, to a hell of a lot. The least I can do is get a pint or two for your whenever you're next in town. Giulio Zizzo and the researchers at the IBM Research Dublin, who have been great collaborators. Fabien Charmet for inviting me back to Japan and to NICT, I thank you, and will always be fond of our calls. Chris Hicks and Vasilios Mavroudis, who 'adopted me' at The Turing. You have been my home-away-from-Imperial, I'm always grateful for the many opportunities I have had to work with and learn from you. Finally, a quick shout-out to Kate for doing one of the many *many* things she does so well: bring people together. *Long live Team Mindrake.*

Dedication

I dedicate this Thesis to the memory of Veronica Christine Dunford, and Roger Bernard Dunford. I hope these 256 pages, 53,904 words, and 24 figures would have made you proud.

Additionally, I have a few small dedications¹².

¹To Miles Atherton (who will appreciate this more than most) and that one reviewer who was very critical of my writing style. You are dedicated all typographical, spelling, punctuation, and grammatical errors.

²To reviewers who considered my research to be ‘largely an engineering work’, I dedicate Sections [1.3](#) and [7.1](#).

‘A learning experience is one of those things that says: *You know that thing you just did?*
Don’t do that.’

Douglas Adams

Contents

Abstract	i
Acknowledgements	iii
Abbreviations	xxi
1 Introduction	1
1.1 Motivation	4
1.2 Objectives	7
1.3 Contributions	7
1.4 Publications	9
1.4.1 Additional Publications	10
1.5 Thesis Outline	11
2 Background	12
2.1 Reinforcement Learning	12
2.1.1 Multi-Armed Bandits	15
2.2 Deep Learning	16
2.2.1 Recurrent Neural Networks	16

2.2.2	Transformers	19
2.2.3	Encoder-Decoder Models	20
2.3	Deep Reinforcement Learning	20
2.3.1	Deep Q-Learning	21
2.3.2	Actor-Critics	23
2.3.3	Proximal Policy Optimisation	25
2.3.4	Curiosity	25
2.4	Fuzzing	26
2.4.1	Deep Learning Approach in Fuzzing	30
2.4.2	Reinforcement Learning Approach in Fuzzing	32
2.5	Web Application Security Testing	34
2.5.1	Crawling Web Applications	36
2.5.2	Grey-Box Testing	38
2.5.3	Black-Box Testing	39
2.6	Takeaways	43
3	Constrained Games for Cross Site Scripting Payload Generation	46
3.1	Overview	48
3.2	Challenges of XSS Detection	48
3.3	Payload Generation as a Hierarchical RL game	52
3.3.1	The Payload Generation Environment	54
3.4	The HAXSS Architecture	56
3.4.1	The Escape Agent	58

3.4.2	Sanitisation Agents	62
3.4.3	Deep Learning Implementation	63
3.4.4	Adapting to New Source-Sink Combinations	64
3.4.5	The XSS Payload Test Bed	65
3.4.6	Training	66
3.5	Benchmark-based Evaluation	69
3.5.1	Experiment Setup	69
3.5.2	Micro-benchmark: Payload Generation	71
3.5.3	Macro-benchmark: Webapp Scanning	73
3.5.4	Ablation Study	77
3.5.5	Payload Analysis	78
3.6	Case studies	82
3.6.1	HTMLy (CVE-2021-42946, CVE-2021-42867)	82
3.6.2	Pixelimity (CVE-2021-42866)	83
3.6.3	Chikitsa (CVE-2021-42868, CVE-2021-42869)	83
3.7	Discussion and Limitations	84
3.7.1	Limitations	85
3.8	Related Work	86
3.8.1	XSS Payload Generation	86
3.8.2	RL for XSS Detection	87
3.9	Takeaways	88

4	A Singular Agent for Generating SQL Injection Payloads	90
4.1	Overview	92
4.2	Motivation	93
4.2.1	SQLi Payload Distribution	95
4.2.2	Existing Scanner Methods	97
4.2.3	Challenges of Using RL for SQLi	97
4.3	An Environment for SQL Injection	100
4.4	The SQIRL Agent	106
4.4.1	Local Worker	106
4.4.2	Global Agent	109
4.4.3	Training	110
4.5	Evaluation	112
4.5.1	Experimental Setup	112
4.5.2	Comparison on SQLi MicroBenchmark	113
4.5.3	Comparison on Production Web Apps	115
4.5.4	Distributed Learning Capability	117
4.5.5	SQIRL Ablation Study	119
4.6	SQLi Payload Analysis	120
4.7	Discussion and Limitations	123
4.8	Related Work	124
4.8.1	Fuzzing for SQL Injection.	124
4.8.2	Defensive Coding Against SQLi.	125
4.9	Takeaways	126

5	REST API Fuzzing By Mutating Test Cases with Reinforcement Learning	128
5.1	Overview	130
5.2	A Framework for Learning to Find Bugs in REST APIs	131
5.2.1	OpenAPI Specification	131
5.2.2	Preprocessing	133
5.2.3	Actions	134
5.2.4	Observations	135
5.2.5	Reward	136
5.2.6	Agent Architecture	136
5.3	Evaluation	137
5.3.1	Training	137
5.3.2	Experiment Setup	139
5.3.3	Coverage and Request Volume	141
5.3.4	Bugs Found and Request Volume	143
5.3.5	Ablation Study	147
5.3.6	Feature Importance Analysis	151
5.4	Limitations	152
5.5	Related Work	153
5.6	Takeaways	154
6	Locating Effective Mutations in Test Cases when Fuzzing JavaScript En-	
	gines.	156
6.1	Overview	158

6.2	Fuzzing JavaScript Engines	160
6.2.1	Fuzzilli	160
6.2.2	FuzzJIT	162
6.2.3	JIT-Picker	162
6.2.4	Challenges in Targeting Mutations	162
6.3	CLUTCH Control	164
6.3.1	Multi-Armed Bandits for Location Selection in JavaScript	164
6.3.2	System Design	166
6.3.3	Actions and Observations	169
6.3.4	Reward	169
6.4	Evaluation	170
6.4.1	Experimental Setup	171
6.4.2	Field Test Comparison	173
6.4.3	CLUTCH Selection	178
6.4.4	Bandit Fuzzing Comparison	182
6.4.5	Wider Bandit Comparison	183
6.5	Limitations	185
6.6	Related Work	186
6.7	Takeaways	188
7	Conclusion	189
7.1	Summary of Thesis Achievements	189
7.2	Limitations	192

7.3	Future Work	194
7.4	Final Thoughts on RL in Cyber Security	196
A	Commands used to test XSS scanners	197
B	Commands used to test SQLi scanners	199
C	DQN Algorithm	201
	Bibliography	201

List of Tables

2.1	The limitations associated with traditional fuzzing techniques.	30
2.2	Fuzzing using learning based approaches.	32
2.3	Web application security testing approaches using deep learning or RL methods.	36
3.1	A: CVE-2021-22889 that uses a built in PHP function incorrectly that can result in XSS. B CVE-2019-10062 that uses a custom regex that can still result in XSS.	51
3.2	Hyperparameter search values and the selected configuration of HAXSS.	64
3.3	Sanitisation rules and output contexts from the XP Train Bed used for training (1-15), and the XP Test Bed for testing (16-25).	65
3.4	Scanners results and request statistics for the XP Test Bed. TP: XSS found; FN: XSS not found; FP: XSS not working.	71
3.5	Scanners results for macro-benchmark. “Not Tried” indicates a failure of the crawler to find the injection point. FN denotes a failure to generate a payload to display the vulnerability.	74
3.6	Additional timing information of the scanners used on all webapps. ‘N/A’ denotes a scanner that had to be stopped after entering an infinite loop.	75
3.7	Performance of different Deep RL algorithms on the XP Train Bed. Ticks (✓) represents where the algorithm has met the win criteria and thus ‘won’ the example.	77
3.8	Sanitisations HAXSS is able to bypass	79

4.1	A: SQLi payload (highlighted in pink) escaping its SQL context to cause the database to pause for 1 second. B-D: SQL queries showing 13 semantically different positions where user input can occur. E: current WordPress example of dangerous parameterised query pattern.	93
4.2	Token space used for SQL payload and statements. SQIRL actions are shown in white, grey for SQL representation.	105
4.3	Context Free Grammar used to generate MySQL statements. Blue indicates terminals.	107
4.4	SQL statements and sanitisations used in the SMB where tasks increase in difficulty. Tasks 1-20 are used for training SQIRL. Tasks 1-30 constitute the positive class for the SMB benchmark.	110
4.5	Hyperparameters used to train the components of SQIRL. We performed a grid search on these values to optimise each model separately.	111
4.6	Comparison of SQIRL against state of the art scanners on the SMB. Dark solid bars (red) emphasise poor performance, light bars (green) emphasise good performance. <i>Spurious Positives</i> are TPs deemed invalid after manual analysis. . .	113
4.7	Comparison of SQIRL and state of the art tools, on production grade web applications.	115
4.8	List of Production Grade Web Applications	116
4.9	Ablations of SQIRL on the SMB.	119
4.10	Payload types and the scanners that can create them. We replace integers, strings, and base64 with INT, TEXT, and BASE64; otherwise leaving the payloads the same. These payloads are gathered from the experiments in Section 4.5. F denotes use of feedback, NF denotes no feedback, and E denotes presence of exceptions.	120
4.11	Features that different scanners can make as observed in the payloads seen. F denotes use of feedback in a web page, E exceptions.	123

5.1	Mutation actions that can be applied by APIRL to the current operation.	134
5.2	Grid search ranges for hyperparameters in APIRL.	137
5.3	Average unique bugs, LoC, and number of requests taken on each test service, with standard deviation shown in brackets.	142
5.4	Ablations of both APIRL and APIRL-cov. Lighter colours represent poor performance, while darker colours represent good performance.	149
6.1	Correctness of generated test cases (five run average) from 24h fuzzing of CLUTCH and baseline fuzzers.	173
6.2	Branch coverage (five run average) from 24h fuzzing of CLUTCH and baseline fuzzers.	175
6.3	Branch coverage per generated test case (five run average) from 24h fuzzing of CLUTCH and baseline fuzzers.	177
6.4	Branch coverage per generated test case (five run average) from 24h fuzzing of CLUTCH and CLUTCHLITE.	183
6.5	Total regret of combinatorial bandits in different settings.	184
6.6	Statistical significance of bandit regret performance compared to CLUTCH, using Vargha and Delaney’s $\hat{A}_{12}$ test and Mann-Whitney U test (p values).	185
6.7	Grid search ranges for hyperparameters neural network models	186
A.1	Configurations for the scanners used as baselines to compare to HAXSS. Where applicable scanners are provided with relevant information to be correctly authenticated with a web application.	198
B.1	Settings used for scanners when conducting experiments of crawling and vulnerability identification. In Section 4.5.3 we also test without crawling, pointing the scanners to each specific vulnerability found by SQIRL.	200

List of Figures

1.1	Number of CVEs reported to The National Vulnerability Database [202]. Showing those the proportion that are related to the most recent OWASP Top 10 [215].	2
2.1	Recurrent Neural Network Architectures. a) Long Short-Term Memory b) Gated Recurrent Unit.	17
2.2	Example DNN architecture of a DQN that given a state s_t computes Q-values for each action	22
2.3	Number of CVEs reported to The National Vulnerability Database [202] from the start of 2017 to the date of submission.	26
3.1	The overall HAXSS payload environment model, showing how the agent is able to interact with the web application.	56
3.2	Conditions for triggering the separate games (syntactically correct payload generation, and bypassing sanitisation) that form HAXSS.	57
3.3	HAXSS The Escape Agent architecture	59
3.4	Number of legitimate XSS payloads that trigger a vulnerability generated by HAXSS training on the XP Train Bed.	67
3.5	Box-plot to show the number of requests for each approach in the XP Test Bed, shown with log-scale. Circles indicate data points beyond the interquartile ranges.	72
4.1	Principal Component Analysis of SQLi payloads.	96

4.2	SQIRL architecture. RL agents (green) compute the mutation of the current payload. The N federated Local Workers (purple) train different agents to gather experiences for the global agent. Environment elements that interact with the web application (blue) then find input locations, submit payloads in requests, and monitor the database for successful injections.	101
4.3	Four federated workers of SQIRL fuzzing random targets of the SMB.	118
4.4	PCA of existing payloads and those from SQIRL.	122
5.1	The REST API testing process using APIRL.	131
5.2	Part of the OpenAPI specification for VAmPI.	132
5.3	Improvements of APIRL compared to baselines.	143
5.4	The unique bugs found by each approach per REST API tested.	144
5.5	SHAPley Values for all features in descending order of importance for APIRL. .	151
6.1	The CLUTCH architecture shown within a fuzzer.	166
6.2	Throughput of test cases (five run average and standard deviation) from 24h fuzzing of JavaScript engines.	174
6.3	Branch coverage (five run average and standard deviation) from 24h fuzzing of JavaScript engines.	176
6.4	Analysis of 10,000 mutated test cases by CLUTCH and the equivalent fuzzer when tested on V8. The left column shows the percentage difference from CLUTCH selection in its top ten selected instructions per mutation. The right column shows the difference between clutch and the baseline fuzzer in terms of the errors caused per mutation.	179
6.5	Distribution of error types caused by 10,000 test cases from different fuzzers, when testing on V8.	180
6.6	Frequency of branches in V8 that are reached from 10,000 test cases from Fuzzilli and FClutch.	182

List of Abbreviations

A2C - Advantage Actor Critic

API - Application Programming Interface

ANN - Artificial Neural Network

AST - Abstract Syntax Tree

CVE - Common Vulnerabilities and Exposures

DBMS - Database Management System

DNN - Deep Neural Network

DQN - Deep Q-Network

DUT - Device-Under-Test

DOM - Document Object Model

GAN - Generative Adversarial Network

GRU - Gated Recurrent Unit

HTTP - Hyper Text Transfer Protocol

LLM - Large Language Model

LSTM - Long Short-Term Memory

MDP - Markov Descension Process

MAB - Multi-Armed Bandit

NVD - National Vulnerability Database

OWASP - Open Web Application Security Project

PCA - Principal Component Analysis

PCM - Payload Control Module

PPO - Proximal Policy Optimisation

REST - REpresentational State Transfer

RL - Reinforcement Learning

RND - Random Network Distillation

RNN - Recurrent Neural Network

Seq2Seq - Sequence-to-Sequence

SMB - SQL injection MicroBenchark

SQLi - SQL injection

URI - Uniform Resource Identifier

URL - Uniform Resource Locator

WAF - Web Application Firewall

Webapp - Web Application

XSS - Cross Site Scripting

XP - XSS Payload

Chapter 1

Introduction

Cyber security has become an increasingly large and diverse battleground in recent years. Adversaries are able to exploit errors and flaws in programs and systems to conduct malicious activities, including: theft of private information, installation of ransomware, and remote code execution. The scale of this problem continues to grow with time, causing a significant impact on users, businesses, and the global economy. According to a recent report from IBM, the average cost of a single data breach resulting from a cyber security attack was estimated to be \$4.88 Million USD in 2024, a 10% increase compared to the previous year [128].

Despite new defences and security best practices, adversaries constantly adapt: bypassing defences, and exploiting new (or zero-day) vulnerabilities. A recent Joint Advisory Report from the Five Eyes¹ cybersecurity centres showed that a) zero-day vulnerabilities were exploited 10% more in 2023 than in previous years, and b) that 9 of the 15 most exploited vulnerabilities related to web security [54].

The persistence of web vulnerabilities is an ongoing concern that has lead to projects like the OWASP Top 10 [215]. This project aims to raise awareness of the critical security risks seen by developers and web application security professionals. To further demonstrate the challenge presented by web security, we need look no further than the US National Institute of

¹A Security and Intelligence alliance comprising Australia, Canada, New Zealand, the United Kingdom, and the United States.

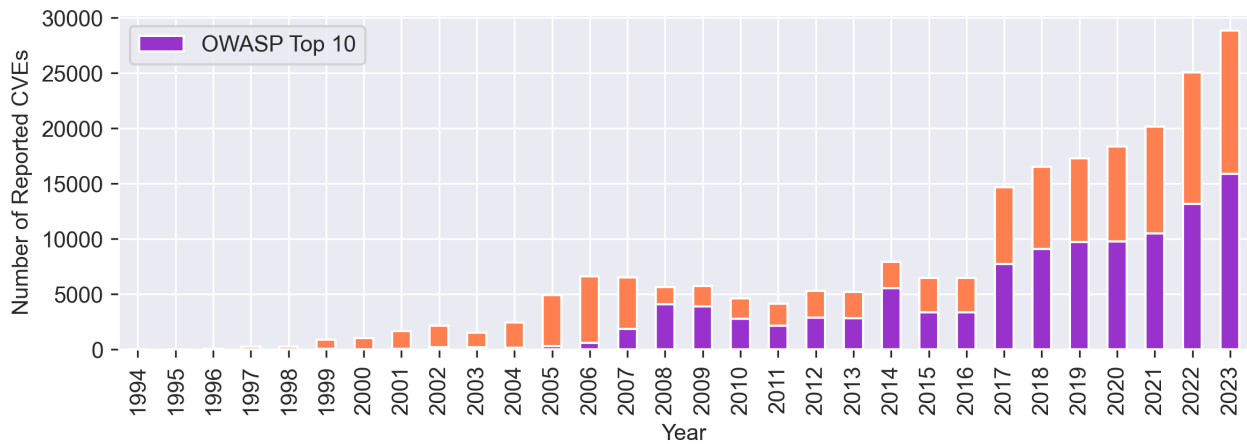


Figure 1.1: Number of CVEs reported to The National Vulnerability Database [202]. Showing those the proportion that are related to the most recent OWASP Top 10 [215].

Standards and Technology’s (NIST) National Vulnerability Database (NVD)². This repository houses confirmed vulnerabilities known as Common Vulnerabilities and Exposures (CVEs). In Figure 1.1 we can see the total CVEs reported from 1994 and the proportion which relate to the most recent OWASP Top 10 [214]. It is clear that vulnerabilities characterised by the OWASP Top 10 present a legitimate threat, with the number of related CVEs increasing year-on-year. In fact, the OWASP Top 10 are prevalent to the point that, in recent years, *they represent more than half of all reported CVEs*.

Indeed the risk presented by web application (in)security has been a growing concern which has focused on a number of areas. A continual threat is present in injection attacks, which account for 29.17% of all reported CVEs in 2024 [202], with Cross Site Scripting and SQL injection being the most, and third most common CVE respectively [188]. Additionally, the reliance on APIs in modern web infrastructure, and the risk they pose, led OWASP to create a ‘Top 10’ for APIs in 2019 [213]. Another area with increased interest in recent years is vulnerabilities in browser engines [96, 246, 248, 296, 297]. The risk from these vulnerabilities is taken serious by developers; for instance, Google Chrome maintainers offer up to \$250,000 USD for security bugs [95].

The cause of web vulnerabilities is commonly due to misconfiguration, flaws, or a lack of im-

²NVD does not contain vulnerabilities specific to any specific country, it includes vulnerabilities reported from around the globe.

plementation and best-practice defences [23, 255]. These causes can all be commonly referred to under the broader category of *bugs*, while vulnerabilities represent security related bugs that can be exploited for malicious purposes. Yet, while non-security related bugs represent a separate and distinct class, a statistically significant correlation between ‘in-production’ bugs and ‘post-release’ vulnerabilities exists [195]. Thus, even non-security related bugs are concerning in their possible consequences, and as a result should be found and patched.

Finding bugs and vulnerabilities is fundamentally important to ensuring the robustness and security of web applications and the systems on which they run. However, systems and applications are becoming increasingly large and complex; resulting in difficulty finding bugs and vulnerabilities which equally becoming more complex [21, 33, 318]. As a result, manual review and testing techniques are often associated with a significant expertise, time, and can lead to false negatives [203].

This has resulted in automated approaches to web security testing including source code analysis [109, 110, 237], and vulnerability scanners [56, 119, 155, 231, 273]. Automated scanners and testers often generate, or mutate, a series of test cases that attempt to find a bug or vulnerability. However, existing approaches [52, 96, 153, 169, 215, 233, 241, 273] are often inefficient, requiring a significant number of attempts. For example, XSSer [234] (a tool for finding Cross Site Scripting (XSS) vulnerabilities) uses 1,293 payloads to test *every* potentially vulnerable input in a web application.

Indeed, the use of random search strategies, manual heuristics, and failure to harness feedback makes testing significantly harder [39, 153]. While in the limit existing automated scanners may find bugs and vulnerabilities, their inefficiency makes them unscalable, and has the potential to *miss legitimate vulnerabilities* [153, 196]. This is to say nothing of the possibility of illegitimate bugs (false positives), which additionally reduce the usability and trust in scanners [179, 196]. Thus, motivating the need for more efficient and effective ways to conduct automated web application security testing.

This Thesis presents a novel approach to web application security testing using Reinforcement Learning (RL). RL can be scaled to high dimensional state-action spaces by training deep learn-

ing models; *deep* RL uses a neural network to learn a policy of exploitation vs exploration [275]. Practically, deep learning architectures can learn representations of complex functions and data. Deep RL has shown better than human level performance in multiple game playing settings including: the game of Go [191, 258, 259], Atari [190], Dota 2 [208], and Gran Turismo [309]. Furthermore, it has shown incredible performance in complex settings, learning to: improve the speed of matrix multiplication [80], perform human-level control with robotics [55], and design both computer chips [187] and protein structures [136].

However, there has been no comparable equivalent in security. While deep RL has been applied to web application security testing tasks, existing works use off-the-shelf deep RL models [36, 51, 150, 153, 160, 184, 246]. As such, they suffer from a number of limitations. First, security testing tasks often lack consistency in their structure making it challenging to form meaningful inputs for the RL model [70, 117, 321]; resulting in the use of simplistic feedback [39, 86, 141]. Second, the diversity of generated outputs and fixed-length nature of neural networks, often leads to bootstrapping from existing expertise [36, 153, 246]. By taking a problem-first approach we develop bespoke deep RL models that adapt to specific web security problems, to increase the efficiency of testing. Specifically, learning policies capable of generating injection payloads, mutating existing API test cases, and targeting mutations in JavaScript tests for browser engines.

1.1 Motivation

Existing Approaches Lack Targeted Search Strategies. Web application security testing tasks often have a large search space, due to the number of possible test cases it may be possible to create. However, the number of *possible* test cases is greater than the number of *valid* test cases that can bypass syntactic and validity checks. Equally, this number of *valid* test cases is larger than the number of test cases that *cause a bug or vulnerability* [107]. Thus, creating test cases that reach this criteria is challenging, as test cases can have a broad range of possible behaviours, not all of which are useful for testing. As a result, many existing approaches

focus on using extensive search strategies that many include redundant testing. This is in an attempt to enumerate the possible behaviours of the test subject by executing test cases as fast as possible, in the hope that some result in abnormal behaviour [156]. Thus, a more effective search strategy over the possible test cases would result in wasting less time and resources on ineffective testing paths.

An RL agent can be trained to conduct an effective search strategy by learning to maximise its reward. Designing, or shaping, rewards can be a challenging task, incorrect design of rewards can lead to *reward hacking*, where an agent exploits rewards learning a worse policy than intended [260]. Similar behaviour has been observed in security applications, leading to poorly performing policies [25]. Therefore, we want to use rewards that result in performant policies, and explore how rewards can effect the learnt policy of agents.

Reinforcement Learning is a Natural Security Tester. RL is the fundamental paradigm for learning from *interactive* experience, balancing the exploration of new states, whilst exploiting known strategies [275]. Similarly, testing principles share many similar concepts with RL. In particular, consider an automated scanner and an RL model. The automated scanner will alter a test case through a series of mutations or rules, with feedback on test case performance to prioritise test cases over time. The RL agent will alter a ‘state’ via a series of ‘actions’, using ‘reward’ to guide the overall strategy. However, RL is able to *learn* from its experiences to target specific behaviours, while such techniques are limited in automated testing [186]. Thus, we postulate that an RL approach to web security testing could lead to improved efficiency and effectiveness.

We want agents to learn multiple ways to complete testing objectives, not approach all problems in the same way. Accordingly, it is important for our RL agents to search over the state space to find multiple terminal states, not just the same one. For example, it should generate different payloads that find vulnerabilities. This could be considered as analogous to completing different levels in a video game. Learning such policies can be a challenging task for agents as they must understand the subtleties of different ‘levels’.

A Problem-First Approach. Web application security testing is a broad domain, with many areas to explore, including: automated vulnerability scanning via generative and mutation techniques, bug detection, and fuzzing. We detail such topics and applications further in Chapter 2. To explore a single area in detail, narrowing in to develop a specific approach may yield impressive results by designing complex systems [51, 221]. However, to do so would narrow the scope of this thesis, and not evaluate RL against a broader backdrop. Thus, we argue for developing RL solutions that target diverse challenges *across* web application security testing, an area that has had little study to characterise the ability of RL [199].

Harness Contextual Information. While some approaches have used RL for web security testing [39, 71, 76, 153, 184], they make use of simplistic, or manually curated feedback for states. Using domain specific knowledge in a representation *may* be enough to find bugs and vulnerabilities using RL. However, using such knowledge can introduce bias in learning, and the resultant RL policy [115]. Furthermore, the variation and diversity of potential state representations, both across and within testing tasks, accentuates the issue, as there is no standardised representation that can be used [117]. We want to leverage information-rich data structures to provide meaningful, application specific, latent representations for RL models.

Improving the Strategy of Testing. RL models are also known for their ability to perform tasks with long term strategy [275]. This ability has had strong performance in a number of other domains, with RL models learning impressive strategies [55, 80, 222, 309]. However, the use of RL in web application security testing has shown comparatively narrow strategy, mainly providing bootstrapping from existing domain knowledge, or framing the approach to restrict the overall capacity of an RL model [51, 71, 72, 153, 159, 323]. Thus, in this thesis we want to determine the ability of RL perform testing. However, to do this we will need to develop RL architectures capable of handling the large state-action spaces, and explore the ways in which we can vary possible strategies.

1.2 Objectives

Based on the motivations set out in the previous section, this Thesis aims to study the ability of RL models to improve the efficiency and effectiveness of performing test case generation and mutation in web application security testing. We formulate the objectives concretely as follows:

1. To determine the applicability, capability, and capacity of RL approaches to overcome challenges in web application security testing.
2. To design appropriate RL approaches that are tailored to solve the unique task specific challenges in web application security testing.
3. To investigate how differing feedback representations in both the state and reward influence how RL models learn strategies.
4. To implement RL approaches that harness information from testing tasks to efficiently search over the state-action space to effectively find bugs and vulnerabilities.
5. To evaluate the proposed approaches rigorously, striving to continuously develop the evaluation methodology.

1.3 Contributions

In this Thesis we present several novel contributions:

- We start by introducing an RL approach to find a single vulnerability. Using a hierarchical RL model, agents specialise in separate games to generate a single payload that causes a vulnerability: XSS. The approach is designed to compartmentalise the strategy of the RL agents to determine if it is possible to learn the sub tasks of generating payloads that find vulnerabilities. We implement this as HAXSS, releasing it as an open-source tool: <https://github.com/ICL-ml4csec/HAXSS>. HAXSS can crawl a web application

for input sources, testing for XSS, and report successful payloads. We compare our approach against a number of state-of-the-art scanners on a new micro-benchmark for XSS payload generation we develop, and on a macro-benchmark of established vulnerable web applications. HAXSS outperforms the other scanners on both benchmarks, identifying 131 vulnerabilities (a 20% improvement over the closest scanner), reporting 0 false positives. Finally, we demonstrate that our approach is practically useful, as HAXSS re-discovers 4 existing CVEs and discovers 5 zero-day CVEs in 3 production-grade web applications.

- Building on the insights from HAXSS we develop a *single* agent that can generate payloads. Focusing on SQLi vulnerabilities, where payloads must adhere to stricter syntax requirements than XSS equivalents, we target a harder class of vulnerabilities for a single agent to find. We develop a deep latent representation for an RL agent based on grey-box feedback from a web application database, we call this approach SQIRL. SQIRL learns to develop diverse payloads from a potentially infinite payload space by taking a variable number of actions, completing three SQLi tasks: fixing syntax, escaping contexts, and bypassing sanitisations. We explain the significance of our model design choices with an ablation study, and make our implementation publicly available at <https://github.com/ICL-ml4csec/SQIRL>. We compare SQIRL against 5 state-of-the-art scanners on a novel benchmark and 14 web applications. SQIRL significantly outperforms other scanners finding the most SQLi vulnerabilities (63) of any scanner, with zero false positives. We have responsibly disclosed 22 novel vulnerabilities found by SQIRL, grouped in 6 CVEs. Impressively our approach does all this efficiently, using substantially fewer requests than most of the state-of-the-art scanners compared with.
- We investigate an important, yet, alternative task compared to the previous two: mutation of existing test cases. By applying a single agent, APIRL, to the task of mutating HTTP request templates, we are able to test REST APIs for bugs in their implementation. This overcomes key challenges in using contextual information in REST APIs by harnessing a pretrained transformer. This provides an improved representation of complex and diverse JSON-structured data present in HTTP responses. As a result, our RL agent benefits from richer information, learning the subtleties relating to test outcomes. We

release our implementation at <https://github.com/ICL-ml4csec/APIRL>. We study the implications of reward functions used in web application security testing via an ablation study of both design choices and 7 reward functions. We show APIRL can find significantly more bugs than the state-of-the-art in real world REST APIs while minimising the number of required test cases.

- We explore a core challenge of fuzzing JavaScript engines, left unexplored in prior work: where to perform mutations in a test case. We postulate it is possible to learn *where* to target mutations to improve efficiency. A challenging problem to solve with heuristics, it is commonly done via random selection. Thus, we present the first study on how to target locations in test cases when fuzzing. We design CLUTCH, a deep contextual bandit that locates where to perform mutations in test cases when testing JavaScript engines. In doing so we overcome several challenges related to location selection, and the use of a learning based architecture. We test CLUTCH by embedding it inside three existing state-of-the-art fuzzers and test on the three most common JavaScript engines. We show that CLUTCH, produces 20.3% more valid JavaScript, covers 2.3% more branches, and increases coverage per test case by 8.9%. We then analyse 10,000 test cases from all fuzzers (with and without CLUTCH) to determine the different strategy used by CLUTCH. We also release our approach at: <https://github.com/ICL-ml4csec/CLUTCH>.

1.4 Publications

- **Myles Foley** and Sergio Maffeis. Haxss: Hierarchical Reinforcement Learning for XSS Payload Generation. In *2022 IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pages 147–158, December 2022. ISSN: 2324-9013 Presented as Conference Paper, and awarded **IEEE Best Student Paper**, it is included in Chapter 3.
- Salim Al Wahaibi, **Myles Foley**, and Sergio Maffeis. SQIRL: Grey-Box Detection of SQL Injection Vulnerabilities Using Reinforcement Learning. In *Proceedings of the 32nd*

USENIX Security Symposium (USENIX Security 23), pages 6097–6114, 2023. Presented as Conference Paper, it is included in Chapter 4.

- **Myles Foley** and Sergio Maffei. APIRL: Deep Reinforcement Learning for REST API Fuzzing. In *Proceedings of the AAAI Conference on Artificial Intelligence*. February 2025. Presented as Conference Paper, it is included in Chapter 5.
- The contents of Chapter 6 is currently under review for a conference publication.

1.4.1 Additional Publications

Additionally, the author has contributed to the following publications which discuss related concepts to this Thesis. However, they are auxiliary to the overarching theme of this Thesis and as such are not discussed here, but provided for interested readers.

- Chris Hicks, Vasilios Mavroudis, **Myles Foley**, Thomas Davies, Kate Highnam, and Tim Watson. 2023. ‘Canaries and Whistles: Resilient Drone Communication Networks with (or without) Deep Reinforcement Learning’. In *Proceedings of the 16th ACM Workshop on Artificial Intelligence and Security*, 91–101. AISEC ’23.
- **Myles, Foley**, Ambrish Rawat, Taesung Lee, Yufang Hou, Gabriele Picco, and Giulio Zizzo. 2023. ‘Matching Pairs: Attributing Fine-Tuned Models to Their Pre-Trained Large Language Models’. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*.
- **Myles, Foley**, Mia Wang, Zoe M, Chris Hicks, and Vasilios Mavroudis. ‘Inroads into Autonomous Network Defence Using Explained Reinforcement Learning’. In *Proceedings of the Conference on Applied Machine Learning in Information Security*, CAMLIS 2022.
- **Myles, Foley** Chris Hicks, Kate Highnam, and Vasilios Mavroudis. 2022. ‘Autonomous Network Defence Using Reinforcement Learning’. In *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security*, 1252–54. ASIA CCS ’22. Awarded **Best Poster**.

1.5 Thesis Outline

- **Chapter 2.** In this Chapter we will investigate the fundamental knowledge required for the rest of this Thesis, including: RL, deep learning, deep RL, fuzzing, and web application testing. During this Chapter we will discuss related work to understand existing approaches for web application security testing, and see where RL has already been used within this broad domain. Doing so will identify the gaps in the literature which this Thesis aims to investigate.
- **Chapter 3.** This Chapter introduces HAXSS, a hierarchical RL approach to finding a single vulnerability: XSS. In an ablation study we explore how different RL approaches perform. By comparing against state-of-the-art approaches we show that hierarchical RL models can develop policies to find real world vulnerabilities.
- **Chapter 4.** This Chapter builds on the findings from Chapter 3, going on to develop a single agent that generates payloads. We show that the single agent approach can find more vulnerabilities in our novel microbenchmark and in 14 production-grade web applications than state-of-the-art approaches.
- **Chapter 5.** In this Chapter we investigate the ability of an RL agent to mutate existing HTTP test cases to find for bugs in REST APIs. This further builds on previous Chapters by improving representations for RL models. We show this approach can efficiently outperform state-of-the-art approaches to find more bugs in fewer requests. We further investigate how reward functions effect learnt policies in web application security testing.
- **Chapter 6.** In this Chapter we develop a deep contextual bandit that learns where to perform mutations in a JavaScript test case. Our novel approach is then placed inside three state-of-the-art JavaScript engine fuzzers, and increases the proportion of valid test case, covering more branches, and increases *efficiency* of coverage per branch.
- **Chapter 7.** This Chapter concludes the Thesis by summarising its achievements, and limitations. Finally, we promote avenues of future work that can be lead on from this Thesis, and briefly discuss the potential of RL in a wider cyber security context.

Chapter 2

Background

In this Chapter, we aim to explore the fundamental knowledge, ideas, and techniques that will be used in the remaining Chapters. We can then explore the domain of web application security testing. This is with a view to understand how different learning based techniques have been used to conduct testing, their appropriate use, and their limitations. To do so will motivate the research directions of this Thesis, and its contributions.

2.1 Reinforcement Learning

To begin we will examine RL, to understand what it is and how it can be used.

RL is a form of machine learning that tries to optimise a given reward, r , whereby a learning agent is not told what to do but, instead must maximise this reward. RL is neither supervised or unsupervised learning but instead follows an exploration vs. exploitation approach [275]. This is an inherently challenging task, as an agent must try new actions and then begin to favour those that work best. To do so immediately could cause a potential loss to future rewards. To maximise the reward over time, RL accounts for the delay of future rewards by factor γ [275].

The learning agent must have knowledge about the environment in which it exists. The information is presented to the agent as a state s . The agent can attempt to affect its state

by taking an action a . This information is in fact the core constituent of a Markov Decision Process (MDP). An MDP is the underlying model of the RL problem. The components of which have been touched upon, but are detailed as:

- A finite set of states, $s \in S$, the agent begins at s_0 progressing to s_t at time t .
- A finite set of actions $a \in A$, with the agent taking action a_t at time t .
- A reward function R , which is a function of taking an action in a given state, $r : s \times a$. We define r_{t+1} as the reward at $t + 1$ for taking action a_t in state s_t
- A probability P that action a is taken to transition from state s to s'
- A discount factor γ that is used to reduce the weight of the long term rewards, this accounts for the longer term rewards in states as t increases.

An agent will learn to perform a given task in an environment. The agent can achieve this task by taking a number of steps in an *episode*. Repeating the task over many episodes the agent will learn a policy π for the task. Given enough time the agent can perform the task efficiently and succinctly, resulting in an optimal policy π^* .

Having an agent effectively explore the environment is important in RL, as this exploration will result in the possibility to find the ‘best’ reward in the environment. A common way to achieve this is via an ε -greedy algorithm [190, 275]. This uses a parameter ε which will take the ‘greedy’ or optimal action with probability $1 - \varepsilon$. In some cases ε can be subject to decay, in such instances it is initially high in the beginning of training decaying to 0, or some predefined minima, this is known as ε -greedy decay.

Work on RL is well established, with a book on its introduction first being published in 1998, and its second revision completed in 2018 [275]. Since then, many related works have followed, including seminal from Mnih *et al.* [190, 191], which we will discuss further in Section 2.3.

When it comes to RL there are two main forms of decision making that fundamentally arise from the MDP, but are subtly different in what, and how, they learn.

Value Based Methods. The goal of this class of methods is to learn a single deterministic action, from a set of discrete actions, by finding the maximum value. Specifically, they learn a *Value Function*. The policy for which action to take can then be derived from the Value Function of the states.

An optimal policy can be formally described by the Bellman Optimality Equation (Eq. 2.1), having two inputs, the action and state, and returns the Q-value for each pair $q(s, a)$ [275].

$$q_\pi(s, a) = \mathbb{E}_\pi[R_{t+1} + \gamma \max_{a'} q_\pi(S_{t+1}, A_{t+1}) \mid S_t = s, A_t = a] \quad (2.1)$$

A common way to learn Q values is via *Q-learning* as defined in Eq. 2.2.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \right] \quad (2.2)$$

Q-learning is an off-policy method, meaning it can learn from a separate policy than the one being learnt (*e.g.* random action selection instead of Q). Thus, the learned function Q approximates the optimal policy q_π independent of the policy being followed. With enough visits to state-action pairs, the policy will update to converge towards q_π .

Policy Based Methods. In such methods the goal is to learn a policy directly, instead of learning the value of states and actions. In doing so a policy is then a mapping function between states and actions. In a tabular setting this is important when state-action spaces are large, as Policy Based Methods do not require the extraction of the policy from the Value Function. The agent achieves its policy by trying to maximise some objective function (J), often seen as a gradient. The Policy Gradient theorem is seen in Eq 2.3.

$$\nabla J(\theta) \propto \sum_s \mu(s) \sum_a q_\pi(s, a) \nabla \pi(a \mid s, \theta) \quad (2.3)$$

Where θ is the policy parameters. The distribution μ is seen to be the normalised frequency of visiting a given state.

2.1.1 Multi-Armed Bandits

The Multi-Armed Bandit (MAB) is a classic RL problem - so called after ‘one-armed bandits’ or slot machines. It is a special case of the MDP where all actions return to the same state, forming an MDP with only one state. In these problems the agent can take pick a single ‘arm’ (or action) and observe the reward; the goal of the agent is to maximise this reward over a given number of time steps. In this way it can be seen to have an episode length of 1, and a maximum number of episodes N . Thus the agent must explore the actions to find the optimal strategy to maximise this reward over N [275].

MABs have seen success in security tasks as they require minimal modelling of the security problem. However, there are few problems that can be modelled in this way. As a result, MABs have primarily been limited to seed prioritisation and selection in mutation-based fuzzing.

Contextual Bandits

A special case of the MAB, *Contextual Bandits* form a ‘halfway house’ between a MAB and MDP. Some tasks may evolve over time, and as a result they require additional information which can be used to inform the choice of arms.

Consider the task of determining which of two slot machines has the higher payout in a casino (A): the task is trivial after several attempts. Now consider a second identical casino (B) where the payout of the machines is flipped. Again, we can find the better machine in B after some error. Finally, consider being randomly placed in A or B and having only one attempt to select the slot machine with the highest payout. As we do not know which casino we are in (as everything is identical), the best possible guess rate is 50%. We can solve this problem by being given some *context*, *i.e.* being informed of which casino we are in.

Providing this contextual information to the bandit is beneficial as it can observe the effect of each of the actions it can take, and detect the shift between different tasks [275]. We will see the utility of contextual MAB approaches in Chapter 6.

2.2 Deep Learning

Deep learning is a complex approach to machine learning that harnesses ideas from neuroscience to create Artificial Neural Networks (ANN). These are comprised of multiple layers of interconnected neurons. Typically these feed-forward neural networks consist of many fully-connected layers to form a Deep Neural Network (DNN). Using the backpropagation algorithm DNNs are able to learn approximation functions based on training data [243]. There are many different architectures that are used in deep learning that can be used in different ways, including Multi-Layer Perceptions (MLPs), and Convolutional Neural Networks (CNNs).

We will delineate here the deep learning architectures that are relevant to later Chapters. We will see in Section 2.3 how deep learning can be used in conjunction with RL to learn a policy. Additionally, we will consider how different deep learning approaches can be used to learn representations. In particular, Recurrent Neural Networks in Chapter 4, Transformers in Chapter 5.

2.2.1 Recurrent Neural Networks

Recurrent Neural Networks (RNNs) are a class of DNN that can model time series sequential data. An RNN utilises ‘memory’ between recurrent units to carry information between timesteps. The memory of an RNN is formed of a hidden layer which is updated using the previous hidden layer and current input. This can be formalised as having an input x_t , and hidden state h_{t-1} , to give output y_t .

$$y_t, h_t = f_\theta(x_t, h_{t-1}) \quad (2.4)$$

where f_θ is a function that represents an RNN with neural network weights θ .

Contemporary RNNs tend to fall into two main architectures: Long Short-Term Memory (LSTM) [91], and Gated Recurrent Networks (GRUs) [46].

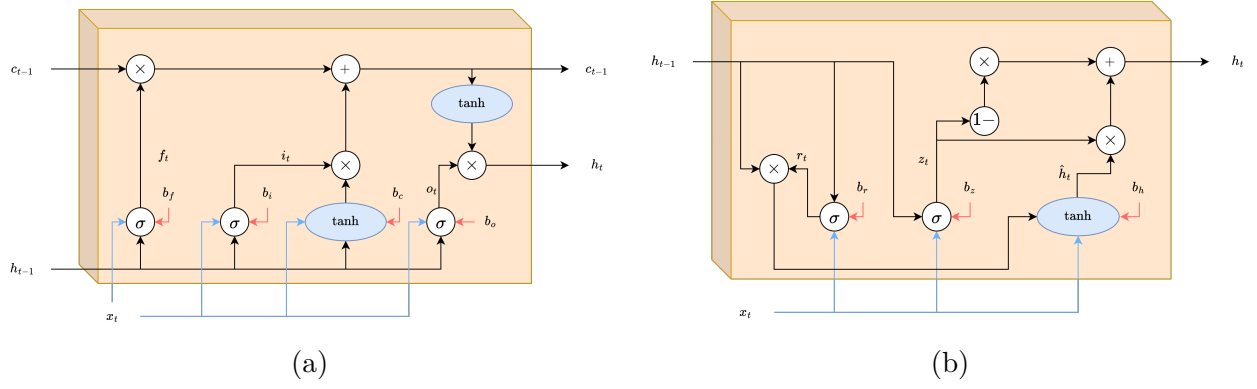


Figure 2.1: Recurrent Neural Network Architectures. a) Long Short-Term Memory b) Gated Recurrent Unit.

Long Short-Term Memory.

The LSTM architecture (shown in Figure 2.1) was first introduced to overcome the vanishing gradient problem [91]. Each LSTM unit comprises a cell c_t , input gate i_t , output gate o_t , and forget gate f_t . The cell functions to ‘remember’ values over time, while gates regulate the flow of information to and from the cell. Forget gates control the information from the previous state by regulating h_{t-1} and x_t , retaining or discarding information. Input gates function similarly to determine the information retained on input to the cell. Output gates control the information that is present in the output of the cell, learning to select the relevant information useful for long term dependencies, to create the hidden state h_t . The LSTM gates and memory cell can be formally defined as:

$$\begin{aligned}
 f_t &= \sigma(W_f x_t + U_f h_{t-1} + b_f) \\
 i_t &= \sigma(W_i x_t + U_i h_{t-1} + b_i) \\
 o_t &= \sigma(W_o x_t + U_o h_{t-1} + b_o) \\
 \tilde{c}_t &= \sigma(W_c \cdot [h_{t-1}, x_t] + b_c)
 \end{aligned} \tag{2.5}$$

Where $\tilde{c}_t$ is the cell input activation vector, σ is the sigmoid activation function, and $W \in \mathbb{R}, U \in \mathbb{R}, b \in \mathbb{R}$ represent weight matrices to be learnt during training. By initialising both c_0 and h_0 to 0, we can derive the final output of the LSTM using the element wise product $\odot$ as

in Eq 2.6 [91].

$$\begin{aligned} c_t &= f_t \odot c_{t-1} + i_t \odot \tilde{c}_t \\ h_t &= o_t \odot \tanh(c_t) \end{aligned} \tag{2.6}$$

LSTMs have been used in a variety of applications, including traffic analysis [120], payload classification and detection [79, 81, 277], anomaly detection [257], and program generation [165].

Gated Recurrent Network. GRUs [46] modify the LSTM architecture to use only two gates: the update gate z_t , and the reset gate r_t . As such they require fewer trainable parameters, while maintaining the ability to model sequential data. GRUs can be defined in a similar way to LSTMs, as we can see from Eq. 2.7:

$$\begin{aligned} z_t &= \sigma(W_z x_t + U_z h_{t-1} + b_z) \\ r_t &= \sigma(W_r x_t + U_r h_{t-1} + b_r) \\ \hat{h}_t &= \tanh(W_h x_t + U_h h_{t-1} + b_h) \end{aligned} \tag{2.7}$$

Where $\hat{h}_t$ is the activation vector, and again $W \in \mathbb{R}, U \in \mathbb{R}, b \in \mathbb{R}$ represent weight matrices to be learnt during training. Now we can define the output of the GRU from the variables computed in Eq. 2.7:

$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \hat{h}_t \tag{2.8}$$

GRUs have also been used for different security tasks including control tasks in Industrial Control Systems [310], network anomaly detection [19, 126], and vulnerability detection [236].

A comparative study between LSTMs and GRUs for web application security testing was conducted by Sablotny *et al.* [244, 245]. This study found that GRUs outperform LSTM based architectures when generating HTML to test web browser.

2.2.2 Transformers

Modern RNN architectures can handle the long range dependencies of sequential data, however in complex, or high dimensional data they can struggle [287]. Vaswani *et al.* [287] proposed the transformer architecture. A key innovation of the transformer is self attention to capture dependencies between inputs, as seen in Eq 2.9.

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \quad (2.9)$$

Where K and V are an encoded representation of the input as key-value pairs, and the previous output is compressed to matrix Q . By computing the weighted sum of the similarities in the input sequence the model *attends* to the relevant parts of the input sequence; meaning the model does not rely on recurrent connections as in RNNs.

Transformers are commonly trained on text based data, such as natural language or source code [110, 287]. Where sequences are represented as a series of tokens, which can be created via a number of popular techniques including Byte Pair Encoding (BPE) [89], WordPiece [252], and Unigram [148]. Regardless of the tokenisation strategy employed the training task is often the same: Masked Language Modelling (MLM). In this task, random tokens within the sequence are masked, and the model must predict the true, original token. By training the model this way it learns a general understanding of the underlying structure of the input sequence, including the syntactic and semantic relationships between tokens.

Transformers of different sizes, including Large Language Models (LLMs) have been used for a variety of tasks in security, including source code vulnerability detection [109, 110], testing firewalls [98, 162], learning representations for hardware testing [171], and fuzzing tasks [61, 185].

2.2.3 Encoder-Decoder Models

Encoder-Decoder or Sequence-to-Sequence (Seq2Seq) models are a type of deep learning model that transform a sequence of input data into an output sequence [47]. In such models the encoder translates the input sequence to a fixed-length vector, creating a compressed representation of the input sequence, a *latent* representation. The decoder uses the encoder output to produce, one element at a time, an output data sequence. Such models can be based on different deep learning architectures including transformers, and RNNs.

Training these models is often done via a form of supervised learning. Using a dataset comprised of input sequences and output sequences the model is trained to predict the output given the input. During training the goal is to minimise the difference between the predicted output sequence and the true output sequence. This is typically modelled in training as a loss function based on cross-entropy.

Commonly these models are used for tasks related to translation, and summarisation tasks [3, 46]. However, these models can also be used for a separate task of learning representations, as an *autoencoder*. In such models the goal is to learn a compressed, latent representation of the data. In doing so the model doesn't learn to predict a new sequence, but learns to predict the same data [267]. During training the model must minimise the reconstruction loss between the input and output representation. As a result, the vector output of the encoder becomes an alternative representation of the input. This will be used in Chapters 4 and 5 to provide information rich state-representations.

2.3 Deep Reinforcement Learning

We have introduced thus far, RL and deep learning as separate topics. However we will now consider the overlapping area of *deep* RL.

While RL is able to succeed at many different tasks, it suffers from the 'curse of dimensionality' as state-action spaces become large [275]. We can utilise deep learning techniques as a way to

approximate the policy and value functions when in new states. By doing so we can account for these larger state-action spaces. In this Section we will discuss several popular approaches to Deep RL that we will utilise in this Thesis.

2.3.1 Deep Q-Learning

Deep Q-learning was first introduced as the Deep Q-Network (DQN), which uses a DNN to estimate the Q-value (Eq. 2.1) of states and actions [191]. An example of a DQN can be seen in Figure 2.2, comprising a DNN as Q-network Q which is used to compute Q-values for the set of actions. By comparison to other deep learning tasks the notion of a labelled dataset is not common in Deep RL, as such the calculation of the loss function must come from elsewhere. Mnih *et al.* [191] proposed using a semi-supervised approach by using a target Q-network ($\hat{Q}$) which is used to determine the loss of the agent. Both Q and $\hat{Q}$ are initialised to have the same random weights. During training the weights of Q are updated via backpropagation, whilst $\hat{Q}$ is periodically updated with the weights of Q after N episodes have occurred. This allows for the convergence of the two networks to the optimal policy π^* . Appendix C shows the DQN training algorithm. In order to increase the stability of the DQN, minibatches of *transitions* are used as input to the DQN; these transitions are tuple of state, action, reward, and next state (s_t, a_t, r_t, s_{t+1}) . These are stored in a replay buffer D and randomly sampled, reducing the correlation of the transitions.

In order to overcome the exploration problem seen in RL, the agent in Algorithm C.1 follows the ε -greedy approach, taking a random action given by ε as described in Section 2.1.

The success of the DQN was seen since its introduction, achieving human and super-human performance in game playing settings [258, 259]. As a result, it has been used on many subsequent tasks including security applications, such as Intrusion Detection Systems [22, 254] and Autonomous Cyber Defence [108].

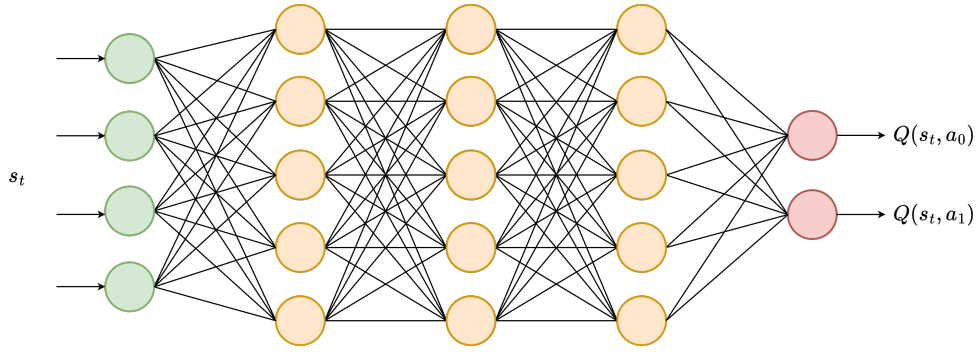


Figure 2.2: Example DNN architecture of a DQN that given a state s_t computes Q-values for each action

Variants on the DQN

There are many variations on the implementation of the DQN which have been suggested to improve its performance in RL game playing settings. For example, Double DQN [285] to reduce overestimation of Q-values, Average DQN [10] that averages network weights over time, DQNs using RNN architectures [159], and approaches that combine the different variants of DQN [116]. We present here some of the key advances in this architecture that we will use in this Thesis.

Duelling DQN. Introduced by Wang *et al.* [304] the Duelling DQN leverages elements of the actor-critic design (discussed in detail in Section 2.3.2) to estimate the value function $V(s_t)$ and the action advantage (Eq. 2.11), these are formed as separate layers in the DQN architecture, but share prior feature learning layers, and are then combined in an aggregation layer to provide the Q for the current state. In particular the Duelling DQN was shown to learn an improved policy compared to the DQN in the case where states have similar values ($V(s_t)$) without requiring changes to the underlying algorithm.

Prioritised Experience Replay. As we have seen the original DQN samples uniformly at random from the transitions in order to avoid the correlation between transitions. Schaul *et al.* [249] argue instead that ‘significant’ transitions should appear more often in the minibatches. Specifically greater ‘significance’ is given to transitions associated with a higher loss. Based on the loss, each transition is then given a ‘priority’ that is used to determine the probability of being sampled. To remove the bias introduced from the change in sampling, the strategy

of importance sampling is used to assign a weight from the sampled transitions used in the updating of Q network weights.

We will see DQN based approaches used in Sections 2.4.2 and 2.5. We will further develop our own approaches in several chapters, including: Chapter 3, 4, and 5. In particular,

2.3.2 Actor-Critics

Training Deep RL methods, such as DQNs, can lead to instability when training and non-optimal solutions. It has been argued in later work by Mnih *et al.* [189] that this often arises from the high variance of policy distributions and rewards. Using a Policy Gradient approach we can redefine the objective function from Eq. 2.3 based on the expected total reward obtained from the environment. By taking a policy π as π_θ (where $\theta \in \mathbb{R}$) we can define:

$$J(\theta) = \mathbb{E}_{\pi_0} \left[\sum_{t=0}^{\infty} \gamma^t r_t \right] \quad (2.10)$$

Then to prevent the variance we can introduce a baseline to subtract from the cumulative reward, thereby reducing the variance and instability. This forms the basis of Actor-Critic methods. In these methods the most prevalent baseline is the use of the Value function. This difference between Q-values and the Value function is referred to as the *advantage* of a policy π_θ taking action a_t in state s_t , this can be seen in (Eq. 2.11) [189, 275].

$$A_{\pi_\theta}(s_t, a_t) = Q_{\pi_\theta}(s_t, a_t) - V_{\pi_\theta}(s_t) \quad (2.11)$$

As r_t can be taken as an estimate of $Q_{\pi_\theta}(s_t, a_t)$ we can use Eq. 2.3 and Eq. 2.11 to rewrite the objective function, J :

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_0} [\nabla_\theta \log \pi_\theta(s_t, a_t) A_{\pi_\theta}(s_t)] \quad (2.12)$$

This can be seen as an actor-critic formulation, where the ‘actor’ is an estimate of the policy,

π_θ , and the ‘critic’ is a baseline, here is seen as $V(s_t)$. This formulation (Eq. 2.12) is known as the advantage actor-critic due to the use of the advantage function [189].

Synchronous and Asynchronous Advantage Actor-Critic. Asynchronous Advantage Actor-Critic, or A3C, was proposed by Mnih *et al.* [189] alongside multiple asynchronous RL implementations. This work utilised ideas from Federated Learning (FL), where a number of *client* models are trained independently for a period, sending the parameters they have learned to a central *server* model, which aggregates them (typically by averaging) and then broadcasts the updated weights back to the clients

Similarly A3C uses an advantage actor-critic model with N parallel workers that exist in their own environments, this increases the diversity through training. These workers are then used to update a global network (and value function) at the end of the episode or when a terminal state is reached.

After which, the workers are reset to have the global network parameters. The policy π has entropy added to it to ensure that the networks do not converge to be sub optimal and deterministic [189].

The A3C implementation also gave rise to its synchronous variant, commonly known as A2C [308]. The problem formulation of this is very similar, still using the advantage actor-critic approach and employing parallelisation. However, in A2C agents must update the global network together, and then all agents begin a new episode updated from the same policy [308]. Both A2C and A3C have shown impressive performance in both game playing settings, and security tasks [153, 163, 177, 189, 308].

We will see how A2C can be used in web application security testing, being implemented as Link [153], a state-of-the-art approach we will use in our evaluation in Chapter 3. In Chapter 3 and 4 we will explore the use of worker agents to improve performance of RL agents.

2.3.3 Proximal Policy Optimisation

Proximal Policy Optimisation (PPO) is an efficient policy gradient method for RL [251]. Similarly to actor-critic methods PPO uses a deep neural-network to approximate state-value function $V_{\pi_\theta}(s)$ and another can model the policy π_θ . The core contribution of PPO is to modify the objective function:

$$L^{CLIP} = \hat{\mathbb{E}}_t [\min(r_t(\theta)\hat{A} - t, \text{clip}(r_t(\theta), 1 - \epsilon, 1 + \epsilon)\hat{A}_t)] \quad (2.13)$$

where ϵ is a hyperparameter, and $r_t(\theta)$ is the probability ratio $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$ (the ratio of probability under the new and old policies, respectively).

The key insight here is that a large step size change in a policy may result in a suboptimal change, without being able to recover. Similarly a small step size will reduce the training efficiency. Thus PPO introduces a clipping function (adjusted via ϵ) when performing gradient descent to reduce the size of the policy update from being too large.

Since its introduction, PPO has shown its ability to achieve state-of-the-art in a range of complex environments [207, 251]. PPO is also popularly used in Autonomous Cyber Defence settings, showing capability to defend simple networks [12, 25, 83, 86, 118].

In Chapters 3 and 5 we will see how PPO performs alongside many of the other Deep RL algorithms described in this Section.

2.3.4 Curiosity

The notion of curiosity is one that has been introduced to RL in order to ensure the agent explores the state-space well in a setting with sparse rewards. Curiosity is implemented with an intrinsic reward function. Rewards are often defined by domain specific knowledge, and in certain cases it can be done with detail to guide the agent to learn its policy. This is known as reward shaping, and is notoriously difficult to do [25]. Instead it is possible to define some

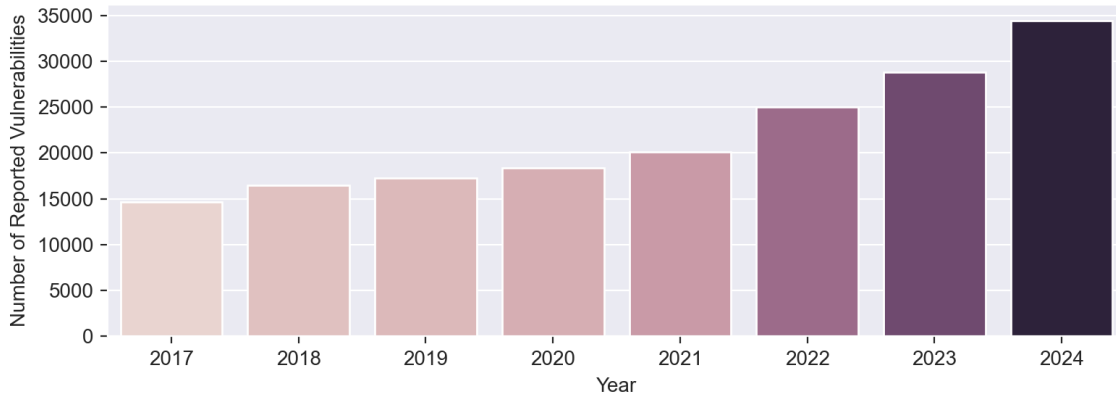


Figure 2.3: Number of CVEs reported to The National Vulnerability Database [202] from the start of 2017 to the date of submission.

intrinsic reward function that is determined not by domain knowledge but by the behaviour of the RL agent. The intrinsic reward also overcomes the problem of the sparse rewards seen in some RL problem settings, which can make it more difficult for the agent to converge on an optimal policy.

The intrinsic reward is a function of how often a given state is visited. The agent should receive a greater reward for states that it has visited less often, and a lower reward for states it has seen often. There are many ways that this can be achieved including: using a simple state visit counter to produce a reward distribution, semi-supervised techniques [222], and randomly initialised neural networks [32].

2.4 Fuzzing

We have seen different learning techniques in previous sections. Here we will delineate the fundamentals of fuzz or software testing, existing approaches, and their weaknesses. We will then go on to consider the ways in which the learning approaches we have discussed have been used in this domain.

As we can see from Figure 2.3, the need to find vulnerabilities before malicious actors is as important as ever, as the number of Common Vulnerabilities and Exposures (CVEs) is constantly on the rise. The detection of vulnerabilities quickly and efficiently before malicious actors is

key to a proactive defence.

To find such vulnerabilities is a challenging task, resulting in many different proposed discovery techniques from source code detection [109, 110, 237], differential testing [28], and fuzz testing [110, 156]. Arguably, one of the most popular of these is fuzz testing, or *fuzzing*, which has lead to many works to find security-related bugs [146, 156, 250, 302].

In fuzzing a variety of inputs (or test cases) are delivered to a Device-Under-Test (DUT), these inputs attempt to trigger abnormal behaviour in the DUT. This unexpected or abnormal behaviour is often classified as a *bug*: a fault or failure in the DUT. Bugs can come in a variety of flavours, including: crashes, logic inconsistencies, memory errors, and security related vulnerabilities.

In practice, the unexpected or abnormal behaviour occurs from a vanishingly small minority of the test cases [107]. Thus, the task of fuzzing is conducted using automated tools that aim to create many test cases, executing these to the DUT hundreds or thousands of times a second [186].

In order to target the bugs and vulnerabilities, three different levels of information about the DUT can be assumed. The most permissive of these is *white-box* testing, where complete access to the DUT is taken, allowing for source-code level analysis [110, 237]. Conversely, *black-box* assumes the opposite, only allowing test cases to be submitted to a DUT, and a response (*e.g.* termination signal) to be observed. This is the most challenging of testing approaches as there is no information to guide the testing process. To reduce the difficulty of interactive black-box testing, and guide the testing approach, *grey-box* testing can be conducted. In such a setting, *instrumentation* is used to collect an intermediary form of feedback on test cases performance from the DUT. The most popular form of feedback for grey-box fuzzing is often test case code coverage [17, 29, 35, 94, 96, 156, 185, 228, 303, 314, 315].

Fuzzers generally fall into one of two categories in their creation of test cases, being either a) generation based, or b) mutation based [156, 206]. In the first case, generation based fuzzers create test cases from scratch during fuzzing, using a grammar, or set of heuristics [16, 20, 113,

156, 296]. While mutation based fuzzers take existing *seeds* from a *corpus*, and perform a series of mutations on seeds to alter their functionality [11, 18, 156, 220].

One of the first popular fuzzing tools was *American Fuzzy Lop* (AFL) [315]. A mutation-based fuzzer, it is able to receive grey-box feedback from the DUT in the form of code coverage. This coverage guides AFL in an attempt to find bugs and vulnerabilities. This work spawned many subsequent research approaches to improve bug detection. In general, these have focused on five main ways in which to improve the performance of fuzzers:

- **Improving throughput** [242, 263, 283]. Such works aim to increase the number of test cases that can be used for testing within a given period of time. One way this can be done is by optimising the fuzzer to increase the rate at which it can output test cases. While alternative approaches attempt to improve the speed at which test cases are executed by the DUT *e.g.* by improved emulation of a DUT.
- **Improving mutations** [96, 174, 248, 290]. Mutations in AFL are low level such as: bit flips, test case splicing, and deleting parts of the test cases. However, improving the mutations to trigger alternative, or targeted behaviour, is desirable to improve fuzzer performance. This has lead to the development of different mutations that are specific to targets, or kinds of behaviour.
- **Improving bug oracles** [43, 97, 194, 299]. As we have mentioned bugs can come in a number of forms, some of which are easier to observe than others. For example, it is clear when a crash has occurred due to a test case; however, the output of logic inconsistencies or access control vulnerabilities are less clear. As a result, different oracles have been proposed to observe different classes of bug [129].
- **Targeting different areas of the DUT** [29, 34, 35]. Exercising the complete functionality of a DUT is only possible ‘in the limit’ provided with enough time or compute power. Fuzzing research has developed approaches to target specific regions within the DUT. In doing so the fuzzer attempts to *direct* its test cases towards such region. However, deciding such regions is a non-trivial problem [306].

- **Improving the instrumentation of the DUT** [17, 256, 297]. Instrumentation is the key feedback mechanism for fuzzers to determine the performance of a test case, *e.g.* providing coverage feedback. However, there are many different forms of instrumentation that exist, from branch coverage, to taint flows. By making effective use of instrumentation can provided different feedback to guide fuzzing.

However, while advances have been made in traditional fuzzing techniques they often still suffer from a number of key limitations [146, 156, 250, 302]. We present these limitations and the specific challenges related to them in Table 2.1, delineating these further here.

The first of these limitations is related to where to apply mutations. In a seed there are only certain locations that mutations can be applied at that will cause a change in the DUT. Once a location has been selected (regardless of it being ‘interesting’ or not), the mutation itself must be selected and applied to the test case. Often in prior fuzzing work these two choices are made at random [17, 242, 248, 315], or following some predefined heuristics [96, 181, 182, 299]. Such a choice is often made by favouring the throughput of a fuzzer. This is in the hope that given enough time and test cases, it can *blindly* select the interesting locations and mutations for a given test case, finding a bug.

The second limitation is the ability to locate bugs in the DUT. The ability to effectively test a DUT is often regarded by the coverage executed by a fuzzer; where a higher coverage implies greater testing ability. However, Böhme *et al.* [35] showed that fuzzers often repeatedly execute the same paths, failing to test ‘deep’ into a DUT. As a result, they are *ineffective* in their ability to exercise the functionality of the DUT.

This follows onto the third limitation of fuzzers: their ability to generate syntactically valid inputs. Protocols exist in both software and hardware that attempt to ensure the validity of an input, thus a fuzzed input must conform with these protocols before the test case is passed to the subject. This presents a problem for fuzzers which can alter the properties of test cases. Particularity when mutations alter the regions of test cases on which validation is performed.

Limitation	Challenge
Mutation Selection	Where to mutate in a test case?
Locating Bugs in a DUT	Given a ‘where’ to mutate, what mutation should be applied?
Syntactic Validity	How to effectively cover the DUT to find bugs?
	How to ensure that test cases pass input validation?

Table 2.1: The limitations associated with traditional fuzzing techniques.

2.4.1 Deep Learning Approach in Fuzzing

We have thus far discussed the way in which fuzzing is conducted, the key research directions in fuzzing, and the key limitations of existing works. We will go on to consider how deep learning approaches, and RL approaches in particular, have been applied to different areas of this problem. We present in Table 2.2 an overview of the different deep and RL based approaches to fuzzing. A variety of different deep learning approaches have been developed to improve the performance of fuzzing [186, 300].

Work from Joffe [135], proposed the use of a DNN to predict the likelihood of a crash occurring from execution traces. The output of this unsupervised method is then used in seed selection as a fitness function. This prioritises the seeds with ‘suspicious’ executions, using an autoencoder to smooth the input traces to a point in an N dimensional space. A similar approach was taken in Neufuzz by Wang *et al.* [303], using an RNN for predicting the likelihood inputs triggering vulnerabilities. This was then used to prioritise seed inputs of the PTFuzzer [317].

Learn&Fuzz [94] was arguably one of the first approaches to use deep learning to *generate* fuzzing test cases. Using an RNN to learn a probability distribution of the next character in a sequence, Learn&Fuzz learns to generate PDF object test sequences. Mutation of the test case occurs randomly during the generation from the RNN, resulting in a different character being selected at random. The model was able to outperform a random baseline in terms of code coverage. A later work DEEPFUZZ [165], used a similar mutation strategy to Learn&Fuzz: having an RNN learn to generate test cases at the character level and then randomly mutating these characters. However, DEEPFUZZ generates valid C programs, and was evaluated on the GCC C compiler, identifying eight new vulnerabilities in the compiler.

Montage [152], is another RNN approach for generation, but targeted for JavaScript. The proposed approach took existing JavaScript code, representing it as an Abstract Syntax Tree (AST). These ASTs were then turned into ‘fragments’, defined as unit subtrees formed of a parent and single child node. By representing fragments as a single sequence this can be used to train an LSTM to predict the next fragment in the sequence. The trained LSTM is then used to generate subtrees that are placed at random locations in the test case. The complete sequence is then turned back into an AST and tested on the target JavaScript Engine, in the case of Montage the authors exclusively target ChakraCore.

Sablotny *et al.* [245] similarly applied LSTM and GRU models to generate HTML code to fuzz HTML compilers. Testing on the FireFox browser, the authors showed that the GRU model was able to generate more syntactically valid source code than its LSTM counterpart. They were also able to show that the trained models achieved approximately 30% more coverage than existing approaches.

The use of large scale transformer architectures, or Large Language Models (LLMs) is explored by Meng *et al.* [185] in CHATAFL. They use a pretrained LLM to guide protocol fuzzing by constructing grammars for messages in the protocol, mutating messages, or predicting next messages. Experimenting on real-world protocols the authors showed efficacy in code coverage and find novel vulnerabilities.

An alternative architecture was developed by Li *et al.* [161], using a Wasserstein Generative Adversarial Network (WGAN) for the fuzzing of Industrial Control Systems (ICS). The WGAN is applied to learn a function that captures distribution of real-world ‘data frames’ or network data packets. The WGAN can then generate data frames with same, realistic, characteristics. In this way the model is able to provide protocol fuzzing for multiple ICS protocols; thus this approach provides diverse data generation producing a greater code coverage - outperforming a Generative Adversarial Network (GAN) used for the same purpose.

	Seed Selection	Generative	Mutational	Deep Learning	Reinforcement Learning	Architecture
Joffe [135]	✓			✓		DNN
NeuFuzz [303]	✓			✓		LSTM
Learn&Fuzz [94]		✓		✓		LSTM
DEEPFUZZ [165]		✓		✓		LSTM
Sablotny <i>et al.</i> [245]		✓		✓		LSTM, GRU
Li <i>et al.</i> [161]		✓		✓		WGAN
Montage [152]		✓		✓		LSTM
Meng <i>et al.</i> [185]		✓	✓	✓		LLM
EcoFuzz [314]	✓				✓	Adversarial Multi-Armed Bandit
Wang <i>et al.</i> [298]	✓				✓	Multi-Armed Bandit
SYZVEGAS [294]	✓				✓	Multi-Armed Bandit
Luo <i>et al.</i> [172]	✓				✓	Multi-Armed Bandit
Fang and Yan <i>et al.</i> [75]		✓			✓	Tabular Q-learning
Spieker <i>et al.</i> [266]	✓			✓	✓	Deep RL
ALPHAPROG [159]		✓		✓	✓	DQN,LSTM
FUZZBOOST [160]		✓		✓	✓	DQN,LSTM
Böttinger <i>et al.</i> [36]			✓	✓	✓	DQN
Kuznetsov <i>et al.</i> [150]			✓	✓	✓	DQN
Zhang <i>et al.</i> [319]			✓	✓	✓	Deep RL
Pham <i>et al.</i> [227]			✓	✓	✓	DQN
Sablotny <i>et al.</i> [246]			✓	✓	✓	DQN

Table 2.2: Fuzzing using learning based approaches.

2.4.2 Reinforcement Learning Approach in Fuzzing

In the previous Section we saw how deep learning techniques can be leveraged to fuzz devices. Many of these works are trained and then used statically during the testing of a DUT. While a variety of different architectures can find vulnerabilities in this way, their problem formulations often result in a lack of feedback from the performance of test cases during testing. Thus, here we will explore works that leverage both RL and Deep RL to improve fuzzing performance.

The first formal work using RL in fuzzing was presented by Böttinger *et al.* [36]. In this inaugural work the authors formalised *mutation* based fuzzing as an RL problem. The authors have a Deep RL agent observe substrings of a test case (viewed as a series of bytes) and select one of 6 low level mutation types (*e.g.* random bit flips). The reward is then modelled using coverage and test case execution time. To validate the approach a DQN model was used, which outperformed a random mutation baseline. However, the authors leave much up to future research, presenting only ‘preliminary’ empirical evidence. One such piece of future work directly following on from this is presented by Kuznetsov *et al.* [150], who increase the action space of the RL agent to be 45 different mutations. Another work that follows a similar

mutation based approach is from Sablotny *et al.* [246]. However, the authors target HTML compilers using a Double DQN. In this setting the agent was able to insert HTML tags into the test cases as they were generated.

Zhang *et al.* [319] study several different Deep RL algorithms ability select from 9 low level mutations on binary based test cases. Zhang *et al.* show that Deep RL methods consistently outperform a random baseline when mutating test cases to fuzz the LAVA-M dataset [64]. Pham *et al.* [227] develop CTFuzz, a DQN based model similar to that developed by Zhang *et al.*. CTFuzz is shown to outperform random selection, AFL, and work from Zhang *et al.* [319] in short time-frame experiment.

ALPHAPROG [159] applies a DQN with LSTM core to generate test cases in a toy compiler for the Brainfuck language. An interesting insight from this paper is the study of 4 different reward functions to help guide the agent, using: coverage, cyclomatic complexity, syntactic validity, and run time. The same group published a concurrent work FUZZBOOST [160], which applies a similar architecture and approach to fuzz the C compiler GCC.

Non-Deep RL architectures have also been used in fuzzing. One of the first works to do this was done by Fang and Yan [75]. Targeting the 4G/LTE protocols on Android devices using a tabular Q-learning agent. The agent generates unique perturbation sequences, causing four vulnerabilities including: location leakage and loss of network connectivity.

Multi-Armed Bandits have had particular success for prioritising seeds in mutational based fuzzers, with several different approaches being developed. The EcoFuzz [314] model uses a MAB for an adaptive probability based search strategy which can be applied on top of AFL, creating an ‘energy saving’ grey-box fuzzer. The ‘game’ for the MAB is to maximise the path coverage using a finite number of trials, which in this case are seeds. However, this game presents a fundamental challenge: the number of possible actions (each seed in the queue) increases as successful test cases are added to the corpus. As fuzzing continues the number of actions increases, while the number of possible paths decreases as does the associated reward. EcoFuzz overcomes these challenges using a variant on the adversarial multi-armed bandit and using separate exploration and exploitation steps. EcoFuzz covers 214% of the paths of AFL,

while reducing the test case generation by 32%.

Wang *et al.* [298] also applied a MAB to the problem of seed selection. However, in this work the problem is modelled as a hierarchical seed scheduler. By using multi-level coverage of seeds (function coverage, edge coverage, and hamming distance) the reward function of the agent can be tailored to increase fuzzing performance to find 20% more bugs compared to AFL. A similar and contemporary work using bandits for seed selections is present in SYZVEGAS [294]. This work focused on improving scheduling of seeds using a custom reward for fuzzing the Linux Kernel. Luo *et al.* [172] also present an alternative MAB approach that requires no hyperparameters to train and is computationally less expensive than other MABs used for seed scheduling.

Deep RL was first used to prioritise seeds in mutational based fuzzers by Spieker *et al.* [266] as RETECS. Using a custom Deep RL approach similar to DQN, RETECS selects and prioritises seeds, using a reward formed on their duration, last execution, and failure history. This approach learnt to prioritise test cases that caused errors when testing Continuous Integration.

Many of the challenges we have discussed here concerning both fuzzing, and using learning based approaches, will reoccur throughout this Thesis. Furthermore, we will develop RL based testing approaches that target specific areas for fuzzing, including: generation in Chapters 3 and 4, mutation in Chapter 5, and explore where to perform mutations in Chapter 6.

2.5 Web Application Security Testing

The internet has revolutionised how life is conducted in the modern era, with 67% of the global population having access to the internet in 2023, up from 51% in 2019 [131, 132]. However, this gives rise to a large security problem: the internet and the applications that run on it were not originally designed to be secure. As we have seen, injection attacks are a prevalent issue in web application security with 2 of the top three most common CVEs being injection attacks [188]. These attacks often arise from the improper sanitation of user input to a trusted web page. Consequently, malicious content is allowed to run alongside trusted content, which

could be used to steal data or commit unauthorised actions. This class of vulnerabilities come in many different flavours, depending on what they are trying to exploit. We briefly cover the most prevalent injection attacks that plague web applications [104, 130, 214], accounting for two thirds of the works included in Table 2.3. Where applicable, they will be expanded in Chapters 3 and 4.

- **Cross-site scripting (XSS).** Using a combination of HTML and JavaScript an XSS payload aims to cause unintended functionality on a web page. These payloads can cause drive-by-downloads, authentication theft, and even remote code execution [130, 214].
- **SQL injection (SQLi).** Aiming to cause a modification or extraction of data from an SQL database, SQLi payloads are injected to inputs in web applications which are included in queries to the database [318].
- **Command injection (CMDi).** Exploiting where web application inputs are interfaced with the system shell, malicious users inject payloads to run commands on the shell [270].

There are many different approaches to detecting and preventing injection attacks. A common defence is to sanitise or filter user inputted values to remove the potentially malicious behaviours. Argyros *et al.* [14] studied sanitisers in a black-box fashion, focusing on **regex** and string sanitisers. They were able to reverse engineer sanitiser functions, going on to correctly determine 90% of filters, and finding SQLi vulnerabilities in 4 common filter libraries.

Around the same time, Mohammadi *et al.* [192, 193] extract encoding functions used by web applications to detect XSS vulnerabilities. The authors describe XSS payloads as having three core parts: ‘pre-escape’, ‘payload’, and ‘post-escape’. These are modelled as a finite state machine and allow the unit test to generate a payload to test each sanitisation filter in an automated unit test. When comparing against an existing scanner, the OWASP Zed Attack Proxy (ZAP) [210], they find that ZAP looks only at string matches in the page leading to false positives. The authors then state a legitimate XSS vulnerability ‘must either call a JavaScript function (e.g. an API), or make an assignment (e.g. change the DOM)’; opting for the latter the XSS is successful if it changes the title of the webpage [230].

	Application	Deep Learning	Reinforcement Learning	Testing Model	Technique
Frempong <i>et al.</i> [87]	XSS payload generation	✓		grey-box	Transformer
Song <i>et al.</i> [265]	XSS payload mutation	✓		grey-box	Double DQN
Suggester [39]	XSS payload generation		✓	black-box	Tabular Q-learning
RLXSS [76]	XSS payload mutation	✓	✓	black-box	Duelling DQN
Link [153]	XSS payload generation	✓	✓	black-box	A2C
Hasegawa <i>et al.</i> [112]	XSS payload generation	✓	✓	black-box	DQN
Yao <i>et al.</i> [311]	XSS payload mutation	✓	✓	black-box	DQN+PER
DeepSQLi [164]	SQLi payload generation	✓		black-box	Transformers
DANuoYI [157]	SQLi payload generation	✓		black-box	Evolutionary algorithms and encoder-decoder models
Gui <i>et al.</i> [98]	SQLi payload generation	✓		black-box	LLM
Erdódi <i>et al.</i> [71]	SQLi payload generation		✓	black-box	Tabular Q-learning
Sommervoll <i>et al.</i> [262]	SQLi payload generation		✓	black-box	Tabular Q-learning
Liu <i>et al.</i> [163]	Crawling		✓	black-box	A2C
WebExplor [323]	Crawling		✓	black-box	Tabular Q-learning
Kim <i>et al.</i> [141]	Bugs in REST APIs		✓	black-box	PPO
Corradini <i>et al.</i> [51]	Bugs in REST APIs	✓	✓	black-box	PPO
WENDIGO [184]	Denial-Of-Service in GraphQL APIs	✓	✓	black-box	PPO
Tsingenopoulos <i>et al.</i> [280]	Bypassing reCAPTCHA	✓	✓	black-box	PPO

Table 2.3: Web application security testing approaches using deep learning or RL methods.

A concurrent work on comparing vulnerability scanners on XSS filters is XSS PEEKER [26]. The authors introduce a benchmark for XSS vulnerability detection, Firing Range. Six state-of-the-art tools are then evaluated on the benchmark, leading to the discovery that approaches are unable to find many vulnerabilities, and cannot optimise for minimal requests.

However, filters and sanitisers may not always be implemented correctly, giving rise to automated testing strategies to try and find bugs and vulnerabilities in web applications. These methods take an alternative approach by viewing the web application as a grey- or black-box. This mimics the behaviour of a malicious actor who may be trying to find weaknesses to exploit. Many different approaches have been taken using automated heuristic, or manually defined techniques [21, 53, 56, 210, 231, 233, 234, 273]. We will focus with particular emphasis on the methods that use deep learning and RL in their testing approaches, in order to determine the potential for research in this domain. An overview of different these approaches can be found in Table 2.3.

2.5.1 Crawling Web Applications

To test web applications in an automated fashion, we first need to be able to find *where* it is possible to test. This task is often achieved by *crawling*: interacting with a web application

to find new web pages, and possible input fields. However, this is a non-trivial task as modern web applications have a complex and dynamic structure, which often require multiple steps to reach ‘deeper’ into a web application.

Doupe *et al.* [66] argue that black-box vulnerability scanners need to access and interact with the more complex elements, *e.g.* Javascript. By achieving such a level of interaction, crawlers are able to test deeper into a web application, otherwise only finding the surface level of the webapps. The authors then present a state aware scanner: Enemy-of-the-State [65]. By defining a state change as an identical request that results in a different response, the model is able to crawl web applications with user-like behaviour, detect a change in a response, and then determine which request changed the state *e.g.* logging a user in. This increases the number of webpages the scanner is able to find, as shown by the increased code coverage in comparison to all other crawlers.

Despite advances in crawling techniques, it is still an ongoing research area as web applications become increasing more complex. Black Widow [74], implemented a ‘data-driven’ approach, building on three fundamentals: navigation modelling, traversing, and tracking inter-state dependencies. Using taint tracking techniques, the Black Widow crawler was able to dynamically verify vulnerabilities to prevent false positives, improve web page coverage, and find additional vulnerabilities.

The vast state-space of potential web applications has even motivated the use of RL approaches. The first of such works was presented by Liu *et al.* [163]. Modelling the selection of crawling workflows as a Partially Observable MDP (POMDP), an A2C agent is trained via expert demonstration selecting from a constrained set of workflows. The authors argue that this reduces the possibility of poor exploration, wasted time, and reduces challenges from a large state-action space. Using a sparse reward that provides positive reward of +1 for successful crawling to new states, the agent learns from only positive samples which are stored in an experience buffer. The authors argue that using only positive samples provides a more ‘powerful’ and expressive replay buffer.

A curiosity-driven model was later developed as WebExplor [323] using tabular Q-learning. The

authors define the state as a combination of HTML and URL, having the RL agent conduct user actions on a web page (*e.g.* clicking buttons, inputting text into elements). The ‘curiosity-driven’ element arises in the reward function, where the agent is rewarded only for finding new states, hence incentivises curious exploration. WebExplor also uses a Deterministic Finite Automata (DFA), that records states and associated visit frequencies. The DFA can then be used when the agent gets ‘stuck’: the DFA moves the agent to a different (low frequency) state in an attempt to bootstrap exploration.

2.5.2 Grey-Box Testing

As we have seen in Section 2.4, grey-box approaches leverage an intermediate level of information to guide testing approaches. Commonly in the case of fuzzing this is through coverage of source-code. However, the structure and diversity of web application environments mean that feedback could be in many different forms, *e.g.* coverage, Hyper Text Transfer Protocol (HTTP) responses, HTTP response codes. Yet, there are few works that use such a level of feedback.

webFuzz [286] applies instrumentation to web applications to create a feedback loop based on code coverage. Using this familiar grey-box testing approach, webFuzz targets XSS vulnerabilities. The testing approach mutates HTTP requests to insert XSS payloads, random strings, web tokens, alter parameter values, or mix HTTP methods (*e.g.* GET and POST). Using a scoring function that prioritises coverage, new mutations, and successful requests webFuzz increases coverage, and finds a novel XSS vulnerability.

Lekies *et al.* [155] use a grey-box testing approach using a custom browser that alters the JavaScript engine, allowing byte-level taint tracking of an input. This allows them to attempt the bypassing of web application sanitisation in an attempt to find XSS vulnerabilities in an automated fashion. In doing so, they can detect the payload context in the webpage to determine the correct pre- and post-escape strings. The model was then tested on the Alexa top 500 web applications, finding that 9.6% of these have at least one XSS vulnerability that can be found in this way. A similar approach is used by Bensalim *et al.* [27], using dynamic data-flow analysis to track potentially vulnerable input elements in web pages.

Human-to-machine translation for developing XSS payloads has been developed by Frempong *et al.* [87]. Using a variety of code snippets to create a dataset, this is then used to train a transformer model to understand the link between source code and XSS payloads. Given a code snippet and text instructions the model then attempts to generate a malicious payload.

Work from Song *et al.* [265] has investigated how RL can be used in grey-box detection of XSS vulnerabilities. Using source code analysis to find potential input points, a Double DQN mutates existing payloads, receiving a sparse reward for finding a vulnerability. The authors show that the Double DQN outperforms policy gradient methods to find all vulnerabilities in two simple and deliberately vulnerable web applications.

2.5.3 Black-Box Testing

Web security testing tasks are generally done in a black-box manner, which can be harder than white-box and grey-box settings due to the lack of knowledge of how the web application works internally. Yet, black-box testing provides a more realistic model of an attacker, so has been standard practice for many years [65, 226]. Among other tasks, automated black-box testing approaches have resulted in a broad and substantial body of work [6, 56, 273, 318, 320].

Finding injection vulnerabilities has been automated in a variety of ways [6, 56, 147]. Jan *et al.* [133] explore the use of search based algorithms when trying to detect XML injection. The same group expand on this later to include learning based approaches in the form of evolutionary algorithms [134]. Commix [270] was developed as an alternative approach for finding CMDi vulnerabilities. Making use of side-channel information including timing and file access techniques to discover CMDi. This innovative approach led to its successful inclusion in the popular security Linux distribution, Kali. These approaches highlight some of the different and novel ideas for detecting vulnerabilities in web applications.

Testing for XSS. Compared with the limited grey-box testing approaches in Section 2.5.2 there have been a number of works to harness both deep learning and RL. An early example using Deep RL for such a task is *RLXSS* [76]. In this model a duelling DQN is trained to

mutate existing XSS payloads¹ by using encodings, substitutions, and adding special characters. These payloads are then tested against existing XSS filters SafeDog [247], XSSChop [40], and Deepxss [77]. RLXSS uses a simple sparse reward function in training to bypass XSS filters. The core contribution is the use of RLXSS to find payloads that bypass filters, which are then used as training data for Deepxss. This data is used to improve the effectiveness of the filter.

The first *scanner* for XSS using RL was developed in the Suggester model [39]. Using multiple tabular Q-learning models for each different payload context, Suggester *suggests* payloads to a user. Payload generation is formulated by the environment ‘moving’ through the different parts of the payload (*e.g.* pre-escape, payload body, post-escape), having the agent generate the different elements. The user then inputs payloads into a web page, providing feedback on payload performance. This informs the reward, which penalises for redundant actions, and rewards for completing the payload. Suggester itself lacks a significant portion of automation: submitting payloads, detection of XSS on a web page, or receiving any form of feedback. The approach had comparable performance with some scanners in terms of the number of vulnerabilities found.

The first Deep RL approach to XSS detection seen in the literature is Link [153], a contemporary effort developed at the same time as our work in Chapter 3. Link uses an off-the-shelf A2C model architecture to generate payloads from a heavily structured action space, and a manually curated state-space. Link can generate payloads, and detect vulnerabilities automatically in web applications. While outperforming state-of-the-art approaches in benchmark applications, Link only finds a comparable number of vulnerabilities in real-world applications.

Hasegawa *et al.* [112] also develop a Deep RL approach using a DQN and a curiosity based reward function. Using a problem formulation similar to Suggester, the agent mutates different parts of the payload sequentially, receiving a similar reward in response. The authors evaluate the model on a single deliberately vulnerable web application, showing that they can find vulnerabilities in fewer requests than open-source tools. However, the evaluation does not compare with alternative RL approaches, or show how many vulnerabilities approaches can

¹XSSed database <http://www.xssed.com/>

find, comparing only in terms of the number of requests used.

Yao *et al.* [311] later develop a Deep RL model using Duelling Double DQN, and PER. This work has a number of similarities to RLXSS: mutating existing payloads in a similar way and being used to bypass filters in Web Application Firewalls (WAFs). The authors use a reward function based on the edit distance algorithm, in an attempt to prevent the agent from getting stuck in local optima by trying to maximise difference between payloads. They show the implemented prototype can outperform other methods, in a simple ablation study.

Testing for SQL injection. SQLi is a prevalent vulnerability, which has led to several different detection approaches. For instance, DeepSQLi [164] uses a transformer architecture as an encoder-decoder model to iteratively generate SQLi payloads. The transformer is pretrained using a diverse dataset of mutated SQLi payloads to learn the semantics of the payloads. DeepSQLi was able to outperform state-of-the-art tooling in terms of the number of vulnerabilities found, and testing resources (number of attempts and CPU consumption). This work led to DANUOYI [157], which uses multiple encoder-decoder models to generate different injection payloads. Using a translation approach viewing different payloads as semantically similar, the model uses a multi-task evolutionary algorithm to evolve payloads for different injection attacks. Testing on three real-world WAFs, DANUOYI generated more payloads that bypass firewall rules compared to its individual components: the combined model was better than the sum of its parts.

Gui *et al.* [98] develop a SQLi payload generation approach using LLMs. Using three state-of-the-art LLMs, they evaluate their potential to generate SQLi Payloads. In particular they use several agentic approach in an exploratory study. Reappropriating the benchmark introduced in Chapter 4, they show that LLMs are able to successfully generate SQLi payloads, but can suffer from instability, hallucination, and test inefficiency.

Work from Erdödi *et al.* [71] develops a Deep RL approach to generate SQLi payloads, modelling the problem as a Capture-The-Flag (CTF). The web server is treated as stateless, while agent states are the history of previous actions. The agent attempts to access a ‘flag’ in the SQL database. Using tabular Q-learning and a DQN, each agent outperforms random action taking,

within the rules defined in the game. This work provides the proof of concept for RL in the context of payload generation for SQL injection. The same group later simulated the ability of RL agents to find different classes of SQLi, finding that RL agents can accurately find such vulnerabilities [262].

We have delineated many of the foundational and state-of-the-art approaches that test web applications, with particular focus on injection attacks. In Chapter 3 we will revisit the generation of XSS payloads in black-box settings using RL agents in separate constrained games. While Chapter 4 will approach grey-box generation of SQLi payloads using a single RL agent.

Web API Testing. Besides injection attacks RL has also been used for testing web APIs. McFadden *et al.* [184] use a PPO based agent to find denial-of-service vulnerabilities in GraphQL APIs. The approach, WENDIGO, uses a large state-action space to manipulate queries, making them increasingly complex. It was seen to outperform both random testing and a state-of-the-art tool, finding vulnerabilities in fewer queries.

The majority of testing approaches applied to web APIs focus on finding bugs. Due to the predominantly stateless implementation of web APIs they can be a challenging target in which to find vulnerabilities [59]. Finding bugs in APIs provides a simple oracle to guide the testing process [13]. As a result there have been many different ways to find bugs [20, 21, 52, 140]. However, these automated testing approaches are similarly *inefficient*. Making use of manual heuristics with limited feedback they often require many requests to find bugs [21, 52, 167].

Kim *et al.* [141] were the first to use RL for this task, developing their approach concurrently with that presented in Chapter 5. Their approach uses multiple tabular Q-learning models to prioritise different parameters and simple feedback in the form of HTTP response codes. The authors were able to show their approach outperformed a number of other state-of-the-art automated testers in a benchmark of simple applications. Another work developed by Corradini *et al.* [51] also used RL. They adopt a complex *multi-learning* model that includes: multiple Multi-Armed Bandits, a Deep RL PPO agent, and an LLM for generating test parameters. This approach was seen to outperform that from Kim *et al.* in the same experimental setup.

Tsingenopoulos *et al.* [280] demonstrate the ability of RL to perform evasive testing in web security, using a deep learning PPO agent to bypass reCAPTCHA. The agent is able to exercise three levels of evasion tactics based on mouse movements and keyboard values. This forms states as the web page viewed from a browser, which can be modified by actions from mouse movements. The authors also develop a staggered reward depending on the level of evasion tactic used, to guide the agent throughout the episode. This approach was able to bypass real world web application reCAPTCHAs. The influential features for the learnt model showed that it picked up on subtleties related to the security of a web page, *e.g.* how cookies were used.

The testing of APIs, in particular REST APIs, will be a focus of Chapter 5. Where we will develop an RL agent with an information rich latent representation to efficiently mutate HTTP test cases.

2.6 Takeaways

In this Chapter we have discussed the details of RL, deep learning, and Deep RL seeing different approaches and techniques to learn. We have also discussed different ways in which testing can be conducted, covering fuzzing, grey- and black-box web security testing.

With regards to fuzzing, many different non-learning approaches have been proposed to improve different aspects of testing. Yet, several common limitations still exist in approaches ranging from 1) the mutation selection processes, 2) how to effectively find bugs in tested systems, and 3) ensuring that test cases are syntactically valid [146, 156, 250, 302].

In an attempt to mitigate some of these limitations several deep learning approaches have been developed. However, deep learning models used in this domain require training datasets, and once trained are used with fixed network weights. For this reason, they are limited in what they can be used for as they cannot learn during the testing process. As a result, deep learning models trained this way are rarely used for the mutation of test cases, and are used for seed selection and generation instead.

Yet, the testing paradigm is highly applicable to RL, sharing many of the same concepts. Therefore, we believe that this motivates RL as an exciting area to improve the efficiency of testing. It has shown to be more efficient for seed selection using MAB approaches and for test case generation and mutation using Deep RL approaches. However, RL approaches for performing mutation or generation often use off-the-shelf models, with relatively simple problem formulations, states, and actions; limiting the problems on which they can be applied.

When it comes to web application security testing, we have delved into the two applicable types of testing: grey- and black-box. Grey-box settings are sparsely used in web security testing compared to their favoured use in fuzzing. Instead, black-box settings that closer mimic behaviours of malicious actors are often preferred. Here we have seen many works making use of deep learning and RL. Yet again the same pattern occurs: deep learning only systems are unable to learn during testing on the DUT. Similarly, the RL models use simple, often manually defined, feedback in order to guide their learning. This pattern has been effective in the settings considered in their respective fields. However, such manually defined feedback can introduce bias in the learnt policy, leading to unexpected behaviours and suboptimal performance [115, 201, 316].

Despite all the research that has been done in web application security testing, there remain several key areas that are either unexplored or not solved:

- **Where to fuzz?** When it comes to mutation based testing, much attention has been placed on developing different mutations, or selecting test cases to mutate. However, there has not been any work on where to perform these mutations.
- **Harnessing feedback.** A common approach to guide web application security testing is to use simple manually defined heuristics. For example, coverage, HTTP status codes, or static inference. While these can be enough to find bugs and vulnerabilities, they don't do so *efficiently*.
- **RL models capable for the task.** Many of the RL models used for web security testing are off-the-shelf methods. Which may not always be the most appropriate choices

in complex environments.

In Chapters 3, 4, 5, and 6 we will develop solutions to these problem areas. We will aim to showcase the ability of RL approaches, tailored to their respective tasks, to perform web application security testing. We demonstrate the real-world impact of these approaches by shifting away from lab-based benchmarks and evaluations which are commonly used in evaluating RL tasks in this domain.

Chapter 3

Constrained Games for Cross Site Scripting Payload Generation

We have discussed some of the problems in web security relevant to this thesis in Chapter 2 and seen how RL has been able to solve myriad problems that are highly complex. Specifically RL has been shown time and again to search over large state-action spaces to find a solution to a given task [191, 207, 246, 251]. The aim of this Thesis is to determine how effective RL is for testing in the domain of web application security.

In this Chapter we begin with a constrained problem to determine if RL is able to complete a relatively simpler task. Yet, such a task should still be challenging for the RL approach so that it can have real-world impact. We do not want to create something that only works in the lab environment.

We begin by applying RL to a single vulnerability: Cross Site Scripting (XSS). XSS has a long standing body of work [27, 147, 155], and yet still presents legitimate threats of data theft and remote code execution [279]. Present in the OWASP Top 10 in 2013 [211], 2017 [212], and 2021 [214], it is also the most rewarded bug bounty by HackerOne¹ from 2019 to 2024 - increasing the average payout to \$577 USD per XSS [101, 102, 103, 104, 239]. Furthermore,

¹HackerOne is a popular bug bounty program that has resulted in over \$300 million USD in bounties as of October 2023 [103].

XSS has been the highest web attack recorded in both 2022 and 2023 [130], leading to over 5000 unique vulnerabilities. Such statistics show that despite the existing work to find and mitigate XSS vulnerabilities, it is still a prevalent security problem.

We argue the reason for XSS remaining a problem, is for two reasons in existing vulnerability scanners. First, we argue that typically scanners use payloads that lack sufficient diversity. This is confirmed in recent work by Buyukkayhan *et al.* [33], who show that XSS payloads require increasing complexity to demonstrate vulnerabilities. Second, is that scanners often suffer from *false positives*, a limitation that means XSS was not effectively triggered, but was erroneously reported by the scanner. We seek to address these limitations using a novel RL approach tailoring payloads to specific sanitisations, finding common and uncommon vulnerabilities, while avoiding false positives.

In this Chapter we gamify black-box web application (webapp for short) testing by implementing an RL environment to handle low level details of web vulnerabilities (e.g. finding sources and sinks, submitting requests, parsing HTML) so RL agents focus on higher level tasks. We formulate XSS payload generation as a hierarchical RL problem, so that agents can learn to fuzz new source-sink combinations, generating diverse payloads and bypassing the associated sanitisation. We implement our approach as the HAXSS scanner. We evaluate HAXSS to show it finds 131 vulnerabilities across our novel micro-benchmark and 5 reference vulnerable webapps, improving on state-of-the-art black-box scanners. HAXSS also reports 0 false positives. Finally, we show that HAXSS is practically useful, finding 4 existing and 5 novel CVEs in production-grade webapps.

Chapter Contributions. This work has been published at the 2022 IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom). The paper was awarded the ‘*IEEE Best Student Paper Award*’. I developed and implemented the HAXSS framework, and designed, performed, and created evaluation metrics for all experiments.

3.1 Overview

Manual code review and static analysis tools help to detect vulnerabilities in source code, when available, at the cost of a significant investment of expertise and time. An alternative approach is to use black-box scanners that interact with webapps without user interaction, attempt to find input sources, trigger vulnerabilities, and report the successful source and payload when they are done. Yet, hard-coded payloads may miss many edge cases, failing to trigger existing but uncommon vulnerabilities, thus leading to *false negatives*.

Our novel idea is to formulate the generation of XSS payloads into two separate tasks or *games* to be solved by different RL agents. The first game will generate payloads that alter the structure of a web page around the user output in order to execute malicious code by *escaping the context*. The second game is triggered when the web application attempts to sanitise a payload that is generated in the first game. Thus the second game generates payloads that are obfuscated to *bypass sanitisation*. Successfully obfuscated payloads are passed back to the first game, which will continue hunting for a vulnerability or reach a failure criterion. Constructing the two games in this way allows a single generated payload to meet several desirable criteria: syntactic validity and context escape in addition to sanitisation bypass. We call our approach *HAXSS: Hierarchical Agents for XSS*.

3.2 Challenges of XSS Detection

XSS is an injection-based vulnerability where user input is accepted in a webapp in one location, the *source*, and is returned to the user as part of the output, in a *sink*. These vulnerabilities are common due to flaws in the way user input is handled. XSS enables adversaries to run code on behalf of their victims, gaining control over a webapp and the associated data [130, 214]. The most common way to prevent XSS is by sanitising user input to render it benign, though incorrect sanitisation is frequent and regularly leads to new vulnerabilities.

We focus our attention on the three main cases of XSS: reflected, stored, and DOM-based.

A Reflected XSS occurs when a malicious payload is provided as part of a web request, and it manifests itself in the returned response (typically an HTML page). A stored XSS occurs when the malicious payload becomes part of the server-side data that is used to generate the HTML of an arbitrary HTML page for the same web app, for instance in a comment section. A DOM-based XSS is only present inside the browser executing the HTML and JavaScript code of the webapp, as the payload directly affects the DOM without being sent to the server.

There are a variety of contexts where submitted payloads can then be found as part of a resulting web page (the sink), and can be used to infer the sanitisation required. We will consider a number of different contexts in which XSS can occur, while not an exhaustive list we cover the majority of sink locations where XSS can be found [39, 74, 155]. Specifically, we consider the following contexts:

1. Inside a HTML tag body: `<div>INPUT</div>`
2. HTML tag attribute value: `<div onLoad=INPUT ></div>`
3. HTML tag: This includes any other location a user input can be seen in a HTML tag such as the tag name (`<INPUT>`) or tag attribute name (`<div INPUT=function()>`)
4. HTML comment: `<!--INPUT -->`
5. JavaScript: This can include any location within a JavaScript program this could include a variable assignment (`x=INPUT`), function declaration (`INPUT()`), and comments (`\*INPUT*\`).

We have briefly recapped the types of XSS that are commonly found, and the contexts in which they occur. However, to generate such payloads in such contexts requires overcoming several non-trivial challenges. These challenges are caused by several factors, including the vulnerability itself, and web applications. We here delineate these challenges in detail.

Listing 3.1: Example HTML and potential user inputs

```
1 <html>
2   <body>
3     <!--USER_INPUT_1-->
4     <style id="USER_INPUT_2"></style>
5   </body>
6 <\html>
```

Finding Sources and Sinks

Before we can inject a payload, and subsequently determine if a webapp contains a vulnerability, we must identify the source and sink locations. Finding source-sink combinations in an automated way often requires the tracking of a unique payload or token. While this is conceptually a simple task, in practice it has many challenges when trying to find a source or sink, this includes: logging into account(s), traversing a specific sequence of pages, interacting with the webapp to display the payload/token. Such challenges describe the basis of crawling, an area of web application testing with its own body of research attempting to improve the coverage of found webpages by traversing ‘deep’ into a webapp [65, 66, 74, 163, 224, 323].

As a result, it is a challenge to cover and find source-sink combinations. We focus on how RL can be used to perform web application security testing and while crawling contributes to this goal, it does so in an auxiliary fashion. Thus we deem developing novel methodologies for crawling beyond the scope of this Chapter, and indeed this Thesis.

Verifying Vulnerabilities

Correctly determining when an XSS vulnerability has been triggered is key to determining the performance of a scanner, and to improving the security of web applications. There are a number of different ways in which one could verify a successful payload. For instance some scanners look for the appearance of successful payloads in the source code of the returned page, such as the OWASP Zed Attack Proxy (ZAP) [215] or XSSer [234]. However, this may report a safe page as vulnerable; for instance `USER_INPUT_1` in Listing 3.1 a payload (`<script>alert()</script>`) reflected on the page inside the HTML comment may be present but not cause a vulnerability, resulting in Line 3 becoming `<!--<script>alert()</script>-->`. In such instances, tools may

Table 3.1: **A:** CVE-2021-22889 that uses a built in PHP function incorrectly that can result in XSS. **B** CVE-2019-10062 that uses a custom regex that can still result in XSS.

```

A: $this->aPageParams['statsBreakdown'] = htmlspecialchars(
    MAX_getStoredValue('statsBreakdown', 'day'));
B: input.replace(/<script\b[^\>]*(?:(!<\script><[^\>]*)*<\script>/gi;,
    '');

```

report a high number of false positives, which can increase the human effort to determine the legitimate vulnerabilities from the illegitimate. In doing so the usability of the tool is reduced, as is its efficiency.

Escaping Contexts

As we have seen when discussing verifying vulnerabilities the *context* in which a payload appears can cause a payload to be rendered benign. Thus, escaping the context can be a crucial step to triggering a vulnerability. Consider `USER_INPUT_1` in Listing 3.1, if we now alter the payload (`<script>alert()</script>`), by inserting a closing HTML comment we can cause the vulnerability to be triggered resulting in Line 3 becoming `<!-- --><script>alert()</script> -->`. Equally, in Line 4 a payload could escape the context by terminating the attribute value, and providing a new attribute leading to code execution, *e.g.* `" onload="alert()`. Failure to change context can result in false positives or false negatives. While changing context can move a legitimate payload to a context where it can be executed.

Escaping the context can be challenging as we must correctly identify the context in the sink, and then use the payload to malform the legitimate content in the web page. This malformation or *escape string* is a character sequence that moves the malicious part of the payload to become executable. In the examples given these could be as simple as a `"` or `-->`, however this may not always be the case. In some cases the sink may occur inside complex JavaScript, making it difficult to determine what a correct escape string may be, or the DOM may alter the structure of the page as a result of the payload. Furthermore, we must be able to detect when using such an escape string has lead to a change in the context.

Bypassing Sanitisation

Even once we have an escape string a payload still may not trigger a legitimate XSS vulnerability, as there may be sanitisation from the web application that can prevent the context from changing, or a payload from reaching the sink. Indeed, standard functions exist to sanitise different contexts *e.g.* `htmlspecialchars` and `htmlspecialchars` in PHP. Yet the use of these functions depends on security knowledge of the developer, the development practices, and other hard-to-control factors. When these functions are not used, or are used improperly, such as in the wrong context, they lead to vulnerabilities. A concrete example of this can be seen in CVE-2021-22889 (Example **A** in Table 3.1), where the lack of specifying the type of quotes required leads to a XSS in a tag attribute.

Furthermore, developers may also implement sanitisations of their own design. Such designs may be due to specific application requirements with dependencies on domain specific knowledge, as a result the number of potentially different sanitisation behaviours is unbounded. Among the vast number of potential sanitisations developed, some of this will inevitably be those that leave the webapp vulnerable. We can see an example of this in CVE-2019-10062 (Example **B** in Table 3.1), where a regex command allows a malicious user to bypass the sanitisation using a payload based on a different HTML tag than `script`. CVE-2019-10062 illustrates that there is also diversity in the sanitisations landscape, and thus the potential space for payloads that can find vulnerabilities is equally diverse and complex [33]. As a result our methods to find such vulnerabilities must be equally diverse.

3.3 Payload Generation as a Hierarchical RL game

In the previous section we have motivated some of the challenges of XSS detection, a common thread is the need for diversity in the payloads we are using. While payload lists cover the most common vulnerabilities, they miss the long tail of ‘edge cases’ that depend on the specifics of a webapp or its developers. While approaches to generate or mutate payloads often require a very large number of attempts in order to find new interesting payloads, as the search space

is extremely large. We postulate here that RL could be efficient in exploring this large state space, by learning a policy to create and tailor payloads. Thus improving the search strategy to find vulnerable edge cases.

In order to use RL for this task we will begin by defining the core challenges specific to this problem.

Agent interaction with webapps

Previously we discussed verifying vulnerabilities and escaping contexts in relation to identifying when XSS occurs. This kind of information is crucial feedback for an RL agent to learn from. Prior work from Caturano *et al.* [39] used a ‘human-in-the-loop’ to input payloads and then observe outcomes. While such an approach may work on small scale tests, it is unfeasible for large scale vulnerability scanning or pentesting. We want our RL model to interact directly with the webapp. Specifically this should include:

1. Finding source-sink locations.
2. Injecting payloads in the source.
3. Traversing to the sink.
4. Checking if the context has changed, or if a payload was successful in finding a vulnerability.

Defining the action space

In typical game playing settings such as the Game of Go, or Dota, actions are clear and defined as part of the game [207, 259]. However, in the case of payload generation, such actions are less clear. The agent must take actions to change its state in the environment, thus conforming to an MDP. The most general choice for this could be bit-flips, or similar low-level string editing operations. However, such an action space would be vast, resulting in highly sparse rewards

and difficulty in developing an effective policy to generate the payload of a single vulnerability, let alone multiple ones.

Adapting to different source-sink combinations

RL excels at finding an optimal policy for a task, guided by some reward function; in doing so it learns to complete a task. How effective the RL agent is at the training task is the usual way in which the agent is evaluated, *e.g.* an agent learning to race in a track in the game Gran Turismo [309]. Yet, for generating payloads, there is no ‘optimal’ payload, as what works for one vulnerable parameter may not work for another. We pose that in the case of injection-based webapp vulnerabilities the same RL methodology is capable of developing diverse payloads for vulnerabilities. Ideally, an agent should learn a policy that generates payloads, adapting to the different sanitisations and contexts in which payloads can occur. This could be considered analogous to learning different strategies in Chess, or Go.

3.3.1 The Payload Generation Environment

Now we can create an environment for the RL agents in HAXSS. Each agent will interact with an instance of the payload generation environment (hence referred to as *environment*, for short) shown at a high level in Figure 3.1. At the conceptual level, the environment receives an action from the agent which is used to mutate the payload and is turned into a concrete action on the webapp via the Injection Interface. This then receives a response and, based on that, determines a reward and a next state to be returned to the agent. The reward depends on the kind of agent that triggered the action, as discussed in Section 3.4. Using different rewards accounts for the different actions of agents, and the behaviours we want them to learn. The environment is implemented in the OpenAI Gym format [31].

We employ an automated *Crawler* component to identify the source-sink combinations present in the webapp. The crawler is implemented in Python and Selenium, based on the Black Widow Crawler by Eriksson *et al.* [74]. Our implementation follows the approach described by

Eriksson *et al.* [74]. It is possible to make changes to the crawler, such as capturing additional requests, modifying headers, or authentication. However, XSS is a client-side vulnerability that is triggered in the browser making further modifications to the crawler of limited value. A detailed exploration into crawling approaches and techniques is auxiliary to the contributions of this Thesis.

For completeness, we explain briefly how the crawler we use functions within HAXSS. To begin, the crawler finds all anchor tags on a webpage and validates that they lead to new pages by clicking on them. If this leads to a new page within the domain-under-test, the URL of the anchor tag is added to the list of pages to be traversed. To avoid the problem of infinitely crawling among pages we need some ruling around URLs. In particular with regard to query components with URLs. We consider query components to be irrespective of identifying new pages. For example, `http://www.example.com/?=value` is considered the same as `http://www.example.com/` which is the same as `http://www.example.com/#FromTheDOM`. While this may result in webpages being misclassified as the same, Eriksson *et al.* [74] show this to be sufficient to find new web pages.

To identify source-sink combinations the Crawler inputs a unique token into each `input`, `textarea` component, and parameters in the URL of a GET request. The unique tokens are then checked for when searching pages to determine if a source-sink combination has been found

The Crawler then stores the source-sink combinations in addition to the information on the HTML tags, that are required to transition to new pages. This information is then passed to an *Injection Interface*.

The environment allows the agent to generate the payload and inject it into the webapp via the Injection Interface (e.g. `input` tags). This Injection Interface sends a HTTP request to the web page with the payload and a unique `token` of 10 digits used to detect vulnerabilities. Tokens identify the current payload in cases where multiple payloads are triggered independently (e.g. in stored XSS). The Interface then observes the HTTP response, and the browser rendered DOM. By monitoring this we can determine if a payload has been sanitised, *i.e.* it has been modified between the source and sink locations. We can also monitor the rendered HTML

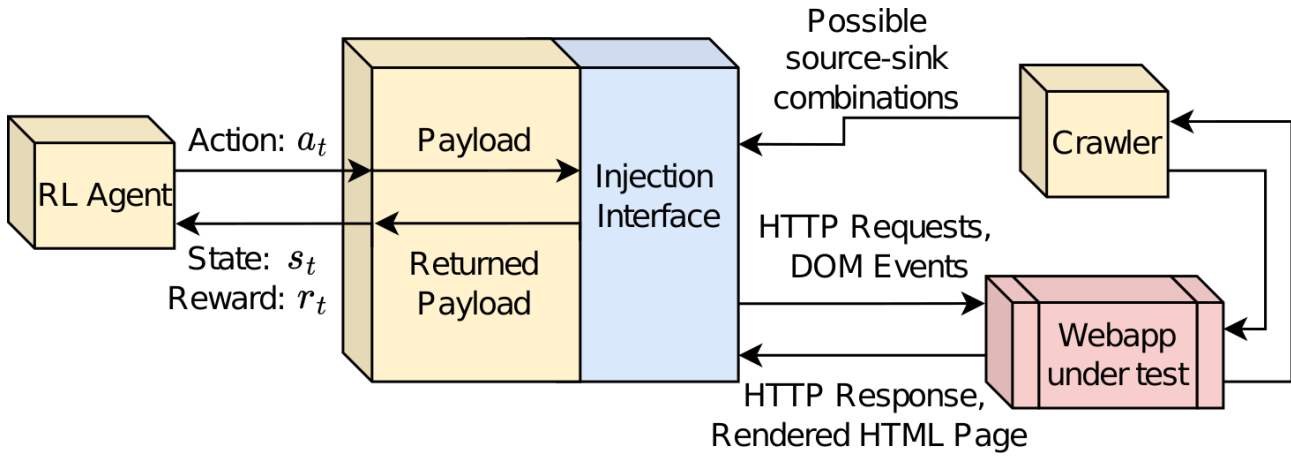


Figure 3.1: The overall HAXSS payload environment model, showing how the agent is able to interact with the web application.

response for successful payloads that trigger an alert containing the `token` value, making it possible to detect DOM-based, stored, and reflected XSS. This provides feedback for the RL agent to learn.

We define a *state* as a tuple of the currently generated payload and the returned payload from the webapp. This encapsulates information for the agent about the payload sanitisation in the state. Both the payload and returned payload are reduced to a numerical value using a vocabulary formed of the agent actions. Thus, `script`, `onLoad=alert(token)`, `>` is represented as 1, 2, and 3 respectively.

To generate a payload a series of actions will be taken to alter the payload state within an episode. Episodes are limited by the number of actions that may be taken (See Section 3.4.3), and additionally terminated when a legitimate vulnerability is found by the Injection Interface. When an episode is ended the state is reset to its starting state (a tuple of new token value and empty string).

3.4 The Haxss Architecture

We have broadly defined the core components of HAXSS that apply to both agents and their respective environments, yet the central idea of HAXSS is to split the payload generation problem into two games. The hierarchical structure, and transitions between these games is shown

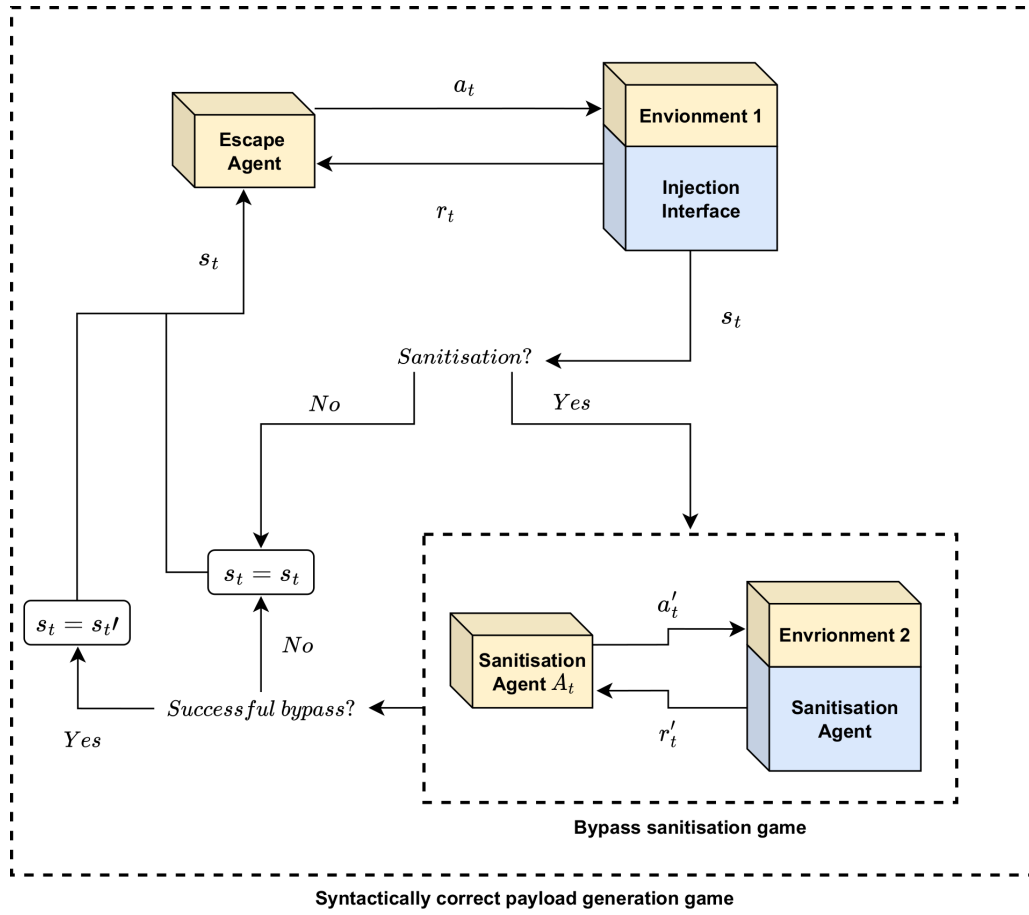


Figure 3.2: Conditions for triggering the separate games (syntactically correct payload generation, and bypassing sanitisation) that form HAXSS.

in Figure 3.2 and algorithmically in Algorithm 3.1. This hierarchy embodies the *RL Agent* of Figure 3.1. We further detail here the separate games that each agent play, providing them with their specific scopes and their interactions.

The first game occurs when there is no sanitisation of the input payload. In this game the Escape Agent learns a policy to generate a variety of syntactically correct payloads of HTML and/or JavaScript that escape their context to trigger an XSS. The task for this agent is to generate a legitimate and correct fragment of HTML and/or JavaScript to trigger an XSS. This allows the Escape Agent to focus on learning a policy that generates a variety of syntactically correct payloads that escape their context.

The second game occurs when payload sanitisation is detected on inspecting the sink output. In this game Sanitisation Agents focus on bypassing sanitisation to inject a given payload into a specific sink.

Algorithm 3.1: HAXSS Payload generation loop, highlighted areas indicate training specific execution.

```

1 Function Haxss(crawl_data):
2     urls, params = crawl_data
3     done = False
4     episode = 0
5     while not done:
6         if win_cond or loss_cond:
7             if next(urls, url) == None:
8                 done = True                                // exit when finished all inputs
9                 break
10            url = next(urls, url)                            // get next URL
11            param = next(params, param)                      // get next parameter
12            esc_env.load(url, param)
13            st = esc_env.reset()
14            for step in range(0, ESC_MAX_STEPS):
15                esc_actions = esc_agent.get_worker_actions(st)
16                payload = esc_env.step(esc_action)
17                st+1, rt, vt, sant = inj_interface(payload, url, param)
18                minibatch = esc_buffer.sample() esc_agent.train_dqn(minibatch)
19                if sant == True:
20                    payload = attempt_bypass_san(payload)    // Began bypass sanitisation game
21            episode ++
22            if episode % 50:
23                esc_agent.update_target_net()
24                san_agent.update_target_net()
25 Function attempt_bypass_san(payload):
26     san_env.load(payload)
27     for EP in range(0, SAN_MAX_EPS):
28         st = san_env.reset()
29         for step in range(0, SAN_MAX_STEPS):
30             san_action = san_agent.get_action(st)
31             san_payload = escape_env.step(escape_san)
32             st+1, rt, vt, bypt = inj_interface(san_payload, url, param)
33             minibatch = san_buffer.sample()
34             san_agent.train_dqn(minibatch)
35         if bypt == True:
36             return san_payload,
37     return payload, None

```

This hierarchical structure reduces the number of different actions needed by each agent. This allows us to determine if the RL agents are capable of achieving their constrained tasks, effectively exploring the action-state space for optimal strategies.

3.4.1 The Escape Agent

We will first consider the Escape Agent of HAXSS, which we challenge with a single task: generating payloads that escape the context in the sink. By doing this we have the agent generate

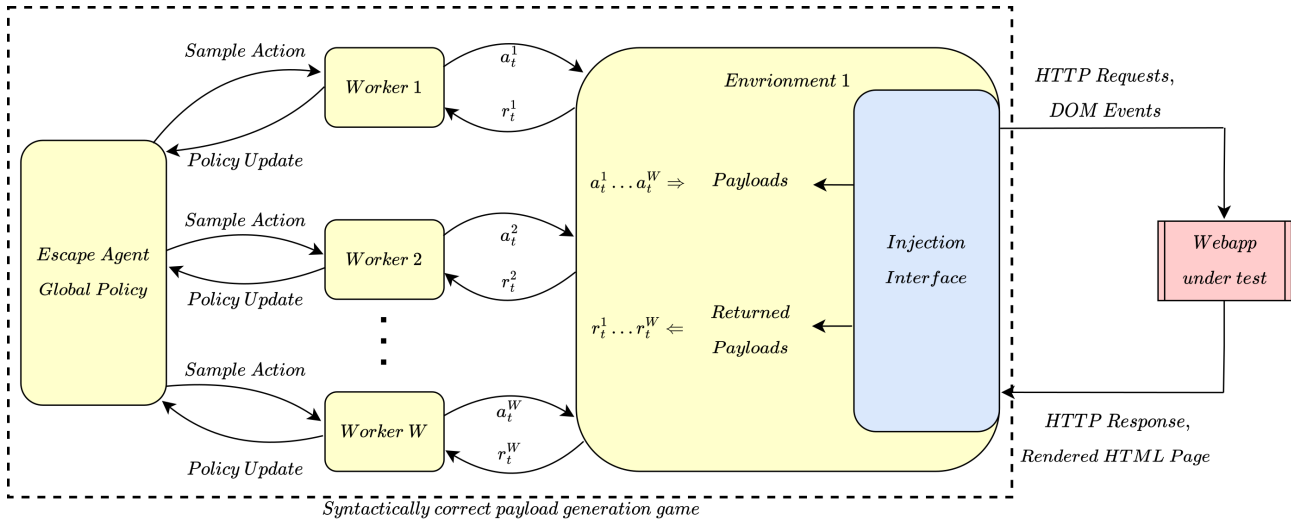


Figure 3.3: HAXSS The Escape Agent architecture

the simplest case of XSS payloads, those that are in plaintext (*i.e.* HTML and JavaScript).

To increase the Escape Agent’s overall efficiency in finding vulnerabilities we use worker agents, as shown in Figure 3.3. These worker agents sample actions from the probability distribution learned by the Escape Agent’s global policy in Figure 3.3. The gathered transitions of s_t , a_t , s_{t+1} , r_t can then be stored in the global buffer of the Escape Agent. The global policy is then updated using the global buffer during back propagation after workers have executed actions. This can be considered off-policy learning as we store the batched data in the replay buffer.

The actions of the Escape Agent can be categorised into 4 classes and can create 6,635,520 unique payloads, actions are taken from the OWASP guidelines on testing for XSS²:

1. **Escaping actions** are used to change the current context. These consist of an escape string, suffix-escape string, closing parent tag, and $>$. Escape strings are generated from the sink when HAXSS begins testing a new source-sink combination. For example, if the user input were found in: `<body onLoad='x = "INPUT"'>` the escape string would be formed of `"'`, to close the JavaScript and enter the context of HTML tag. This is achieved by injecting the unique token into the source and parsing the sink to identify where it appears. Based on such context, we algorithmically build the escape string.
2. **Tag actions**, which insert an open and closing tag into the payload after any parent

²<https://owasp.org/www-community/attacks/xss/>

```

1 var regex = /localhost/i
2 var elements = document.querySelectorAll("img,a,body,style,script");
3 for (var j = 0; j < elements.length; j++) {
4   var src = elements[j].getAttribute('src');
5   if (regex.test(src)) {
6     alert(src.split('http://localhost:666/xss.js?=')[1]);
7   }
8   var href = elements[j].getAttribute('href');
9   if (regex.test(href)) {
10    alert(href.split('http://localhost:666/xss.js?=')[1]);
11  }}

```

Listing 3.2: JavaScript functions called to by HAXSS

closing tag if present, otherwise enclosing the current payload. Tag name actions include: `img`, `script`, `style`, `body`, and `a`.

3. **Tag attributes**, ensure the tag actions trigger vulnerabilities. These include: `onError`, `onLoad`, `onMouseOver`, `onKeyPress`, `src`, `href`. The onEvents are set to `alert(token)` while `src` and `href` call to a webpage with JavaScript that triggers an alert with the token (see Listing 3.2) . These are inserted into an opening HTML tag if present, otherwise appended to the payload.
4. **Standalone**, which can trigger a variety of XSS depending on their use, including: `alert(token)`, `javascript:alert(token)`, and the URL of a page hosting JavaScript to trigger an alert. These are inserted into the payload between an opening and closing HTML tag if present, otherwise appending to the payload.

The Escape Agent receives both extrinsic and intrinsic reward. The extrinsic reward, from the Injection Interface, prompts HAXSS to generate well formed, malicious, payloads using small rewards for changing context, and unsanitised payloads. The extrinsic reward is defined in Eq. 3.1.

$$r = \begin{cases} R_{max} & \text{Successful vulnerability} \\ R_{con} & \text{Change execution context} \\ R_u = R'_u + \frac{1}{6}(R'_u \times t_a) & R'_u < R_u < \frac{1}{2}R'_u \text{ For attribute } t_a \text{ returned in unsanitised payload} \\ R_s = R'_s + \frac{1}{6}(R'_s \times t_a) & R'_s < R_s < \frac{1}{2}R'_s \text{ For attribute } t_a \text{ returned in sanitised payload} \\ R_{pen} & \text{If no payload returned} \end{cases} \quad (3.1)$$

This reward function is motivated by the process of testing for XSS, the most desirable outcome

being to trigger a vulnerability, hence a large positive reward (R_{max}). The reward shaping employed guides the agent to the end goal. In particular it provides a small positive reward (R_{con}) is given for changing the context (*i.e.* from any of the contexts identified in Section 3.1 to any other context).

The remaining rewards are negative. The Escape agent is penalised for a benign unsanitised payload (R_u), where this reward becomes more negative (by $\frac{1}{6}(R_s)$) as the payload grows larger, according to more HTML tags or tag attributes present in the payload (t_a). This is to prevent payloads from growing unnecessarily large. The same function is used when sanitisation occurs (R_s) but with a greater penalty (R'_s). This is to prompt the agent to produce payloads that are not excessively bloated and avoid sanitisation. The values of the constants are determined via a hyperparameter search, details of which can be found in Table 3.2. We report here the exact values used in our implementation:

$$R_{max} = 10 \quad R_{con} = 0.5 \quad R'_s = -3 \quad R'_u = -4.5 \quad R_{pen} = -5$$

The intrinsic reward is generated via Random Network Distillation (RND) [32] which promotes exploration, improving performance when testing new source-sink combinations. This uses two neural networks, both using the current state as input. One network (the predictor) learns to predict the output of the other (the target). Using the mean squared error between the target and predictor networks as a reward for the Escape Agent creates a diminishing value based on how often states are seen. This incentivises exploration of the state-space even after changing to new source-sink combinations, giving a distinct advantage: it results in agents that are able to generalise better when facing new tasks (such as exploiting different XSS vulnerabilities). We clamp this reward between -0.5 and 0.5 to prevent the Escape Agent from learning a purely exploratory policy as recommended by Burda *et al.* [32].

3.4.2 Sanitisation Agents

We have previously introduced the Escape Agent of HAXSS which is constrained to generate only plaintext XSS payloads, without considering any possible obfuscation of the payload. However, we would expect, as outlined in Section 3.2, that sanitisation of payloads would be attempted in a web application. We define a series of *Sanitisation Agents* whose task is constrained to bypass the sanitisation of payloads generated by the Escape Agent.

A Sanitisation Agent is invoked when payloads injected in the source are modified in the sink, and sanitisation has occurred. The agent receives the current payload from the Escape Agent and uses its action budget to bypass the sanitisation. If successful, the new unsanitised payload and the obfuscations used (*e.g.* encoding) are returned to the Escape Agent, allowing the bypass of sanitisation until the end of the episode, as in Figure 3.2. Upon reaching the end of the episode the state is reset to a new token.

We define a Sanitisation Agent for each of the actions available to the Escape Agent, hence we define 18 Sanitisation Agents, (Agents 2.0-2.17). This is under the intuition that these agents will develop techniques to bypass the sanitisation caused by the most recent action of the Escape Agent. For example, if the Escape Agent were to take the tag action of `script` on the existing payload of `alert(token)` the payload would then be: `<script>alert(1)</script>`. If this were then sanitised the Sanitisation Agent for this action learns to bypass the sanitisation specific to the `script` tag. While we could define more general Sanitisation Agents based on the 4 action classes of the Escape Agent, as we are considering a black-box approach, we assume no knowledge of the sanitisation methods. Hence, the only information we have is that the previous action caused a sanitisation to occur.

Each Agents 2.0-2.17 have the same set of actions available to them, these are taken from the OWASP guide on XSS filter evasion³. When these actions are combined together on a payload it can result in 384 unique payload obfuscations. These actions are:

1. **Capitalisation actions**, one to alternate upper and lower case letters in the payload,

³https://cheatsheetseries.owasp.org/cheatsheets/XSS_Filter_Evasion_Cheat_Sheet.html

another to have all lowercase.

2. **Encoding actions**, to encode the plaintext version of the payload, three encodings are possible (UTF-8, URL, and HTML) in addition to a decoding action.
3. **Escaping actions**, apply additional functions to the escape string applied by the action of the Escape Agent, including: backslash escaping quotes, inserting a < before an opening tag, inserting a ; after quote mark(s).
4. **Tag attribute separators**, these replace all separators in the payload, these include %0D, %2F, %0C.
5. **Tag obfuscation**, this inserts %00 into the tag name.
6. **Back ticks**, which replaces any instance of brackets '(' or ')' with a back tick `.`

We reward the Sanitisation Agents using a standard sparse reward functions in RL. This forces the agent to find the optimal solution as quickly as possible the Sanitisation Agents are only rewarded for a successful bypass. At each step agents receive a penalty of -1, unless they are able to successfully bypass sanitisation, in which case they receive a reward of 10.

3.4.3 Deep Learning Implementation

The Escape Agent and the Sanitisation Agents are implemented using the same deep learning architecture: a DQN with Target Q-Network and Q-Network, each having five layers. The input layer takes a minibatch of state tuples, followed by three hidden layers having 64, 96, 64 nodes, an output layer containing a node for each action. Both agents then produce Q-values at their respective output layers that can either be sampled from or the maximum value selected to determine an action. The Escape Agent has an additional output layer of a categorical distribution so the workers can sample actions. We use the `tanh` activation function and the Adam optimiser for each agent in the hierarchy. This architecture was seen to provide a balance between performance and compute time to predict the next action.

Table 3.2: Hyperparameter search values and the selected configuration of HAXSS.

Parameter	Lower Bound	Upper Bound	Selected
γ	0.5	0.999	0.999
Learning Rate	0.05	0.0005	0.005
Batch Size	50	200	100
Update Step	10	150	50
Episode Length	5	20	10, 5 (Escape Agent, Sanitisation Agent)
R_{max}	1	10	10
R_{con}	0	2	0.5
R'_s	-0.3	-13.5	-3
R'_u	-0.45	-9	-4.5
R_{pen}	0	20	-5

Hyperparameters were determined by performing a grid search, details of the range of values tested and those selected are in Table 3.2. Both agents use standard ε -greedy decay with $\varepsilon = 1$ (decaying by 0.999 after each episode). The Escape Agent has episodes of 10 actions to prevent payloads from becoming excessively bloated, while Sanitisation Agents have this reduced to 5 actions to allow for more episodes to attempt bypassing sanitisation. We allow Sanitisation Agents 3 episodes to bypass the sanitisation of a payload before giving up.

The Escape Agent uses a standard Prioritised Experience Replay buffer to increase the frequency at which samples occur in minibatches. Using $\alpha = 0.6$ ensures the model has a higher priority than in the uniform case ($\alpha = 0$), but lower than using complete prioritisation ($\alpha = 1$). The value of β is taken using a common value of 0.4 to reduce the bias of sampling from the buffer [249]. Sanitisation Agents use a uniform buffer, as they will have fewer samples and are less concerned with increasing diversity, aiming instead to bypass sanitisation. Similarly, while the Escape Agent uses curiosity to allow it to adapt to new source-sink combinations faster, Sanitisation Agents do not. Allowing them to focus on the specific task: bypassing sanitisation of the associated action of the Escape Agent.

3.4.4 Adapting to New Source-Sink Combinations

Moving to new source-sink combinations could be considered as analogous to playing a new level in a video game, or a new opponent in a game of chess. In such cases it is important

Table 3.3: Sanitisation rules and output contexts from the XP Train Bed used for training (1-15), and the XP Test Bed for testing (16-25).

Input Field	Sanitisation rule	Context in Page	Context
1	preg_replace('<(.*)s(.*)c(.*)r(.*)i(.*)p(.*)t/i', '', INPUT)	[INPUT] 	Tag body
2	preg_replace('/"/, '\\"', INPUT); preg_replace('/\'/, '\\\\', INPUT);	<script type="text/javascript">	JavaScript
3	preg_replace('<script[^\>]*><script>/', '', INPUT)	function foo() a="[INPUT]"</script>	JavaScript
4	urldecode(INPUT); replace("<","<","INPUT); json_decode(INPUT)	[INPUT] 	Tag body
5	preg_replace('/>/', '>', INPUT); urldecode(INPUT)	<!-- [INPUT] -->	HTML comment
6	urldecode(INPUT); addslashes(INPUT)	<script>' [INPUT]'</script> 	JavaScript
7	urldecode(INPUT); preg_replace('/<(.*)s(.*)c(.*)r(.*)i(.*)p(.*)t/i', '', INPUT);	<script>[INPUT]</script> 	JavaScript
8	preg_replace('/(.*)on(.*)=/', '', INPUT)	<!-- [INPUT] --> 	HTML comment
9	htmlEntities(INPUT, ENT_QUOTES)	[INPUT] 	Tag body
10	str_replace('<script>', '', INPUT)		
11	htmlEntities(INPUT, ENT_QUOTES);		
12	preg_replace('*/(?!<(script img)[^\>]*> <(script img)>/', '', INPUT);		Tag attribute
13	html_entity_decode(INPUT);		
14	preg_replace('/"/, '\\"', INPUT); preg_replace('/\'/, '\\\\', INPUT)	<div onmouseover="x=' [INPUT] '"></div>	Tag attribute
15	preg_replace('/"/, '\\"', INPUT); urldecode(INPUT);	<style>p {color: " [INPUT] "></style>	Tag body
16	preg_replace('/"/, '\\"', INPUT); preg_replace('/\'/, '\\\\', INPUT)	< [INPUT] >	Tag name
17	preg_replace('/"/, '\\"', INPUT); preg_replace('/\'/, '\\\\', INPUT);	<input type="checkbox" name=[INPUT]>	Tag attribute
18	preg_replace('/"/, '\\"', INPUT);		
19	preg_replace('/"/, '\\"', INPUT); preg_replace('/\'/, '\\\\', INPUT);		Tag attribute
20	preg_replace('/"/, '\\"', INPUT); preg_replace('/\'/, '\\\\', INPUT);	< [INPUT] >	Tag name
21	preg_replace('*/(?!<(script img)[^\>]*> <(script img)>/', '', INPUT);		
22	preg_replace('*/(?!<(script img)[^\>]*> <(script img)>/', '', INPUT);		
23	preg_replace('*/(?!<(script img)[^\>]*> <(script img)>/', '', INPUT);		
24	preg_replace('*/(?!<(script img)[^\>]*> <(script img)>/', '', INPUT);		
25	preg_replace('*/(?!<(script img)[^\>]*> <(script img)>/', '', INPUT);		
16 CVE-2020-28919	preg_replace('*/(?!<(script img)[^\>]*> <(script img)>/', '', INPUT);	[INPUT]	Tag body
17 CVE-2021-24884	urldecode(INPUT);html_entity_decode(INPUT);	<a href="https://example.com"	Tag attribute
18 CVE-2021-35043	strip_tags(INPUT, 'audio<video><button><a>');	data-firmverify=" [INPUT] ">	Tag attribute
19 CVE-2019-10062	html_entity_decode(INPUT);	 Vulnerable 	Tag attribute
20 CVE-2021-22889	preg_replace('*/(?!<(script img)[^\>]*> <(script img)>/', '', INPUT);	[INPUT]	Tag body
21	htmlspecialchars(INPUT, ENT_SUBSTITUTE ENT_HTML401);	<input type='checkbox' name='n' value=' [INPUT] ' >	Tag attribute
22	str_replace('<','<','INPUT);str_replace('>','>','INPUT);	[INPUT]	Tag body
23	urldecode(INPUT);	<script>eval(' [INPUT] ')</script>	JavaScript
24	str_replace('<','<','INPUT); str_replace('>','>','INPUT);	< [INPUT] >	Tag name
25	str_replace('<','<','INPUT); json_decode(INPUT);	<script>var d = document.createElement('div');	JavaScript
	preg_replace('/"/, '\\"', INPUT); urldecode(INPUT);	d.innerHTML = '<[INPUT]>';	JavaScript
	urldecode(INPUT); addslashes(INPUT);	document.body.appendChild(d);</script>	JavaScript
			Tag attribute

to understand the changes in the environment and new strategy required. Thus, when HAXSS moves to new source-sink combination it is important for exploration to happen. We enforce this by altering the rate at which random actions are taken: if the value of ε after the previous task is less than 0.3, we increase it to 0.6, and if it is between 0.3 and 0.7 we increase it by 0.2. This is a simple heuristic for changing the rate of exploration, which we determine empirically, along with the hyperparameters. In practice little variation was seen when varying this except in extreme cases.

3.4.5 The XSS Payload Test Bed

To train and evaluate HAXSS we introduce a micro-benchmark of 25 reflected XSS vulnerabilities, the XSS Payload (XP) Train and Test Bed. The benchmark is organised as an easy to crawl HTML page with one form and multiple input fields, each hosting a vulnerability. We designate 15 of these for training and a further 10 for testing. Our aim is to provide a diverse enough set of vulnerabilities that allow an RL agent to learn how to generate legitimate payloads and adapt to unseen scenarios. Such an approach is well aligned with Klees *et al.* [146],

as we can evaluate on a series of known vulnerabilities.

The XP Test Bed includes 5 XSS taken from vulnerabilities with associated CVE numbers. We include these to determine if HAXSS (and other baselines) can find legitimate vulnerabilities. As advocated for by Klees *et al.* using known bugs and vulnerabilities provides a ground truth for evaluation that isn't present when testing unknown bugs. These were selected from the NVD [202], a repository of known vulnerabilities and their associated data. From NVD we select the vulnerabilities according to the availability of the sanitisation rule, context, and a legitimate payload to verify the vulnerability. They range in CVSS 3.x score, a common metric to assess the severity of a vulnerability, from 5.4 to 9.6.

In Table 3.3 we report the different combinations of HTML output contexts and (flawed) sanitisation rules used for both training and testing. For example, field 21 in Table 3.3 removes any instances of > and < from the user input, replacing them with the HTML encoded variants, and then decodes any URL encoded characters, placing the result in the body of a div tag.

3.4.6 Training

Training HAXSS should result in a policy that is capable of generating diverse payloads, and adapting to new injection points by tailoring the payloads it generates. This is achieved by initial training of HAXSS on each of the vulnerabilities on the XP Train Bed. Such training will provide an understanding of how to form a range of payloads with different sanitisation in different contexts. Similarly, this form of training can be considered as analogous to offline learning, as we train the policy initially on the XP Train Bed, and then use the learnt policy in a separate web application.

The initial training of HAXSS is a balance between learning how to generate different payloads, and preventing overfitting to each of the vulnerabilities. To prevent this over fitting and to identify when HAXSS has 'learnt enough' from a task we use a *win* and *loss* criteria. The win criterion is considered met when a worker is able to generate 14 or more payloads within the last 20 episodes, as this demonstrates consistent exploitation of the vulnerability. The loss

criterion is reached by a worker at 200 episodes on a task without meeting the win criterion. This ensures that all vulnerabilities are trained on, with an equal maximum effort budget. Note that even if an agent reaches the loss condition, it may have been able to generate numerous working payloads, and will still benefit from the task.

Training is conducted on a Ubuntu 20.04 Linux computer with 16GB RAM and an Intel core i7 8700k processor. HAXSS is trained using 2 asynchronous workers which can execute in parallel, increasing the speed at which the agent can learn the policy [189].

Haxss training in detail. We report the results of training HAXSS on the XP Train Bed in Figure 3.4 using the number of legitimate XSS payloads that trigger the vulnerability. Each of the fifteen vulnerabilities of the XP Test Bed is shown as an individual peak in Figure 3.4. The use of peaks makes readability of each vulnerability clearer. Once HAXSS has ‘won’ or ‘lost’ against a vulnerability, successful payloads are reset to zero.

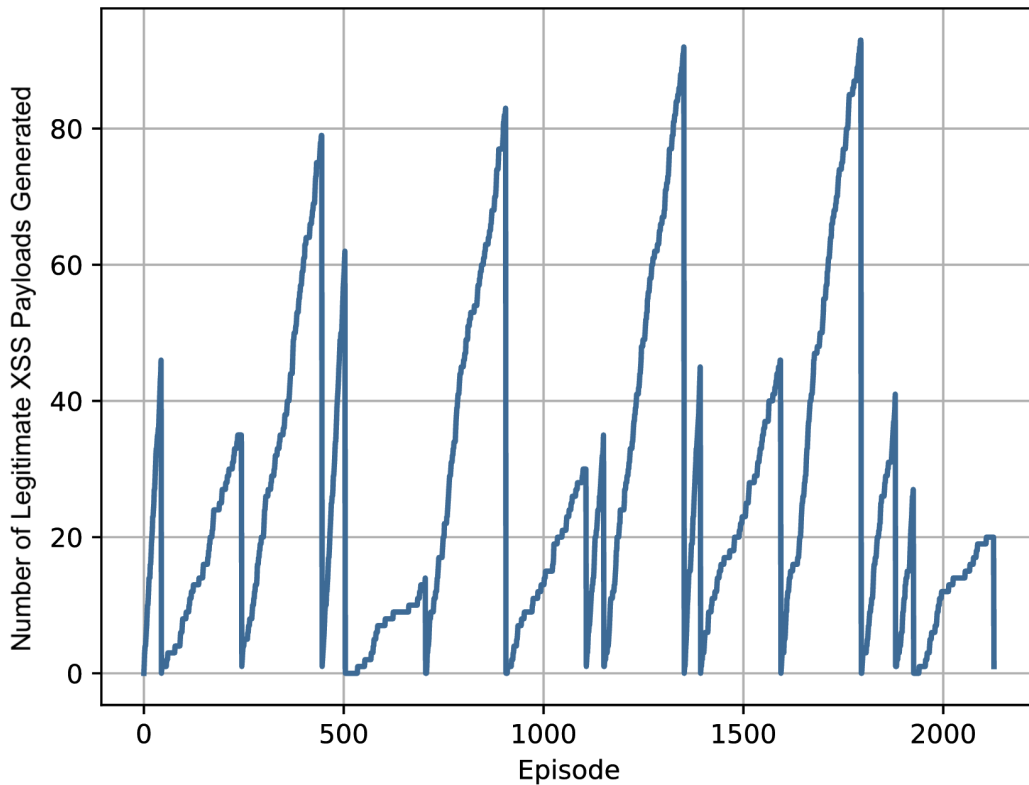


Figure 3.4: Number of legitimate XSS payloads that trigger a vulnerability generated by HAXSS training on the XP Train Bed.

From Figure 3.4 we can see that HAXSS is able to generalise to tasks, testing new source-sink combinations. There are multiple indicators of this, including: the steep slopes that occur after

changing to a different vulnerability, the large number of payloads that HAXSS develops, and the low number of requests required to find the first vulnerability - an average of 103.03 requests per payload. HAXSS is also consistent across training: generating 20 or more vulnerable payloads for 14 out of 15 fields, with a minimum of 14 for field 5 and a maximum of 93 for field 12.

Yet, we see in some instances that HAXSS generates a high number of payloads and then generates relatively fewer. We will examine an instance of this between fields 9 and 10. In field 9 HAXSS must escape from a tag attribute context (`<img src='[INPUT]>`) avoiding the use of `script` and `img` tags. HAXSS is able to generate 89 legitimate payloads for this vulnerability during training, but never meets the win criteria when testing on this field. Conversely field 10 has the Escape agent utilise the escape string action and then close the current div tag. Sanitisation Agent then needs to add backslashes to the escape string to evade sanitisation, resulting in an escape string of `\'\\"></div>`. While a more subtle chain of actions is required to escape the context (and display the subsequent XSS) HAXSS succeeds in meeting the win criterion using only 46 successful payloads. Behaviour like this similarly shows the generalised learning of HAXSS as it can use learnt experiences to adapt to new vulnerabilities.

HAXSS wins 9 fields (1, 3, 4, 6, 8, 10, 12, 13, 14), losing only 6 fields. In particular the lost fields are:

- Field 2: the first change in context for HAXSS.
- Field 5: a change to the JavaScript context is seen.
- Field 7: where the HTML comment context applies a sanitisation of quotemarks, differing from the previous related context that could be bypassed via encoding.
- Field 9: which removes `script` and `img` tags in the first tag attribute context
- Field 11: which occurs in a tag body context with escaping double quote marks
- Field 15: which removes any payloads with three of the five tag actions HAXSS has.

These vulnerabilities take longer to adapt to than the others due to the curriculum of fields. For example losing fields often occur when encountering significant changes in the combination

of context and sanitisation. While an altered curriculum may lead to improving the number of fields that HAXSS wins, optimising such behaviour leads to diminishing returns. Indeed, despite meeting the loss criteria, HAXSS still learns and generalises from these fields, developing over 20 working payloads in each instance. Which allows HAXSS to improve on later vulnerabilities (e.g. 11 leading to better results in 12, 13, and 14).

3.5 Benchmark-based Evaluation

Here we will compare HAXSS to a selection of 9 established vulnerability scanners on our micro-benchmark for XSS exploit generation, and on a macro-benchmark for webapp vulnerability scanning.

3.5.1 Experiment Setup

Using Haxss as a vulnerability scanner

To use HAXSS as a vulnerability scanner after it has been trained on the XP Train Bed we change some of the parameters from training. The aim of this is to reduce bias and overfitting in the new samples, and minimise the number of requests made.

One could keep training HAXSS as described whilst scanning new target applications for vulnerabilities, but that would come at the cost of time, number of requests, and possible degradation of the agent policies from overfitting to specific samples. Instead, we recommend changing some of the hyperparameters before using HAXSS as a vulnerability scanner, as described here.

In order to give more emphasis on the learnt policy, we reduce the exploration parameter to 0.05, thus giving HAXSS a 5% probability of taking a random action. Fixing the rate at which this happens still allows HAXSS to reach new states when in scenarios beyond those in training. To reduce the amount of web traffic generated during scanning, we reduce the number of worker agents from 2 to 1, and change the win and loss criteria. An agent should move on to the next

target as soon as a working payload is generated (win=1) and should stop trying after 150 attempts at worst (loss=15).

Vulnerability scanners

We select 4 state-of-the-art open source general vulnerability scanners: Arachni 1.5.1 [169], OWASP Zed Attack Proxy (ZAP) 2.10.0 [215], w3af 1.6.49 [241], and Wapiti 3.1.0 [273]. We use two state-of-the-art academic scanners for XSS: Black Widow [74] and the RL-based model Link [153]. These have been used in recent studies, hence provide an established benchmark for comparison [39, 74]. We also select 3 XSS-specific scanners, compared in recent work [39]: XSSer 1.8.4, (included in the popular Kali distribution of Linux) [234], XSppear 1.4.1 [105], and the dynamic payload generator XSSMAP [233].

Furthermore, we must also establish the performance of the core design choices of HAXSS, specifically the use of the RL, and the hierarchy of separate agents. Thus, we include two variants of HAXSS, the first of which is AXSS, a non-hierarchical version. AXSS follows the same state-action design and training process as HAXSS, the difference between the two arising from the hierarchy of agents. Specifically, AXSS has only a single agent using both the escape and sanitisation actions. This gives a direct point of comparison for the effect of constraining the ‘games’ HAXSS can play. We also compare against Rand-HAXSS, a variant that takes random actions. This provides a direct comparison to show how RL can *effectively* use the action space provided to it. This is important to consider as much of the prior works in the literature use random selection to create payloads.

All scanners are able to find reflected XSS; all but XSppear and XSSMAP can find stored XSS; only ZAP, Arachni, and XSSer are able to find DOM based XSS. The settings we used for these scanners can be found in Table A.1 (Appendix A). We conduct the evaluation using the same hardware as for training.

Table 3.4: Scanners results and request statistics for the XP Test Bed. TP: XSS found; FN: XSS not found; FP: XSS not working.

Scanner	1	2	3	4	5	6	7	8	9	10	Total TP	Total requests	Avg requests per working payload	Avg requests per true positive
Arachni	TP	FP	FP	TP	TP	TP	FN	TP	FP	FN	5	3710	742	742
Black Widow	TP	TP	TP	TP	FN	FN	FN	FN	FN	TP	5	100	20	20
Link	FN	FP	TP	TP	FP	FN	FN	FN	TP	TP	4	204	51	51
ZAP	FN	FP	TP	TP	TP	FN	FP	FN	TP	FP	4	206	51.5	51.5
w3af	FP	TP	FP	FP	TP	FN	TP	FP	TP	FP	4	512	128	128
Wapiti	TP	TP	TP	TP	TP	FN	FN	TP	TP	TP	8	157	19.6	19.6
XSSer	TP	FN	FN	FN	FN	FN	FN	FN	FN	FN	1	13010	13010	13010
XSppear	TP	FP	TP	TP	FN	FN	TP	FN	FP	TP	5	3868	101.8	773.6
XSSMAP	FN	FN	FN	FN	FN	FN	TP	FN	TP	FN	2	1810	603.3	905
Axss	FN	FN	TP	TP	TP	FN	TP	FN	FN	TP	5	780	156	156
Rand-HAXSS	TP	FN	FN	TP	TP	FN	TP	TP	TP	FN	6	2710	451.67	451.67
HAXSS	TP	TP	TP	TP	TP	TP	TP	TP	TP	TP	10	724	72.4	72.4

3.5.2 Micro-benchmark: Payload Generation

In the first experiment we run each scanner against the XP Test Bed described in Chapter 3.4.5. Using this micro-benchmark provides a ground truth of 10 easy-to-crawl injection points which require independent payloads for each vulnerability, 5 of which have associated CVE numbers. The results are reported in Table 3.4. Using such a benchmark as the XP Test Bed provides a controlled setting where we can evaluate all scanners regardless of their crawling capability, while giving them access to known vulnerabilities.

The best performance among the other scanners comes from Wapiti as it is uses a single payload that is capable of bypassing multiple sanitisations: `<Svg/oNloAd=alert(/TOKEN/)>`. Though this payload always uses angle brackets, meaning Wapiti fails to find XSS for 6 and 7, which escapes payloads with `<` or `>`. Despite this both vulnerabilities can be exploited by URL encoding or changing contexts.

XSSer and XSSMAP do not share the same level of performance, despite being specific for XSS, they detect only 1 and 2 XSS respectively. XSSMAP fails on sanitisations which is surprising since XSSMAP is purportedly an ‘intelligent’ scanner capable of generating XSS payloads based on context and sanitisations. However, we find that while it can change contexts it is unable to escape sanitisations, a common theme amongst the scanners.

Link, an RL based XSS scanner reports two False Positives (FP) due to the use of static analysis

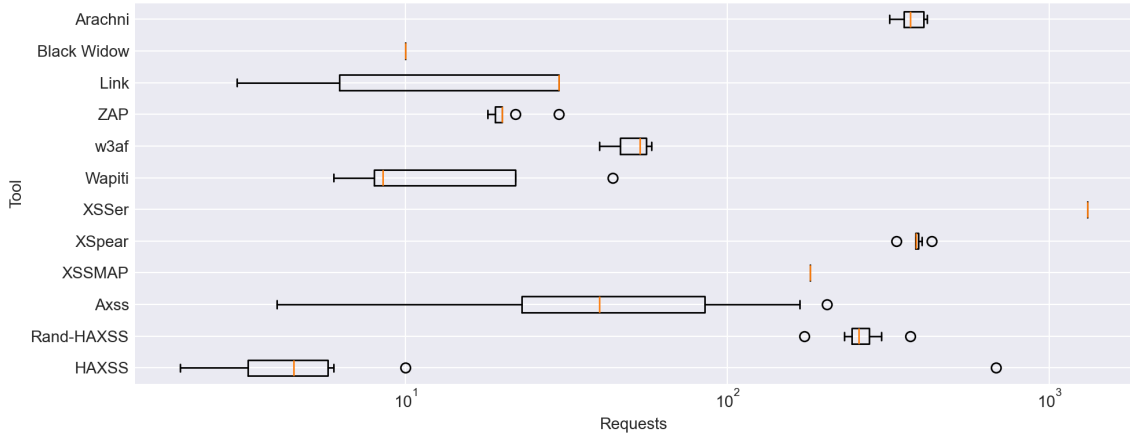


Figure 3.5: Box-plot to show the number of requests for each approach in the XP Test Bed, shown with log-scale. Circles indicate data points beyond the interquartile ranges.

in its vulnerability oracle. In addition it fails to find four XSS (False Negatives) as it fails to bypass sanitisation of tags in 1, angle brackets (via URL and JSON encoding) in 6 and 8, and not finding a legitimate payload for 2, 5, and 7. Overall, Link finds less XSS than our non-hierarchical variant, Axss and Rand-HAXSS.

HAXSS is the best performing of these scanners, finding all vulnerabilities. Note that its novel hierarchical structure plays an important role, as the non-hierarchical variant Axss is only able to detect half of these. However, Axss was unable to find the simpler cases of sanitisation and context e.g. in 1 and 2. It is also unable identify XSS in number 9, where sanitisation also occurs on the escape string, making context escape harder.

The utility of the hierarchical model is further illustrated by the performance of Rand-HAXSS which finds an additional XSS than Axss. Yet, to do so incurs the additional cost of more HTTP requests.

Number of requests

The high volume of traffic generated by automated web testing tools is considered one of their main shortcomings, alerting defenders that an attack is taking place [153]. Moreover, if a scanner needs fewer requests than another to exploit the same injection point, it can be considered more ‘intelligent’ [39]. Therefore, in Table 3.4 we also report the total requests

each scanner made, the number of requests per working payload (in case a scanner reported several different payloads for the same injection point, instead of stopping upon finding the first one), and the number of requests per identified TP (that is, vulnerable field). Additionally, we include Figure 3.5 to show the distribution of requests of each approach.

Black Widow has the lowest number of total requests as it always uses a fixed list of 10 XSS payloads. Yet, the expected number of request per payload found is higher, at 20. Wapiti has the lowest number of requests per true positive and working payload, as it attempts to algorithmically generate payloads that could work for the current context.

HAXSS has a high number of requests as it has an outlier in form 2, which uses 625 requests for the Sanitisation Agents but only 53 for the Escape Agents (seen in Figure 3.5). This is due the sanitisation only allowing certain tags on input that otherwise can't be bypassed. HAXSS does outperform AXSS achieving 83.9% fewer requests per working payload than its non-hierarchical equivalent.

The worst performing scanner is XSSer, as it blindly injects from a payload list of 1,293 payloads, resulting in the highest number of total requests, and average number of requests per TP. Arachni however, has the highest number of requests per working payload, due to an exhaustive search attempting to escape the context and execute a payload.

3.5.3 Macro-benchmark: Webapp Scanning

In the second experiment we assess the ability of the various scanners to generate working XSS payloads in a number of test bench webapps with known vulnerabilities. Since XSSMAP and XSppear do not include a crawling capability, we exclude them from this experiment. Our macro-benchmark is composed of Firing Range (0.48), SCARF (2007), WackoPicko (2018) [65], WebSecLab (0.8.1), and WAVSEP (1.5).

Firing range contains 227 XSS examples including 81 reflected XSS, 45 escaped XSS, 101 DOM XSS in an easy to crawl setting. The remainder of these XSS include Angular, redirected, and remote inclusion XSS. SCARF contains 8 XSS identified in prior work [66, 74], and also puts

Table 3.5: Scanners results for macro-benchmark. “Not Tried” indicates a failure of the crawler to find the injection point. FN denotes a failure to generate a payload to display the vulnerability.

Scanner	Firing Range					SCARF					WackoPicko					WAVSEP					WebSecLab				
	TP	FP	TN	FN	Not Tried	TP	FP	TN	FN	Not Tried	TP	FP	TN	FN	Not Tried	TP	FP	TN	FN	Not Tried	TP	FP	TN	FN	Not Tried
Arachni	42	16	0	137	32	3	1	1	5	0	4	0	1	1	2	36	32	0	22	0	15	4	2	12	7
Black Widow	29	0	0	166	32	8	0	2	0	0	5	0	3	0	0	22	0	0	24	44	N/A	N/A	N/A	N/A	40
Link	45	9	0	104	69	0	0	0	0	8	1	0	0	0	7	33	8	0	49	0	7	6	7	20	0
ZAP	50	17	0	111	49	0	0	2	5	3	3	1	2	1	1	24	46	0	20	0	11	15	4	9	1
w3af	2	55	0	138	32	1	0	1	6	2	1	1	2	3	1	5	46	0	39	0	2	13	5	19	1
Wapiti	43	31	0	121	32	0	0	2	3	5	2	0	2	3	1	12	13	0	65	0	11	4	6	14	5
XSSer	19	2	0	133	68	0	1	0	3	5	1	3	0	3	1	5	27	0	58	0	7	24	0	8	1
AXSS	21	0	0	80	126	5	0	2	3	0	5	0	3	0	0	30	0	0	53	7	9	0	10	20	1
Rand-HAXSS	22	0	0	79	126	3	0	2	5	0	3	0	3	2	0	29	0	0	54	7	10	0	10	19	1
HAXSS	45	0	0	56	126	8	0	2	0	0	5	0	3	0	0	46	0	0	37	7	17	0	10	12	1

the capability of a crawler to the test. WackoPicko is deliberately vulnerable and established in prior work [65, 74]. It contains 6 XSS vulnerabilities, 5 by design and one discovered in [74]. We exclude one of the planted vulnerabilities as it uses the deprecated Flash plugin. WebSecLab contains 23 reflected XSS, 6 DOM XSS and 11 TNs that may be misclassified as false positives (because payloads can be injected, but prevented from executing). In fact, HAXSS discovers that one of these is actually vulnerable, so the final tally is 30 TPs and 10 TNs. These XSS are contained in easy to crawl web pages, and exists in the contexts discussed in Section 3.1. WebSecLab also provides sanitisations that can be bypassed by different techniques: Unicode encoding, URL encoding, and escaping sanitisation attempts. It also contains a vulnerability all scanners miss, where the HTTP `referer` header is echoed on the loaded page. WAVSEP provides 90 examples of XSS (including 4 DOM based), also providing the contexts in Section 3.1 which can be exploited as both GET and POST requests. The sanitisation used includes: partial HTML encoding of characters (angle brackets, single quote, double quote) and tag removal. Two of the examples are also protected by anti CSRF-tokens. All examples are contained in easy to crawl web pages.

Scanner performance

Table 3.5 reports the results of the Macro-benchmark experiment. While our focus is on payload generation rather than scanning ability we include the number of source-sink combinations not identified by a crawler. This metric is important to consider as low numbers of FP/FN may

Table 3.6: Additional timing information of the scanners used on all webapps. ‘N/A’ denotes a scanner that had to be stopped after entering an infinite loop.

Scanner	Firing Range	SCARF	WackoPicko	WAVSEP	WebSecLab	XP Test Bed
Arachni	18min 22s	1hour 2min	N/A	27min 22s	52min 36s	2min 52s
Black Widow	12min 25s	19min 56s	44min 46s	8min 22s	N/A	6min 6s
ZAP	N/A	13min 41s	10min 53s	2min 58s	3min 15s	22.7s
w3af	3min 15s	5min 1s	6min 32s	4min 54s	3min 26s	1min 45s
Wapiti	1min 36s	7s	9s	5min 15s	9s	5s
XSSer	1hour 2min	10min 27s	6min 4s	1hour 14min	43min 39s	53s
AXSS	3hours 48min	2hours 40min	21min 4s	2hours 0mins	1hour 16mins	6min 2s
Rand-HAXSS	3hours 46min	2hours 33min	20min 4s	2hours 12mins	1hour 22mins	8mins 29s
HAXSS	4hours 8min	2hours 43min	23min 21s	2hours 23mins	1hour 46mins	9mins 52s

be in part due to failing to find cases. However, scanners tend to crawl efficiently finding most injection points. There is an exception in Firing Range due to the high number of DOM based XSS, and most scanners inability to access the DOM based sources. We also report in Table 3.6 the time each scanner takes on the benchmarks for completeness. However, such timing results we don’t consider to be a fundamental concern for evaluation, as with enough engineering effort significant improvements in performance can be made to be closer to mature tools such as ZAP, w3af, *etc.*

Black Widow, and HAXSS have perfect performance on SCARF and WackoPicko, finding all known XSS, and reporting no FPs. HAXSS finds the most vulnerabilities on WAVSEP but also suffers from a high number of FNs (37). The majority of these are because the necessary exploit techniques are beyond what HAXSS can generate given its action space and XP Train Bed. Further discussion on this will be in Section 3.5.5. This behaviour mostly arises from the context of VBScript (a deprecated programming language from Microsoft).

Arachni performs consistently well, identifying a high number of XSS in both WebSecLab and WackoPicko, and finding the second most XSS on WAVSEP after HAXSS. It only identifies three XSS in SCARF, reporting 5 FNs. Upon manual inspection, this is a fault of the crawler as it is unable to find the XSS after injection. In WAVSEP, Arachni reports a high number of FPs, as payloads are unable to effectively escape the context, thus are not executable.

The performance of Black Widow on WAVSEP suffers as the scanner fails to identify 44 injection points. It also fails to produce payloads for WebSecLab, as it enters an infinite loop.

ZAP, Wapiti and XSSer have overall a similar performance. They suffer from a high number of FPs in WAVSEP and WebSecLab. Similar to the payloads of Arachni, these scanners all report payloads as successful which appear in the HTML despite failing to escape their contexts to become executable.

Link has poor performance in SCARF and WackoPicko due a failure in finding injection points. More interestingly, in WAVSEP and WebSecLab Link struggles to generate successful payloads from failures to bypass sanitisation, and failures to escape contexts. The inability to escape contexts is further shown in Firing Range, which contains more XSS without sanitisation.

Firing Range is the only case where another scanner (ZAP) finds more TPs than HAXSS. This is because ZAP can generate payloads that trigger DOM based XSS in ways beyond the capabilities of HAXSS. However, HAXSS is still able to generate a high number of XSS despite the crawler identifying fewer source-sink combinations than other scanners, obtaining a better ratio.

w3af does very poorly in all instances reporting high numbers of FPs and FNs, due to its reliance on static analysis to detect success, and lack of bypassing sanitisation.

HAXSS reports the highest number of TPs in all webapps, except Firing Range. Indeed HAXSS is consistent, reporting the highest total number of XSS at 121, 20% more than the closest scanner Arachni (100). HAXSS and Black Widow are the only scanners with 0 FPs. HAXSS is able to do this due to the combination of static and dynamic analysis used to detect the `alert` and `token` by the Injection Interface component that dictates the reward. HAXSS ranks joint third and fourth for the number of FNs in WebSecLab and WAVSEP respectively. This performance is attributed to the actions required to trigger the vulnerabilities. Such examples include: double encoding, introducing additional variables to close contexts, and commenting out the code after the payload. While considerations could be made to add in such features, we consider this to be part of the future work as discussed in Section 7.3. We argue the performance of HAXSS, is currently sufficient to show that RL is capable of finding XSS within its narrow action space. Furthermore, the hierarchy of HAXSS again outperforms both its non-hierarchical (AXSS) and random equivalents (Rand-HAXSS). AXSS performs consistently, but poorly, in most webapps

Table 3.7: Performance of different Deep RL algorithms on the XP Train Bed. Ticks (✓) represents where the algorithm has met the win criteria and thus ‘won’ the example.

Algorithm	Average Requests Until First Successful Payload	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Total ‘Won’
DQN	203.13				✓				✓		✓			✓		✓	5
Duelling DQN	207.14				✓				✓		✓			✓			4
PPO	353.57	✓			✓				✓		✓			✓			5
DQN+PER	178.13	✓			✓				✓		✓			✓		✓	6
DQN+PER +RND	171.42	✓			✓		✓		✓		✓			✓		✓	7
HAXSS	103.03	✓		✓	✓		✓		✓		✓			✓	✓	✓	9

as it has difficulty in identifying complex payloads to bypass sanitisation or escape the context. In some cases, such as in SCARF, it fails to identify even basic payloads.

3.5.4 Ablation Study

We have seen how HAXSS outperforms the current state of the art XSS vulnerability scanners, and other RL based scanners, including Link, AXSS and Rand-HAXSS. However, there are many design choices that have gone into HAXSS to achieve this performance. To demonstrate the contribution of these to HAXSS we perform an ablation study of the different RL components used. Importantly, we aim to evaluate the performance of different deep RL algorithms for conducting web application security testing. We train each different implementation on the XP Train Bed, reporting the average number of requests taken to find the first successful payload and number of fields that meet the success criteria in Table 3.7.

To begin with, we train three RL algorithms with no additions made to them: DQN, Duelling DQN, and Proximal Policy Optimisation (PPO). Of these, Duelling DQN performs the worst in terms of number of fields that meets the win criteria for at 4. This is improved by both DQN, and PPO which win an additional Field. Interestingly, all three algorithms ‘win’ on Fields 4, 8, 10, and 13, which is common to all algorithms in Table 3.7. When it comes to the number of requests until the first successful payload both DQN and Duelling DQN perform similarly, with DQN reducing requests by 1.93%. However, PPO requires significantly more requests on average to find its first payload to display XSS. This is in part due to its inability to generalise

Listing 3.3: Payloads generated by HAXSS

```

❶ %22%3C%3CScRiPt%20SRC%3Dhttp%3A//localhost%3A666/xss.js
%3F%3Dtoken%20onLoad%3Dalert%28token%29%3E%3C/sCrIpT%3E
❷ </tag><<body onkeypress=alert(token)></body>
❸ '></body><body onload=alert(token)></body>
❹ </script><style onload=alert(token)></style>
❺ %3C/tag%3E%3Cbody%20onLoad%3Dalert%28token%29%3E%3C/body%3E
❻ %3C/ScRiPt%3E%3Cbody%20onKeyPress%3Dalert%28token%29%3E%20sRc
%3DhTtP%3A//LoCaLhOsT%3A666/XsS.Js%3F%3Dtoken%3C/body%3E
❼ "\u0022\u003c\u0069\u006d\u0067\u0020\u0053\u0052\u0043\u00%3d
\u0068\u0074\u0074\u0070\u003a\u002f\u002f\u006c\u006f\u0063\u0061
\u006c\u0068\u006f\u0073\u0074\u003a\u0036\u0036\u0036\u002f\u0078
\u0073\u0073\u002e\u006a\u0073\u003f\u003d\u0074\u006f\u006b\u0065
\u006e\u0020\u006f\u006e\u0065\u0072\u0072\u006f\u0072\u003d\u0061
\u006c\u0065\u0072\u0074\u0028\u0074\u006f\u006b\u0065\u006e\u0029
\u0020\u003e\u0060\u003c\u002f\u0069\u006d\u0067\u003e"
❽ onload=alert(token)
❾ </textarea><script href=http://localhost:666/xss.js?=token> alert(token)
</script>
❿ %3Cstyle%20onLoad%3Dalert%28token%29%3E%3C/style%3E
⓫ %0d%0a onload=alert(token)
⓬ \"<style onerror=alert(token) src=http://localhost:666/xss.js?=token
></style>
⓭ "<a onerror=alert(token) src=http://localhost:666/xss.js?=token></a>}"

```

to new fields, which causes it not find any successful payloads for Field 2. From these three algorithms we can see that DQN performs the best in for generalising to different fields.

We see that for each addition we make to the DQN architecture we are able to improve on the number of forms successfully meeting the win criteria. Similarly, each improvement boosts the overall learning and the ability to adapt to new fields, as indicated by the reduction in the average number of requests to find the first successful payload.

3.5.5 Payload Analysis

HAXSS found more vulnerabilities than the other scanners in the XP Test Bed and 4 of the 5 webapps in the macro-benchmark. The route cause of its performance is the payloads it is able to generate. We present a selection of the different payloads that trigger XSS in Listing 3.3. We further present the sanitisation rules that HAXSS can bypass in Table 3.8.

HAXSS is able to bypass the most sanitisations in Firing Range, (see Table 3.8), achieved by

❶, ❷, ❸, and ❹.

Table 3.8: Sanitisations HAXSS is able to bypass

Webapp	Rule
WebSecLab	Remove matches to the regex of <code>' "(?i)<script[>]*> </script></code> (originally marked as secure by designers)
WebSecLab	Unicode sequences (e.g. <code>\u0022</code>) are unescaped
WAVSEP	Strip input of <code>script</code> tags
WAVSEP	HTML entity encode: <code>< ></code>
WAVSEP	HTML entity encode: <code>< > ' "</code>
Firing Range	HTML entity encode: <code>< & ()</code>
Firing Range	Block a payload including <code>'</code> or <code>"</code>
Firing Range	Block a payload including lowercase <code>script</code>
Firing Range	Block a payload including any <code>script</code>

❶ bypasses sanitisation of `script` tags in either upper or lower case. HAXSS does this by encoding non-alphanumeric characters so they do not match the regex of a plaintext `<script>` tag. ❷ avoids sanitisation of `'` or `"` by creating a payload that avoids introducing these characters. Similarly, ❸ avoids sanitisation of lowercase `script` tags by creating a payload that is based on a `body` tag. ❹ bypasses sanitisation of `< & ( )` by forcibly changing the context, sanitisation occurs in the client side in the context of a `script` tag. As the browser accepts this as legitimate it allows the rest of the payload to bypass sanitisation.

Payload ❺ is URL encoded, yet does not to bypass sanitisation. Appearing in the context of a tagname within a `body` tag (`<body><[INPUT]></body>`), HAXSS detects sanitisation occurs as the browser only allows one `body` tag. Thus when HAXSS injects `<body>onload=alert(token)</body>` it is returned in the sink as `<body onload=alert(token)></body>`, meaning ❺ still leads to XSS.

HAXSS cannot bypass two sanitisations in Firing Range: `htmlentities` and URL encoding of `<`, `'`, and `"`. Though this is actually due to the contexts, which require payloads with a different structure than HAXSS can make (e.g. not containing these characters).

Arachni and Wapiti are able to bypass case sensitivity and HTML encoding via specialised payloads using a custom function (Arachni), or an `svg` tag (Wapiti).

HAXSS bypasses two sanitisations in WebSecLab, as shown in Table 3.8, using payloads ❻, and ❼.

❻ bypasses sanitisation for `raw_ifp` which was considered a TN by the WebSecLab authors,

but HAXSS discovered that it is a TP. URL encoding allows HAXSS to bypass regex associated with `raw_ifp:'|"(?i)<script[>]*>|</script>`. This is impressive as a `script` tag must be present in the payload to change to a context where the payload can be executed, which is only done via this particular encoding. ❷ displays XSS via a more complex payload, an `img` tag with a `src` attribute that calls to the web page discussed in Section 3.4.1. This causes an error to trigger the event handler. HAXSS then uses UTF-8 encoding to bypass sanitisation of the non-alphanumeric characters. ❸ and ❹ demonstrate the diversity in payloads that HAXSS develops.

HAXSS fails to bypass two sanitisations in WebSecLab. One requires a payload URL encoded twice which HAXSS cannot do. XSSer and Wapiti are able to discover this vulnerability, but fail bypassing other sanitisations. HAXSS also fails escaping of `"` that itself is not properly escaped. Though this is due to additional sanitisation of all tags in the payload, and HAXSS being unable to comment out the remaining code in the JavaScript context.

Surprisingly, we see that XSSer and Wapiti are able to identify a vulnerability by URL encoding a payload twice, however they fail to bypass any of the other sanitisations including backslash escaping quote marks, and using a unicode encoded input. Failure to bypass sanitisation is a common theme with Arachni, ZAP, and w3af doing this in all cases in WebSecLab and are only able to change execution contexts in simpler cases of single quotes and tag attributes. We observe that AXSS is able to generate a payload that bypasses sanitisation by UTF-8 encoding a payload. It then fails to generate payloads that bypass any further sanitisation. In WAVSEP HAXSS develops XSS payloads despite the sanitisations shown in Table 3.8, using ❷, ❸, ❹, and ❺.

❷ bypasses sanitisation of `script` tags using a payload based around a `style` tag, similar to behaviour observed in Firing Range. ❸ escapes the JavaScript comment context using `%0d%0a`. It then uses the `onload` to trigger the vulnerability from within JavaScript, avoiding angle brackets that would be sanitised.

❹ is particularly impressive as HAXSS uses sanitisation to its advantage. The context is the `src` attribute of an `img` tag, with `htmlentity` encoding of angle brackets. HAXSS escaped the

`src` context and uses sanitisation to trigger an injected `onerror` attribute. HAXSS displays similar behaviour in 13, by using `htmlentity` encoding of angle brackets to trigger the `onerror` in the same context.

However, there is one vulnerability in WAVSEP which HAXSS fails to find, creating a FN, the proof-of-concept for which is: `'"><img src=a onerror=alert(token)>`. Despite generating payloads similar to this format in other cases (for Firing Range, WebSecLab, and WAVSEP), HAXSS fails to generate a valid payload in this instance. This is due to the intrinsic nondeterminism in RL that helps to find novel solutions, where exploration into new state-spaces may be sub optimal. However, HAXSS is able to generate payloads that escape sanitisation via unicode-encoding a payload, and escaping the backslash escaping of quote marks. AXSS struggles to generate payloads effectively bypassing one sanitisation by URL encoding the payload.

In the other scanners, we again see a similar performance: ZAP fails to bypass sanitisation (HTML encoding of angle brackets and quotemarks, removal of script tags), w3af does the same also failing to identify POST request vulnerabilities. Arachni, Black Widow, and Wapiti, bypass sanitisation of `script` tags by using a custom function, `img`, and `svg` tags respectively, but miss the other filters.

In some of the successful payloads generated by HAXSS there are artefacts present from failed payloads in the same episode. For example, 16 contains a `src=...` artifact between the `body` tags. This does not seem to negatively impact performance. In fact, the second new vulnerability found in Section 3.6.3 is found by a payload containing this malformation, which may explain why it was missed by human review. We speculate HAXSS learned to use this same pattern as a polyglot because of the incentive to work with a leaner action space, leading to a better exploration/exploitation trade off.

The payloads HAXSS develops also show that even the Escape Agent avoids sanitisations and develops well formed succinct payloads. We also note that HAXSS displays zero instances of FPs in all webapps, this is due to our mechanism that validates an `alert` function has been executed in the webapp; allowing the feedback for RL to conduct testing.

3.6 Case studies

We have shown that HAXSS is able to outperform the state-of-the-art in XSS payload generation finding more vulnerabilities in our Micro and Macro benchmark evaluations. However, it remains to be seen if HAXSS can find vulnerabilities in production grade webapps. We target vulnerable versions of webapps affected by XSS CVEs reported during 2021 in the NVD database. Among those, we select at the most recent 3 open source webapps that we were able to successfully install, challenging HAXSS to re-discover the CVEs.

HAXSS rediscovers 4 of the 5 reported CVEs and, remarkably, discovers 5 new vulnerabilities which we responsibly disclosed to the webapp developers, and obtained CVE numbers for each.

3.6.1 HTMLy (CVE-2021-42946, CVE-2021-42867)

HTMLy 2.8.1 is a PHP based blogging platform and Content Management System (CMS) containing an admin control panel where users can update content. HTMLy is vulnerable to CVE-2021-36703, as the `Blog title` parameter of the admin panel `/admin/config` can be used to trigger XSS. HAXSS is able to replicate this vulnerability by the payload shown in 14. This payload closes the tag context to insert a new tag with active content.

HAXSS also finds two new vulnerabilities on the same page following a similar pattern: CVE-2021-42946: the `Copyright` parameter is vulnerable by closing the `span` tag and opening a new `body` tag with 15. CVE-2021-42867: the `Description` parameter is vulnerable to 16 in the list that closes the current `body` tag and opens a new one, similar to the prior payloads. 16 also contains an artefact (the `src=http://localhost:666/xss.js?=token` part) of a previous, failed, injection attempt.

14 `</input><style onLoad=alert(token)></style>`

15 `"</span><body onmouseover=alert(token)></body>`

16 `</body><body onmouseover=alert(token)> src=http://local host:666/xss.js?=token</body>`

3.6.2 Pixelimity (CVE-2021-42866)

Pixelimity 1.0 is a PHP CMS, containing an admin panel where the user can alter the content of the webapp. Pixelimity is vulnerable to CVE-2021-29056 in the `/admin/setting.php` page. ①⑦ is generated by HAXSS to trigger this vulnerability closing the current context to then triggering an alert via an `onmouseover` event in a `body` tag. HAXSS also finds a new vulnerability (CVE-2021-42866) using ①⑧, a payload more sophisticated than the ones observed so far, which closes the tag attribute and the parent tag in order to inject the active `style` tag.

①⑦ `'</footer><body onmouseover=alert(token)> href=http://localhost:666/xss.js?=token</body>`

①⑧ `"></input><style onload=alert(token)></style>`

3.6.3 Chikitsa (CVE-2021-42868, CVE-2021-42869)

Chikitsa is a ‘Clinic/Hospital Management System’ that allows users to input patient data, arrange appointments, and make/check payments. In our evaluation we focus on the admin account of version 2.0.2. HAXSS is able to re-discover CVE-2021-38151 and CVE-2021-38149, with payloads ①⑨ and ②⑦. However, while Chikitsa is also vulnerable to CVE-2021-38152, the page containing it could not be found by either the HAXSS crawler or by manual review.

Nonetheless, HAXSS discovers that the `First Name` (CVE-2021-42868) and `Last Name` (CVE-2021-42869) parameters of the `/patient/insert` page are vulnerable to XSS. These two parameters lead to XSS on multiple pages: `/patient/{patient_report,visit_report,bill_detail_report}` and `/appointment/appointment_report`. Payload ②① follows the familiar pattern seen above, closing an existing tag and injecting a new one with an active event handler. Payload ②② instead inserts a `script` tag to directly inject JavaScript, showcasing some of the diversity in the policy learned by the Escape Agent of HAXSS. Note that `onerror=alert(token)` in this context is valid JavaScript code triggering a call to `alert` before assigning its result to the variable `onerror`, although in HTML context this is an exploit-via-event-handler pattern.

①⑨ `></label<style onload=alert(token) href= http://localhost:666/xss.js?=token"></style>`

20 `><body onerror=alert(token)>></body>`

21 `</script><style onload=alert(token)> </style>`

22 `></script><script> onerror=alert(token)</script>`

3.7 Discussion and Limitations

In this section we discuss HAXSS’ overall performance against the other scanners, and limitations of our approach.

HAXSS performance in Section 3.5 demonstrates its ability to bypass sanitisation, finding 131 XSS across the micro- and macro-benchmarks a 20% improvement over the closest scanner, Arachni. This is due to the use of HAXSS Sanitisation Agents, as exemplified in WebSecLab, where HAXSS is able to identify a non-vulnerable parameter (`raw_fp1`) as vulnerable by using URL encoding and capitalisation actions to bypass regex (❹, first row in Table 3.8). Further examples of sanitisations HAXSS bypasses are in Table 3.8.

We also see the key role of the hierarchical design of our model, as it outperforms its non-hierarchical equivalent, Axss, and the other RL based XSS payload generator, Link. Both models use a flat action space and a single agent, hence showing the importance of both bypassing sanitisation and using a hierarchical structure.

Interestingly, we see that even Rand-HAXSS is able to outperform Axss on the XP Test Bed, WebSecLab, and Firing Range. We hypothesise that this is because it is easier to create a successful payload in one game and then bypass sanitisation in another, than it is to do both in a single game.

We have also seen the importance of using RL, as HAXSS outperforms Rand-HAXSS in the micro- and macro-benchmarks. Specifically, the use of RL has allowed HAXSS to search over the large space of possible payloads *effectively*. This particular ability is important as the Escape Agent can generate 6,635,520 unique payloads, and for each of these the Sanitisation

Agents can produce 384 different obfuscations.

3.7.1 Limitations

Our implementation of HAXSS is an illustration of how RL can be used to outperform current state-of-the-art security testing tools, going on to find real world vulnerabilities. Developing a full-featured product goes beyond the scope of this Thesis. As part of this process, we are responsible for a number of limitations that could be addressed with an engineering effort.

When the crawling phase of a scanner fails to identify an input which leads to a vulnerability, the scanner will not be able to find the corresponding XSS, missing a true positive or a false negative. If the input was non-vulnerable instead, the scanner will miss a false positive or a true negative. In order to appreciate the effect of scanning capabilities on the results in Table 3.5, we include the “Not Tried” column, to indicate the number of inputs that are missed by the crawler. Interestingly, where crawlers identify the majority of inputs (Firing range, WAVSEP, and WebSecLab) HAXSS identifies more XSS, even when finding fewer inputs such as in WAVSEP.

We acknowledge the limitations of the HAXSS crawler which misses a large number of inputs on Firing Range. The crawler fails to discover injection points triggered by complex interactions or in reactive web pages which present new injection points under the same URL. Still, the crawler serves its purpose well, enabling HAXSS to find more XSS than the other tools in most cases, and even perform second best on Firing Range.

Similarly, we have only implemented a limited number of user behaviours on a visited page, so the crawler fails to discover injection points triggered by complex interactions. These scenarios strongly depend on the target webapp implementation, and were rarely found during our experiments of Chapters 3.5 and 3.6.

Our method for detecting when a vulnerability has occurred is not fool-proof. The XSS attempt is considered as successful when the payload-specific token is displayed by the alert function. This detection *could* lead to false positives though none were detected in our experiments. One

could imagine a page that detects a suspicious payload and raises an alert with a message reporting the suspicious payload, correctly sanitised. This would result in a false positive detection for HAXSS. This behaviour is highly hypothetical, and as stated was not observed.

A training set determines the performance of any machine learning model. We built the XP Train and Test Bed with the intent of containing didactic examples that contain a variety of contexts and sanitisations. HAXSS performance on the macro-benchmark and its ability to find vulnerabilities in production grade webapps demonstrated that the XP Train and Test Bed served its purpose, yet further examples could be easily added, and it may turn out to further increase the performance of our agents.

The main shortcoming of HAXSS is the limited action space which reduces the diversity of payloads that can be generated, increasing FNs. This is shown in its relatively low performance on Firing Range. At the moment HAXSS does not have the ability to generate the kind of DOM payloads that ZAP uses. Additional actions could be added to improve the ability of HAXSS to generate payloads specifically for DOM XSS. Yet, to do so would serve only to increase the number of DOM XSS that HAXSS can find in a web application; it would not serve to further the research goal of this Chapter: to determine if RL can perform well in a constrained task. In fact, we have seen in Section 3.5 that HAXSS is already capable of outperforming the state-of-the-art scanners. We further saw the real-world impact of HAXSS as it found 5 zero-day CVEs. As such we believe adding more actions to HAXSS is of little interest to this Thesis.

3.8 Related Work

3.8.1 XSS Payload Generation

Alhuzali *et al.* [5] and Bensalim *et al.* [27] use dynamic data analysis to determine input parameters that are potentially vulnerable to XSS. Mohammadi *et al.* [192, 193] extract injection points in white-box settings, then test these with XSS attack strings. Similar work is conducted by Gupta *et al.* [100]. Human-to-machine translation has also been used to develop XSS pay-

loads in grey-box settings by Frempong *et al.* [87]. While the Lekies *et al.* [155] use a custom browser for grey-box taint tracking to generate escape strings for DOM based XSS. A similar methodology is used by Bensalim *et al.* [27]. Khan [138] uses an encoder-decoder architecture based on transformers to generate XSS payloads. However, this uses a white box approach using both server and client side code to generate the payload.

Few of these works use black-box approaches for payload generation, using potentially exhaustive search based strategies [7, 133, 134]. Whereas HAXSS uses RL to optimise this process via exploration and exploitation to identify vulnerabilities.

3.8.2 RL for XSS Detection

Caturano *et al.* [39] use tabular Q-learning agents for different XSS contexts. The authors compare against a number of different scanners, including 5 of those we considered in Section 3.5. This model reports lower TPs than HAXSS in WebSecLab (the only webapp used for testing). The modelling of the problem also differs from HAXSS as it follows a predetermined payload structure, using a ‘human-in-the-loop’ to inject reflected XSS into a webapp.

Hasegawa *et al.* [112] remove the human element of this testing paradigm and introduce a Deep RL agent using a DQN and curiosity module. However, this approach was only shown to reduce the number of requests to find the first vulnerability, not compared in terms of ability to find vulnerabilities. The model could be viewed as analogous to the simpler models evaluated against in the ablation study of Section 3.5.4.

The Link model [153] uses the RL algorithm of PPO to generate XSS payloads using a flat action space. As we demonstrate in Section 3.5, the hierarchical nature of HAXSS outperforms this model.

A grey-box approach developed by Song *et al.* [265] showed how static analysis can be combined to improve the detection capability of a Deep RL agent. In an ablation style evaluation the authors show that Q-learning based methods outperform policy gradient methods for XSS vulnerability detection, confirming the results from Section 3.5.4.

Another example of RL for XSS is seen in by Fang *et al.* [76], where an agent based on DQN forms basic payloads with some obfuscation techniques. This work doesn't generate diverse sets of payloads, but is used for adversarial examples to improve a ML based XSS detector used as a WAF. This results in a lack of application to real webapps, instead testing against two instances of software for detecting XSS. A similar approach to generate payloads for bypassing WAFs via Deep RL is developed by Yao *et al.* [311]. Conversely, HAXSS is able to bypass sanitisation, produce diverse payloads, and is tested against a number of vulnerability scanners on a collection of test bench webapps.

3.9 Takeaways

This Chapter focused on exploring if RL is a suitable paradigm for web application security testing by constraining the more complex problem of generating XSS payloads. We have argued that existing approaches are insufficient for two core reasons: 1) use of a small set of known XSS payloads, missing the potential vulnerabilities that require targeted exploitation and 2) reporting a high number of false positives in their scanning results.

We have then assessed several key challenges of identifying XSS, including: escaping contexts and bypassing sanitisation. We then considered the challenges of using RL for XSS payload generation, including: finding source-sink combinations, determining when XSS occurs legitimately, escaping the intended context of the page, and bypassing sanitisation. As a result we have implemented a hierarchical RL architecture, HAXSS, where different agents specialise on the tasks of escaping contexts with syntactically correct payloads and evading sanitisation. By reducing the search space for a single agent we have seen how, given the constrained games provided, the separate agents can perform better than a single agent could.

We have compared HAXSS to a number of other vulnerability scanners, finding 20% more XSS vulnerabilities across our micro- and macro-benchmarks, while maintaining zero false positives. On analysis of the payloads that HAXSS is able to generate we have seen that RL is capable of generating diverse payloads that bypass multiple different sanitisation rules, even one that was

supposed to be non-vulnerable. HAXSS was able to do this by performing effective obfuscations and tailoring how it is able to escape from intended contexts. While fundamentally a simple approach, executing this is often challenging, as we have seen from the existing state-of-the-art for XSS payload generators. We then went on to test HAXSS “in the wild”, identifying 4 known and 5 new vulnerabilities in 3 production-grade webapps.

The ability of HAXSS to find the quantity of vulnerabilities in a reduced number of requests demonstrates how RL can efficiently conduct web application security testing. As a result it also opens areas for future direction on how to improve the capability of XSS payload generation via RL as outlined in Section 3.7.1. It also suggests the need for further study in the application of RL to other tasks in web application security testing, as we will go on to do in further Chapters.

Chapter 4

A Singular Agent for Generating SQL Injection Payloads

In the previous Chapter we tested the applicability of RL for web application security testing by generating XSS payloads. We developed HAXSS, a hierarchy of agents that, combined, is capable of finding real world vulnerabilities. This hierarchical structure reduces the scope of each agent to focus on specific aspects of payload generation, namely escaping the context, and bypassing sanitisation. This design intentionally compartmentalises the overall strategy, but it does not fully explore how a single agent may behave when tackling the payload generation problem. Therefore, in this Chapter we will explore the suitability of applying a single agent to generate payloads.

Building on the lessons learned from the previous chapter, we will consider SQL injection (SQLi) as the conduit for learning. Such a vulnerability still provides a legitimate threat to user data and manipulation. SQLi is another vulnerability with a substantial body of research [6, 7, 11, 30, 58, 71, 106, 133, 134, 139, 154, 173, 205, 301, 320], which is often considered one of the most common attacks against web applications by recent surveys [130] and the OWASP Top 10 [212, 214]. In fact, SQLi was the most common vulnerability in 2023 according to a recent report it accounted for 19.47% of all web vulnerabilities, 85% more than any other vulnerability [69]. Again, we observe that due to the use of non-diverse payloads that

are not tailored to specific web applications, scanners miss legitimate vulnerabilities and use a high number of requests in doing so [179, 196, 318].

Vulnerability scanners are commonly used to find SQLi vulnerabilities before they are placed in production. In the case of SQLi they sometimes rely upon incorrect handling of errors and feedback from SQL queries on a web page, leading to reduced effectiveness when these are not present. Similar to XSS, scanners such as Wapiti [273], use a small set of payloads from a payload list, with minimal mutation or obfuscation techniques. Others, such as Sqlmap [56], use simple rule-based approaches to cover the most common SQLi payloads. By using payloads which lack diversity, and due to their inability to tailor payloads to specific web applications, scanners miss legitimate vulnerabilities and use a high number of requests in doing so [179, 196, 318].

In the previous Chapter we also saw that off-the-shelf Deep RL models, such as in Link, and AXSS, suffer from a number of limitations, leading to a reduction in performance compared to sophisticated RL architectures. There are two main reasons why we argue there are limitations in previous RL methods for generating injection payloads. First, is the use of fixed, manually curated features that struggle to handle the variable length and complexity of injection payloads [8, 71, 153]. Second, is bootstrapping from manual observations of payloads leads to constrained action spaces. Such biases in design limit the learnable behaviour of a model to within constrained payload definitions [71, 153].

To address these limitations we leverage a grey-box testing approach to ensure the validity of SQLi vulnerabilities. Using this basis we develop the first fully automated SQLi web scanner that uses RL, called SQIRL. Our approach uses a single agent to learn how to fix SQL syntax, escape contexts, and bypass sanitisation. SQIRL uses a bespoke architecture that learns the subtleties of SQL syntax and SQLi payloads to generate payloads in a changing action space, allowing for complete control over the generated payload. SQIRL uses worker agents to improve performance, reducing the number of HTTP requests needed compared to a single agent, and is able to generalise to new, unseen inputs to tailor SQLi payloads.

We go on to show that SQIRL finds more SQLi vulnerabilities in our novel *SQLi MicroBench-*

mark (SMB) and 14 production grade web applications, achieving this in a minimal number of requests compared to the state-of-the-art scanners we compare against. Further payload analysis also demonstrates that SQIRL uses more features, to generate more payload types that are out of distribution for existing payload lists used in scanners.

Chapter Contributions.

This work has been published at the 32nd USENIX Security Symposium 2023. It originated as a graduate thesis project which I co-supervised with Dr Sergio Maffeis, I developed the research direction for the project, including the analysis in Section 4.2. I created the SMB used for training and evaluation of SQIRL. I designed, performed, and created evaluation metrics for all experiments included in Chapters 4.5 and 4.6, and additionally I wrote the majority of the published paper.

4.1 Overview

We develop a novel approach to the automatic discovery of SQLi vulnerabilities which leverages RL to generate payloads targeted to each injection point, specific context, and defences. A single RL agent completes three tasks of SQLi: syntax fixing, context escape, and sanitisation bypass. Building on HAXSS we use worker agents to gather distributed knowledge to be centralised by the core agent in our model, SQIRL.

We design a new RL environment for SQLi, separating the complexities of crawling and interacting with a web application and potentially multiple databases. The environment implements actions for different Database Management Systems (DBMS), so agents are target-agnostic. As part of this environment we leverage database logs, without requiring access to application source code. Such a grey-box approach makes it possible to create an oracle that avoids false positives of SQLi, and effectively identify the application inputs that occur in SQL queries. It encourages best programming practices, and is a useful tool for defenders (which can access logs) and it does not enable uninvited attacks.

Table 4.1: **A:** SQLi payload (highlighted in pink) escaping its SQL context to cause the database to pause for 1 second.

B-D: SQL queries showing 13 semantically different positions where user input can occur.

E: current WordPress example of dangerous parameterised query pattern.

```

A: SELECT * FROM tab WHERE val = ' ' AND SLEEP(1) -- '
B: SELECT input_1 FROM input_2 WHERE input_3 = 'input_4' GROUP BY '
   input_5'
C: UPDATE input_6 SET ("input_7", "input_8")
D: INSERT INTO input_9 (input_10, input_11) SET ('input_12', 'input_13')
E: $query = $wpdb->prepare( "SELECT $id_column FROM $table WHERE meta_key
   = %s", $meta_key );

```

We demonstrate our approach is effective and practical by comparing it against a number of established web application scanners, both on a micro-benchmark, and on production grade web applications. We then compare the advantages of our design choices in a model ablation study. Our approach finds more vulnerabilities than other scanners, and does so using fewer requests. In particular, it discovers 22 new SQLi vulnerabilities, which we have responsibly disclosed, leading to 6 new CVEs. We analyse the payloads of the web scanners we use in our evaluation to determine their payload diversity. SQIRL achieves the greatest payload diversity and uses the most features in doing so.

4.2 Motivation

An injection vulnerability aims to trigger unintended functionality on a web application, typically violating a (possibly implicit) security policy. The particular functionality can vary from remote code execution to data exfiltration. In this Chapter, we focus on SQLi, which injects a payload, typically into the client side of a web application, aiming to cause the execution of an unintended SQL query on the web application database on the server. The simplest form of SQLi produces some feedback on the web page that triggered it, where a query result is returned. A harder to find, but more powerful variant, called *blind* SQLi, instead works even if there is no query feedback on the web page, for example when a web application server only issues generic responses to the client, such as ‘*Thanks for submitting your survey*’. In our research, we address both forms of injection.

Finding SQLi vulnerabilities generally follows the same workflow as for XSS. Beginning with the identification of sources that may be vulnerable (URLs, input tags, and dynamic link elements associated with a web page). These can then be tested for SQLi using payloads that meet 3 criteria:

1. Produce syntactically valid SQL, to avoid errors in execution. Doing so will ensure that the code is actually vulnerable as the vulnerable web application will execute valid SQL.
2. Bypass any blacklist-based filtering applied to the payload. Achieving this is key to displaying the SQLi vulnerability by ensuring the payload is able to run despite any of the possible defensive measures.
3. Change the intended behaviour of the SQL statement. By escaping the intended SQL context the payload and triggering additional functionality beyond the scope of the original statement we demonstrate the vulnerability. For example, assuming no filtering, the payload shown in red in Table 4.1 escapes the ' and executes a `SLEEP` statement. Table 4.1 shows three SQL statements (B-D) that could have 13 different input locations, and 10 different contexts (as inputs in parentheses share the same context). For each context, a different escape pattern is required to trigger a vulnerability.

While methods have been specifically designed to prevent SQLi, they are often ignored or misused [23, 255]. Indeed, this is reflected in the prevalence of SQLi, which was the most common web injection attack in 2020 (68.21%), accounting for over 4 billion attacks [4]. In 2023 this trend continued, with SQLi being the most common vulnerability across the *web*, 85% more common than any other vulnerability [69]. The safest way to prevent SQLi is to use parameterised queries to render user inputs benign. This practice has developers set the SQL code structure before including parameters in the query [255]. The query is defined and binds the parameter to a set type before being inserted into the query, rendering it benign by enforcing type preservation. Although most modern web applications use parameterised queries, these are sometimes used following insecure patterns. For example, current WordPress code [307] includes line E from Table 4.1, which protects its last parameter by coercing `$meta_key` to a

string, but could be vulnerable to SQLi if variable `$id_column` was previously injected with the payload `* FROM accounts --`.

Some modern web applications, and many legacy ones (which constitute the bulk of existing websites), attempt to prevent SQLi by filtering user input. As we have covered in Section 3.2, developers may take many different approaches including basic string manipulation, regex matching, and rule-based approaches. Thus, some instances of these inputs may be vulnerable to SQLi, but require specific, tailored payloads to be found. Indeed to find such vulnerabilities may take existing approaches a high volume of requests and a corresponding substantial execution time.

4.2.1 SQLi Payload Distribution

Work from Buyukkayhan *et al.* [33] showed the increasing complexity of XSS payloads that are required to trigger the vulnerabilities. Zhang *et al.* [318] have shown the same to be true of SQLi payloads, citing several characteristics of existing SQLi payloads. We focus on two observations about SQLi payloads that were made. The first is that existing payloads are sparse, which arises from input validation and sanitisation, as a result SQLi payloads that bypass them are more complex. Second, payloads tend to be similar in nature, having several variations of the same payload that will display the same, or similar, vulnerability. Therefore the distribution of payloads is uneven, leading to clusters of legitimate payloads.

To illustrate this we provide a Principal Component Analysis (PCA) dimensionality reduction of 1028 unique payloads gathered from open-source injection payload lists [217, 276, 278] and scanners (e.g. Wapiti). This is achieved by tokenising the payloads via Byte Pair Encoding [253], extracting 128 feature embeddings of each payload using Doc2Vec[151], and performing PCA to reduce the embeddings from 128 dimensions down to 3.

The three-dimensional PCA in Figure 4.1 demonstrates the uneven distribution of payloads present in the latent payload space, seen by the dense cluster and points that spread around it. Importantly, it further confirms that commonly used SQLi payloads, including those in

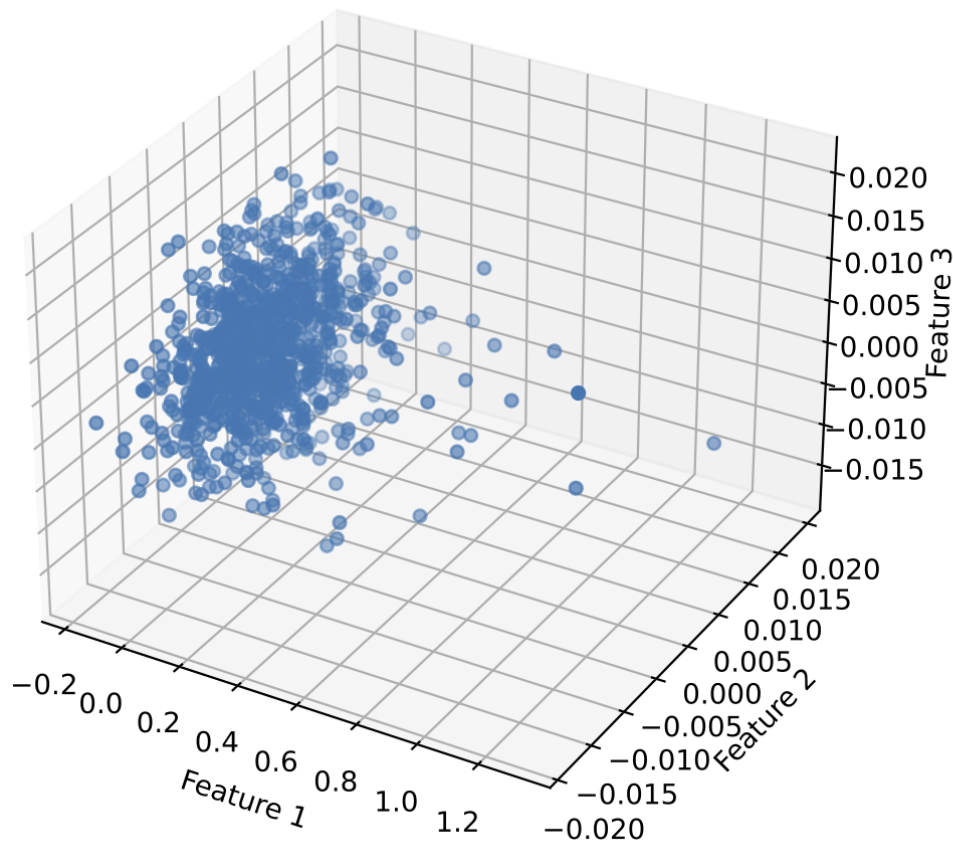


Figure 4.1: Principal Component Analysis of SQLi payloads.

scanners, share similar features in a dense distribution.

It is unlikely that all legitimate payloads are captured in this distribution, as there is a large space left unexplored. We hypothesise that in this space there are alternative payloads that lead to SQLi. An effective exploration of payloads would find these points.

Note however, that not all points in the latent space of Figure 4.1 correspond to legitimate SQLi payloads, as they must also conform to the syntax requirements of SQL. Yet, it is unlikely that all legitimate payloads have been found, an effective exploration of this space would find these points, finding alternative payload structures.

4.2.2 Existing Scanner Methods

While research has been conducted to detect SQLi using traditional fuzzing paradigms, popular static analysis methods such as constraint solvers are unable to solve the different type of constraints precisely, leaving behind a high number of false negatives and false positives [179, 196].

As we have seen in Section 3.2, vulnerability scanners are commonly used to find injection vulnerabilities, including XSS and SQLi. Indeed, they share many of the same limitations, which we briefly outline here. Scanners such as Wapiti, and Arachni [169] use a small number of simple payloads, and do not mutate them, offering no dynamic payload generation methods to bypass sanitisation. While Sqlmap [56] is seen as a “gold standard” for finding for SQLi, it doesn’t focus on generating diverse payloads, instead focusing on breadth of functionality, offering a number of different configurations such as type cast, false positive, and length constraint detection. Whilst payloads in Sqlmap are generated by a rule-based system or automaton, Sqlmap is not able to tailor generation to specific targets, limiting payload diversity. Furthermore, it only considers a small set of anti-sanitisation methods (e.g. for `addslashes` of PHP).

4.2.3 Challenges of Using RL for SQLi

We have seen in Chapter 3 that RL is able to search over the large space of HTML and JavaScript to find vulnerabilities. However, using RL to develop SQLi payloads comes with its own set of challenges, especially when we wish to give a single agent more agency over the payloads.

Given the sparsity and uneven distribution of SQLi payloads, it is difficult to develop automata or heuristics to adequately cover their diversity. An RL-based solution could tailor its strategy to individual test cases, thus efficiently generating diverse payloads. However, there are several challenges that must be overcome.

Interaction with the Web Application and Database

We develop our approach as a grey-box problem. This differs from the methodology of the previous Chapter which treated the web application in a black-box setting to find XSS. This is for good reason, as SQLi can manifest in different ways from XSS, as we have discussed, the blind SQLi variant does not even display through a web application, but requires side channel information. This approach limits the solution we could develop due to the lack of available feedback. It would be challenging to determine success or failure of the payload, let alone the possibility of progressive knowledge.

Yet, similar to what we have seen in the previous chapter, it is unclear how an RL agent should interact with a web application. However, this time we are searching for SQLi, and must additionally consider the database, and DBMS, used. It is difficult to design a general *environment* where an agent is able to interact with different web applications and DBMS. In prior work from Erdődi [71] this challenge was not fully explored, only using RL in contained environments and not on real web applications.

Thus, we will use the SQL logs from the DBMS to provide the level of progressive information to learn from. In this way we can remain black-box to the web application source code in the same way as in Chapter 3, *i.e.* we can't directly *see* the sanitisation code. Yet, we are grey-box to the server-side infrastructure, so we can observe the *effect* of any sanitisation. This is detailed further in Section 4.3.

Maximising Information Usage

Similarly to the previous Chapter the agent must have an understanding of the structure and syntax of the current payload so that it knows which mutations, or actions, to take. How the agent observes this payload will dictate how the agent performs. The question is how to provide this information in a meaningful and concise manner. In prior work, this has been captured by a set of manually defined features about the payloads themselves [71, 153]. While domain knowledge is often useful for RL, each set of bespoke features will impact the agent differently,

and may not contain enough meaningful information. For example, traditional features would struggle to relate the presence of an SQL keyword with the position where it occurs. Hence, we face a challenge of how to maximise the usage of information about the payload we are generating and the whole SQL query we are trying to exploit to improve the learning process.

In the previous Chapter we used a tokenised representation of the payload to avoid this. However, this still has limitations of a fixed length input, and may not capture all the information we may require of the payload. In this Chapter we will more fully explore the way in which we can best utilise alternative representations of such data.

Generating New and Diverse Payloads

The agents in previous work [71, 153], and indeed in the previous Chapter all use a fixed action space. The reasoning behind this is due to the difficulty in adapting to new actions. The use of neural networks in Deep RL also adds to this issue due to their (often) fixed length outputs [36, 83, 86, 118, 122, 153, 159, 160].

With a fixed action space, useful payload-generation tasks such as matching the number of open parentheses in an expression can only be done by relying on nondeterminism, which has a high performance cost. This approach was taken in Chapter 3, by providing an action that inserts a pre-existing escape string. In doing so this removes some of the control from the agent, abstracting the problem and hindering the amount the agent can do.

Using a dynamic action space, an agent can modify a payload at specific meaningful points. For example, consider the query snippet `WHERE (x=1 AND y='$input') LIMIT 1`. A typical first attempt at exploiting the query would be the payload `' OR 1=1 --`, which fails because of unmatched parenthesis. An agent with a dynamic action space could learn to insert `)` directly in the right place to prevent the error and obtain results from the query, using payload `') OR 1=1 --`.

4.3 An Environment for SQL Injection

Having motivated the problem and discussed the challenges around finding SQLi, we can now begin to design and detail the environment in which the agent can interact with the web application and DBMS. This environment encompasses all mechanisms that perform concrete changes to the payload based on agent actions, submitting payloads to the web application, returning observations and rewards to the agent. It also handles crawling the web application to find possible SQLi targets. Hooking into the SQL database we are able to provide feedback on possible targets, further allowing the agent to learn how to tailor SQLi payloads.

Similar to the previous Chapter we will reformulate the SQLi payload generation problem into separate RL games. However, this time the games will be played by a single agent, in an attempt to test the ability of an RL agent to find payloads when it has the complete generation strategy at its disposal.

These games are described below:

- **Context Escape** consists of escaping the intended context of the input in the SQL statement, similar to Chapter 3. The goal is to create a payload that executes as SQL as discussed in Section 4.2.3, thereby reducing false negatives. This is challenging as payloads must be syntactically correct conforming to the correct SQL syntax, and they must do this in the variety of different contexts in SQL. The key element of this game is to have agents implicitly learn and correct the SQL syntax. Examples of SQL contexts are shown in Examples B - D in Table 4.1.
- **Behaviour Change** consists of changing the behaviour of the SQL statement to cause unintended functionality. In SQLi this can be either a change in the output of the query, or a manipulation of the database or its data. Such behaviour raises concerns over data privacy, integrity, and availability, by manipulating data. To prevent any damage on the target, SQIRL demonstrates SQLi with a SLEEP statement as proof-of-concept.
- **Sanitisation Escape** is triggered when trying to solve the first two games. When a web

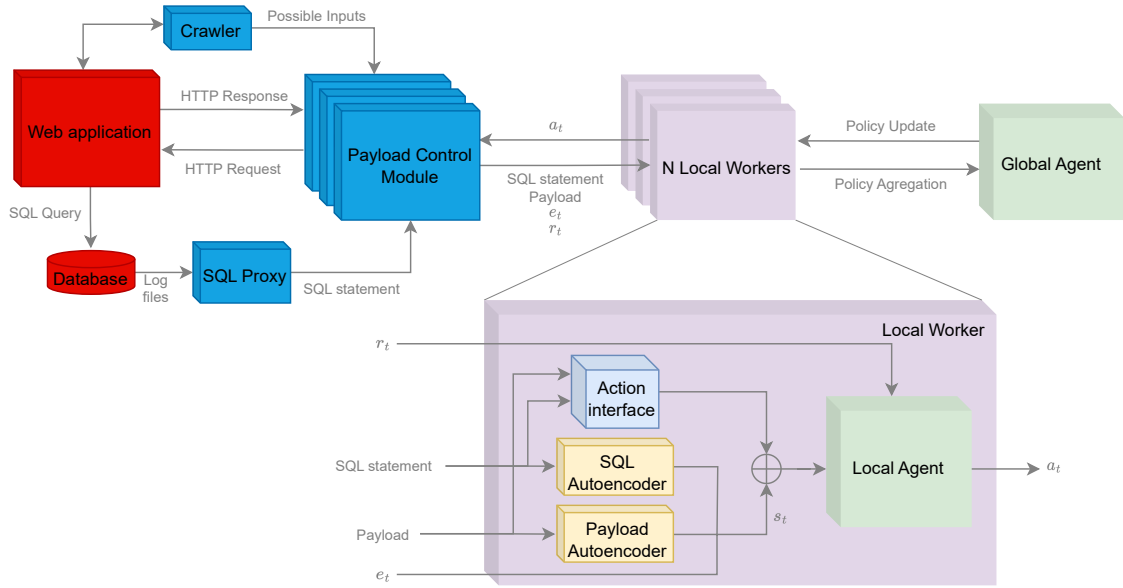


Figure 4.2: SQIRL architecture. RL agents (green) compute the mutation of the current payload. The N federated Local Workers (purple) train different agents to gather experiences for the global agent. Environment elements that interact with the web application (blue) then find input locations, submit payloads in requests, and monitor the database for successful injections.

application changes the payload and prevents it from being effective, SQIRL must attempt to escape sanitisation by replacing the affected statement with a semantic equivalent one, an approach that is limited in existing scanners (see Section 4.2.2).

We previously formed this as two separate games to be solved in a hierarchical fashion. Yet, here we have a single agent perform all games. This is for two reasons, first by having a single agent play these games it will learn a more complete, and combined strategy than separate agents. Second, having three games makes the interaction between the games more complex. Where each game can go to any of the other games, as shown in the transition function in Algorithm 4.1. This moves the SQLi games into the realm of multi-agent settings, making a hierarchical approach unsuitable.

Crawler

We start by designing the Crawler, which follows much of the same behaviour as defined before: recursively looking for anchor tags and URLs within the same domain, queuing them for exploration. On each web page, the Crawler finds two types of possible input: forms and

dynamic URLs. In each input, it injects a unique token that is searched for by the SQL Proxy component, to identify unique input-output combinations that correspond to interaction with the web application database.

As this work is focused on SQLi payload generation, we consider an exhaustive search of the input injection space beyond the scope of this work. Our crawler implementation is thus based on the Black Widow crawler [74]. We implement this as described by Eriksson *et al.*, but extend to increase the attack surface analysed by SQIRL. In particular we capture dynamic AJAX requests. These asynchronous requests can be initiated by the browser, but not result in a page transition. Using these requests means information can be sent to the server without interrupting the crawling and testing. We also integrate the crawler with the new SQL Proxy component, to detect the unique tokens injected in each input without relying on information leakage on a web page, extending our coverage to blind injections.

SQL Proxy

As we have previously stated, we want to maximise the information we that the agent can use. By doing so we aim to provide an RL agent with the means to develop its strategy, while reducing potential bias from domain knowledge [115]. As a result, we have introduced a grey-box model to give more accurate feedback on a payload. We treat the web application ecosystem as grey-box, but black-box to the underlying source code. This avoids the limitations of automated vulnerability detection methods based on deep learning or static analysis that lead to low true positives, and high rates of false negatives [111, 282].

We leverage this grey-box information in the form of an SQL Proxy, which acts between the Payload Control Module and the database. It retrieves an SQL statement from the database log files which contains the unique token present in the current payload, or reports if an error occurred. The SQL Proxy is used for the crawling and fuzzing phase of SQIRL. To reduce redundancy in the fuzzing effort, the SQL Proxy removes duplicate statements, and if multiple SQL statements exist containing the unique token, it selects the one most similar to the original SQL statement triggered during crawling.

Algorithm 4.1: The transition function used to compute the reward, termination condition, and next game.

```

1 Function transition( $e_t$ , payload, step, max_step):
2     done = False
3     step++
4     if  $e_t == 0$ :
5         if behaviourChanged(payload):
6             r = 0
7             done = True
8         elif sanitised(payload):
9             game = sanitisation_escape
10            r = -1
11        elif escapedContext(payload):
12            game = behaviour_change
13            r = -1
14        else:
15            game = context_escape
16            r = -1
17    else:
18        game = context_escape
19        r = -1
20    if step == max_step:
21        done = True
22    return r, done, game

```

State

A *state* at time t is made of three different components: the payload injected at $t - 1$; the SQL statement that has been extracted from the SQL Proxy at time $t - 1$ (in the case of an error and no new SQL statement, we use the one captured at $t - 2$); and a bit e to flag if there was a SQL error at the previous time step. To reduce stochasticity in learning, reduce the space of possible states, and encourage generalisation the payload and SQL statements are converted into a generic form: string and integer values are replaced by the generic tokens **STR** and **INT**, respectively.

In practice this generic form of the payload will be represented as tokens in which we can perform the concrete actions from the agent, as discussed in the Payload Control Module. While the payload is represented as tokens to perform actions, it will be converted to a fixed length vector by the SQIRL worker agents. This follows from the discussion in Section 4.2.3 so that we can maximise information usage, we will describe this process further in Section 4.4.1.

Extrinsic Reward

In order for SQIRL to have knowledge of how the payloads it is generating perform, we define a reward function that motivates finding SQLi vulnerabilities. Our handcrafted, or *extrinsic*, reward is designed to have the agent progress through the three games to reach a legitimate vulnerability. We base this on the standard sparse reward function used in RL, where agents are only rewarded for completing the final objective. However, we use negative rewards to incentivise the agent to find legitimate payloads as quickly as possible [275].

The transition function used to compute the extrinsic reward is shown in Algorithm 4.1. If no error occurs, we detect if the payload changed the SQL behaviour, triggering the successful termination condition and rewarding 0. For instance, if a **SLEEP** keyword is injected and is unsanitised outside of the intended context it would be a legitimate SQLi payload. Otherwise this provides a reward of -1 if the payload: 1) fails to escape the context, 2) is sanitised, 3) doesn't change the behaviour of the executed SQL, or 4) results in a SQL syntax error. The **behaviourChanged**, **sanitised** and **escapedContext** conditions in the transition function are implemented as syntactic checks and regular expressions.

Payload Control Module

In SQIRL the Local Workers, or workers, interact with the web application via the Payload Control Module (PCM). First, the PCM translates an action from a worker into a concrete payload, and injects it into the web application. Then, it computes the resulting state and the reward, and feeds them back to the relevant agent.

To generate concrete payloads, the PCM uses a vocabulary of tokens dependent on the current objective of SQIRL. Tokens can be found in Table 4.2. Tokens are curated using existing payloads found in the literature [11, 320] and a popular GitHub SQL payload list with 809 unique SQLi payloads, 5000 stars and 1,200 forks at time of writing [278].

We split actions into three categories corresponding to each game. During each game the actions associated with that game are provided to the agent. By doing this we reduce the number of

Table 4.2: Token space used for SQL payload and statements. SQIRL actions are shown in white, grey for SQL representation.

Action Type	Action	Example Tokens	Description
Basic Tokens (addition, removal)	Comma	,	
	Comment	#, -, convert # to -	Comments at any location
	Comment tokens	# TOKEN_TO_COMMENT	Comment out specific tokens
	Conditional	1=1, "wdd" = "wq"	A conditional statement
	Identifier (MD5 Hash)	351bf115b49fdc38b92e00482000e9cd	Identifier based on step counter and random salt
	Keyword	SELECT, OR	SQL Keywords
	Number	1.0,2	Number tokens as Float or Int
	Operator	>, =>, /, *	Logical Operands
	Paranthesis	), (	
	Quotes	', ", `	
	Statement	SELECT * FROM Users WHERE SLEEP(0)	Complex SQL statements
	String	"ed"	Abstract strings present in the SQL statement to a simpler form
Behaviour Changing Tokens (addition, removal)	Whitespace	()	A whitespace token
	Hex	0x12A	Hex characters
	AND	AND SLEEP(0)	SQL AND statement
	OR	OR SLEEP(0)	SQL OR statement
	IF	IF (1=1) THEN SLEEP(0) ELSE SLEEP(10)	SQL IF statement
	UNION	UNION SELECT * FROM Users WHERE SLEEP(0)	SQL UNION statement
Sanitisation Escape (token operation)	WHERE	WHERE SLEEP(0)	SQL WHERE statement
	SLEEP	SLEEP(0)	SQL SLEEP statement
	Keyword random capitalisation	SeLEct	Randomly capitalise letters in all selected Keyword
	Whitespace obfuscation	() becomes (**)	Convert all whitespace elements in the payload to comments
	AND obfuscaion	AND becomes &	convert all AND tokens to &
	Char token	CHAR(TOKEN)	Convert the selected token to its numeric ASCII representation and include it in a CHAR operation
	Concat token	CONCAT('TOKEN_PART_1', 'TOKEN_PART_2')	Split the selected token in two parts then include in a CONCAT operation

actions an agent can take, and by extent the state-action search space. Yet, this still allows a single agent to control the overall strategy across all games. These actions are categorised as:

- *Context Escape*: Add or remove *basic tokens* such as ', # , 1=1, SELECT from the payload.
- *Behaviour Change*: Add or remove *behaviour changing* tokens such as AND SLEEP(0), WHERE SLEEP(0), OR SLEEP(0) from the payload.
- *Sanitisation Escape*: Alter existing payload tokens, for example by keyword capitalisation (SeLEct), char token (CHAR(20)), and commenting out white space.

Each of the Context Escape and Behaviour Change tokens can be inserted at any point in the payload. Each of the tokens can also be removed at any point in the payload. This allows SQIRL to search the entire payload space for valid payloads: as tokens can be inserted at any point, there is an infinite number of payloads that SQIRL can create. In practice, SQIRL will learn to

search this space for patterns in the payloads, and tailor payloads to each injection point. The tokens present in Table 4.2 are also used to represent the payload and SQL statement for input into the autoencoders of SQIRL. Greyed tokens are used only for representation, and not for actions. One advantage of our token structure is that the concrete actions on the payload can be changed depending on the specific SQL syntax, allowing for even greater coverage between the different SQL dialects (e.g. PostgreSQL and MySQL) without having to retrain SQIRL.

After forming the concrete payload, the PCM sends it to the relevant input of the web application and receives the new SQL statement executed from the SQL Proxy. The PCM then uses the transition function in Algorithm 4.1 (described in Section 22) to determine the next game, the termination condition, and the extrinsic reward for the payload.

To generate a payload, SQIRL takes a series of actions or steps in an episode to mutate the payload until a vulnerability has been found, or the `max_steps` termination condition is reached. When this occurs, the PCM resets the payload to the empty token (' '), generates a new state, and sets the game to Context Escape.

4.4 The Sqirl Agent

We describe here the overall deep learning architecture of the SQIRL agents, both the Local Workers and the Global Agent. This agent has been designed to leverage complex SQL statements and payloads seen in training to pool diverse experience to improve learning, efficiency, and generalisation.

4.4.1 Local Worker

The Local Worker is the core of SQIRL, containing the logic to handle actions, states, and perform the main learning tasks. Local workers in SQIRL are functionally similar to those used in HAXSS, as they each interact with their own instance of the environment to gather diverse experiences to be used in a global buffer.

Table 4.3: Context Free Grammar used to generate MySQL statements. Blue indicates terminals.

```

1 root: (select_stmt | update_stmt)
2 select_stmt: ('SELECT', selectable, (where_cond | having_cond), (order_by
   | join_stmt), (connector | union_stmt), limit_stmt)
3 union_stmt: ('UNION', select_stmt | '')
4 update_stmt: ('UPDATE', selectable, 'SET', where_cond, join_stmt)
5 where_cond: ('WHERE', var, bool_op, var)
6 having_cond: ('HAVING', var, bool_op, var)
7 order_by: ('ORDER BY', var)
8 join_stmt: ('JOIN', join_op, var)
9 join_op: ('CROSS JOIN' | 'FULL OUTER JOIN' | 'HASH JOIN' | 'INNER JOIN' |
   'LEFT JOIN' | 'LEFT OUTER JOIN' | 'OUTER JOIN' | 'RIGHT JOIN' | '
   RIGHT OUTER JOIN')
10 connector: ((' AND', (select_stmt | update_stmt)) | (' OR', (select_stmt |
   update_stmt)) | (' XOR', (select_stmt | update_stmt)) | '')
11 selectable: '*' | 'COUNT(', var, ')' | 'ABS(', var, ')' | 'AVG(', var, ')' |
   'FLOOR(', var, ')' | 'MAX(', var, ')' | 'BIN(', var, ')' | 'MIN(', var,
   ')' | 'STD(', var, ')' | 'STDDEC(', var, ')' )
12 bool_op: ( '==' | '!=' | '<' | '>' | '<=' | '>=' | 'IN' | 'BETWEEN')
13 limit_stmt: ('LIMIT', INT | '')
14 var: (INT | STRING)

```

Representation Learning

In Section 4.2.3, we argued in favour of representing SQL statements and payloads in a way that captures the inter-dependencies between tokens. Since SQL statements and payloads have no fixed size, we use Gated Recurrent Unit (GRU) autoencoders, which are designed to represent variable length sequences and the semantic relationship between their elements. GRUs have been shown to outperform other recurrent neural networks (RNNs) in web testing tasks [245].

Tokenising and one-hot-encoding techniques also capture dependencies, but rely on fixed length inputs. Clipping payloads and SQL statements to a fixed length introduces noise or discards information valuable to the action-selection process, making such techniques unsuitable. By using GRU models we are able to learn a fixed length representation of variable length data.

We use separate autoencoders for representing SQL statements and payloads, as we expect them to learn different latent representations for the same tokens (for example, ‘’ will have more significance in the latter, and will not appear there in matched pairs).

Starting from the generic representation of the payload and SQL statement, we convert them into sequences of tokens from Table 4.2. Each autoencoder takes as input a token at a time. The first layer produces as output an 800-float vector for the SQL statement, and of 280-float

vector for the payload. These are then fed into the respective GRUs together with the output of the GRUs on the previous tokens of each sequence. The output of each GRU is a 1024-float vector. Using a reconstruction loss, we train the autoencoders in a self-supervised way to reconstruct their inputs (the SQL statements, or the payloads).

To train these models we generate a dataset of 100,000 samples for each autoencoder. Each payload is generated by taking 30 random actions. MySQL statements are generated using the grammar in Table 4.3. Through the process of training, the autoencoders learn to recreate the input data. When running SQIRL, the autoencoder outputs and the state error bit e_t are concatenated, yielding a state of 2049 features.

The use of autoencoders provides SQIRL with an automated feature set that represents the payload in the latent space, avoiding the pitfalls that RL agents can fall into from manual features, such as difficulty learning [88], learning an incorrect policy for the task [15, 316], or exploiting unintended mechanics [125, 316].

The features are then extracted from the autoencoders and input into a DQN. We saw in the previous Chapter that the DQN architecture outperformed other architectures as the off-policy nature of DQN is more data efficient than alternative RL algorithms such as A2C [189] or PPO [251], which discard data after each update, requiring longer training. Tabular Q-learning was considered, but it is infeasible to handle the large number of states resulting from our formulation of the SQLi game.

Action Ranking

In SQIRL, the number of possible actions increases with the payload length, as tokens can be added or removed from any location within the payload. For example, at time $t = 0$ there is only the possibility to add a token at one location, hence 27 possible actions. At $t = 1$, assuming the payload is not sanitised, there is the possibility to add tokens before or after the previous token, or remove the previous token, hence there are 55 possible actions.

The traditional DQN architecture computes a Q-value for each action as the output, selecting

the action with the highest value. Due to the dynamic action space of SQIRL, we are unable to use this same mechanism. Instead we alter the DQN architecture to compute a Q-value one action at a time. We do this by taking the available actions at the current timestep and converting these into a set of action representations, via the Action Interface in Figure 4.2. These representations consist of a 4-tuple that includes: the action number, indices of where the action would mutate the payload (e.g. placing # at index 4, or keyword capitalisation from index 4-9), and action class type (e.g. Comment or Operator).

We are thus able to compute Q-values for a variable number of actions and select the action with the highest associated Q-value as the mutation to perform on the payload. This allows SQIRL to take dynamic actions to generate payloads, going beyond the state-of-the-art practice of using a pre-existing formula, a grammar, or a set of payloads.

Intrinsic Reward

In the previous Chapter we used an intrinsic reward for its ability in making agents ‘curious’ in finding new states. Thus, we similarly use RND [32] to generate an *intrinsic* reward, which encourages the Local Workers to take different actions and find new states.

The RND is implemented by associating to each Local Worker a Target and a Predictor neural networks. A Local Worker feeds a state s_t to both, and over time the Predictor learns to predict the output of the Target. The intrinsic reward is computed as the loss between the outputs of the two networks, and is bound between 0 and 0.7 (see Table 4.5 for the hyperparameter search details.).

4.4.2 Global Agent

Once more we also use worker agents in a vertical federation to accelerate learning. The intuition is that we want to harness different experiences that agents gather from their independent interactions with the same target. To do so, we create a Global Agent containing a global policy, and several Local Workers with their local policies. Each of the workers is able to

Table 4.4: SQL statements and sanitisations used in the SMB where tasks increase in difficulty. Tasks 1-20 are used for training SQIRL. Tasks 1-30 constitute the positive class for the SMB benchmark.

Task	SQL Statement	Sanitisation
1	SELECT * FROM users WHERE name=INPUT	-
2	SELECT * FROM users WHERE name='INPUT'	-
3	UPDATE users SET pass ='pss' WHERE (name='INPUT')	-
4	INSERT INTO 'users' ('name','pass') VALUES ('INPUT','INPUT')	-
5	SELECT * FROM users WHERE name='INPUT' LIMIT 1	-
6	SELECT * FROM users WHERE name='INPUT' group by 'user'	-
7	SELECT * FROM users WHERE name="INPUT"	-
8	SELECT count(name) FROM users group by 'INPUT'	-
9	SELECT count(name) FROM users group by INPUT	-
10	SELECT count(name) FROM users group by ('INPUT')	-
11	SELECT * FROM users WHERE name=(INPUT)	-
12	SELECT * FROM users WHERE name='INPUT' group by ID	-
13	SELECT MIN(name) from users GROUP BY id HAVING id=INPUT	-
14	SELECT * FROM users WHERE id = INPUT	MySQLi real_escape_string
15	SELECT * FROM users WHERE id = INPUT	Filter capitalised and lowercase SQL keywords
16	SELECT * FROM users WHERE id = ('INPUT')	Filter capitalised and lowercase SQL keywords
17	SELECT * FROM users WHERE id = ('INPUT')	Filter out spaces
18	SELECT * FROM users WHERE id = ('INPUT')	Escape all AND keywords
19	SELECT * FROM users WHERE id LIKE 'INPUT'	Filter capitalised and lowercase SQL keywords
20	SELECT * FROM users WHERE name='INPUT1' OR name='INPUT2' OR name='INPUT3' LIMIT 0, 1	MySQLi real_escape_string on input1 and input3
21	UPDATE users SET name='name' WHERE id = INPUT	CVE-2020-8637
22	UPDATE users SET name='INPUT1' WHERE id = INPUT2	CVE-2020-8638
23	SELECT * FROM users WHERE name LIKE 'INPUT'	CVE-2023-30605
24	SELECT * FROM users WHERE name=INPUT	CVE-2023-24812
25	SELECT * FROM users WHERE (id=INPUT1 AND name='INPUT2')	CVE-2020-8841
26	SELECT * FROM users WHERE id='INPUT' LIMIT 0, 1	Remove all inputs containing AND
27	SELECT MIN(name) from users GROUP BY id HAVING id=('INPUT')	Filter capitalised and lowercase SQL keywords
28	INSERT INTO 'users' ('name','pass') VALUES ('INPUT','INPUT')	Filter out spaces
29	SELECT * FROM users WHERE id LIKE "%INPUT%" LIMIT 0, 1	Escape all AND keywords
30	SELECT * FROM users WHERE id LIKE ("%INPUT%") LIMIT 0, 1	Escape all AND keywords

interact with its own instance of the PCM, keeping the objective dynamics separate as the agents execute separate payloads in parallel. This knowledge is then aggregated every 20 episodes by averaging the neural network weights of the workers into the global policy which in turn updates the workers individual weights.

4.4.3 Training

As a source of examples to train SQIRL, and as a basis for comparison with other scanners, we create a novel SMB. The 30 vulnerable samples cover a didactic and realistic range of different SQLi contexts and sanitisations, and are drawn from CVEs and established sources [49, 216, 232]. For each vulnerable sample, we also add to the benchmark a non-vulnerable version of the same injection point, where we parameterise the underlying query and sanitise each parameter. We implement the SMB as an easy-to-crawl PHP web application which reflects the user input in the response web page, so that a crawler can find the test input it submitted.

We used the 20 easiest vulnerable samples to train SQIRL, and we left the harder 10 as a test

Table 4.5: Hyperparameters used to train the components of SQIRL. We performed a grid search on these values to optimise each model separately.

Model	Value	Min	Max	Step	Range	Selected
DQN	ϵ decay	0.999	0.9999	0.0001	-	0.9999
	ϵ minimum	0.1	0.5	0.1	-	0.2
	ϵ start	0.9	0.5	0.1	-	0.7
	Intrinsic reward limit	-	-	-	[0.1,0.3,0.7, 0.9]	0.7
	Batch size	124	2,048	$\times 2$	-	512
	learning rate	10^{-4}	60^{-4}	10^{-4}	-	50^{-4}
	Q-policy update (episodes)	-	-	-	[1, 5, 10, 20, 40]	20
	Network Architecture	-	-	-	(1024-512), (2048-1024-512), (4096-2048-1024-512)	(2048-1024-512)
Autoencoder	learning rate	10^{-4}	90^{-4}	10^{-4}	-	10^{-4}
	Fixed feature size	-	-	-	[10,000,1024,5012]	1,024
	Batch size	124	2048	$\times 2$	-	512
Federated RL agent	Client Aggregation Rate	-	-	-	[10,20,40,80]	20

set to demonstrate its generalisation ability to novel examples not seen during training. During testing, we include also the 30 non-vulnerable samples to estimate false positives. The training set contains multiple sanitisations and a variety of different contexts, and SQIRL can learn the value of all actions in its action space. See Table 4.4 for the SQLi statements and sanitisations.

We conduct training and experiments on a 64GB Intel i7 CPU. For each sample, we train until SQIRL is able to produce 14 valid payloads in the last 20 episodes, or until the maximum of 200 episodes is reached. Each episode consists of up to 30 steps (each corresponding to a fuzzing request). Random actions are taken according to an ϵ -decay policy to ensure exploration of the payload space. This is to balance the policy of an agent to produce valid payloads, and the stochasticity of ϵ taking random actions. We use 4 worker agents for training. A grid-search was conducted on the individual components of the workers to select hyperparameter values, details of which can be found in Table 4.5.

4.5 Evaluation

In our evaluation we aim to improve on the methodology of the previous Chapter, evaluating SQIRL extensively. Our evaluation compares SQIRL to 5 state-of-the-art scanners on the SMB of Section 4.4.3 and on 14 production-grade web applications. The primary performance measure is the ability to find SQLi vulnerabilities (true positives). The high volume of traffic generated by automated web fuzzing tools is considered one of their main shortcomings, alerting defenders that an attack is taking place. Moreover, if a scanner needs fewer requests than another to exploit the same injection point, it can be considered more ‘intelligent’ [39]. Hence, our secondary performance measure is the number of fuzzing requests sent (excluding crawling requests). Finally, we also report average run time, although broadly speaking the total time, running in minutes at most, is not a significant concern for vulnerability scanning.

4.5.1 Experimental Setup

We compare SQIRL against a number of state-of-the-art scanners. We select Sqlmap v1.6, an open source SQLi-specific scanner with 24.6k stars on GitHub, also included in the Kali Linux distribution popular with penetration testers. We further select 4 general security scanners with SQLi functionality: OWASP ZAP v2.11.1, BurpSuite Pro v2022.6.1, Arachni v1.6.1.3, and Wapiti 3.1.2. Whilst we would have liked to compare with existing academic scanners for black-box detection of SQLi, the implementations of these are not available or incomplete [6, 164, 173, 245, 301]. Comparing with a hierarchical architecture of games and agents, similar to Chapter 3, is challenging as the three games can transition to any other depending on the criteria triggered, with no clear hierarchy. Such a problem formulation would create a *multi-agent* problem rather than hierarchical. Furthermore, this Thesis aims to show different ways in which RL can be used to solve key problems in web application security testing. Thus, implementing a hierarchical approach to the SQIRL architecture is more of an engineering overhead adding minimal insight to that from Chapter 3.

During testing, we modify SQIRL by halving the learning rate, and reducing the rate to take

Table 4.6: Comparison of SQIRL against state of the art scanners on the SMB. Dark solid bars (red) emphasise poor performance, light bars (green) emphasise good performance. *Spurious Positives* are TPs deemed invalid after manual analysis.

Tool	Feedback	Exception	Avg Requests per Vuln Input	Avg Requests per Non-Vuln Input	Average Run Time(s)	Spurious Positives	FP	TN	FN	TP	
ZAP	built-in	✓	-	86.9	98.7	1.2	10	21	9	7	23
	advanced	✓	-	7760.1	8208.4	61.1	0	0	30	9	21
	built-in	✗	-	99.8	100.6	1.4	22	22	8	8	22
	advanced	✗	-	8031.9	8208.4	66.2	0	0	30	9	21
Sqlmap	✓	-	2234.5	4524.5	148	0	0	30	13	17	
	✗	-	2212.5	4533.2	145	0	0	30	13	17	
BurpSuite	✓	-	220.5	234.3	47.1	0	0	30	8	22	
	✗	-	279.5	279.0	46.9	0	0	30	13	17	
Arachni	-	✓	117.7	121.5	17.8	0	0	30	0	30	
	-	✗	121.3	121.6	17.2	0	0	30	30	0	
Wapiti	-	✓	23.8	35.0	1.0	0	0	30	9	21	
	-	✗	35.1	34.6	1.9	0	0	30	30	0	
Rand-SQIRL	-	-	92.9	300.0	5.7	0	0	30	8	22	
SQIRL	-	-	24.8	300.0	65.5	0	0	30	0	30	

random actions to 0.1, using the same decay. This places greater emphasis on the learnt policy but still allows for exploration of new states. We allow SQIRL a maximum of 10 episodes per parameter tested, relying on its ability to discover payloads quickly, and limiting unnecessary attempts for non-vulnerable inputs.

As an additional fuzzing baseline, we use a random version of SQIRL, called Rand-SQIRL. This version uses the same action space, and constrained games as Sections 4.3 and 4.4. As in Chapter 3, this provides an important point of comparison to understand to what extent the RL component of SQIRL can learn to effectively use the action space.

Finally, we run each of the scanners for a maximum of 3 hours to ensure consistency in the scanning. We report scanner configurations in Table B.1 (Appendix B).

4.5.2 Comparison on SQLi MicroBenchmark

In this experiment we investigate the performance of SQIRL and the state-of-the-art scanners of Section 4.5.1 on the SMB of Section 4.4.3, consisting of 30 different examples of SQLi (*positives*),

which use a variety of different contexts and sanitisations, and their 30 fixed variants (*negatives*).

Sqlmap, ZAP, and BurpSuite can benefit from receiving the SQL query result in the web page as feedback. Arachni and Wapiti instead can benefit from seeing on the web page the exception trace caused by malformed queries. We test each of these scanners twice: in the default configuration of the benchmark (reflect input only), and with the additional beneficial output (query results or exception trace, as needed). ZAP can use its built-in SQLi plugin, or use an advanced plugin to improve performance: we test it in both configurations. Table 4.6 shows the aggregate results of this experiment. The key performance metric is the number of vulnerabilities found (TP). SQIRL and Arachni are the only scanners able to find all the 30 vulnerabilities. BurpSuite, Rand-SQIRL (22), ZAP-advanced and Wapiti (21) find two-thirds of the vulnerabilities, and Sqlmap (17) finds half.

Wapiti and Arachni (0) are ineffective when the exception trace is not shown on the page, indicating that their performance on production web applications may suffer. Sqlmap does not appear to leverage the feedback, as it finds 17 injections irrespective of feedback. Rand-SQIRL and SQIRL need neither feedback nor exceptions, as they leverage the SQL Proxy information.

ZAP-built-in suffers from a high percentage of FPs (two thirds). Moreover, upon manual inspection, it turns out that most of its TPs (10 with feedback, 22 without) are *spurious*. This is due to tests which, on certain inputs, fail to verify the presence of SQLi, yet lead ZAP to report SQLi. A concrete example is a test that compares the page output length after injecting two payloads, expecting that the first would cause the query to return some results (ZAP ' OR '1'='1' --) and that the second would not (ZAP ' AND '1'='2' --). The injection fails (say single quotes are sanitised), but ZAP thinks it succeeds, because it mistakenly detect a difference in page size due to the reflected (ineffective) payloads.

The average number of requests per vulnerable input measures how ‘intelligent’ a scanner is, in terms of using effective payloads instead of brute-force. Wapiti and SQIRL, with less than 25 requests, are the most effective. Wapiti leverages a curated set of 35 payloads, and SQIRL benefits from the use of RL (Rand-SQIRL takes 4 times as many requests). Thanks to its small set of payloads, Wapiti is also the scanner that uses fewer requests to detect a TN, as the

Table 4.7: Comparison of SQIRL and state of the art tools, on production grade web applications.

Tool	Average Requests per Vuln Input	Average Time (s)	WordPress & Plugins			B2evolution			Sourcecodester e-learning			Sparks Hotel Management			Total		
			FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP
ZAP (bulit in and advanced)	4414.0	452.5	5	4	4	0	1	0	0	2	4	0	1	17	5	8	25
Sqlmap	2280.5	778.2	0	1	7	0	0	1	0	4	2	0	6	12	0	11	22
BurpSuite	211.0	276.6	0	3	5	0	0	1	0	1	5	0	9	9	0	13	20
Arachni	720.0	446.8	0	6	2	0	1	0	0	1	5	0	5	13	0	13	20
Wapiti	30.0	63.1	0	7	1	0	0	1	0	6	0	0	14	4	0	27	6
Rand-SQIRL	475.0	133.1	0	0	8	0	0	1	0	0	6	0	1	17	0	1	32
SQIRL	111.1	159.2	0	0	8	0	0	1	0	0	6	0	0	18	0	0	33

other tools give up finding a vulnerability after a higher number of attempts. Sqlmap’s rule-based approach shows some effectiveness in payload generation, in that it takes half as many attempts in exploiting a vulnerable input than in exhausting a non-vulnerable one. ZAP-advanced instead, with over 8k requests, seems to rely mostly on brute force.

In terms of time elapsed, the average time spent on an input varies between 1 second and 2.5 minutes. Simpler approaches, in terms of number of payloads (Wapiti), or design (Rand-SQIRL) are faster than high volume (Sqlmap, ZAP-advanced) or intelligent scanners (SQIRL).

4.5.3 Comparison on Production Web Apps

In this experiment, we compare the ability of all scanners to find vulnerabilities in production grade web applications. We select at random 10 open source web applications from a popular repository [204], and 4 web applications with SQLi CVEs reported in 2021 with CVSS 3.1 scores more than 7.1. One of these is WordPress, for which we select 7 popular plugins with more than 2k downloads. The list of selected applications and plugins is available in Table 4.8.

In order to avoid any bias due to the use of different crawlers, we test all of the scanners on each input (URL and parameter) found by SQIRL, and we also let the scanners crawl the entire web applications on their own, to verify that SQIRL’s crawler did not miss any additional vulnerabilities. We also provide each scanner the session cookie, to ensure they are logged in. For ZAP, we enable both the built-in and advanced plugins. Table 4.7 reports the results of the experiment on the 4 applications for which vulnerabilities were found. SQIRL is the only scanner able to identify all of the 33 vulnerabilities. Arachni, which found all

Table 4.8: List of Production Grade Web Applications

1. WordPress core v6.0 and plugins (Download Monitor WordPress V 4.4.4, WP User Frontend 3.5.25, Sliced Invoices 3.8.2, Plugin Photo Gallery 1.5.34, Supsystic Ultimate Maps 1.1.12, WP Statistics 13.0.7, JoomSport)
2. B2evolution v7.2.3-stable
3. BBpress v2.6.9
4. Big tree CMS v4.4.16
5. Drupal v9.3.18
6. Joomla v4.2.0
7. Admidio v4.0
8. Gila CMS
9. Media wiki v1.38.2
10. Pbboard v3.0.3
11. Impresscms v1.4.4
12. WackoWiki v6.0.31
13. Sourcecodester E-learning System v1.0,
14. Sparks Hotel Management System v1.0

vulnerabilities in the SMB, found only 20 in production. A similar drop in performance is observed in Wapiti, detecting only 6. This was expected as both scanners performed poorly on the SMB without exception feedback, and production applications should not expose exception traces. For example, Arachni fails to detect the vulnerability CVE-2022-2489 in Sourcecodester e-learning. The vulnerability is on a web page with a search bar, which internally runs the following vulnerable query: `SELECT * FROM posts WHERE body LIKE '%USER_INPUT%' AND courseCode='class101_a' ORDER BY id DESC`. The search results are shown on the page itself, and several scanners are able to find this vulnerability successfully. For example, Burp Suite uses the payload `41896576' or 1334=1334--`. Arachni instead attempts payloads of the form `%TEXT"'--` which intentionally trigger an exception (due to the ' between the ' and the comment). Since Sourcecodester does not reveal the exception on the page, returning instead a default empty search result, Arachni fails to detect success, and reports an FN.

SQIRL has the lowest average number of requests (111) of any effective scanner. ZAP, which is the third-best scanner with 25 TPs, requires 44 times more requests than SQIRL, takes 3 times longer, and is the only scanner to produce false positives.

Rand-SQIRL finds 32 TPs, missing only one, but takes more than 4 times as many requests as SQIRL, for a comparable execution time. Importantly, this shows the advantage of the RL,

both in the context of SQIRL and web application security testing. RL is able to *effectively* use the action space to generate diverse payloads, substantially reducing both the number of requests required, and the variance in the payload discovery process. Indeed, SQIRL has perfect performance in both experiments, whereas Rand-SQIRL had middling performance on the SMB. The ability of SQIRL to identify in total 63/63 vulnerabilities with a low number of requests, after being trained on 20 of them, shows that it has effectively learned, and was able to generalise to new data.

Remarkably, 22 of the 33 vulnerabilities discovered by SQIRL are new zero-days in web applications or versions which had already been reviewed for vulnerabilities (the remaining 11 vulnerabilities), demonstrating that the present approach is practically useful and effective. We have responsibly disclosed the new vulnerabilities in WordPress (CVE-2022-2717, CVE-2022-2718), Sparks (CVE-2022-38918, CVE-2022-38919, CVE-2022-38920), Sourcecodester (CVE-2022-38921), and B2evolution.

4.5.4 Distributed Learning Capability

SQIRL uses multiple worker agents which fuzz the same SQLi and reconcile their learning via the global agent every 20 episodes. This could be considered the simplest case of federated learning, a *vertical* approach. There does exist an alternative *horizontal* approach, where workers interact with different environments [235]. We investigate if SQIRL could benefit from applying the workers to different tasks (input targets, or even web applications), sharing their experiences, in a potentially privacy-preserving way, as in traditional FL.

We therefore use federated RL using 4 Local Workers to interact with different, randomly chosen SMB vulnerabilities. We call this variant FED-SQIRL. We assess its learning performance by measuring the reward the workers receive. We compare with two baselines: a version of FED-SQIRL that can take only random actions, and SQIRL with 4 workers using the same budget of episodes (hence all workers train on the same vulnerabilities).

We show the experimental results in Figure 4.3. The random baseline shows significant variance,

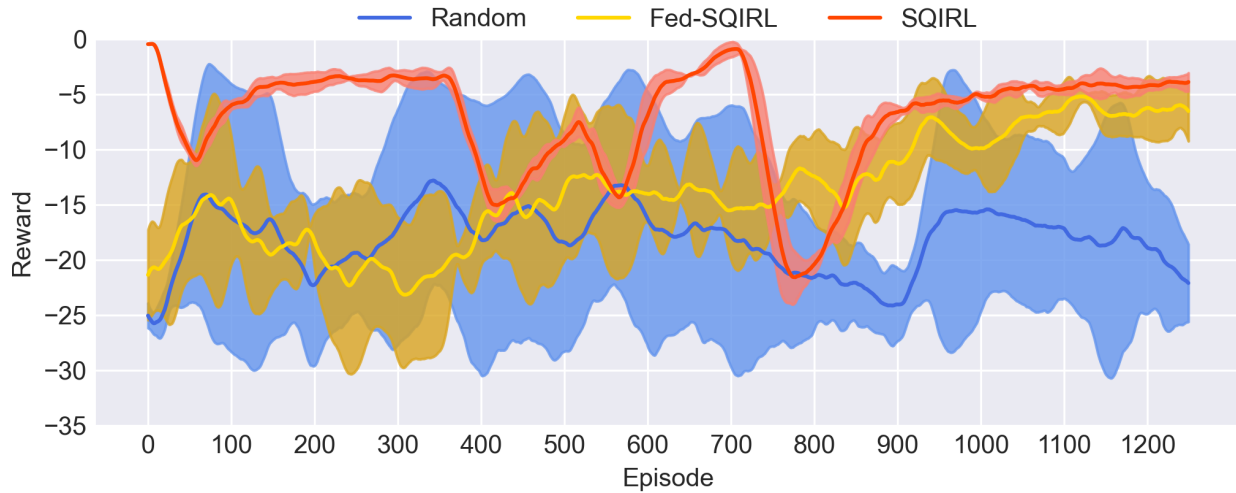


Figure 4.3: Four federated workers of SQIRL fuzzing random targets of the SMB.

as expected, as each worker targets a training sample of different complexity. SQIRL shows minimal variance, as it benefits from each worker targeting the same training sample. The irregularity in the learning curve, corresponds to the switch to different training samples, where learning ‘restarts’ and eventually stabilises. FED-SQIRL shows a significant variance in the workers moving averages. This is due to the different SQLi that each worker is exploiting, requiring the generation of different payloads. Yet, the incremental increase in the moving average, and the gradual reduction of the variance around it, demonstrate that learning is still happening, as the agents improve the quality of the exploits.

The significance of FED-SQIRL is that, with a small loss of performance from SQIRL, we can distribute the learning across Local Workers which could be deployed across different organisations, targeting proprietary web applications, yet mutually improving their SQLi detection capabilities.

This Chapter has aimed to determine the capability of an RL model to perform a complete task: generating SQLi payloads. Indeed we have shown with SQIRL it is possible for an RL model to do this. While FED-SQIRL indicates the potential for a distributed learning system to find injection vulnerabilities using RL. The application and utility of such a model is unclear. Future work remains open to determine the performance of distributed models such as FED-SQIRL for vulnerability detection.

Table 4.9: Ablations of SQIRL on the SMB.

Agent	Avg Cumul Reward	SQLi Found	Avg Time (s) per Vuln Input	Avg Requests per Vuln Input
Rand-SQIRL	585.6	22	5.7	91.9
DQN, 1-hot encoded	682.4	25	191.7	65.8
DQN, AEs	765.3	26	26.8	50.1
DQN, AEs, RND	799.1	30	31.1	41.8
SQIRL: Multi-Worker	984.3	30	27.9	24.0
DQN, AEs, RND				
FED-SQIRL: Distributed	917.7	29	29.7	33.0
DQN, AEs, RND				

4.5.5 Sqirl Ablation Study

To measure the impact of the various design choices behind SQIRL, we here present an ablation study of the model. We start from Rand-SQIRL, as described previously, and gradually introduce and compare the use of RL, its enhancement with autoencoders (AEs) and RND, and the use of the multi-worker architecture (SQIRL) and distributed learning (FED-SQIRL). Such an ablation study is important to show the effect of introducing the autoencoder representation in SQIRL, in addition to the other design choices.

We train each model on the SMB (Section 4.4.3) and challenge each to find its 30 vulnerabilities. The results can be found in Table 4.9. We include the average cumulative reward, normalised so that it is analogous to common RL practice, where greater positive reward implies better performance. At time T , the reward is computed as follows:

$$r_{sum} = \sum_{t=0}^T \min(40 + r_t, 30) \quad (4.1)$$

where r_t is the average reward of the the workers in the episode t .

We observe that each improvement from Rand-SQIRL to SQIRL increases the number of vulnerabilities detected and decreases the number of requests necessary to do so, justifying its introduction. RND improves the generalisation of the model, leading to an increase in reward and reduction in number of requests. The AEs drastically reduce the average time. The multi-worker architecture almost halves the number of requests used. Finally, the introduction of

Table 4.10: Payload types and the scanners that can create them. We replace integers, strings, and base64 with **INT**, **TEXT**, and **BASE64**; otherwise leaving the payloads the same. These payloads are gathered from the experiments in Section 4.5. **F** denotes use of feedback, **NF** denotes no feedback, and **E** denotes presence of exceptions.

Payload Type	Built-in	F	ZAP Adv.	F	Adv.	NF	Sqlmap F	NF	BurpSuite F	NF	Arachni E	Wapiti E	SQIRL
TEXT' OR '1'='1' --	✓												
TEXT%" --	✓												
TEXT') UNION ALL SELECT CONCAT(HEX,HEX,HEX),NULL,NULL#			✓										
(SELECT * FROM (SELECT(SLEEP(5)))TEXT)			✓										
TEXT') AND (SELECT * FROM (SELECT(SLEEP(5)))TEXT) AND ('TEXT'='TEXT			✓		✓								
TEXT') TEXT (SELECT (CASE WHEN (INT=INT) THEN HEX ELSE 0x28 END)) AND ('TEXT'='TEXT')			✓				✓	✓					
TEXT' RLIKE (SELECT * FROM (SELECT(SLEEP(5)))TEXT) AND 'TEXT'='TEXT			✓		✓		✓	✓					
(SELECT * FROM (SELECT(SLEEP(5)))INT)					✓								
TEXT AND (SELECT * FROM (SELECT(SLEEP(5)))TEXT)								✓					
TEXT' UNION ALL SELECT NULL,CONCAT(HEX,HEX,HEX),NULL--							✓						
id=INT') UNION ALL SELECT CONCAT(CONCAT('TEXT','TEXT'),'TEXT'),NULL,NULL- TEXT							✓						
(SELECT (CASE WHEN (INT=INT) THEN INT ELSE (SELECT INT UNION SELECT INT) END							✓						
'+(select*from(select(sleep(20)))TEXT)+'									✓	✓			
(select*from(select(sleep(20)))TEXT)									✓	✓			
\'									✓				
INT or INT=INT									✓				
and (select*from(select(sleep(20)))TEXT)--									✓				
INT' or INT=INT-- and INT' or INT=INT--									✓				
INT' or 'INT'='INT and INT' or 'INT'='INT									✓				
' and INT=INT-- and ' and INT=INT--									✓				
TEXT\"' --											✓		
INT'%TEXT\"' --											✓		
INT'=sleep(16)='											✓		
INT' and sleep(16)='											✓		
0 or sleep(16) #											✓		
INT_'"(												✓	
AND INT=INT AND INT=INT												✓	
BASE64 AND SLEEP (0.0)													✓
BASE64" And SLEEP (0.0)' AND SLEEP (0.0)#													✓
BASE64)' and SLEEP (0.0))#													✓
BASE64' aND sLEEp (0.0)#													✓
BASE64'/**/&/**/**/SLEEP/**/(0.0))#													✓
BASE64/**/And/**/**/SLEEP/**/(0.0)"/**/AND/**/**/SLEEP/**/(0.0)#													✓
BASE64 and sleep(0.0)													✓
BASE64' & SLEEP (0.0)#													✓
BASE64' AND SLEEP (0.0)-- --													✓
BASE64'#													✓

distributed learning introduces only a small performance cost, to the potential benefit of a broader application scenario.

4.6 SQLi Payload Analysis

The payloads generated by scanners reveal insight about their general capabilities. In Table 4.11, we display the different features that each scanner generated, including context escape (comments, quote and parentheses) and sanitisation bypass (concatenation of parameters, capitalising key features, whitespace escaping, AND escaping). These are not exhaustive, as they come from the payloads observed during the experiments in Section 4.5. We provide SQIRL payloads in Listing 4.1, and payloads from all scanners in Table 4.10.

ZAP-built-in is only able to insert comments to clean up the SQL after the payload, even when there is feedback on the page, generating two payload types. With the advanced plugin, it

Listing 4.1: Generic representation of payloads generated by SQIRL.

```

23 BASE64 AND SLEEP (0.0)
24 BASE64" And SLEEP (0.0)' AND SLEEP (0.0)#
25 BASE64) anD SLEEP (0.0))#
26 BASE64' aND sLEEp (0.0)#
27 BASE64' /**/&/**/ /**/SLEEP/**/(0.0))#
28 BASE64/**/AnD/**/ /**/SLEEP/**/(0.0)"/**/AND/**/ /**/SLEEP/**/(0.0)#
29 BASE64 and sleep(0.0)
30 BASE64' & SLEEP (0.0)#
31 BASE64' AND SLEEP (0.0)-- --
32 BASE64' #

```

escapes both single quotes and parentheses. Relying only on these features causes ZAP to miss the most complex cases in the SMB and the real world web applications. ZAP tends to use payloads that are similar in structure, with little variance. It is the only scanner to have payload structure overlap with another scanner, Sqlmap, with two in common. Sqlmap is the only scanner to use the `CONCAT` function to avoid sanitisations. BurpSuite displays consistency in the features it uses to generate payloads, irrespective of feedback. It creates simple quote, `select`, and `or` based payloads, making the most use of feedback compared to other scanners, producing 6 additional payloads. Arachni uses both comments and single quotes in SMB to identify all SQLi when there is an exception present in the page. It uses two simple payloads (`''--`, and `INT!%TEXT\''--`) to escape context and use a comment to clear up the remaining SQL. Wapiti, like Arachni, uses simple payloads that attempt to escape multiple contexts without using SQL keywords (`''()`). This lets it identify 13 SQLi when the exception trace is present in the SMB. For the web applications, Wapiti only uses one alternative payload structure: `AND INT=INT AND INT=INT`. This limited variety in payloads leads to the poor performance observed in Table 4.7.

As intended, SQIRL generates a greater variety of payload structures than other scanners (Listing 4.1). SQIRL uses capitalisation obfuscation in payloads 26, 27 to solve 12 and 13 in SMB (where other scanners require feedback or exceptions to identify this). We see SQIRL's ability to generalise as it reuses 26 to identify CVE-2021-24786. SQIRL is also the only scanner that makes use of whitespace obfuscation to exploit 14, and that replaces `AND` with `&` (in 30) to bypass sanitisation of 15.

We can see that SQIRL favors payloads of the form similar to 23, mutating this to include

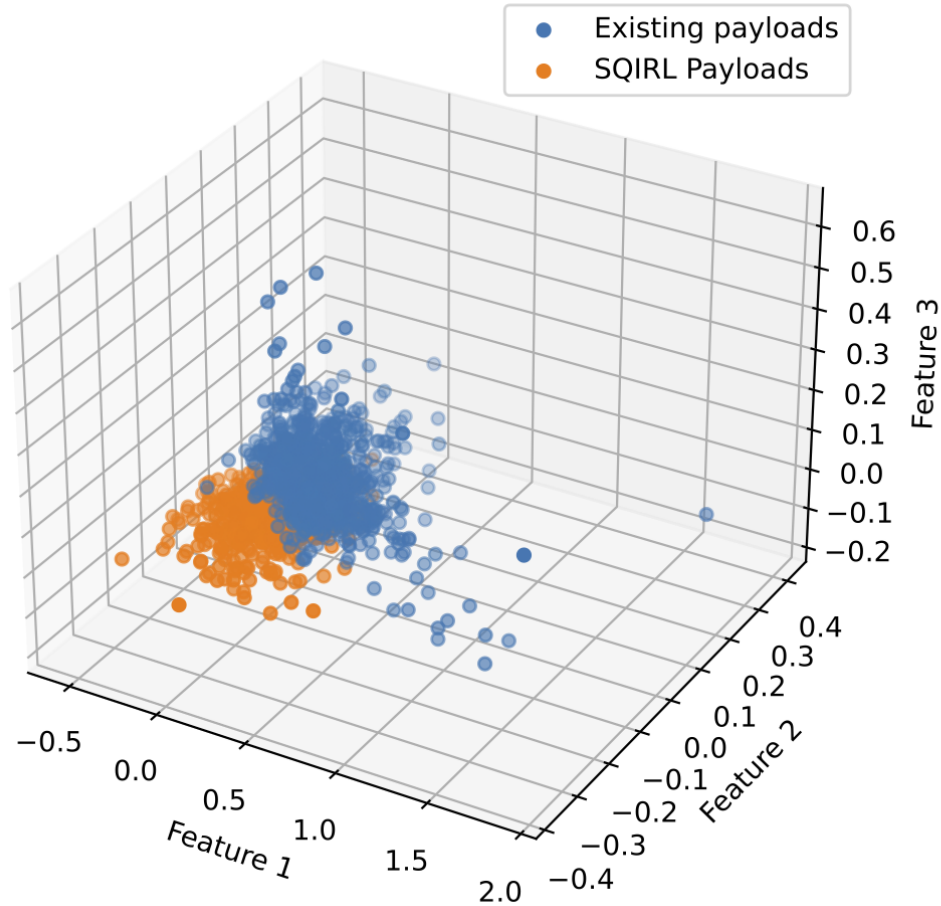


Figure 4.4: PCA of existing payloads and those from SQIRL.

quote marks that escape the context, comments (`--`, `#`) to prevent issues with trailing SQL, in addition to the obfuscation techniques. This shows that SQIRL has leveraged reinforcement learning to develop a minimal payload and then mutate this to tailor it to each input case, resulting in the low number of requests seen in Section 4.5.

Using the same technique as in Section 4.2.1, we generate latent representations and perform a PCA of existing payloads and payloads generated by SQIRL. Figure 4.4 shows that some payloads from SQIRL overlap with existing payloads (faint orange points are visible towards the right-center of the distribution) yet a significant fraction occupies new space, showing that SQIRL has learnt to use the action space to produce out-of-distribution payloads compared to existing payloads. Note that only a fraction of the latent space is occupied, confirming the intuition that the space of valid SQLi payloads is smaller than all possible payloads.

Table 4.11: Features that different scanners can make as observed in the payloads seen. **F** denotes use of feedback in a web page, **E** exceptions.

Tool	F	E	Comment	Payload Types	Single Quote Escape	Parentheses Escape	Concat	Caps Escape	Whitespace Escape	AND Escape
ZAP (built-in)	✓	–	✓	2						
ZAP (advanced)	✓	–	✓	5	✓	✓				
ZAP (built-in)	✗	–		0						
ZAP (advanced)	✗	–		4	✓	✓				
Sqlmap	✓	–	✓	5	✓	✓	✓			
	✗	–		3	✓	✓				
BurpSuite	✓	–	✓	8	✓			✓		
	✗	–	✓	2	✓			✓		
Arachni	–	✓	✓	5	✓					
	–	✗	✓	0						
Wapiti	–	✓		2	✓	✓				
	–	✗		0						
SQIRL	–	–	✓	10	✓	✓		✓	✓	✓

4.7 Discussion and Limitations

Grey-Box Approach. SQIRL makes use of information from the SQL database, but does not have access to the web application source code, making it a grey-box tool. Thus, we compare it against scanners that require similar levels of access to web applications, and white-box approaches are out of scope. We argue that the grey-box approach is well-suited for developers and security practitioners, for two reasons. First, vulnerabilities can be found without relying on the insecure practice of leaking error information on the web pages themselves (SQL statement results or exceptions). Second, the approach is effective for pentesters, whom can be allowed to obtain access to the database logs, whereas it is not useful for malicious actors targeting deployed web applications.

Training vs Testing. Note that 20 of the SMB positives were already used to train SQIRL: their reuse for testing is standard RL practice, as success during training is already evidence of the RL agent success in its task. The agent does not receive feedback from the labels of the samples, but only from changes to the observable state. The case is different for supervised learning, where training feedback ‘leaks’ information about the sample labels, which should therefore not be used for testing. Similar to Chapter 3 we trained SQIRL on a small training set of 20 samples. We found that the SMB was enough to find all the vulnerabilities in the experiments of Section 4.5, achieving a better performance than established scanners.

Crawling. In our evaluation, we have taken steps to remove crawling-induced bias from our

experiments. The crawler used in SQIRL was developed as a tool for the payload generation by extending a state-of-the-art crawler, and we do not regard it as a key contribution. Our crawler is unable to utilise CSRF tokens and unique tokens used in forms and dynamic links unless the web page is accessed again. This was investigated, but would double the number of requests sent to the web application. As with Chapter 3, crawling is not the main focus of our work, but provides a means to answer our research questions related to RL and web application security testing.

4.8 Related Work

4.8.1 Fuzzing for SQL Injection.

One of the first papers to separate mutations into different classes used rule-based mutation [11]. While SQIRL uses a similar concept to separate actions for our agents, using RL means it can learn the grammar, yet also violate it, to develop SQLi. Lei *et al.* [154] identify inputs in web applications that are ‘most likely’ to lead to a vulnerability. This work focuses mainly on crawling web applications, using predefined payloads, and heuristic mutations. Zhao *et al.* [320] use combinatorial testing to select mutations to apply to a SQL statement. This worked better against sanitisations, but using predefined SQL statements limits the scanner when finding edge cases, as opposed to SQIRL.

DeepSQLi [164] uses a neural language model to develop SQLi payloads. Starting from a user input, DeepSQLi iteratively generates a new payload using a transformer, guided by a beam search to improve diversity, until it manages to exploit a vulnerability. Similarly to our approach, they use a SQL proxy to provide feedback on the payload success. Comparing against Sqlmap, DeepSQLi achieves a maximum improvement of 29.5% in terms of vulnerabilities found. In contrast, SQIRL achieves a 70.2% improvement on Sqlmap in our experiments. This comparison is only indicative, as the target web applications are different.

A later work from Gui *et al.* [98] examines the use of LLMs to generate SQLi payloads. In

particular they use three state-of-the-art LLMs configured as agents in an exploratory study. Re-presenting the SMB for evaluation purposes the authors show that LLMs can generate SQLi payloads, but can be unstable, hallucinate, and be inefficient, using a higher number of requests than SQIRL.

Aliero *et al.* [6] present an algorithmic approach, using an object-oriented model. This is however limited in scope, requiring error responses and regex for login conditions. Luo [173] uses a genetic algorithm approach to mutate predefined payloads, minimising the number of attempts in controlled settings. RL has also been used to modify SQLi payloads in order to bypass web application firewalls [301]. Del Verme *et al.* [58] discuss the difficulties of exploring the diverse payload space and the importance of mutations in SQLi. Erdődi *et al.* [71] then develop two simple RL agents to generate payloads using tabular Q-learning and a DQN. The models are demonstrated on limited cases and are not extended to handle the variety of SQLi of interest for practical use in web applications.

4.8.2 Defensive Coding Against SQLi.

SQLi can also be prevented via defensive coding methods. Su and Wassermann [271] track user inputted queries, at runtime, to block those where substrings change the syntactic nature of the query. Valeur *et al.* [284] and Halfond *et al.* [106] develop statistical models of queries, using anomaly detection to prevent queries that do not conform to normal behaviour. Context-Sensitive String Evaluation (CSSE) [229] is application agnostic, using only metadata about queries to determine untrusted user inputs to enforce strict channel separation. Sun and Beznosov [272] present another application agnostic method, which intercepts queries to enforce conformity of input type and intention of the SQL. Kasim [137] presents ensemble methods, including tree search and regex-based feature extraction, to classify SQL queries as benign or malicious.

A widespread adoption of the defensive techniques surveyed here, and of parameterised queries, would drastically reduce the occurrence of SQLi. Yet these solutions have been around at least

since 2005, and SQLi still plagues web applications, as we saw with a concrete example from WordPress in Table 4.1 (Line E). Justifying further research on detection techniques.

4.9 Takeaways

In this Chapter we have focused on increasing the capacity of a single RL model to perform testing in web application security. We developed a *single* agent for the problem of *generating* SQLi payloads to find vulnerabilities. SQL payloads must adhere to the strict syntax in which they are found, which by comparison with XSS payloads do not benefit from the interaction of JavaScript and HTML engines which can trigger vulnerabilities from malformed payloads. Therefore, SQLi presents a harder target to find vulnerabilities.

We use a grey-box approach to combine executed SQL queries from the database logs with the payloads that were submitted to a web application. This information provides greater detail on the defences the web application may have in place, and the SQL query which an agent must exploit. However, how to best represent this data to maximise the information for an agent is challenging.

Similarly, having ‘good’ data to learn from is also challenging. In this Chapter, we generated SQL statements and payloads via a context free grammar. However, such mechanisms to generate data may not always be readily available in other applications.

We have improved on the representation used in the previous Chapter, and avoided the pitfalls from representations which can lead to poorer performance from agents [88, 125, 316]. SQIRLs tailored architecture, based on GRU models, can observe variable length SQL and payloads to take a variable number of actions when generating a payload. From the diverse space of payloads that SQIRL could make, it learns to generate a different distribution of tailored payloads compared to existing state-of-the-art tools. Furthermore, SQIRL was able to find more SQLi vulnerabilities in our SMB and 14 production grade web applications, achieving this in fewer requests compared to state-of-the-art scanners we compare against. Vulnerabilities identified by SQIRL have currently led to 6 CVEs.

This Chapter has built on the lessons from HAXSS to develop a single agent capable of finding a single vulnerability by generating tailored payloads. While impressive results have been seen from both HAXSS and SQIRL building on these further is to focus on the single, and narrow, domain of scanning for injection based web vulnerabilities. Indeed, there are still unanswered research questions remaining relating to the use of RL for such a domain, including: the generalisation of a single agent to multiple vulnerabilities [157], polyglot payloads [144, 178], and mutation strategies of payloads [157, 164]. However, we wish to evaluate the performance of RL for the broader domain of web application security testing. This encompasses, related but different research areas besides generation based vulnerability scanning (Chapter 3, and 4) including: mutation based testing (Chapter 5) and fuzzing (Chapter 6). Furthermore, research into injection based vulnerability scanning via RL would require significant engineering effort to advance the performance of crawlers and testing environments, as used in more mature scanners [56, 210, 231, 273].

Chapter 5

REST API Fuzzing By Mutating Test Cases with Reinforcement Learning

In previous Chapters, we have developed RL architectures to improve the ability of RL agents to conduct web testing tasks. Starting from multiple agents learning to play separate games of payload generation, we expanded the architecture to a single agent that learns all the separate elements to create a payload.

In this Chapter, we again have a single agent perform a web application testing task. However, instead of *generative* testing as investigated in the previous two Chapters, we will investigate the use of RL for *mutation* of test cases. We also focus on a different testing subject: REpresentational State Transfer (REST) APIs.

REST APIs have become the standard way to interact with Web services and resources. They are used by organisations such as Amazon, Google, and OpenAI to integrate with a wide array of systems, processes, and resources. These APIs consist of operations, specified by a Uniform Resource Locator (URL) accessible by HTTP, and a series of associated parameters. Atlidakis *et al.* [21] show bugs in these operations are typically hard to find, due to the complex interactions and diversity of APIs. Some bugs have security implications that result in the extraction of information or data manipulation, as summarised in the OWASP API Security Top 10 [218]. Hence, it is crucial to test the robustness of REST APIs using sophisticated techniques.

Compared with injection payload generation, techniques for testing REST APIs use mutation based fuzzing approaches to discover bugs or security vulnerabilities. In doing so, they iteratively modify HTTP request test cases, and monitor for abnormal behaviour such as a server-side error. Early works achieve this using manual rule-based approaches to mutate test cases [145, 225]. Subsequent works then automatically mutate test cases using black-box fuzzing techniques [20, 167]. While such approaches have lead to improvements in REST API fuzzing, they lack targeted search strategies, or do not harness contextual information. This behaviour limits the potential of any fuzzing framework due to the varying schema requirements of each API, their diversity, and the unique behaviour of endpoints. As a result, these models can be *inefficient*, requiring a very large number of test cases to find bugs.

We have seen in previous Chapters the benefit of RL in web testing tasks, additionally we have also seen how improving the representation of feedback for RL states leads to better agent performance. However, similar to REST API testing [21, 53, 167, 176], RL-based software testing often uses simple heuristics as feedback [153, 159], which may fail to fully capture the subtleties of the problem. While work from Kim *et al.* [141] and Corradini *et al.* [51] has shown that RL can find bugs in REST APIs, they suffer from the same issue: using only simple heuristics for feedback. The key challenge in using contextual feedback arises from the variation across different APIs and the diversity in individual endpoint responses. Yet, alternative, information-rich data structures could provide valuable feedback for test case mutation.

Thus, in this Chapter we address these challenges by developing a Deep RL agent, APIRL, that learns how to manipulate varied HTTP requests to efficiently target bugs. Our RL problem formulation uses a transformer architecture we pre-train to harness JSON and natural language in HTTP responses. Transformers capture long range dependencies more effectively than RNNs. Thus, they can learn the subtleties of HTTP response data, providing information-rich feedback to facilitate adaptation to operations after training [287]. We then introduce a Markov Decision Process (MDP) for the mutation of HTTP requests as a number of sequential changes from an initial HTTP request (referred to as *request template*). Between mutations, the new test cases are submitted to an API, using the response code and execution information to determine the reward used in training. The trained policy uses the mutation strategy to find bugs in unseen

APIs with a minimal number of HTTP requests, overcoming a significant limitation of web based testing approaches [153].

Chapter Contributions.

This work has been published at the 2025 AAAI Conference on Artificial Intelligence. I designed, developed, and implemented the APIRL framework. I also conducted the evaluation and further analysis of the experiments.

5.1 Overview

In this Chapter, we introduce APIRL, a *mutation* based testing approach leveraging deep Q-learning to mutate HTTP requests and find bugs in REST APIs, indicated by 5XX response codes. We represent REST API testing as an MDP for a deep RL agent: Figure 5.1 shows the process and agent architecture.

This approach differs from what we have seen in previous Chapters: instead of generating new HTTP requests to test API endpoints, an RL agent will mutate a HTTP *request template*. Furthermore, we can test the ability of RL in a different, and important, area of web application security testing: REST APIs. This area in particular is well suited to *mutation* testing, with many works focused on it [20, 21, 140, 145, 167]. REST API fuzzing often begins with the OpenAPI specification that details the intended functionality of all endpoints (for further detail see Section 5.2.1) [20, 21, 52, 150, 167].

At a high level, an agent takes actions to mutate HTTP request templates. The environment implements actions as concrete mutations, submitting mutated requests to API endpoints, and feedback is received that forms the reward. APIRL takes a HTTP request-response pair as the input state s_t . Maximal information is then extracted from functional features and an embedded representation via a pre-trained transformer (Section 5.2.4). Using diverse and variable length features such as HTTP headers and body data, a neural network selects a corresponding action a_t from Table 5.1 to mutate the HTTP request.

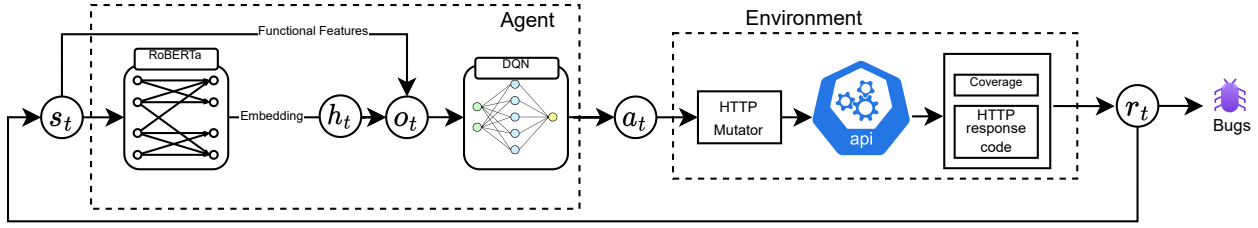


Figure 5.1: The REST API testing process using APIRL.

Using the agent-selected action, the environment then performs concrete mutations on the HTTP request template forming *test cases*. The model evaluates the test case performance using the HTTP status code and the execution trace of the REST API. We develop these to form varied reward functions to study their effect in training (Section 5.2.5 and 5.3.5). The model evolves to optimise for the reward as it performs more mutations, and performance evaluations.

We compare the learnt policy against a state-of-the-art learning and non-learning tools. Finally, in an ablation study we evaluate seven reward functions and key design choices.

5.2 A Framework for Learning to Find Bugs in REST APIs

In this section, we define our RL-based REST API testing approach and detail the model design of APIRL.

5.2.1 OpenAPI Specification

The OpenAPI Specification [209] is a standard for describing REST APIs, documented in the YAML or JSON format. It is also the standard starting point for REST API testing frameworks [21, 167], and where we begin with the APIRL approach.

It specifies the URL and HTTP method (POST, GET, PUT, PATCH, and DELETE) which forms an

```

1  "/users/v1/{username}/email":
2    put:
3      security:
4        - bearerAuth: [ ]
5      parameters:
6        - name: username
7          in: path
8          required: true
9          schema:
10             type: string
11             example: 'name1'
12      requestBody:
13        content:
14          application/json:
15            schema:
16              type: object
17              properties:
18                email:
19                  type: string
20                  example: 'mail3@mail.com'
21            required: true
22      responses:
23        204:
24          content: {}
25        400:
26          content:
27            application/json:
28              schema:
29                type: object
30                properties:
31                  status:
32                    type: string
33                    enum: ['fail']
34                    example: 'fail'
35                  message:
36                    type: string
37                    example: 'Please Provide a valid email address.'

```

Figure 5.2: Part of the OpenAPI specification for VAmPI.

operation. Each operation includes the schemas for associated parameters, and the required authentication schema. Figure 5.2 shows a snippet of the specification for the PUT endpoint `/users/v1/{username}/email` in VAmPI [73]. This example includes the `username` parameter in the URL (Line 1) and key-value pair in the request body (Line 12) of `{"email": "value"}`. Each parameter is assigned a name, type (`integer`, `string`, `object`, *etc.*), and can contain details such as examples (*i.e.* `mail3@mail.com` on Line 20), or even if it is required in the operation (*i.e.* Line 21). Note the HTTP response code specified in Figure 5.2 of ‘204’, indicates a successful request. However the response could be in the range 400-499 (4XX) indicating a user error as on Line 25, or 500-599 (5XX) indicating a server error or bug.

We define bugs as 5XX responses from each operation that originate from different lines of

source code. A status code of 5XX, implies that there has been an error in the server causing a fatal crash. The 5XX response should be distinguished from 4XX codes which handle bad requests gracefully, providing the relevant status code to inform the user of their error. Such a definition is in line with several previous works [13, 21, 52, 141].

5.2.2 Preprocessing

From a specification, we extract the operations and their associated parameters. This forms a list of request templates (HTTP method, URL, parameters details) for the RL agent to mutate. For example, the operation in Figure 5.2 would have a request template such as: (PUT, /users/v1/{username}/email, {name:username, in:path, required:true, type:string}). We also find related parameters to form valid requests, as recommended by the literature [180]. This is done by initial observation of HTTP responses of all operations. Initially we iterate over all operations twice to capture all related key-value pairs. This is under the intuition that we may need to utilise some of these key-value pairs in order to gather additional parameters. If there are no dependencies we default to using randomly generated values that correspond to the parameter type, (*e.g.* a string of 6 random characters, or random integers).

We then determine related key-value pairs by the common values seen in responses. As an example, two separate endpoints may respond with parameters `username:alice1`, `name:Alice`, and `username:alice1`, `user_id:1`. We can then see that `username:alice1` appears in both responses, thus we consider the three key-value pairs to be related to each other.

While it would be possible to select the most preferable keys in certain settings (*e.g.* `username` over `name`) we treat keys agnostically, as it is not possible to determine key preference in all settings, (*e.g.* if keys such as `username`, `id`, `user_id` were used in a REST API response). Using the concrete values most recently seen, we improve the chances of using *valid* related values. This allows us to be agnostic of the specific operations, this is important as there are many different combinations of parameters and operations that could occur.

Table 5.1: Mutation actions that can be applied by APIRL to the current operation.

Action Type	Action Number	Example	Description
Auth Token Refresh	1		Refresh the any authorization token.
Switch Auth Token	2		Use the authorization of another user.
Switch Parameter	3		Change to the next parameter in request template.
Change Parameter Type	4	<code>Int, String, Bool, Array, Object</code>	Randomly change the parameter value type to a different one.
Duplicate Parameter	5	<code>{parameter1:[value1, value1]}</code>	Duplicate the parameter value.
Remove Parameter	6	<code>{}</code>	Remove the parameter from the request template.
Extension	7- 9	<code>.txt, .pdf, .doc</code>	Append a file extension to the value of the parameter.
Append	10	<code>{parameter1:[value1, newValue]}</code>	Convert the parameter to a JSON array and append an additional value from observed values.
Request Method	11	<code>POST → PUT, PUT → POST</code>	Change HTTP method from PUT to POST or POST to PUT.
Add Parameter	12	<code>{parameter1:value1, newParameter:newValue}</code>	Add an additional parameter to the request template from parameters related to the parameter.
Wildcard	13-15	<code>*, .*, %</code>	Append a wildcard to the value of the parameter.
Change ID number	16-17	<code>1, -1</code>	Increment the <code>Int</code> value of the value by 1.
Set Parameter value	18-21	<code>'admin', -1, 999999999, "</code>	Set the parameter value to a default value.
Set Existing Value	22	<code>{parameter1:value2}</code>	Set the parameter value to a related value from the API.
Set admin	23	<code>{parameter1:value1, admin:TRUE}</code>	Set <code>admin:True</code> in an <code>Object</code> .

5.2.3 Actions

We provide the RL agent with a fixed action space of 23 actions a_t that are adapted from prior work [21, 24, 53]. To do so, we reimplement each action in the APIRL framework, and use distinct values where possible *e.g.* actions 13-21. Actions from Barabanov *et al.* [24] do not give concrete definitions of mutations, but instead the conditions for which they can occur. We give them concrete definitions in Table 1. Remaining actions follow their original implementations where possible to avoid introducing any potential bias or advantage in implementation.

Actions are performed on requests and are independent of specific applications so APIRL does not need further training or feedback from rewards when encountering new REST APIs, thus aiding generalisation.

Actions are detailed in Table 5.1, and can be broadly divided in three categories. Actions 1 and 2 that alter the authorisation token by either refreshing the current authorization token (if an endpoint allows for this), or switching to an alternative authorization token if one has been provided. Action 3, that allows the agent to switch to begin mutating the next parameter for this operation. This then loops to the start of the parameters upon reaching the end. Finally, actions 4-23 that manipulate the request template to perform REST API fuzzing functionality. We utilise those that may trigger bugs by duplicating or removing parameters, using default values, and finding alternative endpoints.

Using actions in Table 5.1 the agent mutates a request at each timestep. After a mutation the request is sent to the REST API and, in training, runtime information is collected for the reward (Section 5.2.5). The episode then terminates if a bug is found (as indicated by a 5XX status code) or if the maximum number of timesteps have elapsed.

5.2.4 Observations

Making use of information from real world tasks for RL observations can be challenging due to the complexity, high dimensionality, or format (such as natural language). We aim to make APIRL bridge this semantic gap with a pre-trained RoBERTa [168] transformer model. We use transformers as they can effectively capture long range dependencies compared to GRUs. This is an important property as HTTP responses which can be significantly larger than stand-alone strings (*e.g.* the SQL statements of Chapter 4). APIRL’s transformer takes as input the HTTP response from the test cases to produce a latent representation. This means APIRL can harness complex JSON and natural language from the structure, text, headers, and encoding of API Responses. The latent representation forms a vector of 768 features from the RoBERTa transformer (its standard output feature dimension), shown as h_t in Figure 5.1. The observation further includes: the HTTP method of the current operation, the HTTP response code, the variable type of the parameter currently being manipulated, and the normalised index of this parameter out of all parameters in the request template. These features are represented as a vector of length four. Both feature vectors are combined (o_t in Figure 5.1) and passed to

the DQN as a single vector of 772 ($768 + 4$) features. See Section 5.3.1 for further details on pre-training the transformer.

5.2.5 Reward

Two main signals can be used to guide REST API testing: a) coverage of the REST API; and b) the HTTP response code of the request. Coverage reflects the ability to explore the API behaviour, blindly aiming to exercise as much of the implementation code as possible, hoping to trigger bugs in the process. On the other hand, the HTTP response code of the request provides information to guide testing, including the validity of requests in terms of authorisation (*e.g.* 401), parameters (*e.g.* 200 or 400), and server-side errors or bugs (*e.g.* 500). We define the reward for APIRL based on the HTTP response code as it provides more nuanced feedback on test case performance (Eq. 5.1). We will consider alternative rewards, including coverage, in an ablation study (Section 5.3.5).

$$R_{sc} = \begin{cases} 10, & 5XX \text{ HTTP status response} \\ 1, & 2XX \text{ HTTP status response} \\ -1, & \text{Otherwise} \end{cases} \quad (5.1)$$

R_{sc} incentivises the agent most for the desired behaviour of finding bugs on the server-side of the REST API. However, as this can be a sparse reward, we provide interim feedback for performing correct requests. In all other cases we discourage the behaviour using a negative reward [275]. This reward is *consistent* as training progresses so the agent learns to develop diverse requests, covering more of the back-end of the REST API.

5.2.6 Agent Architecture

To develop mutational strategies that can be dynamically altered to specific operations and REST APIs, we develop a Deep RL agent based on the Deep Q-Network [191]. We implement

Table 5.2: Grid search ranges for hyperparameters in APIRL.

Model	Hyperparameter	Lower Bound	Upper Bound	Selected
DQN	γ	0.5	0.999	0.9
	α	0.05	0.0005	0.005
	Batch Size	32	256	128
	Update Step	10	200	100
	Episode Length	5	20	10
PPO	γ	0.5	0.999	0.9
	α	0.05	0.0005	0.005
	Entropy Coef	0.01	1.0	0.15
	Clip Coef	0.01	1.0	0.1
	Episode Length	5	20	10
RoBERTa	Batch Size	32	256	64
	Training Epochs	5	20	10

the neural network in PyTorch, with an input layer of size 772, and hidden layers of size 64, 96, 64, with an output layer of 23, corresponding to the actions in Table 5.1. We use the ReLU activation function after each hidden layer. The agent learns which mutation, or combination of mutations to apply, while reducing the computational complexity of the neural network. We use a standard ε -greedy decay with $\varepsilon = 1$ (decaying by 0.999 after each episode). We select $\gamma = 0.9$, $\alpha = 0.005$, and batch size 128. These are selected via gridsearch, further details and parameter values can be found in Appendix 5.2. To select the episode length and additional hyperparameters we perform a grid search. We report the values of this grid search in Table 5.2; showing the lower and upper bounds in addition to the selected values.

5.3 Evaluation

5.3.1 Training

Training should result in an RL agent that can manipulate HTTP requests, generalising to find bugs in different APIs. Thus, we train APIRL on multiple endpoints, targeting each operation for a set number of episodes. We have seen in previous Chapters how this form of curriculum learning prevents over-fitting by dividing training equally across diverse operations. Specifically, APIRL is trained using an open-source REST API used for manual testing containing known bugs: Generic University [223]. APIRL trains on each of Generic University’s 11 operations for

10,000 episodes, with the maximum steps per episode of 10 (see Table 5.2). APIRL also uses a standard ε -greedy from $\varepsilon = 1$, decaying by 0.999 after each episode

We pre-train a RoBERTa transformer using HTTP responses from 103 different REST APIs, comprising 1283 operations. This has two advantages: limiting the bias from HTTP responses in training, and reducing instability in training the RL agent [219]. API specifications were taken from a public OpenAPI specification platform. The REST API dataset consists of 2566 responses from 1283 operations, responses were collected from the REST APIs on April 29th 2023. REST APIs were identified from an open-source OpenAPI Specification platform¹. This dataset can be found in the APIRL repository. REST APIs used in the dataset range from popular cryptocurrency platforms such as coindesk, online dictionary services such as Wiktionary, image generation tools, and open-source database services. We save the HTTP response data from each endpoint collating it with its corresponding HTTP request.

While it may be possible to fine-tune an existing variant of RoBERTa, we opt to pre-train on a smaller corpus of data. Training on API HTTP requests and responses provides a heavily structured dataset differing from the kinds of data structures seen in natural language. HTTP requests are formulated using set a specified structure, whilst natural language doesn't follow the same rigid pattern. Thus, we speculate that a smaller, but highly structured dataset is sufficient to learn the semantic relation between the tokens.

This data was captured through the several steps:

1. Crawling the open-source OpenAPI Specification platform to extract specifications. This is with the aim to collect only *publicly* available data, to reduce the potential concerns from personal or private information.
2. Manually verify that each of the REST APIs has a public facing endpoint that can be accessed. This ensures that we can gather data from the REST API endpoints.
3. Using the OpenAPI specification and the APIRL framework we send HTTP requests to each operation, capturing the responses. We also retain the parameter values from these

¹<https://app.swaggerhub.com/search>

responses in order to use their dependencies in making well formed requests.

4. We then send a request to each of the endpoints a second time, using the previously seen parameter values. By sending a second request to each endpoint after we have collected initial data, we can reuse this to make well-formed requests, increasing the likelihood of making valid requests [180]. This approach is similar to what we have seen in Section 5.2. Achieving this is important to increase the number of responses that contain meaningful information from which the RoBERTa model can learn.

When training the RoBERTa model used in APIRL we preprocess each of the responses to form string based *sentences* for input to the feature extractor, including: the status code, headers (removing those that contribute to noise in learning, such as `Date`, `etag`, `report-to`, or `Last-Modified`), encoding, HTTP protocol, and any redirects. A vocabulary of 52,000 tokens is formed via Byte-Pair Encoding [89]. The transformer is trained by masked language modelling [62]. A gridsearch is used to select hyperparameters as in Table 5.2.

Through training the transformer learns relationships between parameters, to provide a meaningful, generalised, embedding for the agent [2]. As a result, both the state and action space are target agnostic, meaning that the subsequent policy (APIRL) can be used on unseen REST APIs without the high number of training iterations required to reach optimal performance. An advantage of this behaviour is no further learning or feedback for the reward (*e.g.* code coverage) is required and APIRL can be run in black-box fashion.

5.3.2 Experiment Setup

To test APIRL we make several modifications to its setup: we limit the number of episodes per operation to three, and reduce the probability of taking random actions (ε) to 5%. As in previous chapters this is to place emphasis on the learnt policy. Experiments are run on Ubuntu Linux, with 16GB RAM and Intel core i7 8700k processor. To mitigate the intrinsic stochasticity of fuzzing, we repeat each experiment five times [146, 250].

REST APIs. We evaluate our framework on smaller REST APIs, including: VAmPI [73], vAPI v1.3 [149], and c{api}tal [183]. We conduct large scale tests for bugs in APIs from large-scale projects. Spree Commerce v4.4.0 (a popular e-commerce platform with 12.5k stars on github) containing 2 APIs, 17 APIs from BitBucket v8.2.1 (a popular git hosting service with over 15 million users), and 4 APIs from WordPress v6.6.1 (an opensource web platform used in over 5 million websites). These 26 APIs represent 823 separate operations.

We run tests on VAmPI, vAPI, and c{api}tal for a maximum of 1.5 hours, and 10 hours on BitBucket, Spree Commerce and Wordpress APIs, due to their complexity and increased number of operations. Testing these tools for such time limits is in line with other studies [52, 141, 142, 167, 176]. Additionally, broader studies in fuzzing, such as by Böhme *et al.* [35] have shown that most fuzzers find the majority of coverage early in testing, after which they asymptotically converge.

Due to ethical concerns over data privacy, integrity, and availability, *we do not test on live REST APIs*, but instead deploy them locally. In this way we prevent APIRL from potentially storing, manipulating, or changing real user data. Additionally, it prevents potential outages in deployed systems. Both effects are undesirable for live systems, as this can have unknown effects, *e.g.* changing source-code in a repository hosted on BitBucket could have down stream effects on third party applications. Each REST API is initialised with *non-sensitive* data. After each test we restore it to the state prior to testing to remove any bias.

Baselines. In order to compare with the state of the art, we select five black-box baselines. *MINER* [176] builds call sequences using feedback from executed requests and an attention based neural network to generate mutational parameters. *ARAT-RL* [141] is a *tabular*-Q-learning based REST API tester with separate Q-tables for: operations, parameters, and parameter value mappings. *DeepREST* [51] is a multi-learning approach using an LLM to generate key value pairs, a Deep RL agent to select operations, and a series of Multi-Armed Bandits (MABs) per parameter to select mutations. *EvoMaster* [13] is based on evolutionary algorithms and heuristics. *RestTestGen* [52] uses the specification to generate Operation Dependency Graphs (ODGs) to create call sequences. We also use *Rand-APIRL*, a variant of our

approach that selects actions at random.

Evaluation Criteria. We use several different criteria in our evaluation. *Line coverage* (LoC) measures precise coverage performance. Coverage collection can be a non-trivial task depending on the system under test. We make use of standard libraries where possible to ensure consistently with prior work. In particular we use Xdebug [240] for PHP and Jacoco [121] for Java. For code collection in Python we use a hook function implemented by Atlidakis *et al.* [21]. This is implemented using standard libraries `sys` and `functools`. While this is a custom hook, it was designed specifically for collecting coverage of Python based REST API implementations.

Note that coverage cannot be collected from BitBucket as it is closed-source [167], and Spree Commerce does not support coverage collection across its multi-server architecture. As we have in previous Chapters, we use the *request volume* as recommended by Caturano *et al.* [39]. It represents how ‘intelligent’ an approach is, where better scanners achieve similar results in fewer requests. The *number of bugs* found in REST APIs is a key criteria and the goal of testing. Achieving high coverage of REST APIs may trigger logic flaws or server-side errors, as indicated by a 5XX status code. As defined in Chapter 5.2.1 unique bugs are 5XX responses from each operation that originate from different lines of source code. Detecting bugs this way provides a simple and effective oracle consistent with prior work [13, 21, 52, 141].

The experiment results are shown in Table 5.3, and the average improvement of APIRL over the respective baselines is shown in Figure 5.3. These results account for 1,035 hours of compute time (6 models tested 5 times on 3 APIs for 1.5 hours, and 3 APIs for 10 hours).

5.3.3 Coverage and Request Volume

Coverage is commonly used to determine tester performance, with higher coverage increasing test completeness to find more bugs. However, we also consider the *efficiency* of approaches in terms of the requests used.

APIRL outperforms the random baseline in all cases, covering at least 7.0% more code, using 15.4% fewer requests. The distributions in results are also different between APIRL and Rand-

Table 5.3: Average unique bugs, LoC, and number of requests taken on each test service, with standard deviation shown in brackets.

Test Service	Operations	Metric	EvoMaster	RestTestGen	DeepREST	ARAT-RL	MINER	Rand-APIRL	APIRL
Capital	16	Unique Bugs	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
		LoC	500.2 (21.7)	592.6 (9.9)	596.8 (19.6)	385.8 (74.2)	446 (0)	727.4 (11.3)	751.3 (11.1)
		Requests	1670.2 (310.7)	1074.6 (0.5)	1075.8 (1.1)	3437.8 (3574.9)	15785.4 (1113.5)	680 (0)	680 (0)
VAmPI	13	Unique Bugs	1.4 (1.1)	2.4 (0.5)	2.4 (1.5)	2 (1.4)	2 (0)	1.8 (1.3)	3 (0.7)
		LoC	245.8 (83.2)	327.6 (42.8)	278 (9.1)	272.4 (10.4)	277 (0)	327.2 (51.4)	382.8 (8.7)
		Requests	4564.6 (1190.6)	3990.6 (874.6)	5056.6 (1651.3)	2723 (2072.8)	10701.2 (150.7)	526.2 (11.8)	521 (4.4)
vAPI	17	Unique Bugs	5.5 (0.6)	6.4 (0.5)	7.2 (0.8)	2.2 (1.5)	6 (0)	7.6 (1.1)	9.5 (0.6)
		LoC	362.2 (1.6)	349 (0)	359.8 (6.6)	359.2 (17.9)	344 (0)	362.8 (16.5)	385.7 (6.5)
		Requests	629.75 (41.3)	1330.6 (68.5)	520.8 (249.6)	1375.6 (53.1)	1952.8 (500.7)	394.6 (70.4)	243.75 (14.2)
Spree	68	Unique Bugs	1.7 (1.5)	0 (0.0)	0.5 (0.6)	0.3 (0.6)	1.75 (1.5)	2.25 (1.0)	10.2 (0.8)
		Requests	1657.0 (99.0)	2034.2 (1.0)	2439 (396.0)	2169.3 (232.0)	16019.75 (2802.1)	1459.5 (78.0)	1128.6 (37.6)
BitBucket	518	Unique Bugs	6.2 (1.3)	0 (0)	3.6 (4.2)	3.4 (4.3)	6.3 (1.5)	5 (1)	6.6 (1.1)
		Requests	1315.8 (438.6)	1194 (0)	3168.2 (276.8)	1079.4 (595.3)	1485.25 (46.5)	10593.6 (11.9)	10562.8 (7.5)
WordPress	191	Unique Bugs	0.6 (1.3)	1.2 (0.4)	0 (0)	1 (1.0)	0.6 (0.5)	3 (2.8)	5.2 (3.5)
		LoC	5658.5 (514.1)	7843.8 (105.8)	7075.2 (29.7)	5295 (0.0)	5639.8 (331.6)	7995 (159.5)	8122.2 (117.5)
		Requests	6327.6 (5138.8)	2298.4 (35.3)	5835.4 (125.3)	5064.8 (889.7)	7373.6 (3380.3)	5685.3 (25.9)	5678.4 (55.4)

APIRL, as there is no overlap in their first standard deviation. APIRL always uses the same number of requests in c{api}tal as no bugs were found, resulting in episodes always reaching maximum length. MINER consistently achieves low coverage. Indeed, a lack of targeted mutations is apparent from the significantly high number of requests, as clear from both Table 5.3 and Figure 5.3. While ARAT-RL and RestTestGen uses minimal requests in two case studies their performance is inconsistent, as is their coverage performance. The evolutionary approach of EvoMaster reduces requests compared to heuristic counterparts. Yet, on average, it still requires 173.8% more requests than APIRL, resulting in 38.9% less coverage. DeepRESTs multi-learning approach doesn't impact the rate of requests it can send as it achieves a high number of requests when testing endpoints. Yet, DeepREST is unable to use this to find increased coverage, having a middling performance achieving 13.9% less coverage than APIRL, using 81.9% more requests. Overall, APIRL improves over RestTestGen, ARAT-RL, and MINER by 13.6%, 49.7%, 40.6% respectively in terms of coverage. The consistent performance of APIRL results in higher efficiency of coverage per request, on average 64.8% higher. The lower standard deviation of APIRL further demonstrating the *consistency* in targeting mutations to improve over both Rand-APIRL and state-of-the-art.



Figure 5.3: Improvements of APIRL compared to baselines.

5.3.4 Bugs Found and Request Volume

We now investigate the ability to find bugs in the systems under test. APIRL outperforms all baselines, finding at least 6.4% more bugs, and significantly more bugs on average (Figure 5.3). APIRL finds these bugs with lower or competitive standard deviation. In only two instances is APIRL's standard deviation greater than its random counterpart. Indeed, Rand-APIRL has an expectedly high standard deviation, which leads to an overlap in number of bugs found in BitBucket, VAmPI, and WordPress. Yet, Rand-APIRL's best case performance does not outperform APIRL, which finds more bugs in fewer requests. ARAT-RL displays the worst bug finding performance of all approaches, which combined with the high number of requests results in low bugs found per request. While EvoMaster and MINER find bugs across diverse REST APIs, they find 55.5% and 51.6% fewer bugs than APIRL respectively. RestTestGen inconsistently finds bugs due to its heuristic matching of parameter names for creating the call graph ODG. DeepREST, being based on RestTestGen, suffers from similar performance issues, finding 63.8% fewer bugs than APIRL.

In BitBucket we see an anomaly, APIRL uses more requests than other baselines. This is due to the larger number of REST API endpoints that do not contain bugs. In the configuration of APIRL during testing it has a maximum of 3 episodes and ten requests sent per episode. For example, in BitBucket, the upper limit of requests for the 518 operations is $3 \times 10 \times 518 = 15,540$. This upper limit could be reduced for Test Services with a larger number of operations, but to do so would not be a legitimate comparison in this evaluation.

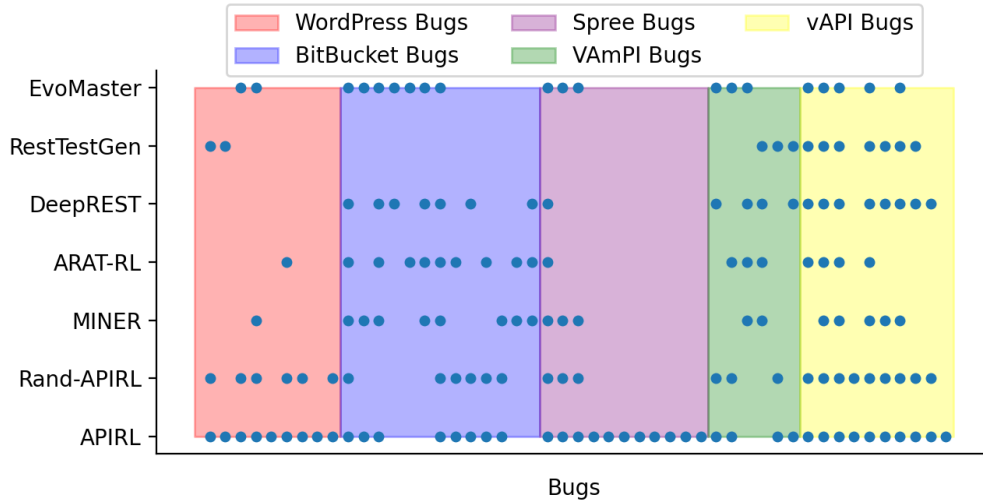


Figure 5.4: The unique bugs found by each approach per REST API tested.

Interestingly, in the other production grade REST APIs this behaviour is less prevalent. While in WordPress APIRL does not achieve the fewest number of requests it is competitive with all other approaches as MINER uses the most requests. In Spree we see a different story, as APIRL achieves the lowest number of requests of all approaches.

Using our state-action space improves results compared state-of-the-art as Rand-APIRL occasionally finds more bugs than APIRL. However, the learnt policy uses the state-action space effectively, by always finding more bugs in fewer test cases, highlighting the ability of RL to efficiently target mutations, even when testing on unseen REST APIs.

In Figure 5.4 we see the unique bugs found by each approach on for each REST API project. There are 49 unique vulnerabilities, of which APIRL also finds the most (42). APIRL misses only 7 of these, the least missed by any scanner by at least 31.8%, as Rand-APIRL misses 22 bugs, followed by EvoMaster which misses 29 bugs. Rand-APIRL achieves this performance thanks to it using the APIRL test framework, giving it the same state-action space. Being able to select actions that are all likely to trigger bugs, we reason gives it an ‘edge’ over the other testing strategies.

MINER, DeepREST, and ARAT-RL can find 5 of the 7 bugs missed by APIRL, but as we have seen this comes with an increased cost in the number of requests required, and a lower overall number of bugs found. Many other tools aim to chain API calls together to trigger

Listing 5.1: Mutations made by APIRL to parameter `user` that finds a bug in Spree Commerce.

```
1 "user": {"email": "spree@example.com", "first_name": "John", "
    last_name": "Snow", "password": "spree123", "
    password_confirmation": "spree123"}
2 "user": {}
3 "user": ""
4 "user": "*"

```

additional functionality, whilst APIRL tests endpoints one after another. Additionally, their mutation abilities include additional keys-values pairs in requests with default (and intentionally incorrect) values. As key values don't exist in the schema, this causes errors in the APIs when parsing the request. For example a request body including `{"fuzz":false}` causes a bug in BitBucket from the mishandling of the "fuzz" parameter. As APIRL is only able to include parameter keys which are present in the schema, it cannot trigger this functionality.

Interestingly, both of the two bugs in VAmPI are due to the chaining of API calls used in the approach of other tools. As a result other approaches call frequently to the endpoints: `users/v1/register` and `json/createdb`. The DB is then unable to handle the number of calls made to it to reinitialise the database and becomes inaccessible, throwing an error.

APIRL misses no bugs in WordPress, Spree, or vAPI. Additionally, 10 of the bugs found are unique to APIRL. APIRL is able to trigger these bugs in a number of ways, including altering the HTTP request type, using the existing parameters, removing key parameters in URLs causing errors when trying to redirect to endpoints that don't exist. APIRL learns to cast objects to alternative types, causing bugs in 7 endpoints in Spree which are missed by other approaches. A concrete example of this behaviour is seen in Listing 5.1. Such learnt behaviour helps APIRL to find bugs in the production grade software (*e.g.* WordPress and Spree), where other approaches struggle to find bugs.

Example Bugs.

APIRL can find bugs that have a number of different root causes, including: `NULL` errors, incorrect handling of undefined endpoints, re-routing errors, `Type` errors, `Index` errors, and even SQL errors from unique constraints, and undefined table columns. APIRL tries to access

parameter values that don't exist, leading to the `Null` and `Index` errors. It also triggers handling errors by inserting `Int` and `String` objects, and removes required parameters.

We now delve further into the types of bugs that can be found by APIRL. `Type` errors can be found by APIRL in different ways. For example in Spree Commerce and WordPress, APIRL casts `Int` or `String` values to `Array`'s. Which leads to unhandled exceptions when inbuilt functions attempt to cast the value to an integer. A similar bug is found in BitBucket by duplicating a parameter value in a request such as `"all": [true, true]` this triggers a `Type` error when the `Array` is cast as a `Bool`. In VAmPI APIRL manipulates the HTTP request to change the HTTP method leading to errors when redirecting to the requested operation, it also manipulates the request path to add, and remove, parameters to try and access undefined endpoints leading to unhandled exceptions.

APIRL has also learnt to trigger unhandled SQL errors. In VAmPI it inserts existing parameter values into the request that fail unique constraints. In Spree Commerce APIRL inserts an additional parameter that is included in the SQL query, leading to the query trying to access a column that does not exist.

APIRL performs a chain of mutations to find a bug in Spree Commerce's `api/v2/storefront/account` endpoint when trying to update a user through a series of `PATCH` requests. APIRL modifies the `user` parameter mutating it as in Listing 5.1: removing the `user` object (Line 2), casting it as an empty string (Line 3), then adding a wild card (Line 4) that bypasses input sanitisation to cause an error when trying to access the attributes of the user.

In Listing 5.2 we see a bug in WordPress's `POST wp/v2/posts/id` operation. This bug is triggered by inserting an additional `password` parameter into the body of the request (Line 6). Due to the incorrect type being an `Int` and not a `string`, WordPress correctly throws an error, however this results in an authentication check using the password parameter. Finally, this attempts to check the hashed POST password against the real password, which throws a fatal error.

²<https://core.trac.wordpress.org/ticket/61837>

Listing 5.2: Example request from APIRL that leads to a fatal error in WordPress².

```

1 {
2     "date": "2024-08-07T02:04:37",
3     "date_gmt": "2024-08-07T02:04:37",
4     "slug": "hello-world",
5     "status": "publish",
6     "password": 99999999,
7     "author": 1,
8     "featured_media": 0,
9     "comment_status": "open",
10    "ping_status": "open",
11    "format": "",
12    "sticky": false,
13    "template": "PE9PCU5W",
14    "categories": [],
15    "tags": []
16 }
```

These bugs, and the variety that APIRL finds in comparison to other scanners, demonstrate the reward function achieves the objective: policies that find bugs in unseen REST APIs, in a limited number of requests.

5.3.5 Ablation Study

We wish to determine the extent to which the core elements of APIRL contribute to its performance. Thus, we conduct an ablation study, presenting the results in Table 5.4. In particular, we vary rewards based on coverage (APIRL-cov) and response code (APIRL), we then ablate transformer embeddings (APIRL-m), and change the RL training algorithm to use PPO (APIRL-p), see Table 5.2 for hyperparameters. We will further present a feature importance study in Section 5.3.6 to further confirm the utility of transformer embeddings.

Reward Variations. Finding the correct reward function is fundamentally important in RL as it provides the feedback for learning. To prevent agents from finding exploitative strategies in complex reward functions we have designed simple reward functions [275]. However, the choice of reward for an RL agent is key to unlocking its potential. Little work has been done to study the effects of reward in real-world settings, such as web application testing. Prior work from Li *et al.* [159] studied how reward functions alter behaviour of RL agents when generating

code for compiler testing. Further work from Bates *et al.* [25] showed the challenges of designing rewards for RL applications in cyber-security.

Thus, we investigate reward functions for *manipulation* of HTTP request templates in automated testing of REST APIs. By using diverse sets of rewards based on different testing signals, we can study how subtleties in rewards influence learning policies that test real-world systems. Both coverage and status code are commonly used to provide feedback for RL agents, yet it is unclear how they affect learning [51, 141, 159]. Thus, here we evaluate multiple reward functions based off of these. This allows for a direct comparison between the effects of learning from coverage, and response code. We develop several reward variants:

- *A coverage based reward (Eq. 5.2).* Similar to R_{sc} , R_{cov} rewards most for increasing coverage, giving a smaller reward for recovering the same code, and penalises otherwise. We refer to this agent as APIRL-cov.

$$R_{cov} = \begin{cases} 10, & \text{Newly executed}(LoC_{new}) \\ & \text{code} \\ 1, & \text{Repeating code } (LoC) \\ -1, & \text{Otherwise} \end{cases} \quad (5.2)$$

- *A sparse uniform reward.* We train APIRL-u by rewarding 1 for 2XX and 5XX response codes and -1 otherwise. Equally for APIRL-cov-u we train a coverage variant that rewards 1 for new unique LoC and -1 otherwise. These rewards compare the difference between sparse rewards without the use of a ‘large’ reward for the overall objective (finding bugs). Such rewards are commonly used in the literature to train RL agents for security testing, making them an ideal candidate for evaluation in our ablation [71, 153, 171, 262].
- *A reward ratio.* We train APIRL-r using a reward in the form of status code ratio: $r = \sum(5XX + 2XX) / \sum(XXX)$. We also train APIRL-cov-r using $r = \sum(LoC_{new}) / \sum(LoC)$. Ratio based rewards are also used in the literature to train RL models for testing tasks, again making a class of rewards important to compare [36, 159, 160].

Table 5.4: Ablations of both APIRL and APIRL-cov. Lighter colours represent poor performance, while darker colours represent good performance.

APIRL Variant	Successful Requests	Error Requests	Invalid Requests	Coverage	LoC per Request $\times 100$
Rand-APIRL	88048	1019	14423	81.3%	6.575331
APIRL-r	83756	1533	6930	87.2%	7.914464
APIRL-u	91597	1436	2805	92.4%	8.069743
APIRL	90079	1591	2805	95.6%	8.469669
APIRL-m	90535	1562	2421	87.2%	7.721958
APIRL-p	100086	728	0	85.7%	7.115172
APIRL-arat	90222	1037	4531	90.7%	7.925243
APIRL-cov-r	93748	5256	5987	92.4%	7.366231
APIRL-cov-u	92150	5499	7961	97.6%	7.735177
APIRL-cov	87633	11197	6138	96.4%	7.670852
APIRL-cov-m	91936	4562	3498	92.4%	7.734189
APIRL-cov-p	104979	11	11	90.8%	7.237988

- *An ARAT-RL style reward.* Similar to R_{sc} , Kim *et al.* [141] design a status code based reward that gives 1 for 4XX and 5XX responses, penalising -1 for 2XX. Thus, we wish to compare with the same reward function. The model trained with this reward is APIRL-arat.

Effect of Reward Function. The variations of APIRL-cov use a higher number of requests than APIRL equivalents, which reduces the coverage per request. As APIRL models try to find bugs quickly they terminate episodes early, which in turn increases efficiency of coverage per request. Such results suggest that APIRL-cov models send valid requests to increase coverage, not to find bugs, which results in the higher number of requests.

Compared to learnt policies, Rand-APIRL makes 108% more invalid requests (4XX). Rand-APIRL also achieves the worst coverage and coverage per request of any model.

APIRL and APIRL-cov-u achieve the highest code coverage for each reward function type (request based, and LoC). Additionally, APIRL finds the most error requests (5XX) out of APIRL models, indicating both the breadth and depth of the learnt model. The reward based ratio

(APIRL-r and APIRL-cov-r) achieves the low coverage and successful requests (2XX) in both reward ablations. This is likely due to diminishing returns, *i.e.* the delta introduced by a single step has small impact compared to the denominator as training progresses. Uniform rewards for APIRL-u, APIRL-cov-u show the inability to differentiate between successful and error requests as they result in high numbers of invalid requests and lower errored requests. Similarly, APIRL-arat’s reward results in the model being unable to distinguish between finding bugs, and the undesirable behaviour of invalid requests. APIRL-cov finds the most bugs of the ablations (89% more than the next APIRL-cov variant). However, it achieves lower coverage in comparison to APIRL-cov-u, as APIRL-cov can still receive positive reward when recovering the same code.

These results highlight the intricacies of reward functions, showing how even well considered rewards may not yield as good of an outcome. We see that coverage based rewards outperform their request counterparts in terms of coverage, though they may take additional requests to do so. Furthermore, APIRL outperforms the request rewarded models achieving more coverage, error requests, and LoC per request than other reward functions.

Effect of Transformer. The ablation of transformer embeddings in APIRL-m leads to an 8.4% reduction in coverage. Similarly, APIRL-cov achieves a higher coverage compared to APIRL-cov-m.

Thus, as in Table 5.4, transformer embeddings lead to the highest efficiency of LoC per request. The non-transformer variants perform comparably in terms of successful requests. However, both transformer based models find more bugs, with APIRL-cov finding the most errored requests of any model. These results showcase the utility of using the structured HTTP responses for feedback in learning. As policies learn to effectively maximise the learning objective (finding bugs, or increasing coverage).

Effect of RL Training Algorithm. Using PPO leads to minimal invalid requests, and has the highest number of 2XXs. However, it finds the fewest bugs and the lowest coverage, 6.8% and 9.9% lower coverage than APIRL-cov and APIRL. Upon manual inspection it is due to the PPO model entering local-optima, replicating action sequences, rarely deviating from

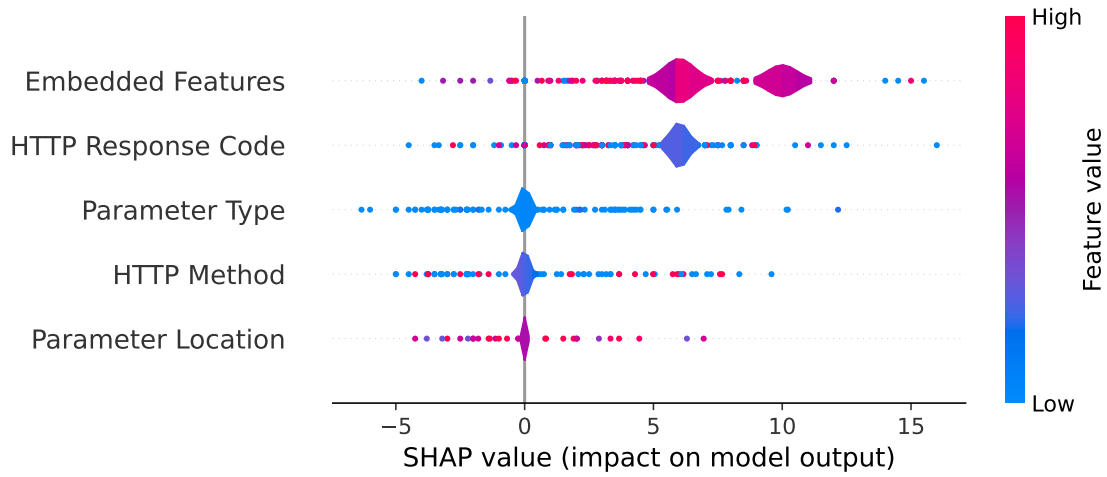


Figure 5.5: SHAPley Values for all features in descending order of importance for APIRL.

these to maximises successful requests. We speculate the off-policy nature of DQN learns a general mutation strategy, while PPO struggles to adjust over the curriculum, similar to the performance observed in Chapter 3. Specifically, the replay buffer in the DQN architecture provides a history of experiences, resulting in a greater degree of generalisation over the different REST API endpoints.

5.3.6 Feature Importance Analysis

We also investigate how APIRL makes use of the state representation via feature importance analysis. We use the well known SHapley Additive exPlanations [170] (SHAP) framework used in our feature importance analysis. It uses a game theoretic approach to determine the importance of input features on model outputs, doing so in an implementation agnostic way. SHAP connects the optimal credit allocations and local explanations to determine *SHAPley values*, providing a way to determine the contribution of individual features within the complete feature space [170].

Specifically, we use the states seen in testing to compute SHAPley values representing the impact of each feature, and how their variance effects the RL agent. Figure 5.5 shows how each variable influences APIRL, from most to least influential in descending order. APIRL shows a positive correlation (particularly in the highest ranking features), indicating the policy

uses these optimally to create successful requests. The deep embeddings from the transformer rank as most influential, indicating that this representation of additional response information, *e.g.* ‘value missing’ or ‘Example is invalid’, provides meaningful granular feedback beyond a simple 400 HTTP code. It is unsurprising that the HTTP response code is second with a wide distribution, indicating high influence. This shows that the policy has learnt the response code is important to dictate strategy, correlating with its importance in previous works [21, 93]. The remaining features (Parameter Type, HTTP Method, and Parameter Location) then have a smaller distribution and influence. This is due to relatively fewer number of actions that can effect these features compared to other actions. Thus, these features are less influential than the Embedded Features and HTTP Response code.

5.4 Limitations

To train APIRL there are two components that require separate training: the RL agent, and the transformer. Any training set will bias a neural networks final performance. To limit bias and overfitting we use data from a diverse range of real REST APIs in the training set of APIRLs transformer.

We use a pre-trained element in the APIRL framework as transformers in RL can lead to training instability, requiring longer training, if they converge at all [219]. We also select an open-source example for training the policy element of APIRL. This is with the intention that a trained agent can be used to test other REST APIs without the high number of training iterations that are required to reach optimal performance. While larger, more complex REST APIs could be used, the empirical results demonstrate the performance of the agents from training on the didactic training set used. We also do not test on live REST APIs due to ethical concerns over manipulation of API data or potential service outages. However, we use production grade REST APIs locally in an effort to bridge the gap.

APIRLs action set is also constrained. In particular in regard to the ability to place in additional key-value pairs in the request. Other approaches can add in targeted or guessable key-value

pairs using known heuristics or generative methods [51, 176]. To add additional ways to generate key-value pairs for APIRL are possible, but would require more engineering effort. As a result, given the performance of APIRL in Section 6.4 we believe adding in this functionality serves limited purpose.

5.5 Related Work

Diverse techniques have been used to test REST APIs. One of the first automated works on fuzzing REST APIs was presented as a stateful fuzzer, RESTler [21, 93]. Starting from a single request this approach incrementally builds valid request sequences. NLP techniques have been shown by Kim *et al.* [140] to enhance the OpenAPI specification, and to successfully generate parameters for REST API testing in MINER [176]. RestTestGen [52] traverses an ODG to develop call sequences. EvoMaster [13] presents an evolutionary algorithm that uses only the HTTP response in black-box settings to guide its fitness function. By comparison, APIRL parses direct feedback to a latent representation resulting in significantly more bugs in fewer requests.

ARAT-RL [141] uses separate Q-tables to prioritise parameters and value-mapping functions when testing REST APIs. ARAT-RL also uses a reward based on response code, however our extensive experiments show that less granular rewards, such as those used by ARAT-RL can harm performance. Additionally, APIRLs deep architecture learns which parameters to include, how to manipulate values, and how to alter the HTTP method and auth token.

DeepREST [51], is a concurrent work that uses a multi-learning architecture to test REST APIs. It uses a PPO based agent to select different operations. DeepREST then uses a multi-armed bandit *per parameter* to select data values from a variety of different sources including: user provided data, previously seen data, and LLM generated data. Finally, they apply mutators to the selected data by random selection. APIRLs architecture differs significantly from this by using a single deep learning element to select parameters or apply mutations. APIRL also only requires training on a single REST API and can then test other APIs, whilst DeepREST

requires training on each REST API.

Other RL approaches for fuzzing have used off-the-shelf architectures, to find bugs in software [36, 159, 160]. These works use similar rewards based around code coverage, expressing it as a function of *how much* new coverage is achieved. Our experiments suggest this can lead to diminishing returns as the total coverage achieved increases, placing greater weight on the rewards achieved early in training. APIRL demonstrates the utility of consistent rewards.

RL has also been used for automation tasks, often using simple heuristics [36, 159, 160, 323] or manually defined features [84, 153]. RL has even been used to test GraphQL APIs for denial-of-service attacks by McFadden *et al.* [184]. Yet, unlike APIRL, these approaches are limited in the feedback they can use. Specifically, they are unable to a) use input of unbound length, b) use diverse input that contains subtleties relating to the test case, and c) generalise to unique, unseen request-templates without the need for retraining.

5.6 Takeaways

This Chapter has aimed to test the ability of RL to perform an important task in web application security testing: *mutation* of existing test cases (in this case a HTML template). Due to their extensive use, testing REST APIs is key to ensuring the continued functioning of web infrastructure. However, existing approaches in this domain lack the ability to effectively use feedback from test cases. Such challenges arise from the diverse and complex nature of different endpoints, their responses, and their behaviours. As a result, the main form of feedback comes from simple heuristics that may not fully capture the subtleties of the problem.

We overcome the key challenge of using contextual information when testing REST APIs by creating a novel Deep RL architecture, APIRL. Building on the knowledge from previous Chapters we develop a deep representation in the form of a pre-trained transformer to harness the JSON and natural language in the HTTP responses from API endpoints. We have seen that APIRL can outperform existing state-of-the-art tools, including both deep learning, and

RL based approaches. APIRL consistently achieves higher code coverage and finds more bugs than the state-of-the-art, in a lower request budget.

We have investigated how different rewards can impact learning of RL agents. We show that rewards based on ratios, unless carefully implemented, can lead to diminishing returns for the agent, effectively weighting rewards (and by association state and action transitions) earlier in training.

In this Chapter have seen how an RL agent can learn yet another common security testing technique, by developing mutation strategies on pre-existing test cases. This finding is in addition to what we have observed in Chapters 3 and 4, demonstrating the utility and performance of RL for web application security testing. However, there is an additional application of RL for this domain which we will investigate in the next Chapter: *where* to perform mutations in test cases.

Chapter 6

Locating Effective Mutations in Test Cases when Fuzzing JavaScript Engines.

We have seen in Chapters 3 and 4 how RL can generate payloads to find vulnerabilities. We then saw in Chapter 5 how RL can apply mutations to existing test cases. However, there is a question related to applying mutations we have not yet explored: *where* to perform mutations in a testcase in a more effective manner. In the last Chapter APIRL could choose to iterate through the variables in a test case, but its main focus was to apply a series of mutations, *i.e.* moving to different variables was a by-product to facilitate APIRL to *apply* mutations. Hence, all but one action were related to the different mutations, not the locations they could be performed.

In this Chapter we will apply a Deep RL approach to the problem of optimising the locations in which to perform mutations in a test cases when fuzzing JavaScript engines. JavaScript is commonly used across the web, with a consistent year-on-year increase [123, 124]. This prevalence is in part due to the frameworks that allow for easy and lightweight deployment such as jQuery, React, and core-js. JavaScript is a dynamically typed, prototype-based object-oriented language, allowing for flexibility of types and properties during runtime as the code

executes. It is used for a variety of tasks in modern websites, including: animations, logging, content loading, and API calls. Due to the large volume of JavaScript on the web, and its varied functionality JavaScript engines use Just-In-Time (JIT) compilers. These JIT compilers have become a focus of security concerns, due to the rising number of vulnerabilities they contain [220].

Fuzzing of JavaScript engines has spawned different methodologies ranging from neural networks for code generation [152], ensuring semantic and syntactic consistency [107, 220, 248], using AST representations [48, 296], and generation via structured grammar [16, 295].

However, no such work has investigated where to perform such mutations in test cases, as there are several core challenges: 1) the large search space of test cases, and 2) the complexity of JavaScript. We expand on these challenges in Section 6.2.4. We address these challenges by introducing CLUTCH, a *deep contextual bandit* that can learn from feedback on test cases. This means we can leverage greybox information such as coverage as a signal for learning, in addition to information on the test case produced such as syntactic validity or complexity.

We demonstrate the utility of CLUTCH by embedding it inside three state-of-the-art JavaScript fuzzers, comparing approaches focusing on their *efficiency* over the number of test cases they are able to produce. We show that CLUTCH, produces 20.3% more valid JavaScript, covers 2.3% more branches, and increases coverage per test case by 8.9%. We study what CLUTCH has learnt by taking 10,000 test cases and analysing where it has learnt to apply mutations. We further compare against an alternative contextual bandit approach to demonstrate the advantages of our approach.

Chapter Contributions.

This work is currently under review for conference publication. It was in collaboration with co-authors from the National Institute of Information and Communications Technology, who were often present at meetings, and could contribute to the general discussion. I developed and implemented the CLUTCH framework, and designed, performed, and created evaluation metrics for all experiments.

6.1 Overview

Recent research has focused specifically on the JIT compiler of JavaScript Engines to target the deeper functionality and harder-to-find vulnerabilities contained within. Several of these approaches are based on Fuzzilli [96], which provided modular and customisable templates for generating JavaScript for particular engines; in addition to mutations designed to trigger JIT compilation. This fuzzer has been extended in multiple directions, including differential testing in JIT-Picker [28], and with custom JIT triggers in-the-testing-loop and a non-crash oracle in FuzzJIT [299].

Broadly, these approaches generate new JavaScript and then mutate test cases to increase the diversity in the hope of triggering new paths in the JavaScript engine and find potential vulnerabilities or bugs. However, while they focus on developing new mutations, testing heuristics, or oracles, they apply mutations at random locations within test cases. Behaviour such as this reduces the number of successful test cases, and leads to inefficient search of the test subject. Work from Böhme *et al.* [35] has shown that most test cases trigger the same few shallow paths, as generated inputs after mutation are often invalid.

There is good reason to apply such mutations at random, as it is challenging to select a given location in a test case. For example, the test case in Listing 6.1 causes a bug due to the following behaviour. On line 4 `v14` is assigned as a `BigInt`, and on line 6 a call to `asUintN` always throws an error as the result is too large. The optimiser then does not assign the result to `v15`, leaving it in the cache. Finally, due to the repeated calls from the `for` loop this causes a segmentation fault in two separate instances¹². However, if variable `v14` on line 4 was mutated to an other type instead, *e.g.* `Int` this would not trigger the crash.

Selecting a given location to mutate in a JavaScript program requires a deep understanding of its structure, intended behaviour, and mutation to be applied. To understand JavaScript fundamentals alone is challenging as it is defined by 15th ECMA standards, which comprises over 800 pages of technical detail [68]. Thus, to design any heuristic for such purpose, or indeed

¹https://bugzilla.mozilla.org/show_bug.cgi?id=1747777

²https://bugzilla.mozilla.org/show_bug.cgi?id=1745907

Listing 6.1: Javascript test case that causes a segmentation fault in SpiderMonkey².

```

1  for (let v3 = 0; v3 < 100; v3++) {
2      function v4(v5,v6) {}
3      function v7(v8,v9) {
4          v14 = BigInt(-1.7976931348623157e+308);
5          try {
6              v15 = BigInt.asUintN(639625512,v14);    // Always
                                                       throws an error so optimises it away without
                                                       clearing the cache leading to a segmentation fault.
7          } catch(v17) {}
8      }
9      v18 = v4.prototype;
10     v18.b = v7;
11     v19 = new v4();
12     v20 = v19.b();
13 }

```

static analysis techniques, one would have to handle an immense search space, ignoring many edge cases.

Instead, we aim to utilise greybox information such as coverage and test case information (*e.g.* syntactic validity or complexity) as a mechanism to improve location selection via RL. However, this is a non-trivial task with several challenges. First, the agent must select a location in a given JavaScript test case, receive an immediate indicator of performance, and then receive a new test case to begin the process again. Second, the RL agent must be able to generalise to the large state-action space of potential test cases generated. Third, given a dynamic and variable length input representation the agent must select a location from the input. Forth, we must be able to dynamically control the rate at which we explore new possibilities, as the rate of exploration should change with the depth of testing into the JavaScript engines.

To overcome these challenges, we present the first study on how to target locations in test cases when fuzzing JavaScript engines. Particularly, we develop CLUTCH, a deep contextual bandit that given a representation of a JavaScript test case, and a mutation to apply, will select where to apply the mutation, immediately receiving feedback from test case performance that is used to guide learning via a reward. CLUTCH uses an attention-based neural network with Concrete Dropout, allowing it handle different sizes of JavaScript test case, and adapt its exploration as fuzzing progresses. We test CLUTCH by embedding it inside three existing state-of-the-art

fuzzers and test on the three most common JavaScript engines. We evaluate the mutation selection strategy learnt by CLUTCH to show how this links to its performance improvements.

6.2 Fuzzing JavaScript Engines

Fuzzing JavaScript engines is fundamentally no different from other fuzzing tasks, where the goal is to trigger abnormal behaviour, often corresponding to a bug or vulnerability in an application. Fuzzing tools either tend to generate new test cases or mutate existing ones, at random or following some predefined heuristics. As we have seen, code coverage is a common metric to evaluate the performance of test cases, providing a practical estimate of the portion of the search space explored. However, only a small fraction of test cases will find bugs in the test program, leading to a high number of executed tests [35]. Hence, a key objective for practical fuzzing solutions is to increase the efficiency of bug finding by reducing the volume of test cases that do not result in new behaviour or that are invalid (those that do not pass input validation) [156, 302].

There have been many works that focus on fuzzing JavaScript browser engines, either as stand-alone JavaScript engines [48, 96, 107, 220, 248, 296], or in context of the web browser [63, 143]. Prior work selects at random the location where to apply a mutation in JavaScript code, sacrificing efficiency for speed and simplicity. Our approach, called CLUTCH, instead uses RL to select the locations to mutate. CLUTCH is independent from the JavaScript engine, and the fuzzer. We will place it inside three state-of-the-art fuzzers for evaluation in Section 6.4. We detail these works here, motivating their different approaches, and the key elements that will be relevant to CLUTCH.

6.2.1 Fuzzilli

Fuzzilli [96] is a greybox fuzzer targeting crashes in the JIT compiler of JS engines. Fuzzilli leverages coverage feedback and semantic correctness preservation in order to improve its ability

Listing 6.2: Example source code of a JavaScript test case.

```

1  const v2 = new Uint32Array(256);
2  const v3 = [0, 1, 2, 3, 4, 5, 6];
3
4  for (const v4 of v2) {
5      ~v3[v4];
6  }
7
8
```

Listing 6.3: Fuzzilli IR of the JavaScript test case in Listing 6.2.

```

1  v0 <- LoadInteger '256'
2  v1 <- LoadBuiltin 'Uint32Array'
3  v2 <- Construct v1, [v0]
4  v3 <- CreateFloatArray [0, 1, 2, 3, 4, 5, 6];
5  BeginForOfLoop v2 -> v4
6      v5 <- GetComputedProperty v3, v4
7      v6 <- UnaryOperation '~', v5
8  EndForOfLoop
```

to fuzz. In particular, Fuzzilli applies mutations to the Intermediate Representation (IR) of the JIT compiler, instead of the JavaScript source code (see Listings 6.2 and 6.3), reducing the chance of semantically meaningless mutations.

Fuzzilli generates its own corpus of test cases, which are then mutated at the IR level. Key mutations include:

- **InputMutator**: replace one input instruction with another. For example, changing `v4` on line 6 of Listing 6.3 to `v0`.
- **InputMutator** (type aware): replace one input instruction with another, ensuring type preservation.
- **OperationMutator**: mutate parameters of an Operation (*e.g.* 256 on line 1 of Listing 6.3 could be replaced with 64).
- **CombineMutator**: insert part of a program into another, renaming variables to avoid collisions/syntax errors.
- **CodeGenMutator**: generate new, random code, placing this in the program, renaming variables to avoid collisions/syntax errors.
- **SpliceMutator**: copy a self contained part of a program within the corpus, placing this into another to combine their features.

These mutations are performed on IR instructions located *at random* in test cases, making them ideal candidates for prioritisation by CLUTCH.

6.2.2 FuzzJIT

FuzzJIT [299] is a tool based on the Fuzzilli architecture, but develops JIT-specific oracles and triggers. Specifically, FuzzJIT wraps each generated function in a wrapper that can be called repeatedly to trigger JIT compilation. The wrapped output can then be compared before and after JIT compilation to determine errors that result from the JIT. FuzzJIT implements its oracle to perform type-specific comparisons, using deep equality to find if there are differences in objects, arrays, or individual values, and in such cases reporting an error.

FuzzJIT uses a subset of the Fuzzilli mutators, of which many can utilise CLUTCH including: `CombineMutator`, `InputMutator`, `InputMutator` (type aware), and `OperationMutator`.

6.2.3 JIT-Picker

JIT-Picker [28] uses differential testing on the backbone of Fuzzilli to detect subtle non-crash bugs. It uses both the interpreter and the JIT compiler as separate bug oracles, using the fact that both demand the interpreted and compiled code to perform computations in a strict and substitutable way. This abstracts away from implementation-dependent behaviour, allowing JIT-Picker to perform differential testing in individual engines without need to instrument the JavaScript code being tested.

JIT-Picker also uses a subset of the Fuzzilli mutators including: `CodeGenMutator`, `InputMutator`, `InputMutator` (type aware), `CombineMutator`, and `OperationMutator`.

6.2.4 Challenges in Targeting Mutations

Mutational JavaScript engine fuzzers traditionally focus on developing new ways to perform mutations [48, 152, 220, 296], or new oracles to detect bugs and vulnerabilities [28, 96, 297]. Mutations introduce diversity in the testing set, which may increase coverage, and find bugs. In comparison, the problem of *where* to apply these mutations has been overlooked, and they are typically applied at random locations. This may be in part due to the difficulty of disentangling

the effects on performance of the applied mutation operator and the chosen location. To the best of our knowledge, there has been no attempt so far to target *where* to perform these mutation, either in JavaScript engine fuzzing [16, 48, 82, 96, 152, 220, 248, 296, 299, 315], or in the broader domain [43, 97, 156, 174, 242, 263, 283, 290, 315]. We detail below two key challenges.

Large search space

First, JavaScript is a highly structured and dynamically typed programming language. This means that a variable can have arbitrary types dependent on the point of execution. Thus, the combinations of possible programs that can be created in JavaScript constitutes an enormous search space [220].

Mutation based fuzzers, attempt to reduce the problem of creating test cases for fuzzing JavaScript engines by applying mutations to existing, valid, JavaScript. Conversely, some generation based fuzzers use a constrained set of rules, or grammars, to increase the proportion of valid samples. Yet, despite this, the search space of test cases that cover new paths or trigger bugs remains vanishingly small compared to the space of syntactically valid JavaScript [107].

Complexity of JavaScript

Fuzzers traditionally focus on throughput, trying to generate as many test cases as quickly as possible. They try to do this with some understanding of the ‘sensible’ behaviours that could lead to bugs. Such behaviours are generally understood by software testers and security practitioners, and used to form the basis of mutations. In performing these mutations, fuzzers generate a diverse range of JavaScript to stress an engine.

However, also accounting for the location of where to place these mutations makes the problem significantly harder. Consider again Listing 6.1: had a mutation on Line 4 caused variable `v14` to be of type `int`, the bug would not have occurred. We can retrospectively see that this was the ‘best place’ for the mutation by analysing the behaviour of the test case. Such post-hoc

analysis is not possible during fuzzing, as the mutation has been applied, and the test case executed.

Finding a good place for a mutation requires identifying locations, assign them preference, resolve conflicting candidates. The human effort to design such heuristics would require significant investment to handle to edge cases from the diverse JavaScript a fuzzer will inevitably produce.

6.3 Clutch Control

In this Section we introduce CLUTCH, a novel deep contextual bandit that learns the effective location to mutate JavaScript code to increase efficiency in fuzzing JavaScript engines. First we discuss considerations specific to designing a learning based system for this task, and then provide technical details of CLUTCH.

6.3.1 Multi-Armed Bandits for Location Selection in JavaScript

As discussed in Section 6.2.4, selecting an effective location to mutate is a challenging task that involves covering a large search space, deeply understanding JavaScript, and determining what affects the performance of test cases. Given these challenges, we believe RL may provide a solution, where we use a Multi-Armed Bandit (MAB) to interact with a JavaScript engine and receive feedback in the form of coverage.

The MAB based approach is motivated by the fuzzing problem itself. Consider the JavaScript in Listing 6.1. At timestep t , an RL agent selects a location to mutate in the test case. The mutated test case is executed, and feedback is received on its performance. At the following timestep, $t+1$, the agent must select a location in a different test case prioritised by a scheduler, irrespective of the location chosen by the agent in the previous step and test case. It follows that the code observed at each timestep is, in RL terms, independent.

The MAB is a classic RL problem (an extension of ‘one-armed bandits’ or slot machines) where different actions return to the same state, and there is only one state. The bandit must explore the actions to find the effective strategy to maximise reward over all test cases [275]. Framing the problem as a MAB is also advantageous for the computational complexity of fuzzing, as we need to call the bandit only once per test case, reducing overhead.

In the traditional MAB actions and observations do not require any kind of association. This means that in stationary problems the bandit learns the single ‘best action’, and in non-stationary tasks it learns the ‘best action’ over time [275].

In our fuzzing scenario, given a JavaScript test case and a mutation (the observation), at each timestep the bandit must select a location to perform the mutation (the action). Test cases have different structures and, as in the example in Listing 6.1, will have different locations to best perform a given mutation. Thus, we can consider each test case as a separate task, to optimise for coverage, bugs, or testing deeper into the JavaScript engine. As we fuzz, generated test cases will be included in the corpus increasing the number of tasks the agent has to learn. Therefore, it is important that we provide a *context* of the current test case to the bandit, enabling the association of actions to observations. As a result, we need a *contextual* bandit that understands the subtleties of the JavaScript test cases.

Contextual bandits learn to select one of a fixed number of arms (actions), given an observation. This is beneficial as the bandit can observe the effect of each of the actions it can take, and detect the shift between different tasks. Yet, the nature of fuzzing deviates from this format. The JavaScript test cases the bandit sees may be of different structure *and* size, thus the number of locations that it can select may vary significantly across test cases, as can the code structure and the relevant mutations. Our challenge is to rapidly adapt to the number of actions that we see at each timestep, while reducing the uncertainty in the location we select.

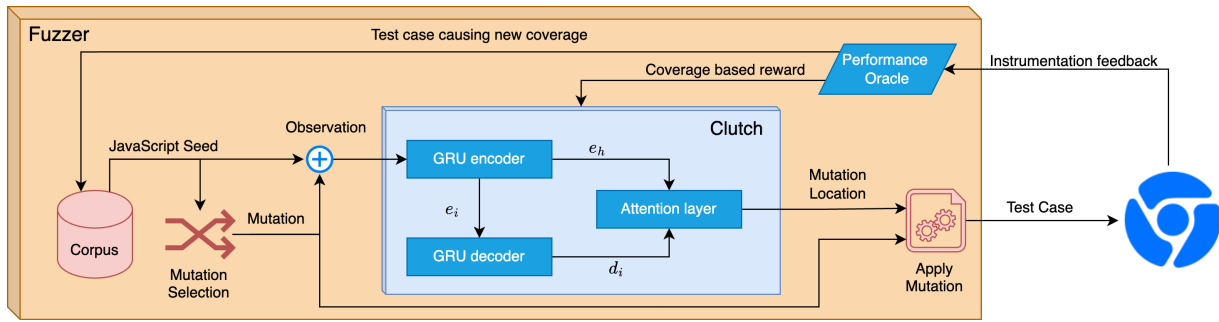


Figure 6.1: The CLUTCH architecture shown within a fuzzer.

6.3.2 System Design

In Figure 6.1 we show a high level system diagram of CLUTCH, our approach to overcome the challenges above. CLUTCH receives a representation of the test case and mutation in the form of an *observation*, and with its bespoke deep learning architecture it selects a location to perform a mutation. The fuzzer then performs the mutation and submits this to the JavaScript engine, collecting trace information from the instrumentation. Using this feedback, we provide a reward that CLUTCH uses to improve its location selection strategy.

Model Architecture

To handle the representation of JavaScript that is used in observations and actions for CLUTCH we use a deep learning architecture. Specifically we use a sequence-to-sequence model with attention mechanism in the form of a Pointer Network [291].

In CLUTCH we use a Gated Recurrent Unit (GRU) model as an encoder-decoder. GRUs have been shown to outperform other recurrent neural networks (RNNs) in web testing tasks [245]. We saw in Chapter 4 how GRUs can learn a useful latent representation for web application security testing. Indeed, the GRU handles a variable length input and produces an output of the same size. Yet, it does not directly provide the output location we require. However, we can use the encoder and decoder states $e_1, \dots, e_n$ and $d_1, \dots, d_m$, to compute an *attention* vector a^t over the locations available at each timestep t , where parameters v^T , W_1 and, W_2 are

learnable.

$$\begin{aligned} u_j^t &= v^T \tanh(W_1 e_j + W_2 d_j) \quad j \in (1, \dots, n) \\ a^t &= \text{softmax}(u^t) \end{aligned} \tag{6.1}$$

As a result, CLUTCH can take as input the IR of a JavaScript test case, that constitutes a number of locations, and output an attention value for each location (vector a^t). The location with the highest value can then be selected as the location to perform the mutation.

The last challenge is the rate of exploration that CLUTCH should use to learn over time. In previous Chapters and indeed traditional RL, this is often handled by an epsilon-decay algorithm, that reduces a probability, ε of taking random actions over time. However, to determine the optimal hyperparameters to use when fuzzing JavaScript engines would require large compute time and cost. Not to mention that we may want to adjust this randomness during testing to explore different parts of the engine. Instead, we automatically tune this exploration using Concrete Dropout. Concrete Dropout treats the dropout parameter p as a parameter to be trained via gradient descent [90]. To control the exploration-exploitation trade-off regularisation parameters, $r^1, \dots, r^4$, are used in the objective function. As the training increases and more data is seen, the dropout regularisation loss reduces as the uncertainty on the weights reduces [50]. This reduces the exploration, resulting in more exploitation. Equally, the network can have higher uncertainty in the weights which will increase the regularisation, effectively increasing the exploration.

Concrete Dropout is motivated by dropout, which randomly samples from a masked weight matrix in the neural network [50]. This dropout probability and weight matrix can be approximated to the probabilistic deep Gaussian process [57]. By taking this approximation we can apply a Bayesian deep learning model to a neural network of L layers and θ variations parameters. This allows us to place a prior on the network weights and then learn the posterior $q_\theta(\omega)$ distribution over the weights ($\omega = \{W_l\}_{l=1}^L$) [90]. This can be formally written as the objective function as in Eq. 6.2.

$$\hat{\mathcal{L}}_{MC}(\theta) = -\frac{1}{M} \sum_{i \in S} \log p(\mathbf{y}_i | \mathbf{f}^\omega(\mathbf{x}_i)) + \frac{1}{N} \text{KL}(q_\theta(\omega) || p(\omega)) \quad (6.2)$$

where S is a set of M data points, N is the number of data points, and θ is the optimisable parameters, $\mathbf{f}^\omega(\mathbf{x}_i)$ is the model output given $\mathbf{x}_i$, and $p(\mathbf{y}_i | \mathbf{f}^\omega(\mathbf{x}_i))$ is the model's likelihood.

The KL term $\text{KL}(q_\theta(\omega) || p(\omega))$ can be considered the 'regularisation' term to ensure that the posterior $q_\theta(\omega)$ and prior $p(\omega)$ distributions do not diverge too far. Gal *et al.* [90] show that the KL term can be approximated to depend only on the probability of turning off a Bernoulli random variable (node) with probability p . From this we can derive Concrete Dropout [90], where we continuously relax dropout's discrete mask.

Implementation

The implementation of this architecture is as follows. Program representations are passed through a single embedding layer. This embedding is then passed through a single GRU layer of width 64, to produce both encoder states and the terminal hidden state. The terminal hidden state is used to initialise the decoder, and give the decoder output. The encoder and decoder output are passed through the respective hidden layers W_1 , W_2 (of size 512) with Concrete Dropout applied. We are then able to have the model select the action from the attention vector (which is the predicted reward of the arm) a_t , and the log variance, φ_t .

$$\begin{aligned} a_t &= v^T \otimes (e_t \otimes W_1 + d_t \otimes W_2) \\ \varphi_t &= v^m \otimes (e_t \otimes W_1 + d_t \otimes W_2) \end{aligned} \quad (6.3)$$

where $\otimes$ represents multiplication and Concrete Dropout. We can then determine the loss function for CLUTCH following that of Gal *et al.* [90] to include the regularisation terms of each

layer $(r^1 \dots r^4)$ as follows:

$$l = \frac{1}{T} \sum_{t=1}^T (e^{-\varphi_t} + (r_t - a_t)^2 + \varphi_t) + \sum_{t=1}^T r_t^1 + r_t^2 + r_t^3 + r_t^4 \quad (6.4)$$

6.3.3 Actions and Observations

Actions are the ‘arms’ that a MAB can select. CLUTCH will select actions that correspond to the location we can mutate within the test case. As we have outlined it is important for CLUTCH to understand specifics of the JavaScript that will be mutated and the mutation that will be applied. Thus, at each timestep we give CLUTCH an observation of the available locations that can be selected and the mutation that will be performed. However, we prevent CLUTCH from selecting the mutation as an action by only sampling from the possible locations.

Both the actions and observations of CLUTCH are agnostic of the fuzzer for which it will select locations. This has the added advantage of reducing the overhead when selecting locations as no further processing is needed. For instance, the IR used by Fuzzilli provides a series of instructions, corresponding to locations that can be selected as an action. For CLUTCH, each of these instructions can be uniquely mapped into a vocabulary. An observation is the vocabulary representation of the IR (the set of locations), in addition to the mutation that will be applied.

6.3.4 Reward

Rewarding for MABs is similar to traditional RL: for each action taken we can assign a numerical value or reward. We have seen in Chapter 5 how different reward functions led agents to learn differing policies. In some cases rewards are inconsistent across training, due to their diminishing returns. This is particularly relevant when using coverage based rewards where a) the total possible coverage can be quite large, and b) the majority of coverage is often found early in testing. Hence, we design a reward for CLUTCH that is consistent across training.

$$R_{sc} = \begin{cases} 1 + \frac{CC_t}{CC_{max}} + |\max(\text{Var}_{i=0}^t(B_i) - \text{Var}_{i=0}^{t-1}(B_i))|, & \text{Syntactically Correct Test} \\ -1, & \text{Otherwise} \end{cases} \quad (6.5)$$

Where CC is the cyclomatic complexity and B_t represents the vector of branch coverage at time t . This reward function balances validity, complexity, and coverage. When a test case is not syntactically valid it penalises the choice, with a negative reward of -1 . When test cases are valid, it rewards 1 for the validity, additionally rewarding for how complex the test case is, as a ratio of current cyclomatic complexity (CC) and the maximum CC seen historically.

Finally, Eq 6.5 rewards for coverage using a term that incentivises the agent to explore new branches. At each timestep we compute an updated running variance of each branch ($b \in B$) representing this as a vector [41], $\text{Var}_{i=0}^t(B_i)$. In doing so we can estimate the variance of each branch without retaining all the historic branch coverage. We then take the difference in variance between time t and $t - 1$, isolating the difference in variance (novelty in coverage) from the test case at time t . The maximum value indicates the largest difference in coverage compared to the previous timestep, *i.e.* where branches which are less frequently covered. We then take the absolute value to reward for changing the variance. Meaning that CLUTCH is incentivised not to hit the same branches frequently, but instead to explore other branches, including lower frequency branches.

6.4 Evaluation

In this Section we evaluate the effectiveness of CLUTCH at improving the performance of state-of-the-art fuzzers, we compare it to a simpler CLUTCHLITE alternative, and we analyse the underlying behaviours that CLUTCH has learnt.

6.4.1 Experimental Setup

Baselines

At its core, CLUTCH is a location-based selection methodology that can be applied to different JavaScript engine fuzzers. We evaluate its effectiveness by inserting CLUTCH into three different state-of-the-art JavaScript engine fuzzers: Fuzzilli [96], FuzzJIT [299], and JIT-Picker [28]. We refer to these variants as FClutch, FJClutch, and JPClutch respectively. We did not include in the comparison OptFuzz [297], another extension of Fuzzilli, as the available prototype relied on a version of `llvm` incompatible with current operating systems and JavaScript engines.

Implementation

We implement CLUTCH in Python 3.11 using PyTorch. We use the PythonKit [289] framework to interface CLUTCH with the fuzzers above (implemented in Swift), so they can share JavaScript representations, coverage information, and locations. Locations in each of these models are represented by the IR instructions used in each of the three fuzzers we place CLUTCH into. We run experiments on two identical machines running Ubuntu 24.04 with Intel i7 8700k and 64GB RAM.

JavaScript Engines

We test CLUTCH on JavaScriptCore (used in Webkit and Safari), V8 (used in Chrome and Chromium), and SpiderMonkey (used in Firefox). These three comprise the main JavaScript engines in use, and have been targeted by previous work from academics, security researchers, and project maintainers [28, 96, 107, 152, 220, 248, 296, 297, 299].

Metrics

To assess the performance of CLUTCH we compare across several key metrics, with the particular view to understanding *efficiency*. We include the *throughput* of each fuzzer, to see the volume of

test cases produced and, by association, the fuzzer overhead. We consider the *correctness* of the test cases, *i.e.* the test cases that are valid JavaScript, not causing one of the four error types of JavaScript: `SyntaxError`, `ReferenceError`, `TypeError`, and `RangeError`. Generating JavaScript that does not cause such errors results in test cases that pass initial semantic and syntactic checks. Testing deeper into the engine triggers less frequently used paths, a common metric to measure this is code *coverage*. There are many different forms of coverage that one can measure, including: line, branch, and block. Each of these offers different levels of granularity of the test subject. We use branch coverage as our metric as it is often used in the literature, providing a point of comparison with prior work [28, 96, 299].

Experiment Design

Each trial (JavaScript engine and fuzzer) is run for 24 hours and repeated 5 times, overall constituting 3,240 CPU hours across all experiments. Repeating experiments reduces the effect of randomness that is inherent in JavaScript engine fuzzing approaches. Additionally it allows for statistical testing, as advocated for by Schloegel *et al.* [250] and Klees *et al.* [146]. In particular, we use the Mann-Whitney U test to determine the statistical significance, and Vargha and Delaney’s $\hat{A}_{12}$ test to quantify the magnitude of difference or effect size. The performance is considered significant when $p < 0.1$, and outperforms with a large effect size when $\hat{A}_{12} \geq 0.71$.

Ablation

In Section 6.4.4 we compare CLUTCH with CLUTCHLITE, a contextual bandit architecture that does not use deep learning. CLUTCHLITE uses the Adaptive Contextual Combinatorial Upper Confidence Bound (ACCUUB) algorithm [200], which similarly to CLUTCH can handle a variable number of actions. We cannot compare with other contextual bandits such as Lin-UCB [158] and Thompson Sampling with online bootstrap (BTS) [67] because they only handle a fixed set of actions.

Table 6.1: Correctness of generated test cases (five run average) from 24h fuzzing of CLUTCH and baseline fuzzers.

Engine	Metric	FuzzJIT	FJClutch	Fuzzilli	FClutch	JIT-Picker	JPClutch
V8	Average	66.4%	80.5%	65.9%	79.3%	73.3%	85.8%
	Relative Change		8.321%		20.32%		8.585%
	$\hat{A}_{12}$		1.000		1.000		1.000
	p		0.011		0.002		0.011
SpiderMonkey	Average	59.8%	77.4%	64.9%	77.5%	76.5%	89.5%
	Relative Change		9.843%		19.35%		8.615%
	$\hat{A}_{12}$		1.0		1.0		1.0
	p		0.008		0.003		0.036
JavaScriptCore	Average	70.5%	82.1%	63.7%	78.7%	71.8%	85.1%
	Relative Change		5.054%		23.52%		12.23%
	$\hat{A}_{12}$		1.000		1.000		1.000
	p		0.008		0.001		0.008

6.4.2 Field Test Comparison

We now measure the effect of embedding CLUTCH into existing JavaScript fuzzers.

Correctness

We will first measure the ability of CLUTCH to improve the rate of generation of valid JavaScript code, that is code that does not cause one of the four JavaScript error types. We report the validity results in Table 6.1.

All CLUTCH variants achieve a higher proportion of valid test cases compared to their non-CLUTCH counterparts. Indeed CLUTCH can quickly learn from its experiences to find strategies that improve the rate of valid test cases. Impressively, this performance is consistent across different fuzzing approaches, from differential testing to JIT-specific mutators. CLUTCH improves correctness by at least 16.42% (compared with FuzzJIT), and on average by 20.3%. In all cases we show that these results are statistically significant, and have a large effect size.

JIT-Picker and JPClutch achieve the highest rate of correctness of baseline fuzzers, and CLUTCH variants, respectively. JIT-Picker is designed for differential testing, having a focus on instrumentation and probing. It uses a subset of mutators from prior work [96], which causes this

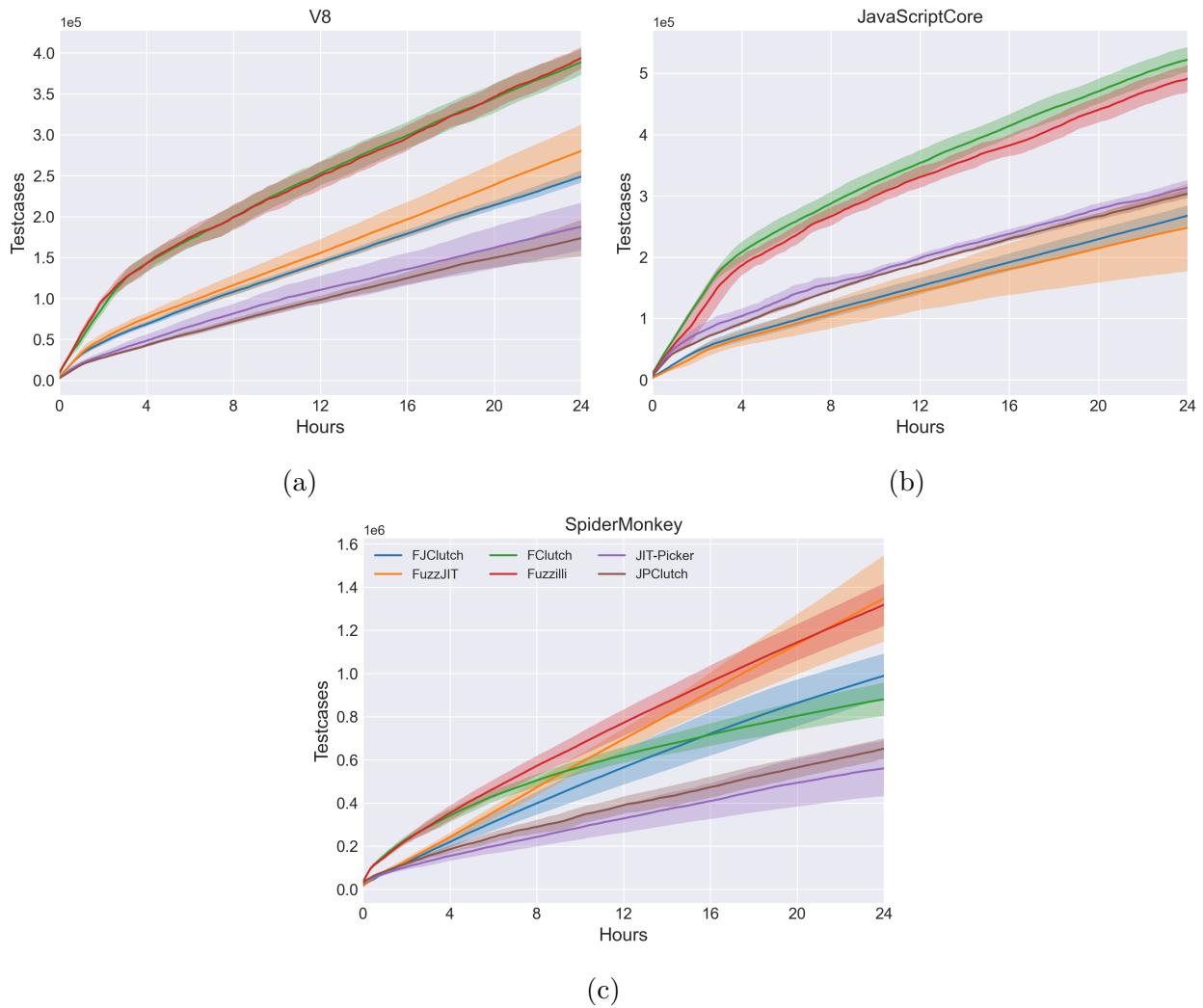


Figure 6.2: Throughput of test cases (five run average and standard deviation) from 24h fuzzing of JavaScript engines.

improvement in performance compared to the other approaches.

Throughput

The throughput of a fuzzer is important to consider as, typically, the more test cases we execute the higher the likelihood of triggering a bug. We monitor the throughput of each fuzzer, reporting the results in Figure 6.2. We would expect that CLUTCH variants will take longer to execute a single test case compared to their relative counterparts, as calling an external implementation of a neural network in Python will inevitably introduce a higher overhead compared to a random selection implemented directly in the fuzzer. Increasing the rate correctness of test cases may also reduce throughput: by having test cases that are valid they test deeper into

Table 6.2: Branch coverage (five run average) from 24h fuzzing of CLUTCH and baseline fuzzers.

Engine	Metric	FuzzJIT	FJClutch	Fuzzilli	FClutch	JIT-Picker	JPClutch
V8	Average	11.26%	11.08%	10.69%	10.67%	6.854%	7.212%
	Relative Change		-1.62%		-0.16%		5.224%
	$\hat{A}_{12}$		0.440		0.640		0.640
	p		0.099		1.000		0.953
SpiderMonkey	Average	21.30%	20.73%	24.50%	22.70%	16.18%	16.97%
	Relative Change		-2.68%		-7.33%		4.898%
	$\hat{A}_{12}$		0.000		0.000		0.667
	p		0.055		0.001		0.112
JavaScriptCore	Average	17.64%	18.21%	16.49%	16.64%	14.58%	14.55%
	Relative Change		3.229%		0.923%		-0.18%
	$\hat{A}_{12}$		1.000		0.680		0.520
	p		0.859		0.594		0.755

the engine, and in doing so, they take longer to execute.

Overall, Figure 6.2 shows that each of the fuzzers have a different rate at which they execute test cases (similar to prior work), and that CLUTCH versions have a lower throughput.

Coverage

Coverage is often a primary metric for evaluation of fuzzers. We show the total branch coverage in Table 6.2.

Figure 6.3 shows the average coverage in 24 hours of fuzzers and CLUTCH variants on each of the tested JavaScript engines.

We observe a limited and mixed effect on overall coverage by adopting CLUTCH, as shown by the average and relative change percentages, and the small values of $\hat{A}_{12}$ in four instances. Still, 4 of the CLUTCH based models do outperform the fuzzer baselines in terms of coverage achieved. In fact, on average, across all combinations of fuzzer and JavaScript engine CLUTCH improves the coverage by an average of 2.3%.

Given the lower number of test cases that CLUTCH variants execute, this aggregate performance is remarkable. CLUTCH leverages its ability to select the location for mutations, improving the efficiency of testing. To perform a more legitimate comparison on this efficiency we compare the coverage relative to the number of test cases executed in Section 6.4.2.

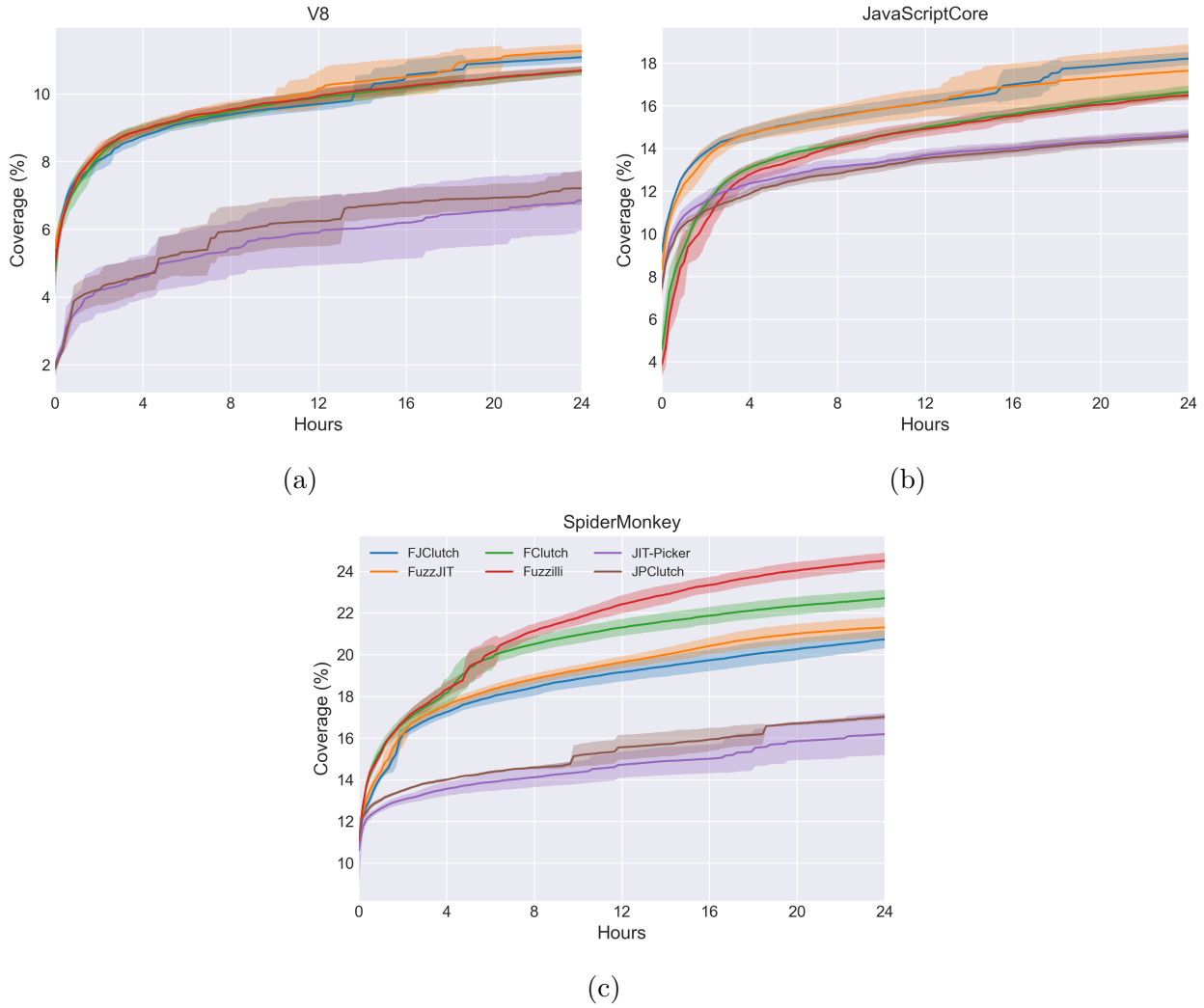


Figure 6.3: Branch coverage (five run average and standard deviation) from 24h fuzzing of JavaScript engines.

Test Case Efficiency

Recent work from Caturano *et al.* [39] has argued in favour of RL for its testing efficiency, stating that ‘more intelligent’ scanners achieve the same (or better) results with a given number of attempts. We report the efficiency as coverage found per test case in Table 6.3. This allows us to determine how efficient each approach is with its test case budget.

We can see that CLUTCH is generally able to improve the number of branches covered per execution, by an average of 8.9%. Indeed, CLUTCH increases the branches covered per execution *consistently*, doing so in six of the nine combinations of fuzzer and JavaScript engine. Specifically, each CLUTCH variant improves over the baseline fuzzer in two of the three engines. When

Table 6.3: Branch coverage per generated test case (five run average) from 24h fuzzing of CLUTCH and baseline fuzzers.

Engine	Metric	FuzzJIT	FJClutch	Fuzzilli	FClutch	JIT-Picker	JPClutch
V8	Average	0.566	0.627	0.382	0.387	0.514	0.585
	Relative Change		10.81%		1.152%		13.72%
	$\hat{A}_{12}$		0.560		0.680		0.880
	p		0.099		0.513		0.594
SpiderMonkey	Average	0.050	0.067	0.059	0.082	0.092	0.081
	Relative Change		32.55%		38.56%		-11.9%
	$\hat{A}_{12}$		1.000		1.000		0.222
	p		0.001		0.001		0.287
JavaScriptCore	Average	0.530	0.508	0.250	0.238	0.344	0.358
	Relative Change		-4.25%		-5.01%		4.12%
	$\hat{A}_{12}$		0.000		0.200		1.000
	p		0.768		0.055		0.096

comparing per JavaScript engine we see that CLUTCH always increases efficiency when fuzzing V8. This is due to a combination of the size of the engine and the throughput of the fuzzer approaches. As the size of V8 is larger compared to JavaScriptCore and SpiderMonkey, the search space for CLUTCH is larger, giving it an advantage to target it efficiently. The throughput on V8 is also similar between CLUTCH variants and baseline fuzzers in comparison to the other JavaScript engines, which also accounts for efficiency improvements of CLUTCH, as baseline fuzzers achieve less coverage per test case. Both FJClutch and FClutch have marginally lower efficiency in JavaScriptCore (-4.63% on average), and significantly greater efficiency in SpiderMonkey (35.56% on average). Yet, JPClutch does not follow this same behaviour, achieving 11.9% fewer branches per execution on SpiderMonkey, and 4.12% more on JavaScriptCore. This arises from the different approaches in fuzzing taken by JIT-Picker compared to Fuzzilli and FClutch, which leads to the highest rate of valid JavaScript generation from both JIT-Picker and JPClutch. As a result JIT-Picker has a higher base efficiency compared to other fuzzers.

We can see from Vargha and Delaney’s $\hat{A}_{12}$ values that the effect size is often large. Similarly, we can see that when CLUTCH outperforms a fuzzer the result is often statistically significant, as indicated by $p < 0.1$.

6.4.3 Clutch Selection

We have seen that CLUTCH improves the efficiency of fuzzing by increasing the correctness of test cases, their coverage, and the coverage per test case. However, it is important to consider the underlying cause. To investigate this we have recorded the mutations performed by baseline fuzzers and CLUTCH variants on 10,000 test cases. We capture where mutations are performed and run the test cases on V8 to capture branch coverage and JavaScript errors.

We provide the distribution of errors in Figure 6.5. Figures 6.4b, 6.4d, 6.4f show the difference in the errors across the mutations. Figures 6.4a, 6.4c, and 6.4e show the distribution of IR instructions per mutation. For example Figure 6.4a shows the difference between Fuzzilli and FClutch. The `CodeGenMutator` bar shows where CLUTCH deviates from the strategy of Fuzzilli, *i.e.* the `CallFunction` is selected 5.2% less by CLUTCH, and `SetProperty` is chosen 1% more. For readability we only report the top ten IR instructions, thus the difference may not sum to zero for each bar. Note that not all mutations are used by all fuzzers, as discussed in Section 6.2.

By comparing with the random selection of the fuzzers we can show what CLUTCH has learned to do. Importantly, we do so while accounting for the original distribution of test cases, where some instructions, or mutations, may occur more frequently than others. For instance, `InputMutator` cannot be applied to instructions which do not have input variables, such as a `BinaryOperation`.

Distribution of Mutations

We let CLUTCH observe the IR of a test case and the mutation that will be applied, with the intention for it to learn *where* to apply different mutations. Signs of this occurring can be seen in Figure 6.4 from the difference in the IR instructions chosen by CLUTCH for different mutations. For instance, `BinaryOperation` is present in the top ten IR locations selected by CLUTCH regardless of the fuzzer approach. Yet, the preference of location to perform mutations varies across different CLUTCH variants. Consider the differences in selection between JPClutch (Figure 6.4e) and FClutch (Figure 6.4a). JPClutch performs 9.8% more `CodeGenMutator` mutations on `BinaryOperations` than the random selection of JIT-Picker, whereas FClutch has learnt an

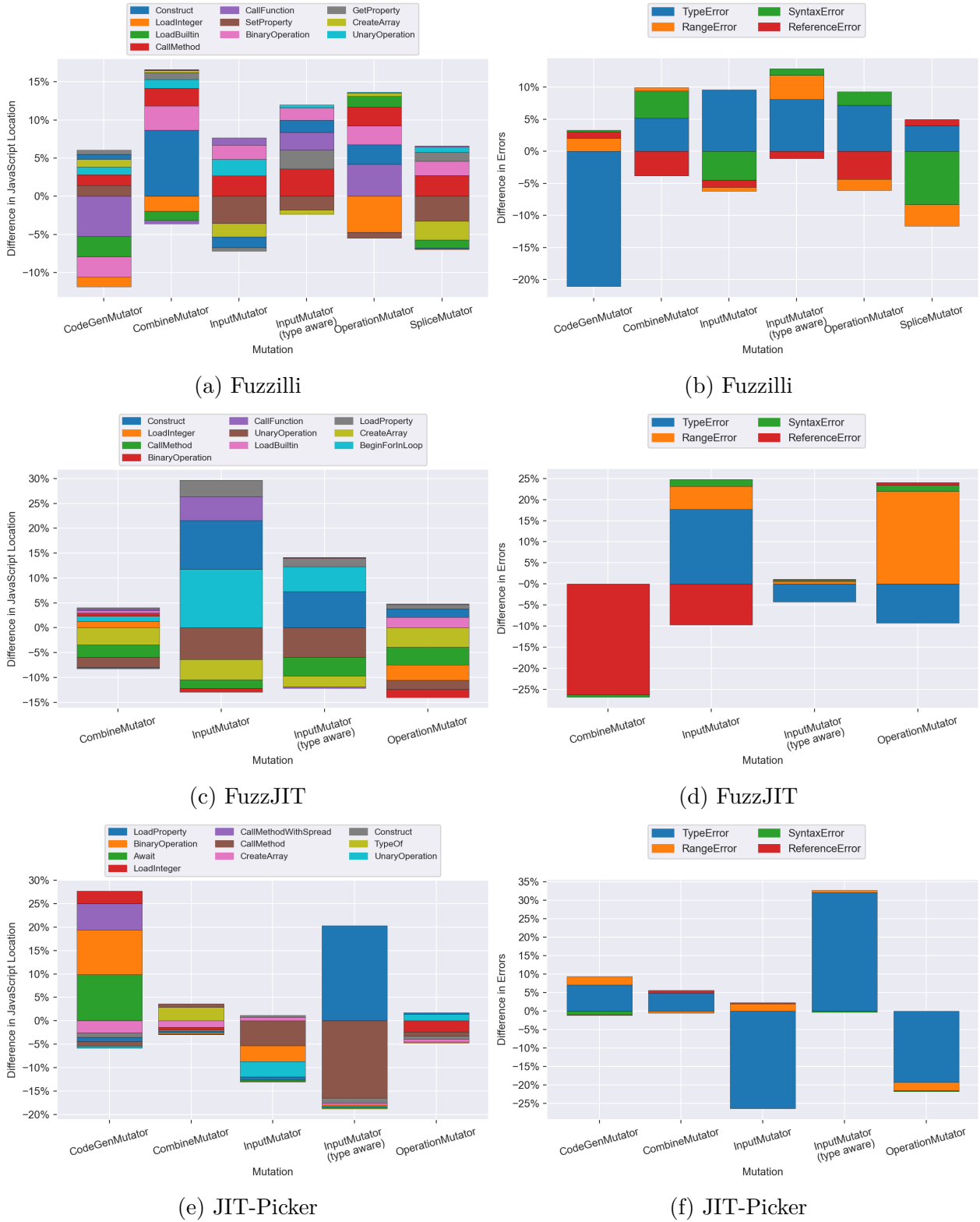


Figure 6.4: Analysis of 10,000 mutated test cases by CLUTCH and the equivalent fuzzer when tested on V8. The left column shows the percentage difference from CLUTCH selection in its top ten selected instructions per mutation. The right column shows the difference between clutch and the baseline fuzzer in terms of the errors caused per mutation.

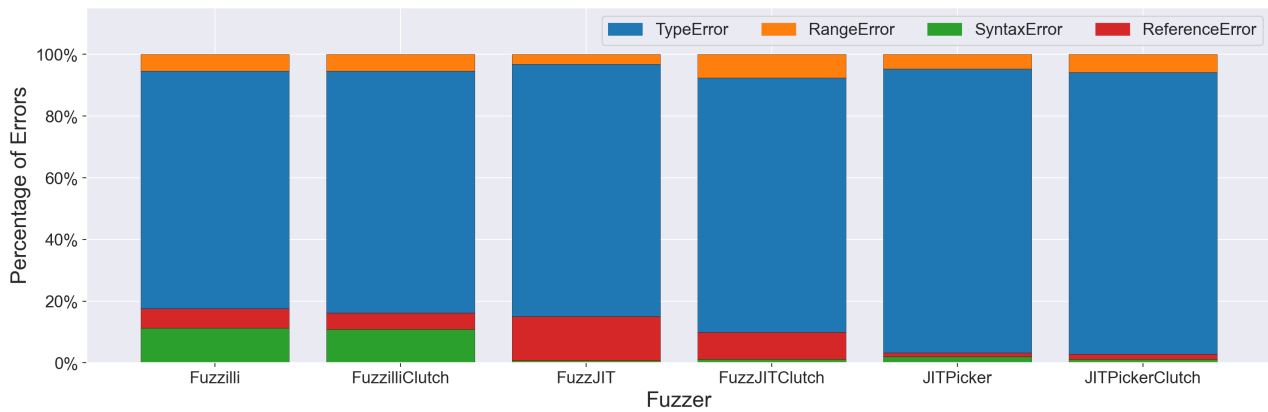


Figure 6.5: Distribution of error types caused by 10,000 test cases from different fuzzers, when testing on V8.

alternative strategy performing this combination 2.6% less than Fuzzilli.

However, CLUTCH variants also perform mutations at similar locations, as six of the instructions are present in all variants. Consider FJClutch and FClutch, which both favour the **Construct** instruction to perform mutations, indicated by its prevalence in Figure 6.4a and 6.4c. This seems a reasonable choice as **Construct** instructions create and initialise variables. By mutating these CLUTCH effects the structure and properties of test cases in an attempt to trigger additional functionality.

Distribution of Errors

In Section 6.4.2 we saw how CLUTCH increased the number of test cases that executed successfully. We investigate here why this is the case by studying what errors occur, and where. Figures 6.4b, 6.4d, 6.4f show the difference in the errors caused by test cases, and Figure 6.5 shows the distribution of errors across all fuzzers and CLUTCH variants.

Figure 6.5 shows a similar distribution of errors in the CLUTCH variants and baseline fuzzers, with some slight variation apparent between the different fuzzer types. Overall, fuzzers cause **TypeErrors** most frequently, which account for at least 78.2% for FClutch, and in the worst case of JIT-Picker causing 92.1% of errors. This is not an unexpected distribution of errors as JavaScript is a dynamically typed language where variable types can change type during the execution of a program. Combining this with the prevalence of mutations that change the type

of a variable, either explicitly or implicitly, increases the likelihood of a `TypeError`.

A more granular analysis of these errors, from Figures 6.4b, 6.4d, 6.4f, reveals a different story: errors occur because of different underlying causes. These Figures show the difference in error types between CLUTCH variants and baseline fuzzers. Consider Figure 6.4d, where FJClutch causes fewer `ReferenceErrors` when performing both `Input` and `Combine` mutations, leading to an overall reduction of these errors by 35%. However, while there is a commensurate increase in both `TypeError`s and `RangeErrors`, the overall number of errors is reduced as seen in Section 6.4.2. Such behaviour shows CLUTCH has learnt the syntax of test cases, selecting the most appropriate location to perform mutations. By comparison with Figure 6.4c we can see how applying mutations to different instructions causes a shift in errors. For instance, the use of `Construct` and `CallFunction` likely leads to an increase in `TypeError`s.

Similar comparisons can be made in the performance of JPClutch. In Figure 6.4f, the main difference in errors compared to JIT-Picker is from `TypeError`s. We can see that the `InputMutator` (type aware) leads to an increase in `TypeError`s. When comparing with Figure 6.4e, the cause of this is the shift in mutations; applying mutations more often to `LoadProperty`, and less to `CallMethod`. The redistribution of mutations to different instructions also reduces the `TypeError`s in other locations: `InputMutator` and `OperationMutator`. By doing this CLUTCH reduces the overall number of errors (Table 6.1), and ratio of `TypeError`s (Figure 6.5).

Frequency of Branch Coverage

We have argued in favour of using CLUTCH to increase the efficiency of testing JavaScript engines. To demonstrate this further we compare branch frequency from the 10,000 generated test cases generated by Fuzzilli and FClutch on V8. We report these results in Figure 6.6. We can see that both Fuzzilli and FClutch find a comparable number of branches, covering 0.25% of the V8 engine, accounting for over 34,000 branches. Comparable performance is also seen from the highest frequency of a single branch, 1320 for Fuzzilli and 1228 for FClutch. However, the frequency density of the the two highlights the difference in testing strategy. On average Fuzzilli requires 150.28 test cases to cover each branch, 23.5% more test cases than CLUTCH.

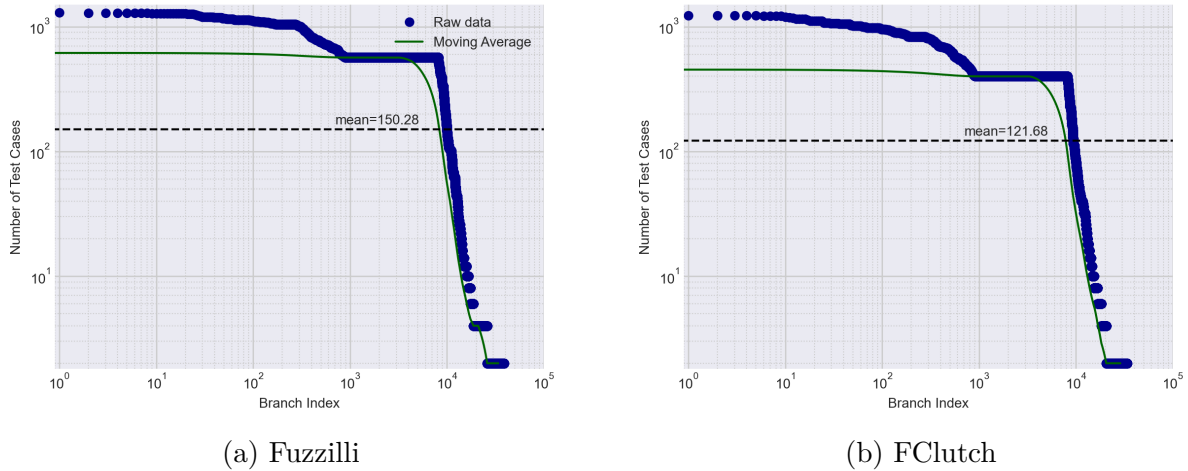


Figure 6.6: Frequency of branches in V8 that are reached from 10,000 test cases from Fuzzilli and FClutch.

Test cases from Fuzzilli trigger the same shallow paths due to higher proportion of errors they contain. Indeed, such results show that CLUTCH explores a maximal number of low frequency paths [35]. CLUTCH does so by learning to perform mutations in locations which are, a) valid for JavaScript, b) able to probe deeper in the engine more consistently.

6.4.4 Bandit Fuzzing Comparison

We have shown how CLUTCH outperforms equivalent fuzzing baselines. Compared with random selection, CLUTCH learns from its experience to improve efficiency, and a targeted testing strategy. We designed CLUTCH to handle the many complex problems that come with selecting a location to perform mutations, as outlined in Sections 6.2.4 and 6.3. Yet, it remains to be seen if alternative bandit architectures could learn in such a setting.

Thus we introduce CLUTCHLITE, an alternative bandit model based on the ACCUCB [200] algorithm. While ACCUCB was originally used for solving ‘dynamic resource allocation problems’ using a tree based structure, we have modified it to work within CLUTCH, by providing it with the ability to increase the search space over time (*i.e.* the number of instructions seen from test cases over time). CLUTCHLITE can then handle the context of selecting instructions from the IR, and the variable number of arms that are seen over the course of fuzzing. As a

Table 6.4: Branch coverage per generated test case (five run average) from 24h fuzzing of CLUTCH and CLUTCHLITE.

Engine	Metric	FJClutch	FJClutchLite	FClutch	FClutchLite	JPClutch	JPClutchLite
V8	Average	0.627	0.620	0.387	0.382	0.585	0.438
	Relative Change		-1.13%		-1.24%		-25.1%
	$\hat{A}_{12}$		0.560		0.480		0.880
	p		0.841		1.000		0.056
SpiderMonkey	Average	0.067	0.058	0.077	0.088	0.081	0.103
	Relative Change		-13.4%		13.52%		26.06%
	$\hat{A}_{12}$		1.000		0		0.111
	p		0.008		0.008		0.143
JavaScriptCore	Average	0.508	0.671	0.240	0.235	0.358	0.363
	Relative Change		32.20%		-2.07%		1.968%
	$\hat{A}_{12}$		0.000		0.680		0.500
	p		0.008		0.421		0.571

result we can compare CLUTCH to another contextual bandit for selecting locations to mutate. We compare CLUTCH and CLUTCHLITE for their efficiency of branches found per execution, reporting the results in Table 6.4.

We can see that CLUTCHLITE learns to select appropriate locations for mutations, as evidenced by the comparable performance in efficiency compared to CLUTCH. At best, CLUTCHLITE increases efficiency by 32.2%, and at worst reduces efficiency by 25.1%. This result shows that even an alternative bandit algorithm is capable of learning to optimise mutations when fuzzing. While CLUTCHLITE has been able to learn, we still see the superior performance of CLUTCH, resulting in greater branch coverage per execution in the majority of engines, increasing the aggregate efficiency by 30.81%, and 3.42% on average.

6.4.5 Wider Bandit Comparison

We must also consider the performance of CLUTCH in relation to alternative bandit approaches. However, to do so in the context of JavaScript engines is challenging as it is computationally expensive to train bandits, and there is no ground truth to determine regret performance. Instead, we use evaluation methodologies used in prior work, providing a ground truth reward, and relatively inexpensive training time. Specifically, we consider the Gollawa Dataset in the same configuration as Nika *et al.* [200] for volatile bandits, and use three hidden rewards as Hwang *et al.* [127] for non-volatile settings. We use the hyperparameters and experimental

Table 6.5: Total regret of combinatorial bandits in different settings.

Setting	Selected Arms	Volatile	Total Regret							
			ACC-UCB	CC-UCB	CombLinTS	CombLinUCB	CN-UCB	CN-TS	CN-TS(M=1)	CLUTCH
Gowalla Dataset	2	✓	4915.64 (± 57.00)	12328.73 (± 20.31)	-	-	-	-	-	1077.68 (± 962.21)
	4	✓	985.70 (± 20.35)	3688.81 (± 15.15)	-	-	-	-	-	125.42 (± 105.05)
$R_1(\mathbf{x}) = \mathbf{x}^\top \mathbf{a}$	4		-	-	2865.87 (± 321.02)	2667.51 (± 227.48)	74.43 (± 3.95)	67.87 (± 2.12)	70.80 (± 3.97)	65.04 (± 2.51)
$R_2(\mathbf{x}) = (\mathbf{x}^\top \mathbf{a})^2$	4		-	-	1140.77 (± 23.56)	1155.06 (± 14.31)	310.01 (± 83.50)	253.60 (± 28.48)	242.26 (± 14.97)	216.54 (± 4.37)
$R_3(\mathbf{x}) = \cos(\pi \mathbf{x}^\top \mathbf{a})$	4		-	-	2501.58 (± 35.68)	2495.12 (± 48.78)	948.33 (± 96.76)	997.02 (± 158.80)	991.70 (± 83.72)	325.86 (± 40.11)

setup as in the prior work, running experiments for 10,000 rounds and repeating 10 times. We present the results in Table 6.5.

Volatile Arms. In this setting we consider a crowd sourcing problem used by Nika *et al.* [200], using the Gowalla dataset. At each round t a random number of user check-ins are sampled, where each check-in is an arm. Context is the location (normalised longitude and latitude lying in $[0, 1]^2$), and the reward is the ‘willingness to work’, *i.e.* battery status. Bandits must then select $K \in 2, 4$ locations. We compare against two state-of-the-art bandits that handle the volatile nature of the arms: ACC-UCB [200] and CC-UCB [44]. From Table 6.5 we see that bandits improve when selecting more arms due to the increased likelihood of selecting ‘correct’ arms. Furthermore, CLUTCH has the best performance in both $K \in 2, 4$, achieving at least 78.1% fewer regret than other bandits.

Non-Volatile Arms. In this setting we consider three different, unknown, reward functions: $R_1(\mathbf{x}) = \mathbf{x}^\top \mathbf{a}$, $R_2(\mathbf{x}) = (\mathbf{x}^\top \mathbf{a})^2$, and $R_3(\mathbf{x}) = \cos(\pi \mathbf{x}^\top \mathbf{a})$. Where x is the set of contexts for arms and a has the same dimension as x and is randomly generated from a unit ball, remaining fixed during the experiment. Bandits must select from $K = 4$ arms at each round, observing a context of 80 features. Such an evaluation is inline with Hwang *et al.* [127] and Zhou *et al.* [324]. We compare against contextual neural bandits CN-UCB and CN-TS, and their linear equivalents CombLinUCB and CombLinTS [127]. We additionally include CN-TS(M=1), a special case of CN-TS drawing only one sample per arm. Volatile bandits require context features to be $N = 2$, which not originally considered by Hwang *et al.* [127] which uses 80 features ($N = 80$). From Table 6.5 we again see that the deep CBs outperform the linear counterparts. We also

Table 6.6: Statistical significance of bandit regret performance compared to CLUTCH, using Vargha and Delaney’s $\hat{A}_{12}$ test and Mann-Whitney U test (p values).

Setting	Volatile	Selected Arms	Features	ACC-UCB	CC-UCB	CombLinTS	$\hat{A}_{12}, p$ CombLinUCB	CN-UCB	CN-TS	CN-TS(M=1)
Golwalla Dataset	✓	2	2	1.0, 0.0002	1.0, 0.0002	-	-	-	-	-
		4		1.0, 0.0002	1.0, 0.0002	-	-	-	-	-
$R_1(\mathbf{x}) = \mathbf{x}^\top \mathbf{a}$		4	2	0.8, 0.03	0.9, 0.03	0.78, 0.04	0.9, 0.03	1.0, 0.0002	1.0, 0.0002	1.0, 0.0002
			80	-	-	1.0, 0.0002	1.0, 0.0002	0.98, 0.0003	0.81, 0.02	0.86, 0.007
$R_2(\mathbf{x}) = (\mathbf{x}^\top \mathbf{a})^2$		4	2	1.0, 0.0002	1.0, 0.0002	1.0, 0.0002	1.0, 0.0002	1.0, 0.0002	1.0, 0.0002	1.0, 0.0002
			80	-	-	1.0, 0.0002	1.0, 0.0002	1.0, 0.0002	0.96, 0.0006	1.0, 0.0002
$R_3(\mathbf{x}) = \cos(\pi \mathbf{x}^\top \mathbf{a})$		4	2	1.0, 0.0002	1.0, 0.0002	1.0, 0.0002	1.0, 0.0002	1.0, 0.0002	1.0, 0.0002	1.0, 0.0002
			80	-	-	1.0, 0.0002	1.0, 0.0002	1.0, 0.0002	1.0, 0.0002	1.0, 0.0002

see that across all rewards CLUTCH outperforms the deep CBs, achieving between 4.1%-67.3% fewer regret.

Furthermore, across both volatile and non-volatile settings CLUTCH outperforms with large effect size ($\hat{A}_{12} \geq 0.78$) and statistical significance ($p < 0.04$) (see Table 6.6).

6.5 Limitations

The selection of hyperparameters in a learning architecture may depend on domain knowledge, leading to bias, and may in general affect the performance of the model unpredictably. To mitigate this, CLUTCH controls its own rate of experimentation, uses hyperparameters from prior work where applicable [50, 90, 291], and uses a grid-search to find optimal values for remaining hyperparameters. We present the details of this grid-search in Table 6.7, including lower and upper bounds, and the selected value.

We did not place CLUTCH inside an exhaustive list of all JavaScript engine fuzzers. In particular, we have excluded OptFuzz [297] and Superion [296]. OptFuzz, while publicly available, requires recompiling the `11vm` library with custom hooks, and does not fully document the dependencies required to do so. Superion is used for fuzzing both XML and JavaScript engines, extending the AFL fuzzing model. We considered the approach and underlying system out-of-scope. Indeed, it is non-trivial to place CLUTCH inside fuzzers. Instead, we selected three state-of-the-art fuzzers to evaluate the performance of CLUTCH. We argue these fuzzers demonstrate the generalisability of our approach, as they use different approaches and testing mutations, ranging from targeting the JIT to differential testing. We further test each fuzzer

Table 6.7: Grid search ranges for hyperparameters neural network models

Hyperparameter	Lower Bound	Upper Bound	Selected
γ	0.5	0.999	0.9
Learning Rate	0.05	0.0005	0.005
Update Step	10	10000	200
Update Type	Exponential	Linear	Linear

and CLUTCH variant on three separate and commonly used JavaScript engines. This removes bias from test subjects, showing the real world application of CLUTCH.

To mitigate for randomness in the different approaches we conducted large scale experiments. We repeated experiments in Sections 6.4.2, and 6.4.4 five times, accounting for 3,240 compute hours of experiments. In reporting our results we have followed guidance from Schlosser *et al.* [250] and Klees *et al.* [146], reporting metrics which provide deeper statistical comparison between different sets of data. Additionally, in Section 6.4.3, we use 10,000 test cases to compare between CLUTCH and the fuzzer baselines to provide an in-depth study on their behavioural differences.

6.6 Related Work

Fuzzing Strategies. To our knowledge there has been no prior work in fuzzing to determine where to perform mutations in a test case in fuzzing. Orthogonal works have focused on different aspects of fuzzing. Weissberg *et al.* [306] examined different methods to select areas in the *target* to fuzz in directed fuzzing [18, 35, 174, 268]. Lyu *et al.* [174] develop an algorithm to schedule mutators, this to later work EMS [175] which incorporated historic data for the same purpose. Zhao *et al.* [322] investigated the scheduling of seeds using Monte-Carlo Tree Search approach. While these works investigate selection of target code, scheduling mutators, or scheduling seeds, CLUTCH selects where in the *test case* to perform mutations in greybox fuzzing.

Some work, including that of Yue *et al.* [314], Wang *et al.* [298], and Luo *et al.* [172] explored using bandits. Specifically for seed scheduling. This is substantially different from CLUTCH,

which uses a *contextual* bandit for selecting locations to mutate, not the seed testcases.

Fuzzing JavaScript Engines. Fuzzing has become the de facto approach to find bugs and vulnerabilities in JavaScript engines, leading to a number of works on this. LangFuzz [48], creates an Abstract Syntax Tree (AST) of JavaScript functions, selecting nodes to mutate at random. IFuzzer [288] makes use of a similar strategy, but improves effectiveness with genetic programming, and a fitness function from greybox feedback on the test case. Montage [152] uses Natural Language Processing (NLP) to generate JavaScript sub trees at random locations in a test case’s AST. DIE [220] is a mutation based fuzzer that aims to preserve type and structural properties of test cases. CodeAlchemist [107], reduces test cases to fragments, using static and dynamic analysis to infer types, and then reassemble test cases. In both Superion [296], and SkyFire [295], the authors use grammar based methods for mutation and generation, respectively. Nautilus [16] takes this a stage further: generating its own test cases and then mutating their ASTs. While Salls *et al.* [248] use a token-level approach to perform mutations at a higher level, to avoid using grammars.

As we have discussed Fuzzilli [96] focused on finding bugs in the JIT compiler using an IR. The approach used in Fuzzilli has since been adopted by FuzzJIT [299] and JIT-Picker [28]. OptFuzz [297], also builds on the Fuzzilli architecture to use optimisation paths as a performance metric. The authors of OptFuzz use an approximation of the optimisation path coverage as feedback to guide seed scheduling and preservation.

Importantly, none of the JavaScript engine fuzzers make use of an RL feedback approach, with only Montage using a learning based fuzzer. Nor do these approaches target where to perform the mutations in the test cases, whilst CLUTCH uses an RL approach to target mutations in a test case.

6.7 Takeaways

Across this Thesis we have investigated how RL could improve efficiency and be more effective than state-of-the-art approaches. In this Chapter we have tackled another key problem, not just in web application security testing, but in the broader domain of fuzzing. We aimed to determine if an agent can learn where to perform mutations in a test case when fuzzing JavaScript engines. Such a challenge has the potential to open up new avenues for improving the efficiency of testing. Yet this challenge has not been explored in prior work, which has focused on oracles, instrumentation, and heuristics for mutations.

We implement our approach as CLUTCH, a deep contextual bandit that observes a testcase and mutation, selecting the location where the mutation will be performed. Being based on a deep learning architecture with attention, CLUTCH can overcome several challenges: 1) the large search space of test cases, 2) the complexity of JavaScript, and 3) the varied structure of test cases. Implementing our approach in a fuzzer agnostic way, we can place CLUTCH inside existing fuzzers.

We evaluated CLUTCH in three state-of-the-art fuzzers, testing on three JavaScript engines. Our results show CLUTCH learns to increase the correctness of test cases, increases aggregate coverage, and efficiency of coverage per test case. We have shown that CLUTCH can learn specific testing strategies that are distinct from the random selection strategy used in existing fuzzers. As a result, it can reduce the overall number of errors. We believe CLUTCH demonstrates an impressive performance in locating *where* to perform mutations in a test case. This showcases how RL can be used to make web application security testing more intelligent. With more engineering effort, CLUTCH and approaches like it, can be made faster to increase the throughput, *e.g.* reimplementing CLUTCH in native fuzzer languages.

Chapter 7

Conclusion

In this Thesis we have studied ways to improve the effectiveness and efficiency of web application security testing using Reinforcement Learning (RL). RL is the paradigm where agents learn policies to complete tasks. Without requiring a dataset to learn from, agents collect experience and reward-based feedback through interaction with an environment, *i.e.* they can take *actions* to effect *states*. As a result, RL has shown impressive, even better than human, performance in a variety of tasks in different domains [80, 136, 187, 208, 309]. Across this Thesis we have proposed and implemented RL approaches to problems in the domain of web application security testing, including: generating tailored injection payloads, utilising feedback to mutate HTTP test cases, and target effective locations to perform mutations in JavaScript test cases. In this Chapter, we will summarise the achievements of the Thesis and outline its limitations. We then hope to inspire further research by suggesting future work in this domain. Finally, we discuss RL in a broader context of cyber security.

7.1 Summary of Thesis Achievements

“It is more than possible; it is probable.”

—*The Adventures of Sherlock Holmes* by Sir Arthur Conan Doyle

This Thesis has aimed to determine if, how, and where RL can be used in the domain of web application security testing. Thus, in Chapters 3, 4, 5, and 6 we have created RL approaches and designed architectures that are targeted to different testing strategies: mutation and generation, grey- and black-box. When it comes to the generation of payloads we see that RL models can find 20% more XSS and 26% more SQLi vulnerabilities, using fewer requests than state-of-the-art approaches. This trend continues in mutation based testing when we compare our RL agent, APIRL, to other approaches. APIRL finds 85% of all bugs found, and finds the most bugs undetected by other approaches (22%). Furthermore, it does so improving the coverage and bug finding capability per request. Finally, we have seen that RL can effectively target where to perform mutations in isolation from within a fuzzer increasing the proportion of valid test cases, and the efficiency in terms of the coverage per test case.

As a result of these investigations we have shown how novel and bespoke RL architectures are capable of learning diverse testing strategies. The learnt policies also transfer to different, but related tasks. For instance RL models in Chapters 3, 4, and 5 are trained on a curriculum of didactic examples, and then redeployed on related but alternative subjects. Impressively, the learnt policies can find a variety of bugs and vulnerabilities unseen in training. Concrete examples of this are: CVE-2021-42946, CVE-2021-42866, CVE-2021-42867, CVE-2021-42868, CVE-2021-42869, CVE-2022-2717, CVE-2022-2718, CVE-2022-38918, CVE-2022-38919, CVE-2022-38920, and CVE-2022-38921; all of which are zero-day vulnerabilities found by our RL approaches. Indeed, in Chapter 5 we have further seen how RL can trigger different classes of bugs in REST APIs. The learnt policies are even able to find the same classes of bugs using different testing strategies. Thus, we have demonstrated that RL can be used in a plethora of web application security testing tasks.

“Never trust to general impressions, my boy, but *concentrate yourself upon details.*”

—*The Adventures of Sherlock Holmes* by Sir Arthur Conan Doyle

We have compared the approaches detailed in this Thesis in different domains. Through this series of experiments we have seen that our RL approaches outperform the state-of-the art techniques, varying from: random selection and manual heuristics, to evolutionary approaches.

We have further seen how using off-the-shelf RL models can lead to sub-optimal performance, by comparing with prior work, and ablations of our own architectures. We argue that this shows a compelling case for the considered design of RL approaches in web application security testing. The bespoke architectures we develop focus on trying to include detail in information rich latent representations, and improve the level of autonomy agents can have in their the strategies. Such a focus on design and on the definition and model of an agent is aligned with Abel *et al.* [1], who argue that agent models require further research beyond existing understanding. We argue our results show tailored agents are more *efficient* and *effective* than alternative state-of-the-art approaches. Furthermore, they show *consistency* across the evaluations and ablations in Chapters 3, 4, 5, and 6.

“You see, but you do not observe. The distinction is clear.”

—A *Scandal in Bohemia* by Sir Arthur Conan Doyle

We aimed to increase the strategy of testing by harnessing information beyond the use of domain knowledge and heuristics. How to represent real-world web data as observations or rewards for an RL agent is a fundamental challenge. This arises from the diversity and complexity of data, *e.g.* JavaScript test cases, or payloads. There is a breadth of possible signals and data that we could harness in the testing domains investigated in this Thesis. We have explored how representations effect the learning of an RL agent by using increasingly rich latent representations, ranging from tokens, RNN encodings, and transformer embeddings. We have been able to show that such representations improve the performance of agents when observations are complex, containing important and subtle semantic relations, including: SQL, JSON, and natural language.

We have further challenged the design of reward functions when learning policies for testing. Investigating reward functions that use domain knowledge in different ways, we have seen how rewards can unintentionally result in poor testing policies. We speculate that such reward functions bias earlier periods in training as ineffective reward functions lead to diminishing returns later in training. Through these insights we hope to promote a carefully considered approach to the design of RL for web application security testing.

7.2 Limitations

This Thesis has aimed to be rigorous and continually strive for improvement in design, implementation, and evaluation. However, it is not without its limitations, which we discuss here:

- **Smarter does not equal Faster.** We have seen in previous chapters the advantages of using RL approaches for web application security testing. However, the increased efficiency and effectiveness often comes at the cost of computational complexity. By introducing a deep learning architecture we inevitably introduce additional overhead in the time used to perform testing in comparison with static heuristics or random selection methods. For example, in Chapter 3, HAXSS, can find more vulnerabilities in a fewer number of test cases. However, in some cases it takes a significantly longer time to complete the testing compared with more mature scanners. In Chapter 6, we placed CLUTCH inside other scanners to determine its performance. Similarly we found that it could increase efficiency per test case, but at the cost of increasing the overhead of test cases.

With enough engineering effort the more ‘intelligent’ techniques introduced in this Thesis can be made faster. On the other hand, to do so comes at the cost of a significant time investment which would not increase the quality of performance in any other dimension, *i.e.* the number of bugs or vulnerabilities found. Thus, such an investment is of minimal value to the research that has been conducted in this Thesis. We hope that the ideas and techniques from this Thesis can be taken further and used more broadly in both the RL and web application testing communities.

- **Uniform Evaluation.** We set out in the objectives for this Thesis to *continuously* develop the evaluation methodology. As a result, we have included additional metrics and experiments across the different testing paradigms. However, this means that the evaluations are not uniform across Chapters 3, 4, 5, and 6. Such uniform evaluation is desirable, showcasing all possible metrics and cross comparing against further architec-

tures. However, we believe our evaluations are rigorous, demonstrating the capacity of RL to test the security of web applications. In the evaluations we have consistently compared against existing state-of-the-art approaches and, where possible, included related approaches from the literature. We have been transparent in our reporting of evaluations, showing the strengths and weaknesses of different approaches, including those developed in this Thesis. Furthermore, we have set out to evaluate the real-world impact of our approaches, testing on production grade test subjects, showcasing zero-day CVEs, and bugs.

- **Good Faith Testing.** We have viewed web application security testing through the lens of the defender trying to secure systems. In such cases, web security practitioners often have limited time for testing, and require confidence in their testing tools [153, 179, 203]. Therefore, we have harnessed the power of RL for its ability to effectively search over large state-action spaces to improve the efficiency of web security testing. We have done so for these techniques to be useful for security practitioners. While such techniques *could* conceivably be used by malicious actors, their interest in such tools is very unlikely, especially in the case of grey-box testing.

We have further not accounted for the possibility of adversarial samples, backdoors, or poisoning [293]. Such behaviour could be possible, leading to the deterioration of the learnt policy. However, the threat model would be overly permissive to the adversary, and in such a scenario one would have more severe consequences than a suboptimal policy.

- **Using Domain Knowledge.** We have aimed to improve the ability of RL agents testing strategies by maximising the useful information given to them in both states and rewards. In doing so, we have tried to move away from using heuristics that can constrain the ability of agents. However, in some cases it is challenging to decouple domain knowledge from the RL implementation. For example, in Chapter 3 we used a constrained state-action space, and shaped reward to guide HAXSS to find payloads. In Chapter 5 we used a combination of explicit features and latent embeddings; showing that embeddings improved performance, and were meaningful to the agent. Finally, we believe this Thesis

examines the use of deep representations, which is not commonly found in the domain of RL testing [36, 51, 71, 141, 153, 314].

7.3 Future Work

In this Thesis, we have presented research that aims to improve the efficiency and effectiveness of web application security testing by developing novel RL architectures and problem formulations. We believe that this opens up many interesting areas for future research.

We have seen how to harness the unique power of RL to perform web application security testing; motivated by the scale of web vulnerabilities we have focused on specific classes of vulnerabilities. However, there are related domains that have not been within the remit of this Thesis. In particular the application of RL for testing alternative subjects such as deep learning libraries. The use of deep learning libraries, and their integration into many modern systems, has caused concern in the security of the underlying libraries [61]. Fundamentally, such libraries share many of the same potential causes of bugs as we have examined in this Thesis: incorrect logic, API calls, and user defined input parameters. Thus, such libraries can be treated similarly, harnessing many of the approaches developed here. As yet, there has been no work to use RL to test such systems. We believe this to be a rich area to explore, as the potential search space of these testing domains is large, and could benefit from the target approaches that RL offers.

Furthermore, the increased use of deep learning models, particularly LLMs, in production grade systems presents another dimension in which testing can be conducted [261]. Such systems often have user input components that can be accessed through the web, *e.g.* LLMs can be accessible via APIs. Thus, these web exposed systems can be treated in a similar way to test subjects in this Thesis. However, they introduce additional complexities from the deep learning model. This presents a new dimension in which we can test systems [37, 38]. Test cases could be developed to target: a) the underlying deep learning model, b) the interface between the model and its pipeline, and c) the remaining system. There has been some prior work to show that

RL can be effective in prompting for LLM based systems [60, 85]; which has been expanded to jailbreak LLM models [45]. This suggests that RL could have the capability to test *systems* that use LLMs and similar deep learning architectures, to effectively uncover their insecurity.

We have investigated the ability of RL agents to learn strategies in web application security testing. To evaluate the performance of the developed models we have mainly focused on the generated and mutated outputs. We further compared the action choices of agents, and analysed how features impacted the choice of the agent. Yet, we have done so in a relatively shallow capacity with regards to the explainability of a model. There has been work to determine the underlying causes of RL agent performance in more general settings [114, 305]. However, this has been comparatively little evaluation in cyber security settings [86, 280]. The potential for explainability methods for security settings has been recently introduced by Yu *et al.* [42, 313]. Importantly, such methods have the potential to inform better reward functions, and state representations. Integrating these kinds of approaches, and developing further ways to explain model policy and behaviour could reduce the reliance on domain knowledge and increase performance.

As we have seen in Chapter 2, many approaches to deep RL in this domain tend to commonly use off-the-shelf-methods. We have tried to develop novel approaches to agent design in this space, providing augmented representations, and designing new deep architectures. However, there are many more techniques developed and used in the deep RL community that have not yet made their way into the security or testing domains. Approaches like upside-down RL [269] and inverse RL [197] represent alternative models to learn policies and complete tasks. The use of these techniques for web application security testing represents an underutilised area of research. An exploration into how alternative learning models can perform in such domains could unlock additional performance and further research.

7.4 Final Thoughts on RL in Cyber Security

We conclude this Thesis with some final thoughts on the use of RL in the broader context of cyber security. In more traditional RL settings there are clearly defined rules, states, and actions, *e.g.* in video games; and RL has shown to perform well in these settings [191, 251, 309]. RL has additionally been used in a variety of complex, real world settings where these definitions are less clear [55, 80, 187]. Yet, RL applications in cyber security often lack the same level of performance, impact, and realism, due to the unique challenges in mapping problems to MDPs and environments [292]. Such challenges arise from difficulty in recreating real-world systems and making environments efficient enough to train RL agents. In this Thesis, we have designed our environments to be applicable to real-world settings. The further development of security environments that focus on real-world impact is key to demonstrate the ability of RL agents beyond testing.

On the other hand, this Thesis has also argued for the development and research of agent models that are well suited to their specific task. Indeed, recent work from Abel *et al.* [1] has argued for a greater focus on the development of better agent models. This salient point is echoed in Richard Sutton’s *The Bitter Lesson* [274]; arguing for focus on agents capable of searching through state-action spaces to discover their own strategies, *without* a priori bootstrapping from domain knowledge.

In this Thesis, we have attempted to evaluate and determine the potential of RL for web application security testing. In doing so we have provided RL agents with better representations of real world problems, and shifted away from using constrained domain knowledge. We have shown that RL is a capable tester in terms of both effectiveness and efficiency. We believe RL can be further harnessed to develop solutions to broader cyber security problems. There has been notable work using RL in a number of key areas beyond web application security, including: automated cyber defence [12, 83, 86, 117, 122, 292, 312], penetration testing [166, 198, 281], red teaming AI models [45, 92, 99], and malware detection and evasion [9, 78, 238, 264]. Thus we advocate for the development of real-world RL environments, and RL agents capable of learning effective security policies.

Appendix A

Commands used to test XSS scanners

Table A.1: Configurations for the scanners used as baselines to compare to HAXSS. Where applicable scanners are provided with relevant information to be correctly authenticated with a web application.

Arachni: `./bin/arachni [url] --check=xss* --browser-cluster-pool-size=2
--plugin autologin:url=[url],parameters="username=[username]\&
password=[password]",check="[check]"`

Black Widow: `python crawl.py --url [url]`

Wapiti: `./bin/wapiti -u [url] -a [username]#[password] -m xss`

XSppear: `XSppear -u '[url]'`

XSSer: `xsser -u [url] --auto -c=18 --Cl --auth-type=Basic --auth-cred=[
username]:[password]`

XSSMAP: `python XssMap.py '[url]' -x`

w3af: We used the following plugin settings providing the correct authentication:

1. audit: xss
2. crawl: web_spider
3. auth: generic

ZAP: We used the following settings to run ZAP:

1. traditional spider
 2. AJAX spider with headless chrome
 3. context with the relevant authentication
 4. active using all of the XSS options
-

Appendix B

Commands used to test SQLi scanners

Table B.1: Settings used for scanners when conducting experiments of crawling and vulnerability identification. In Section 4.5.3 we also test without crawling, pointing the scanners to each specific vulnerability found by SQIRL.

Arachni:

```
./bin/arachni [url] --check=sqli --browser-cluster-pool-size=2
--http-cookie-string='Path=[cookie_file]'
```

BurpSuite: We used the following settings:

1. Default crawling settings
2. Set the authentication using active cookie
3. Audit using all SQL injection options

Sqlmap:

```
sqlmap -u [url] --cookie=[cookie] --crawl 10 --batch --forms
```

Wapiti:

```
./bin/wapiti -u [url] -c [cookie_file] -m sql
```

ZAP: We used the following settings:

1. Traditional spider
 2. AJAX spider with headless chrome
 3. Context with the relevant authentication
 4. Active scans using SQL injection option, using the built-in/advanced plugin
-

Algorithm C.1: Deep Q-Network training algorithm as proposed by Mnih *et al.* [191]

```
1 Initialise replay memory  $D$  to capacity  $N$ 
2 Initialise action-value function  $Q$  to random weights  $\theta$ 
3 Initialise target action-value function  $\hat{Q}$  with weights  $\theta^- = \theta$ 
4 for  $episode = 1, M$ :
5     Initialise Sequence  $s_1 = x_1$ 
6     for  $t = 1, T$ :
7         With probability  $\varepsilon$  select random action  $a_t$ 
8         otherwise select  $a_t = \arg \max_a Q(\phi(s_t), a; \theta)$ 
9         Execute action  $a_t$  and observe reward  $r_t$  and the next image  $x_{t+1}$ 
10        Set  $s_{t+1} = s_t, a_t, x_{t+1}$ 
11        Store transition( $s_t, a_t, r_t, s_{t+1}$ ) in  $D$ 
12        Sample a random mini batch of transitions  $s_j, a_j, r_j, s_{j+1}$  from  $D$ 
13         $y_j = \begin{cases} r_j, & \text{if episode terminates at step } j + 1 \\ r_j + \gamma \max_a \hat{Q}(s_{t+1}, a; \theta^-), & \text{otherwise} \end{cases}$ 
14        Perform a gradient descent on  $(y_j - Q(s_j, a_j; \theta))^2$  with respect to the network parameters  $\theta$ 
15        Every  $N$  steps reset  $\hat{Q} = Q$ 
```

Appendix C

DQN Algorithm

Algorithm C.1 presents the training algorithm used for the DQN as developed by Mnih *et al.* [191].

Bibliography

- [1] D. Abel, M. K. Ho, and A. Harutyunyan. Three Dogmas of Reinforcement Learning. *Reinforcement Learning Journal*, 1(1), July 2024. arXiv:2407.10583 [cs].
- [2] L. Adolphs and T. Hofmann. LeDeepChef: Deep Reinforcement Learning Agent for Families of Text-Based Games, Sept. 2019. arXiv:1909.01646 [cs, stat].
- [3] W. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang. Unified pre-training for program understanding and generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 2655–2668, Online, June 2021. Association for Computational Linguistics.
- [4] Akamai. Phishing for Finance. *State of the Internet*, 7, 2021.
- [5] A. Alhuzali, R. Gjomemo, B. Eshete, and V. N. Venkatakrishnan. NAVEX: Precise and Scalable Exploit Generation for Dynamic Web Applications. In *USENIX Security Symposium*, 2018.
- [6] M. S. Aliero, I. Ghani, K. N. Qureshi, and M. F. Rohani. An algorithm for detecting SQL injection vulnerability using black-box testing. *Journal of Ambient Intelligence and Humanized Computing*, 2020.
- [7] I. Alsmadi, A. AlErroud, and A. A. Saifan. Fault-based testing for discovering SQL injection vulnerabilities in web applications. *International Journal of Information and Computer Security*, 16, 2021.
- [8] M. Amouei, M. Rezvani, and M. Fateh. RAT: Reinforcement-Learning-Driven and Adaptive Testing for Vulnerability Discovery in Web Application Firewalls. *IEEE Transactions on Dependable and Secure Computing*, 19(5):3371–3386, Sept. 2022.
- [9] H. S. Anderson, A. Kharkar, B. Filar, D. Evans, and P. Roth. Learning to Evade Static PE Machine Learning Malware Models via Reinforcement Learning. *arXiv:1801.08917 [cs]*, Jan. 2018. arXiv: 1801.08917.
- [10] O. Anschel, N. Baram, and N. Shimkin. Averaged-DQN: variance reduction and stabilization for

- deep reinforcement learning. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ICML'17, pages 176–185, Aug. 2017.
- [11] D. Appelt, C. Nguyen, L. Briand, and N. Alshahwan. Automated testing for sql injection vulnerabilities: An input mutation approach. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, 2014.
- [12] A. Applebaum, C. Dennler, P. Dwyer, M. Moskowitz, H. Nguyen, N. Nichols, N. Park, P. Rachwalski, F. Rau, A. Webster, and M. Wolk. Bridging Automated to Autonomous Cyber Defense: Foundational Analysis of Tabular Q-Learning. In *Proceedings of the 15th ACM Workshop on Artificial Intelligence and Security*, AISec'22, pages 149–159, New York, NY, USA, Nov. 2022. Association for Computing Machinery.
- [13] A. Arcuri. Automated Black- and White-Box Testing of RESTful APIs With EvoMaster. *IEEE Software*, 2021.
- [14] G. Argyros, I. Stais, A. Kiayias, and A. D. Keromytis. Back in Black: Towards Formal, Black Box Analysis of Sanitizers and Filters. In *2016 IEEE Symposium on Security and Privacy*, page 19, 2016.
- [15] D. Arp, E. Quiring, F. Pendlebury, A. Warnecke, F. Pierazzi, C. Wressnegger, L. Cavallaro, and K. Rieck. Dos and Don'ts of Machine Learning in Computer Security. In *Proceedings of the 31st USENIX Security Symposium (USENIX Security 22)*, pages 3971–3988, 2022.
- [16] C. Aschermann, T. Frassetto, T. Holz, P. Jauernig, A.-R. Sadeghi, and D. Teuchert. NAUTILUS: Fishing for Deep Bugs with Grammars. In *Proceedings 2019 Network and Distributed System Security Symposium*, San Diego, CA, 2019. Internet Society.
- [17] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz. REDQUEEN: Fuzzing with Input-to-State Correspondence. In *Proceedings 2019 Network and Distributed System Security Symposium*, San Diego, CA, 2019. Internet Society.
- [18] F. Y. Assiri and A. O. Aljahdali. Software Vulnerability Fuzz Testing: A Mutation-Selection Optimization Systematic Review. *Engineering, Technology & Applied Science Research*, 14(4):14961–14969, Aug. 2024. Number: 4.
- [19] M. V. Assis, L. F. Carvalho, J. Lloret, and M. L. Proença Jr. A GRU deep learning system against attacks in software defined networks. *Journal of Network and Computer Applications*, 177:102942, 2021. Publisher: Elsevier.
- [20] V. Atlidakis, R. Geambasu, P. Godefroid, M. Polishchuk, and B. Ray. Pythia: Grammar-

- Based Fuzzing of REST APIs with Coverage-guided Feedback and Learning-based Mutations. *arXiv:2005.11498*, 2020.
- [21] V. Atlidakis, P. Godefroid, and M. Polishchuk. RESTler: Stateful REST API Fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019.
- [22] M. Bachl, F. Meghdouri, J. Fabini, and T. Zseby. SparseIDS: Learning Packet Sampling with Reinforcement Learning. In *2020 IEEE Conference on Communications and Network Security (CNS)*, pages 1–9, June 2020.
- [23] D. Balzarotti, M. Cova, V. Felmetzger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications. In *2008 IEEE Symposium on Security and Privacy (sp 2008)*, 2008.
- [24] A. Barabanov, D. Dergunov, D. Makrushin, and A. Teplov. Automatic detection of access control vulnerabilities via API specification processing. *arXiv:2201.10833*, 2022.
- [25] E. Bates, V. Mavroudis, and C. Hicks. Reward Shaping for Happier Autonomous Cyber Security Agents. In *Proceedings of the 16th ACM Workshop on Artificial Intelligence and Security*, pages 221–232, Copenhagen Denmark, Nov. 2023. ACM.
- [26] E. Bazzoli, C. Criscione, F. Maggi, and S. Zanero. XSS PEEKER: Dissecting the XSS Exploitation Techniques and Fuzzing Mechanisms of Blackbox Web Application Scanners. In J.-H. Hoepman and S. Katzenbeisser, editors, *ICT Systems Security and Privacy Protection*, IFIP Advances in Information and Communication Technology, pages 243–258, Cham, 2016. Springer International Publishing.
- [27] S. Bensalim, D. Klein, T. Barber, and M. Johns. Talking About My Generation: Targeted DOM-based XSS Exploit Generation using Dynamic Data Flow Analysis. In *Proceedings of the 14th European Workshop on Systems Security*, 2021.
- [28] L. Bernhard, T. Scharnowski, M. Schloegel, T. Blazytko, and T. Holz. JIT-Picking: Differential Fuzzing of JavaScript Engines. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS '22*, pages 351–364, New York, NY, USA, Nov. 2022. Association for Computing Machinery.
- [29] W. Blair, A. Mambretti, S. Arshad, M. Weissbacher, W. Robertson, E. Kirda, and M. Egele. HotFuzz: Discovering Algorithmic Denial-of-Service Vulnerabilities Through Guided Micro-Fuzzing. In *Proceedings 2020 Network and Distributed System Security Symposium*, San Diego, CA, 2020. Internet Society.

- [30] S. Boyd and A. Keromytis. SQLrand: Preventing SQL Injection Attacks. In *Applied Cryptography and Network Security*, 2004.
- [31] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. OpenAI Gym. *arXiv:1606.01540 [cs]*, 2016.
- [32] Y. Burda, H. Edwards, A. Storkey, and O. Klimov. Exploration by Random Network Distillation. *arXiv:1810.12894 [cs, stat]*, Oct. 2018. International Conference on Learning Representations (ICLR).
- [33] A. S. Buyukkayhan, C. Gemicioglu, T. Lauinger, A. Oprea, W. Robertson, and E. Kirda. What’s in an Exploit? An Empirical Analysis of Reflected Server XSS Exploitation Techniques. In *Research in Attacks, Intrusions, and Defenses*, 2020.
- [34] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury. Directed Greybox Fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS ’17*, pages 2329–2344, New York, NY, USA, Oct. 2017. Association for Computing Machinery.
- [35] M. Böhme, V.-T. Pham, and A. Roychoudhury. Coverage-based Greybox Fuzzing as Markov Chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, CCS ’16*, pages 1032–1043, New York, NY, USA, Oct. 2016. Association for Computing Machinery.
- [36] K. Böttinger, P. Godefroid, and R. Singh. Deep Reinforcement Fuzzing. In *2018 IEEE Security and Privacy Workshops (SPW)*, pages 116–122, May 2018.
- [37] N. Carlini and M. Nasr. Remote Timing Attacks on Efficient Language Model Inference, Oct. 2024. *arXiv:2410.17175 [cs]*.
- [38] N. Carlini, F. Tramèr, E. Wallace, M. Jagielski, A. Herbert-Voss, K. Lee, A. Roberts, T. Brown, D. Song, U. Erlingsson, A. Oprea, and C. Raffel. Extracting Training Data from Large Language Models. In *Proceedings of the 30th USENIX Security Symposium*, Aug. 2021.
- [39] F. Caturano, G. Perrone, and S. P. Romano. Discovering reflected cross-site scripting vulnerabilities using a multiobjective reinforcement learning environment. *Computers & Security*, 103:102204, Apr. 2021.
- [40] Chaitin. XSSChop. <https://xsschop.chaitin.cn/>.
- [41] T. F. Chan, G. H. Golub, and R. J. LeVeque. Updating Formulae and a Pairwise Algorithm for Computing Sample Variances. In H. Caussinus, P. Ettinger, and R. Tomassone, editors,

- COMPSTAT 1982 5th Symposium held at Toulouse 1982*, pages 30–41. Physica-Verlag HD, Heidelberg, 1982.
- [42] F. Charmet, H. C. Tanuwidjaja, S. Ayoubi, P.-F. Gimenez, Y. Han, H. Jmila, G. Blanc, T. Takahashi, and Z. Zhang. Explainable artificial intelligence for cybersecurity: a literature survey. *Annals of Telecommunications*, 77(11):789–812, Dec. 2022.
- [43] J. Chen, W. Diao, Q. Zhao, C. Zuo, Z. Lin, X. Wang, W. C. Lau, M. Sun, R. Yang, and K. Zhang. IoTfuzzer: Discovering Memory Corruptions in IoT Through App-based Fuzzing. In *Proceedings 2018 Network and Distributed System Security Symposium*, San Diego, CA, 2018. Internet Society.
- [44] L. Chen, J. Xu, and Z. Lu. Contextual Combinatorial Multi-armed Bandits with Volatile Arms and Submodular Reward. In *Advances in Neural Information Processing Systems*, volume 31, 2018.
- [45] X. Chen, Y. Nie, W. Guo, and X. Zhang. When LLM Meets DRL: Advancing Jailbreaking Efficiency via DRL-guided Search, Oct. 2024. arXiv:2406.08705.
- [46] K. Cho, B. v. Merriënboer, C. Gülcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. In A. Moschitti, B. Pang, and W. Daelemans, editors, *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL*, pages 1724–1734. ACL, 2014.
- [47] J. Chorowski and N. Jaitly. Towards better decoding and language model integration in sequence to sequence models. *CoRR*, abs/1612.02695, 2016.
- [48] H. Christian, K. Herzig, and A. Zeller. Fuzzing with code fragments. In *Proceedings of the USENIX Security Symposium*, pages 445–458, 2012.
- [49] J. Clarke. *SQL Injection Attacks and Defense*. Syngress, second edition, 2012.
- [50] M. Collier and H. U. Llorens. Deep Contextual Multi-armed Bandits. *arXiv.org*, July 2018.
- [51] D. Corradini, Z. Montolli, M. Pasqua, and M. Ceccato. DeepREST: Automated Test Case Generation for REST APIs Exploiting Deep Reinforcement Learning, Aug. 2024. arXiv:2408.08594.
- [52] D. Corradini, A. Zampieri, M. Pasqua, and M. Ceccato. RestTestGen: An Extensible Framework for Automated Black-box Testing of RESTful APIs. In *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2022.

- [53] D. Corradini, A. Zampieri, M. Pasqua, E. Viglianisi, M. Dallago, and M. Ceccato. Automated black-box testing of nominal and error scenarios in RESTful APIs. *Software Testing, Verification and Reliability*, 2022.
- [54] Cybersecurity and Infrastructure Security Agency, Federal Bureau of Investigation, National Security Agency, Australian Signals Directorate’s Australian Cyber Security Centre, Canadian Centre for Cyber Security, New Zealand National Cyber Security Centre, Computer Emergency Response Team New Zealand, and UK National Cyber Security Centre. 2023 Top Routinely Exploited Vulnerabilities. Joint Cybersecurity Advisory, Cybersecurity and Infrastructure Security Agency, Nov. 2024.
- [55] D. B. D’Ambrosio, S. Abeyruwan, L. Graesser, A. Iscen, H. B. Amor, A. Bewley, B. J. Reed, K. Reymann, L. Takayama, Y. Tassa, K. Choromanski, E. Coumans, D. Jain, N. Jaitly, N. Jaques, S. Kataoka, Y. Kuang, N. Lazic, R. Mahjourian, S. Moore, K. Oslund, A. Shankar, V. Sindhwani, V. Vanhoucke, G. Vesom, P. Xu, and P. R. Sanketi. Achieving Human Level Competitive Robot Table Tennis, Aug. 2024. arXiv:2408.03906.
- [56] B. Damele and M. Stampar. sqlmap: automatic SQL injection and database takeover tool. <https://sqlmap.org/>, 2023.
- [57] A. Damianou and N. D. Lawrence. Deep Gaussian Processes. In *Proceedings of the Sixteenth International Conference on Artificial Intelligence and Statistics*, pages 207–215. PMLR, Apr. 2013. ISSN: 1938-7228.
- [58] M. Del Verme, A. Sommervoll, L. Erdodi, S. Totaro, and F. Zennaro. SQL Injections and Reinforcement Learning: An Empirical Evaluation of the Role of Action Structure. In *Secure IT Systems*, 2021.
- [59] G. Deng, Z. Zhang, Y. Li, Y. Liu, T. Zhang, Y. Liu, G. Yu, and D. Wang. NAUTILUS: Automated RESTful API Vulnerability Detection. In *Proceedings of the 32nd USENIX Security Symposium*, 2023.
- [60] M. Deng, J. Wang, C.-P. Hsieh, Y. Wang, H. Guo, T. Shu, M. Song, E. P. Xing, and Z. Hu. RLPrompt: Optimizing Discrete Text Prompts with Reinforcement Learning. In *Empirical Methods in Natural Language Processing*. arXiv, Oct. 2022. arXiv:2205.12548 [cs].
- [61] Y. Deng, C. S. Xia, P. Haoran, Y. Chenyuan, and L. Zhang. Large Language Models Are Zero-Shot Fuzzers: Fuzzing Deep-Learning Libraries via Large Language Models. In *Proceedings of*

- the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023*, pages 423–435, Seattle, WA, USA, July 2023. Association for Computing Machinery, Inc.
- [62] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding, May 2019. arXiv:1810.04805 [cs].
- [63] S. T. Dinh, H. Cho, K. Martin, A. Oest, K. Zeng, A. Kapravelos, G.-J. Ahn, T. Bao, R. Wang, A. Doupe, and Y. Shoshitaishvili. Favocado: Fuzzing the Binding Code of JavaScript Engines Using Semantically Correct Test Cases. In *Proceedings 2021 Network and Distributed System Security Symposium*, Virtual, 2021. Internet Society.
- [64] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan. LAVA: Large-Scale Automated Vulnerability Addition. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 110–121, May 2016. ISSN: 2375-1207.
- [65] A. Doupé, L. Cavedon, C. Kruegel, and G. Vigna. Enemy of the State: A State-Aware Black-Box Web Vulnerability Scanner. In *Proceedings of the 21st USENIX Security Symposium*, pages 523–538, 2012.
- [66] A. Doupé, M. Cova, and G. Vigna. Why Johnny Can’t Pentest: An Analysis of Black-Box Web Vulnerability Scanners. In C. Kreibich and M. Jahnke, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, Lecture Notes in Computer Science, pages 111–131, Berlin, Heidelberg, 2010. Springer.
- [67] D. Eckles and M. Kaptein. Thompson sampling with the online bootstrap, Oct. 2014. arXiv:1410.4009 [cs, stat].
- [68] Ecma International. ECMAScript 2024 language specification. Technical Report 15, Ecma International, June 2024.
- [69] EdgeScan. 2024 Vulnerability Statistics Report. Vulnerability Statistics 9, EdgeScan, 2024.
- [70] H. Emerson, L. Bates, C. Hicks, and V. Mavroudis. CybORG++: An Enhanced Gym for the Development of Autonomous Cyber Agents, Oct. 2024. arXiv:2410.16324.
- [71] L. Erdődi, A. A. Sommervoll, and F. M. Zennaro. Simulating SQL injection vulnerability exploitation using Q-learning reinforcement learning agents. *Journal of Information Security and Applications*, 2021.
- [72] L. Erdődi and F. M. Zennaro. The Agent Web Model – Modelling web hacking for reinforcement learning. *arXiv:2009.11274 [cs]*, Sept. 2020. arXiv: 2009.11274.
- [73] erev0s. VAmPI. <https://github.com/erev0s/VAmPI>, 2023.

- [74] B. Eriksson, G. Pellegrino, and A. Sabelfeld. Black Widow: Blackbox Data-driven Web Scanning. In *Proceedings of the 42nd IEEE Symposium on Security and Privacy (S&P)*, page 16, San Fransisco (CA), May 2021.
- [75] K. Fang and G. Yan. Emulation-Instrumented Fuzz Testing of 4G/LTE Android Mobile Devices Guided by Reinforcement Learning. In J. Lopez, J. Zhou, and M. Soriano, editors, *Computer Security, ESORICS*, volume 11099, pages 20–40. Springer International Publishing, Cham, 2018. Series Title: Lecture Notes in Computer Science.
- [76] Y. Fang, C. Huang, Y. Xu, and Y. Li. RLXSS: Optimizing XSS Detection Model to Defend Against Adversarial Attacks Based on Reinforcement Learning. *Future Internet*, 11(8):177, Aug. 2019. Number: 8 Publisher: Multidisciplinary Digital Publishing Institute.
- [77] Y. Fang, Y. Li, L. Liu, and C. Huang. DeepXSS: Cross Site Scripting Detection Based on Deep Learning. In *Proceedings of the 2018 International Conference on Computing and Artificial Intelligence, ICCAI 2018*, pages 47–51, New York, NY, USA, Mar. 2018. Association for Computing Machinery.
- [78] Z. Fang, J. Wang, J. Geng, Y. Zhou, and X. Kan. A3CMal: Generating adversarial samples to force targeted misclassification by reinforcement learning. *Applied Soft Computing*, 109:107505, Sept. 2021.
- [79] A. A. R. Farea, C. Wang, E. Farea, and A. Ba Alawi. Cross-Site Scripting (XSS) and SQL Injection Attacks Multi-classification Using Bidirectional LSTM Recurrent Neural Network. In *2021 IEEE International Conference on Progress in Informatics and Computing (PIC)*, pages 358–363, Dec. 2021. ISSN: 2329-6259.
- [80] A. Fawzi, M. Balog, A. Huang, T. Hubert, B. Romera-Paredes, M. Barekatin, A. Novikov, F. J. R. Ruiz, J. Schrittwieser, G. Swirszcz, D. Silver, D. Hassabis, and P. Kohli. Discovering faster matrix multiplication algorithms with reinforcement learning. *Nature*, 610(7930):47–53, Oct. 2022.
- [81] M. Feng and H. Xu. Deep reinforcement learning based optimal defense for cyber-physical system in presence of unknown cyber-attack. In *2017 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 1–8, Honolulu, HI, Nov. 2017. IEEE.
- [82] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse. AFL++ : Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, Aug. 2020.

- [83] M. Foley, C. Hicks, K. Highnam, and V. Mavroudis. Autonomous Network Defence using Reinforcement Learning. In *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '22, pages 1252–1254, New York, NY, USA, May 2022. Association for Computing Machinery.
- [84] M. Foley and S. Maffei. Haxss: Hierarchical Reinforcement Learning for XSS Payload Generation. In *2022 IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pages 147–158, Dec. 2022. ISSN: 2324-9013.
- [85] M. Foley, A. Rawat, T. Lee, Y. Hou, G. Picco, and G. Zizzo. Matching Pairs: Attributing Fine-Tuned Models to their Pre-Trained Large Language Models. In A. Rogers, J. Boyd-Graber, and N. Okazaki, editors, *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7423–7442, Toronto, Canada, July 2023. Association for Computational Linguistics.
- [86] M. Foley, M. Wang, Z. M, C. Hicks, and V. Mavroudis. Inroads into Autonomous Network Defence using Explained Reinforcement Learning, June 2023. arXiv:2306.09318 [cs].
- [87] Y. Frempong, Y. Snyder, E. Al-Hossami, M. Sridhar, and S. Shaikh. HIJaX: Human Intent JavaScript XSS Generator:. In *Proceedings of the 18th International Conference on Security and Cryptography*, 2021.
- [88] S. Fujimoto, W.-D. Chang, E. J. Smith, S. S. Gu, D. Precup, and D. Meger. For SALE: State-Action Representation Learning for Deep Reinforcement Learning, Nov. 2023. arXiv:2306.02451.
- [89] P. Gage. A New Algorithm for Data Compression. *C Users Journal*, 1994.
- [90] Y. Gal, J. Hron, and A. Kendall. Concrete Dropout. In *31st Conference on Neural Information Processing Systems (NIPS 2017)*, 2017.
- [91] F. A. Gers, N. N. Schraudolph, and J. Schmidhuber. Learning precise timing with lstm recurrent networks. *J. Mach. Learn. Res.*, 3:115–143, Mar. 2003.
- [92] A. Gleave, M. Dennis, C. Wild, N. Kant, S. Levine, and S. Russell. Adversarial Policies: Attacking Deep Reinforcement Learning, Jan. 2021. arXiv:1905.10615 [cs, stat].
- [93] P. Godefroid, B.-Y. Huang, and M. Polishchuk. Intelligent REST API data fuzzing. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 725–736, Virtual Event USA, Nov. 2020. ACM.
- [94] P. Godefroid, H. Peleg, and R. Singh. Learn Fuzz: Machine learning for input fuzzing. In *2017*

- 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), pages 50–59, Oct. 2017.
- [95] Google. Chrome Vulnerability Reward Program Rules | Google Bug Hunters. <https://bughunters.google.com/about/rules/chrome-friends/5745167867576320/chrome-vulnerability-reward-program-rules>, Aug. 2024.
- [96] S. Groß, S. Koch, L. Bernhard, T. Holz, and M. Johns. FUZZILLI: Fuzzing for JavaScript JIT Compiler Vulnerabilities. In *Proceedings 2023 Network and Distributed System Security Symposium*, San Diego, CA, USA, 2023. Internet Society.
- [97] Z. Gui, H. Shu, F. Kang, and X. Xiong. FIRMCORN: Vulnerability-Oriented Fuzzing of IoT Firmware via Optimized Virtual Execution. *IEEE Access*, 8:29826–29841, 2020.
- [98] Z. Gui, E. Wang, B. Deng, M. Zhang, Y. Chen, S. Wei, W. Xie, and B. Wang. SqliGPT: Evaluating and Utilizing Large Language Models for Automated SQL Injection Black-Box Detection. *Applied Sciences*, 14(16):6929, Jan. 2024. Number: 16 Publisher: Multidisciplinary Digital Publishing Institute.
- [99] W. Guo, X. Wu, S. Huang, and X. Xing. Adversarial Policy Learning in Two-player Competitive Games. In *Proceedings of the 38th International Conference on Machine Learning*, pages 3910–3919. PMLR, July 2021. ISSN: 2640-3498.
- [100] S. Gupta and B. Gupta. Robust injection point-based framework for modern applications against XSS vulnerabilities in online social networks. *International Journal of Information and Computer Security*, 2018.
- [101] HackerOne. Top 10 Most Impactful and Rewarded Vulnerability Types, 2020.
- [102] HackerOne. 8th annual hacker-powered security report 2022, 2023.
- [103] HackerOne. Hacker-powered security report 2023, 2023.
- [104] HackerOne. 8th annual hacker-powered security report 2024/2025, Nov. 2024.
- [105] HAHWUL. XSpear. <https://github.com/hahwul/XSpear>, 2021.
- [106] W. G. J. Halfond and A. Orso. AMNESIA: analysis and monitoring for NEutralizing SQL-injection attacks. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, 2005.
- [107] H. Han, D. Oh, and S. K. Cha. CodeAlchemist: Semantics-Aware Code Generation to Find Vulnerabilities in JavaScript Engines. In *Proceedings 2019 Network and Distributed System Security Symposium*, San Diego, CA, 2019. Internet Society.

- [108] Y. Han, B. I. P. Rubinstein, T. Abraham, T. Alpcan, O. De Vel, S. Erfani, D. Hubczenko, C. Leckie, and P. Montague. Reinforcement Learning for Autonomous Defence in Software-Defined Networking. *arXiv:1808.05770 [cs, stat]*, Aug. 2018. arXiv: 1808.05770.
- [109] H. Hanif and S. Maffei. VulBERTa: Simplified Source Code Pre-Training for Vulnerability Detection. In *2022 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, July 2022. ISSN: 2161-4407.
- [110] H. Hanif, M. H. N. Md Nasir, M. F. Ab Razak, A. Firdaus, and N. B. Anuar. The rise of software vulnerability: Taxonomy of software vulnerabilities detection and machine learning approaches. *Journal of Network and Computer Applications*, 179:103009, Apr. 2021.
- [111] M. Hanif and M. Hazim. *Deep representation learning for software vulnerability detection*. PhD thesis, Imperial College London, May 2023. Accepted: 2024-03-22T14:57:19Z Publisher: Imperial College London.
- [112] K. Hasegawa, S. Hidano, and K. Fukushima. Automating XSS Vulnerability Testing Using Reinforcement Learning:. In *Proceedings of the 9th International Conference on Information Systems Security and Privacy*, pages 70–80, Lisbon, Portugal, 2023. SCITEPRESS - Science and Technology Publications.
- [113] N. Havrikov and A. Zeller. Systematically Covering Input Structure. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 189–199, San Diego, CA, USA, Nov. 2019. IEEE.
- [114] C. He, B. G. Leon, and F. Belardinelli. Do Androids Dream of Electric Fences? Safety-Aware Reinforcement Learning with Latent Shielding, Dec. 2021. arXiv:2112.11490.
- [115] M. Hessel, H. v. Hasselt, J. Modayil, and D. Silver. On Inductive Biases in Deep Reinforcement Learning, July 2019. arXiv:1907.02908.
- [116] M. Hessel, J. Modayil, H. v. Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. Azar, and D. Silver. Rainbow: Combining Improvements in Deep Reinforcement Learning, Oct. 2017. arXiv:1710.02298.
- [117] C. Hicks and V. Mavroudis. Autonomous cyber defence: Beyond games?, Apr. 2024.
- [118] C. Hicks, V. Mavroudis, M. Foley, T. Davies, K. Highnam, and T. Watson. Canaries and Whistles: Resilient Drone Communication Networks with (or without) Deep Reinforcement Learning. In *Proceedings of the 16th ACM Workshop on Artificial Intelligence and Security, AISec '23*, pages 91–101, New York, NY, USA, Nov. 2023. Association for Computing Machinery.

- [119] K. Highnam, Z. Hanif, E. V. Vogt, S. Parbhoo, S. Maffei, and N. R. Jennings. Adaptive Experimental Design for Intrusion Data Collection. In *Proceedings of the Conference on Applied Machine Learning for Information Security*. arXiv, Oct. 2023. arXiv:2310.13224.
- [120] K. Highnam, D. Puzio, S. Luo, and N. R. Jennings. Real-Time Detection of Dictionary DGA Network Traffic Using Deep Learning. *SN Computer Science*, 2(2):110, Feb. 2021.
- [121] M. Hoffmann, B. Janiczak, and E. Mandrikov. JaCoCo Java Code Coverage Library. <https://github.com/jacoco/jacoco>, 2023.
- [122] C. Hou, M. Zhou, Y. Ji, P. Daian, F. Tramer, G. Fanti, and A. Juels. SquirRL: Automating Attack Analysis on Blockchain Incentive Mechanisms with Deep Reinforcement Learning. In *Proceedings 2021 Network and Distributed System Security Symposium*, Feb. 2021. arXiv: 1912.01798.
- [123] HTTP Archive et al. Web Almanac 2021: HTTP Archive’s Annual State of the Web Report. Technical report, HTTP Archive, 2022.
- [124] HTTP Archive et al. Web Almanac 2022: HTTP Archive’s Annual State of the Web Report. Technical report, HTTP Archive, 2023.
- [125] H. Hu, A. Lerer, A. Peysakhovich, and J. Foerster. "Other-Play " for zero-shot coordination. In *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *ICML’20*, pages 4399–4410. JMLR.org, July 2020.
- [126] Y. Huang, Z. Zhao, and C. Shi. Network Security Perception System Integrating Improved CNN Algorithm and Improved GRU Algorithm. *IEEE Access*, 2024. Publisher: IEEE.
- [127] T. Hwang, K. Chai, and M.-H. Oh. Combinatorial Neural Bandits. In *Proceedings of the 40th International Conference on Machine Learning*. PMLR, 2023.
- [128] IBM. Cost of a data breach 2024. Technical report, IBM, 2024.
- [129] A. R. Ibrahimzada, Y. Varli, D. Tekinoglu, and R. Jabbarvand. Perfect is the enemy of test oracle. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2022, page 70–81, New York, NY, USA, 2022. Association for Computing Machinery.
- [130] INTEL471. The 471 Cyber Threat Report 2024, 2024.
- [131] ITU. Measuring digital development: Facts and figures 2020, 2020.
- [132] ITU. Measuring digital development: Facts and figures 2024, 2024.
- [133] S. Jan, C. D. Nguyen, A. Arcuri, and L. Briand. A Search-Based Testing Approach for XML In-

- jection Vulnerabilities in Web Applications. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 356–366, Mar. 2017.
- [134] S. Jan, A. Panichella, A. Arcuri, and L. Briand. Automatic Generation of Tests to Exploit XML Injection Vulnerabilities in Web Applications. *IEEE Transactions on Software Engineering*, 45(4):335–362, Apr. 2019.
- [135] L. Joffe. Machine Learning Augmented Fuzzing. In *2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 178–183, Oct. 2018.
- [136] J. Jumper, R. Evans, A. Pritzel, T. Green, M. Figurnov, O. Ronneberger, K. Tunyasuvunakool, R. Bates, A. Žídek, A. Potapenko, A. Bridgland, C. Meyer, S. A. A. Kohl, A. J. Ballard, A. Cowie, B. Romera-Paredes, S. Nikolov, R. Jain, J. Adler, T. Back, S. Petersen, D. Reiman, E. Clancy, M. Zielinski, M. Steinegger, M. Pacholska, T. Berghammer, S. Bodenstein, D. Silver, O. Vinyals, A. W. Senior, K. Kavukcuoglu, P. Kohli, and D. Hassabis. Highly accurate protein structure prediction with AlphaFold. *Nature*, 596(7873):583–589, Aug. 2021.
- [137] O. Kasim. An ensemble classification-based approach to detect attack level of SQL injections. *Journal of Information Security and Applications*, 59, 2021.
- [138] S. Khan. LL-XSS: End-to-End Generative Model-based XSS Payload Creation. In *2024 21st Learning and Technology Conference (L&T)*, pages 121–126, Jan. 2024.
- [139] A. Kieyzun, P. Guo, K. Jayaraman, and M. Ernst. Automatic creation of sql injection and cross-site scripting attacks. In *2009 IEEE 31st International Conference on Software Engineering*, 2009.
- [140] M. Kim, D. Corradini, S. Sinha, A. Orso, M. Pasqua, R. Tzoref-Brill, and M. Ceccato. Enhancing REST API Testing with NLP Techniques. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1232–1243, Seattle WA USA, July 2023. ACM.
- [141] M. Kim, S. Sinha, and A. Orso. Adaptive REST API Testing with Reinforcement Learning, Sept. 2023. arXiv:2309.04583 [cs].
- [142] M. Kim, Q. Xin, S. Sinha, and A. Orso. Automated test generation for REST APIs: no time to rest yet. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2022, pages 289–301, New York, NY, USA, July 2022. Association for Computing Machinery.
- [143] S. Kim, Y. M. Kim, J. Hur, S. Song, G. Lee, and B. Lee. FuzzOrigin: Detecting UXSS vul-

- nerabilities in browsers through origin fuzzing. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 1008–1023, Boston, MA, Aug. 2022. USENIX Association.
- [144] R. Kirchner, J. Möller, M. Musch, D. Klein, K. Rieck, and M. Johns. Dancer in the Dark: Synthesizing and Evaluating Polyglots for Blind {Cross-Site} Scripting. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 6723–6740, 2024.
- [145] KissPeter. APIFuzzer — HTTP API Testing Framework. <https://github.com/KissPeter/APIFuzzer>, 2018.
- [146] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks. Evaluating Fuzz Testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS '18*, pages 2123–2138, New York, NY, USA, Oct. 2018. Association for Computing Machinery.
- [147] A. Klein. DOM Based Cross Site Scripting or XSS of the Third Kind. *Web Application Security Consortium*, July 2005.
- [148] T. Kudo. Subword regularization: Improving neural network translation models with multiple subword candidates. In I. Gurevych and Y. Miyao, editors, *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 66–75, Melbourne, Australia, July 2018. Association for Computational Linguistics.
- [149] T. Kulkarni. vAPI. <https://github.com/roottusk/vapi>, Mar. 2023.
- [150] A. Kuznetsov, Y. Yeromin, O. Shapoval, K. Chernov, M. Popova, and K. Serdukov. Automated Software Vulnerability Testing Using Deep Learning Methods. In *2019 IEEE 2nd Ukraine Conference on Electrical and Computer Engineering (UKRCON)*, pages 837–841, July 2019.
- [151] Q. Le and T. Mikolov. Distributed Representations of Sentences and Documents. In *Proceedings of the 31st International Conference on Machine Learning*, 2014.
- [152] S. Lee, H. Han, S. K. Cha, and S. Son. Montage: A neural network language Model-Guided JavaScript engine fuzzer. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2613–2630. USENIX Association, Aug. 2020.
- [153] S. Lee, S. Wi, and S. Son. Link: Black-Box Detection of Cross-Site Scripting Vulnerabilities Using Reinforcement Learning. In *ACM Web Conference*, 2022.
- [154] X. Lei, J. Qu, G. Yao, J. Chen, and X. Shen. Design and Implementation of an Automatic Scanning Tool of SQL Injection Vulnerability Based on Web Crawler. In *Security with Intelligent Computing and Big-data Services*, 2020.
- [155] S. Lekies, B. Stock, and M. Johns. 25 million flows later: large-scale detection of DOM-based

- XSS. In *ACM SIGSAC Conference on Computer & Communications Security*, Berlin, Germany, 2013. ACM Press.
- [156] J. Li, B. Zhao, and C. Zhang. Fuzzing: a survey. *Cybersecurity*, 1(1):6, Dec. 2018.
- [157] K. Li, H. Yang, and W. Visser. DaNuoYi: Evolutionary Multi-Task Injection Testing on Web Application Firewalls. *IEEE Transactions on Software Engineering*, pages 1–21, 2023.
- [158] L. Li, W. Chu, J. Langford, and R. E. Schapire. A contextual-bandit approach to personalized news article recommendation. In *Proceedings of the 19th international conference on World wide web*, WWW '10, pages 661–670, New York, NY, USA, Apr. 2010. Association for Computing Machinery.
- [159] X. Li, X. Liu, L. Chen, R. Prajapati, and D. Wu. ALPHAPROG: Reinforcement Generation of Valid Programs for Compiler Fuzzing. *Proceedings of the AAAI Conference on Artificial Intelligence*, 36(11):12559–12565, June 2022.
- [160] X. Li, X. Liu, L. Chen, R. Prajapati, and D. Wu. FuzzBoost: Reinforcement Compiler Fuzzing. In *Information and Communications Security: 24th International Conference, ICICS 2022, Canterbury, UK, September 5–8, 2022, Proceedings*, 2022.
- [161] Z. Li, H. Zhao, J. Shi, Y. Huang, and J. Xiong. An Intelligent Fuzzing Data Generation Method Based on Deep Adversarial Learning. *IEEE Access*, 7:49327–49340, 2019.
- [162] H. Liang, X. Li, D. Xiao, J. Liu, Y. Zhou, A. Wang, and J. Li. Generative Pre-trained Transformer-Based Reinforcement Learning for Testing Web Application Firewalls. *IEEE Transactions on Dependable and Secure Computing*, pages 1–15, 2023.
- [163] E. Z. Liu, K. Guu, P. Pasupat, T. Shi, and P. Liang. Reinforcement Learning on Web Interfaces Using Workflow-Guided Exploration. *arXiv:1802.08802 [cs]*, Feb. 2018. International Conference on Learning Representations (ICLR).
- [164] M. Liu, K. Li, and T. Chen. DeepSQLi: deep semantic learning for testing SQL injection. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2020, pages 286–297. Association for Computing Machinery, July 2020.
- [165] X. Liu, X. Li, R. Prajapati, and D. Wu. DeepFuzz: Automatic Generation of Syntax Valid C Programs for Fuzz Testing. *Proceedings of the AAAI Conference on Artificial Intelligence*, 33(01):1044–1051, July 2019. Section: AAAI Technical Track: Applications.
- [166] X. Liu, W. Yu, F. Liang, D. Griffith, and N. Golmie. On deep reinforcement learning security for Industrial Internet of Things. *Computer Communications*, 168:20–32, Feb. 2021.

- [167] Y. Liu, Y. Li, G. Deng, Y. Liu, R. Wan, R. Wu, D. Ji, S. Xu, and M. Bao. Morest: Model-based RESTful API Testing with Execution Feedback, Apr. 2022. Number: arXiv:2204.12148 arXiv:2204.12148 [cs].
- [168] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov. RoBERTa: A Robustly Optimized BERT Pretraining Approach, 2019.
- [169] S. LLC. Arachni: Crawl coverage and vulnerability detection. <https://www.arachni-scanner.com/features/framework/crawl-coverage-vulnerability-detection/>, Sept. 2014.
- [170] S. M. Lundberg and S.-I. Lee. A unified approach to interpreting model predictions. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, 2017.
- [171] M. Luo, W. Xiong, G. Lee, Y. Li, X. Yang, A. Zhang, Y. Tian, H.-H. S. Lee, and G. E. Suh. AutoCAT: Reinforcement Learning for Automated Exploration of Cache-Timing Attacks. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 317–332, Feb. 2023. ISSN: 2378-203X.
- [172] S. Luo, A. Herrera, P. Quirk, M. Chase, D. C. Ranasinghe, and S. S. Kanhere. Make out like a (Multi-Armed) Bandit: Improving the Odds of Fuzzer Seed Scheduling with T-Scheduler. In *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*, ASIA CCS '24, pages 1463–1479, New York, NY, USA, July 2024. Association for Computing Machinery.
- [173] Y. Luo. SQLi-Fuzzer: A SQL Injection Vulnerability Discovery Framework Based on Machine Learning. In *2021 IEEE 21st International Conference on Communication Technology (ICCT)*, 2021.
- [174] C. Lyu, S. Ji, C. Zhang, Y. Li, W.-H. Lee, Y. Song, and R. Beyah. MOPT: Optimized Mutation Scheduling for Fuzzers. In *Proceedings of the 28th USENIX Security Symposium (USENIX Security)*, pages 1949–1966, 2019.
- [175] C. Lyu, S. Ji, X. Zhang, H. Liang, B. Zhao, K. Lu, and R. Beyah. EMS: History-Driven Mutation for Coverage-based Fuzzing. In *Proceedings 2022 Network and Distributed System Security Symposium*, San Diego, CA, USA, 2022. Internet Society.
- [176] C. Lyu, J. Xu, and S. Ji. MINER: A Hybrid Data-Driven Approach for REST API Fuzzing. In *Proceedings of the 32nd USENIX Security Symposium*, 2023.

- [177] R. Maeda and M. Mimura. Automating post-exploitation with deep reinforcement learning. *Computers & Security*, 100:102108, Jan. 2021.
- [178] J. Magazinius, B. K. Rios, and A. Sabelfeld. Polyglots: crossing origins by crossing formats. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security, CCS '13*, pages 753–764, New York, NY, USA, Nov. 2013. Association for Computing Machinery.
- [179] Z. Marashdeh, K. Suwais, and M. Alia. A Survey on SQL Injection Attack: Detection and Challenges. In *2021 International Conference on Information Technology (ICIT)*, 2021.
- [180] A. Martin-Lopez, S. Segura, C. Muller, and A. Ruiz-Cortes. Specification and Automated Analysis of Inter-Parameter Dependencies in Web APIs. *IEEE Transactions on Services Computing*, 2021.
- [181] B. Mathis, R. Gopinath, M. Mera, A. Kampmann, M. Höschel, and A. Zeller. Parser-directed fuzzing. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019*, page 548–560, New York, NY, USA, 2019. Association for Computing Machinery.
- [182] B. Mathis, R. Gopinath, and A. Zeller. Learning input tokens for effective fuzzing. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2020*, page 27–37, New York, NY, USA, 2020. Association for Computing Machinery.
- [183] R. Mazon, L. Levy, Y. Nizry, and G. Lyuboshits. Checkmarx/capital. <https://github.com/Checkmarx/capital>, Oct. 2024.
- [184] S. McFadden, M. Maugeri, C. Hicks, V. Mavroudis, and F. Pierazzi. WENDIGO: Deep Reinforcement Learning for Denial-of-Service Query Discovery in GraphQL. In *IEEE Workshop on Deep Learning Security and Privacy (DLSP)*, Apr. 2024.
- [185] R. Meng, M. Mirchev, M. Böhme, and A. Roychoudhury. Large Language Model guided Protocol Fuzzing. In *Proceedings 2024 Network and Distributed System Security Symposium*, San Diego, CA, USA, 2024. Internet Society.
- [186] S. Miao, J. Wang, C. Zhang, Z. Lin, J. Gong, X. Zhang, and J. Li. Deep Learning in Fuzzing: A Literature Survey. In *2022 IEEE 2nd International Conference on Electronic Technology, Communication and Information (ICETCI)*, pages 220–223, May 2022.
- [187] A. Mirhoseini, A. Goldie, M. Yazgan, J. W. Jiang, E. Songhori, S. Wang, Y.-J. Lee, E. Johnson, O. Pathak, A. Nova, J. Pak, A. Tong, K. Srinivasa, W. Hang, E. Tuncer, Q. V. Le, J. Laudon,

- R. Ho, R. Carpenter, and J. Dean. A graph placement methodology for fast chipdesign. *Nature*, 594(7862):207–212, June 2021.
- [188] MITRE. CWE - 2024 CWE Top 25 Most Dangerous Software Weaknesses, Nov. 2024.
- [189] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu. Asynchronous Methods for Deep Reinforcement Learning. In *International Conference on Machine Learning*, pages 1928–1937. PMLR, June 2016. ISSN: 1938-7228.
- [190] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing Atari with Deep Reinforcement Learning. *arXiv:1312.5602 [cs]*, Dec. 2013. arXiv: 1312.5602.
- [191] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, Feb. 2015.
- [192] M. Mohammadi, B. Chu, and H. R. Lipford. Detecting Cross-Site Scripting Vulnerabilities through Automated Unit Testing. In *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, 2017.
- [193] M. Mohammadi, B. Chu, H. R. Lipford, and E. Murphy-Hill. Automatic web security unit testing: XSS vulnerability detection. In *Proceedings of the 11th International Workshop on Automation of Software Test - AST '16*, 2016.
- [194] M. Muench, J. Stijohann, F. Kargl, A. Francillon, and D. Balzarotti. What You Corrupt Is Not What You Crash: Challenges in Fuzzing Embedded Devices. In *Proceedings 2018 Network and Distributed System Security Symposium*, San Diego, CA, 2018. Internet Society.
- [195] N. Munaiah, F. Camilo, W. Wigham, A. Meneely, and M. Nagappan. Do bugs foreshadow vulnerabilities? An in-depth study of the chromium project. *Empirical Software Engineering*, 22(3):1305–1347, June 2017.
- [196] C. Nagy and A. Cleve. A Static Code Smell Detector for SQL Queries Embedded in Java Code. In *2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2017.
- [197] A. Y. Ng and S. J. Russell. Algorithms for Inverse Reinforcement Learning. In *Proceedings of the Seventeenth International Conference on Machine Learning, ICML '00*, pages 663–670, San Francisco, CA, USA, June 2000. Morgan Kaufmann Publishers Inc.

- [198] T. D. Nguyen, S. Marchal, M. Miettinen, H. Fereidooni, N. Asokan, and A.-R. Sadeghi. D\IoT: A Federated Self-learning Anomaly Detection System for IoT. *arXiv:1804.07474 [cs]*, May 2019. arXiv: 1804.07474.
- [199] T. T. Nguyen and V. J. Reddi. Deep Reinforcement Learning for Cyber Security. *IEEE Transactions on Neural Networks and Learning Systems*, pages 1–17, 2021. arXiv: 1906.05799.
- [200] A. Nika, S. Elahi, and C. Tekin. Contextual Combinatorial Volatile Multi-armed Bandit with Adaptive Discretization. In *Proceedings of the Twenty Third International Conference on Artificial Intelligence and Statistics*, pages 1486–1496. PMLR, June 2020. ISSN: 2640-3498.
- [201] E. Nikishin, M. Schwarzer, P. D’Oro, P.-L. Bacon, and A. Courville. The Primacy Bias in Deep Reinforcement Learning. In *Proceedings of the 39th International Conference on Machine Learning*, pages 16828–16847. PMLR, June 2022. ISSN: 2640-3498.
- [202] NIST. NVD - Home. <https://nvd.nist.gov/>, 2024.
- [203] W. Niu, X. Zhang, X. Du, L. Zhao, R. Cao, and M. Guizani. A deep learning based static taint analysis approach for iot software vulnerability location. *Measurement*, 152:107139, 2020.
- [204] Nodiscc. Awesome-Selfhosted. <https://github.com/awesome-selfhosted/awesome-selfhosted>, 2022.
- [205] L. Ntagwabira and S. Kang. Use of query tokenization to detect and prevent sql injection attacks. In *2010 3rd International Conference on Computer Science and Information Technology*, volume 2, pages 438–440, 2010.
- [206] P. Oehlert. Violating assumptions with fuzzing. *IEEE Security & Privacy*, 3(2):58–62, Mar. 2005.
- [207] OpenAI. Dota 2 with Large Scale Deep Reinforcement Learning. *arXiv:1912.06680 [cs, stat]*, 2019.
- [208] OpenAI, C. Berner, G. Brockman, B. Chan, V. Cheung, P. Debiak, C. Dennison, D. Farhi, Q. Fischer, S. Hashme, C. Hesse, R. Józefowicz, S. Gray, C. Olsson, J. Pachocki, M. Petrov, H. P. d. O. Pinto, J. Raiman, T. Salimans, J. Schlatter, J. Schneider, S. Sidor, I. Sutskever, J. Tang, F. Wolski, and S. Zhang. Dota 2 with Large Scale Deep Reinforcement Learning. *arXiv:1912.06680 [cs, stat]*, Dec. 2019. arXiv: 1912.06680.
- [209] OpenAPI. Openapi initiative. <https://www.openapis.org/>, 2023.
- [210] OWASP. OWASP ZED ATTACK PROXY (ZAP). <https://www.zaproxy.org/>.

- [211] OWASP. OWASP Top 10 2013: The Ten Most Critical Web Application Security Risks. Technical report, OWASP, 2013.
- [212] OWASP. OWASP Top Ten Web Application Security Risks. <https://owasp.org/www-project-top-ten/2017>, 2017.
- [213] OWASP. OWASP API Security Top 10 2019RC, 2019.
- [214] OWASP. OWASP Top 10 Web Application Security Risks. <https://owasp.org/Top10/>, 2021.
- [215] OWASP. OWASP ZED ATTACK PROXY (ZAP). <https://www.zaproxy.org/>, 2021.
- [216] OWASP. SQL Injection | OWASP Foundation. https://owasp.org/www-community/attacks/SQL_Injection, 2022.
- [217] OWASP. Testing for sql injection. https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/07-Input_Validation_Testing/05-Testing_for_SQL_Injection, 2022.
- [218] OWASP. OWASP API Security Top 10 2023RC, 2023.
- [219] E. Parisotto, H. F. Song, J. W. Rae, R. Pascanu, C. Gulcehre, S. M. Jayakumar, M. Jaderberg, R. L. Kaufman, A. Clark, S. Noury, M. M. Botvinick, N. Heess, and R. Hadsell. Stabilizing Transformers for Reinforcement Learning. In *Proceedings of the 37th International Conference on Machine Learning*, 2020.
- [220] S. Park, W. Xu, I. Yun, D. Jang, and T. Kim. Fuzzing JavaScript Engines with Aspect-preserving Mutation. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1629–1642, May 2020. ISSN: 2375-1207.
- [221] G. D. Pasquale, I. Grishchenko, R. Iesari, G. Pizarro, L. Cavallaro, C. Kruegel, and G. Vigna. {ChainReactor}: Automated Privilege Escalation Chain Discovery via {AI} Planning. In *Proceedings of the 33rd USENIX Security Symposium*, pages 5913–5929, 2024.
- [222] D. Pathak, P. Agrawal, A. A. Efros, and T. Darrell. Curiosity-driven Exploration by Self-supervised Prediction, May 2017. arXiv:1705.05363 [cs, stat].
- [223] K. Paxton-Fear. Generic University. <https://github.com/InsiderPhD/Generic-University>, 2023.
- [224] G. Pellegrino, C. Tschürtz, E. Bodden, and C. Rossow. jÄk: Using Dynamic Analysis to Crawl and Test Modern Web Applications. In H. Bos, F. Monrose, and G. Blanc, editors, *Research in Attacks, Intrusions, and Defenses*, volume 9404, pages 295–316. Springer International Publishing, Cham, 2015. Series Title: Lecture Notes in Computer Science.

- [225] J. Pereyda. jtpereyda/boofuzz. <https://github.com/jtpereyda/boofuzz>, 2015.
- [226] A. Petukhov and D. Kozlov. Detecting Security Vulnerabilities in Web Applications Using Dynamic Analysis with Penetration Testing. In *OWASP Application Security Conference*, 2008.
- [227] V.-H. Pham, D. Thi Thu Hien, N. Phuc Chuong, P. Thanh Thai, and P. The Duy. A Coverage-Guided Fuzzing Method for Automatic Software Vulnerability Detection Using Reinforcement Learning-Enabled Multi-Level Input Mutation. *IEEE Access*, 12:129064–129080, 2024.
- [228] V.-T. Pham, M. Böhme, A. E. Santosa, A. R. Căciulescu, and A. Roychoudhury. Smart Greybox Fuzzing. *IEEE Transactions on Software Engineering*, 47(9):1980–1997, Sept. 2021.
- [229] T. Pietraszek and C. V. Berghe. Defending Against Injection Attacks Through Context-Sensitive String Evaluation. In *Recent Advances in Intrusion Detection*, 2006.
- [230] J. Portner, J. Kerr, and B. Chu. Moving Target Defense Against Cross-Site Scripting Attacks (Position Paper). In *International Symposium on Foundations and Practice of Security*, volume 8930, pages 85–91, Apr. 2015.
- [231] PortSwigger. Burp Suite - Application Security Testing Software. <https://portswigger.net/burp>, 2021.
- [232] PortSwigger. What is SQL Injection? Tutorial & Examples | Web Security Academy, 2022.
- [233] K. Prole. XSSMap. <https://github.com/secdec/xssmap>, 2021.
- [234] psy. epsylon/xsser. <https://github.com/epsylon/xsser>, 2021.
- [235] J. Qi, Q. Zhou, L. Lei, and K. Zheng. Federated Reinforcement Learning: Techniques, Applications, and Open Challenges, 2021.
- [236] R. Rabheru, H. Hanif, and S. Maffei. DeepTective: detection of PHP vulnerabilities using hybrid graph neural networks. In *Proceedings of the 36th Annual ACM Symposium on Applied Computing, SAC '21*, pages 1687–1690, New York, NY, USA, Apr. 2021. Association for Computing Machinery.
- [237] R. Rabheru, H. Hanif, and S. Maffei. A Hybrid Graph Neural Network Approach for Detecting PHP Vulnerabilities. In *2022 IEEE Conference on Dependable and Secure Computing (DSC)*, pages 1–9, June 2022.
- [238] H. Rathore, S. K. Sahay, P. Nikam, and M. Sewak. Robust Android Malware Detection System Against Adversarial Attacks Using Q-Learning. *Information Systems Frontiers*, Nov. 2020.
- [239] D. Raywood. XSS is Most Rewarding Bug Bounty as CSRF is Revived. *Infosecurity Magazine*, June 2019.

- [240] D. Rethans. Xdebug - Debugger and Profiler Tool for PHP. <https://xdebug.org/>, 2019.
- [241] A. Riancho. w3af. <http://w3af.org/>, 2007.
- [242] J. Ruge, F. Gringoli, J. Classen, and M. Hollick. Frankenstein: Advanced Wireless Fuzzing to Exploit New Bluetooth Escalation Targets. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, page 19. USENIX Association, 2020.
- [243] D. E. Rumelhart and J. L. McClelland. Learning Internal Representations by Error Propagation. In *Parallel Distributed Processing: Explorations in the Microstructure of Cognition: Foundations*, pages 318–362. MIT Press, 1987.
- [244] M. Sablotny. *Fuzzing software with deep learning*. PhD, University of Glasgow, 2023.
- [245] M. Sablotny, B. S. Jensen, and C. W. Johnson. Recurrent Neural Networks for Fuzz Testing Web Browsers. In K. Lee, editor, *Information Security and Cryptology – ICISC 2018*, Lecture Notes in Computer Science, pages 354–370, Cham, Jan. 2019. Springer International Publishing.
- [246] M. Sablotny, B. S. Jensen, and J. Singer. Reinforcement learning guided fuzz testing for a browser’s HTML rendering engine, July 2023. arXiv:2307.14556 [cs].
- [247] Safedog. Safedog. http://free.safedog.cn/website_safedog.html.
- [248] C. Salls, C. Jindal, J. Corina, C. Kruegel, and G. Vigna. Token-Level Fuzzing. In *Proceedings of the 30th USENIX Security Symposium (USENIX Security 21)*, pages 2795–2809, 2021.
- [249] T. Schaul, J. Quan, I. Antonoglou, and D. Silver. Prioritized Experience Replay. *arXiv:1511.05952 [cs]*, 2016.
- [250] M. Schloegel, N. Bars, N. Schiller, L. Bernhard, T. Scharnowski, A. Crump, A. Ale-Ebrahim, N. Bissantz, M. Muench, and T. Holz. SoK: Prudent Evaluation Practices for Fuzzing. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 1974–1993, May 2024. ISSN: 2375-1207.
- [251] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal Policy Optimization Algorithms. *arXiv:1707.06347*, 2017.
- [252] M. Schuster and K. Nakajima. Japanese and korean voice search. In *International Conference on Acoustics, Speech and Signal Processing*, pages 5149–5152, 2012.
- [253] R. Sennrich, B. Haddow, and A. Birch. Neural Machine Translation of Rare Words with Subword Units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2016.
- [254] K. Sethi, R. Kumar, D. Mohanty, and P. Bera. Robust Adaptive Cloud Intrusion Detection System Using Advanced Deep Reinforcement Learning. In L. Batina, S. Picek, and M. Mondal,

- editors, *Security, Privacy, and Applied Cryptography Engineering*, Lecture Notes in Computer Science, pages 66–85, Cham, 2020. Springer International Publishing.
- [255] L. K. Shar and H. B. K. Tan. Defeating SQL Injection. *Computer*, 46, 2013.
- [256] D. She, Y. Chen, A. Shah, B. Ray, and S. Jana. Neutaint: Efficient Dynamic Taint Analysis with Neural Networks. *arXiv:1907.03756 [cs]*, Sept. 2019. arXiv: 1907.03756.
- [257] Y. Shen, E. Mariconti, P. A. Vervier, and G. Stringhini. Tiresias: Predicting Security Events Through Deep Learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 592–605, Toronto Canada, Oct. 2018. ACM.
- [258] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, Jan. 2016.
- [259] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. v. d. Driessche, T. Graepel, and D. Hassabis. Mastering the game of Go without human knowledge. *Nature*, 550(7676):354–359, Oct. 2017. Number: 7676 Publisher: Nature Publishing Group.
- [260] J. Skalse, N. Howe, D. Krasheninnikov, and D. Krueger. Defining and Characterizing Reward Gaming. *Advances in Neural Information Processing Systems*, 35:9460–9471, Dec. 2022.
- [261] S. Somasundaram. Successful Real-World Use Cases For LLMs (And Lessons They Teach). *Forbes*, Forbes Technology Council, Mar. 2024.
- [262] A. A. Sommervoll, L. Erdödi, and F. M. Zennaro. Simulating all archetypes of SQL injection vulnerability exploitation using reinforcement learning agents. *International Journal of Information Security*, 23(1):225–246, Feb. 2024.
- [263] D. Song, F. Hetzelt, D. Das, C. Spensky, Y. Na, S. Volckaert, G. Vigna, C. Kruegel, J.-P. Seifert, and M. Franz. PeriScope: An Effective Probing and Fuzzing Framework for the Hardware-OS Boundary. In *Proceedings 2019 Network and Distributed System Security Symposium*, San Diego, CA, 2019. Internet Society.
- [264] W. Song, X. Li, S. Afroz, D. Garg, D. Kuznetsov, and H. Yin. MAB-Malware: A Reinforcement Learning Framework for Attacking Static Malware Classifiers. *arXiv:2003.03100 [cs]*, Apr. 2021. arXiv: 2003.03100.

- [265] X. Song, R. Zhang, Q. Dong, and B. Cui. Grey-Box Fuzzing Based on Reinforcement Learning for XSS Vulnerabilities. *Applied Sciences*, 13(4):2482, Jan. 2023. Number: 4 Publisher: Multidisciplinary Digital Publishing Institute.
- [266] H. Spieker, A. Gotlieb, D. Marijan, and M. Mossige. Reinforcement learning for automatic test case prioritization and selection in continuous integration. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2017, 2017.
- [267] N. Srivastava, E. Mansimov, and R. Salakhutdinov. Unsupervised learning of video representations using lstms. *CoRR*, abs/1502.04681, 2015.
- [268] P. Srivastava, S. Nagy, M. Hicks, A. Bianchi, and M. Payer. One Fuzz Doesn't Fit All: Optimizing Directed Fuzzing via Target-tailored Program State Restriction. In *Proceedings of the 38th Annual Computer Security Applications Conference*, ACSAC '22, pages 388–399, New York, NY, USA, Dec. 2022. Association for Computing Machinery.
- [269] R. K. Srivastava, P. Shyam, F. Mutz, W. Jaśkowski, and J. Schmidhuber. Training Agents using Upside-Down Reinforcement Learning, Sept. 2021. arXiv:1912.02877.
- [270] A. Stasinopoulos, C. Ntantogian, and C. Xenakis. Commix: automating evaluation and exploitation of command injection vulnerabilities in Web applications. *International Journal of Information Security*, 18(1):49–72, Feb. 2019.
- [271] Z. Su and G. 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*, 2006.
- [272] S.-T. Sun and K. Beznosov. Retrofitting Existing Web Applications with Effective Dynamic Protection Against SQL Injection Attacks. *International Journal of Secure Software Engineering (IJSSE)*, 1, 2010.
- [273] N. Surribas. Wapiti. <https://wapiti.sourceforge.io/>, 2021.
- [274] R. Sutton. The Bitter Lesson. https://www.cs.utexas.edu/~eunsol/courses/data/bitter_lesson.pdf, Mar. 2019.
- [275] R. S. Sutton and A. G. Barto. *Reinforcement learning: an introduction*. Adaptive computation and machine learning series. The MIT Press, second edition edition, 2018.
- [276] swisskyrepo. Payloads all the things. <https://github.com/swisskyrepo/PayloadsAllTheThings>, 2022.

- [277] P. Tang, W. Qiu, Z. Huang, H. Lian, and G. Liu. Detection of SQL injection based on artificial neural network. *Knowledge-Based Systems*, 190:105528, Feb. 2020.
- [278] I. Tasdelen. payloadbox/sql-injection-payload-list. <https://github.com/payloadbox/sql-injection-payload-list>, 2022.
- [279] P. Touseef, K. A. Alam, A. Jamil, H. Tauseef, S. Ajmal, R. Asif, B. Rehman, and S. Mustafa. Analysis of Automated Web Application Security Vulnerabilities Testing. In *Proceedings of the 3rd International Conference on Future Networks and Distributed Systems*, ICFNDS '19, pages 1–8, New York, NY, USA, July 2019. Association for Computing Machinery.
- [280] I. Tsingenopoulos, D. Preuveneers, L. Desmet, and W. Joosen. Captcha me if you can: Imitation Games with Reinforcement Learning. In *2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P)*, 2022.
- [281] I. Tsingenopoulos, D. Preuveneers, and W. Joosen. AutoAttacker: A reinforcement learning approach for black-box adversarial attacks. In *2019 IEEE European Symposium on Security and Privacy Workshops (EuroS PW)*, pages 229–237, June 2019.
- [282] S. Ullah, M. Han, S. Pujar, H. Pearce, A. Coskun, and G. Stringhini. LLMs Cannot Reliably Identify and Reason About Security Vulnerabilities (Yet?): A Comprehensive Evaluation, Framework, and Benchmarks . In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 862–880, Los Alamitos, CA, USA, May 2024. IEEE Computer Society.
- [283] Unicorn. Unicorn – The Ultimate CPU emulator. <https://www.unicorn-engine.org/>.
- [284] F. Valeur, D. Mutz, and G. Vigna. A Learning-Based Approach to the Detection of SQL Attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, 2005.
- [285] H. van Hasselt, A. Guez, and D. Silver. Deep Reinforcement Learning with Double Q-Learning. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence (AAAI-16)*, page 7. Association for the Advancement of ArtificialIntelligence, Feb. 2016.
- [286] O. van Rooij, M. A. Charalambous, D. Kaizer, M. Papaevripides, and E. Athanasopoulos. webFuzz: Grey-Box Fuzzing for Web Applications. In *Computer Security – ESORICS 2021*, volume 12972, 2021.
- [287] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is All you Need. In *31st Conference on Neural Information Processing System*, 2017.
- [288] S. Veggalam, S. Rawat, I. Haller, and H. Bos. IFuzzer: An evolutionary interpreter fuzzer using

- genetic programming. In *Proceedings of the European Symposium on Research in Computer Securit*, pages 581–601, 2016.
- [289] P. J. P. Vieito. pvieito/PythonKit. <https://github.com/pvieito/PythonKit>, Oct. 2024.
- [290] V. Vikram, I. Laybourn, A. Li, N. Nair, K. OBrien, R. Sanna, and R. Padhye. Guiding grey-box fuzzing with mutation testing. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2023, page 929–941, New York, NY, USA, 2023. Association for Computing Machinery.
- [291] O. Vinyals, M. Fortunato, and N. Jaitly. Pointer Networks. *arXiv:1506.03134 [cs, stat]*, Jan. 2017. arXiv: 1506.03134.
- [292] S. Vyas, J. Hannay, A. Bolton, and P. P. Burnap. Automated Cyber Defence: A Review, Mar. 2023. arXiv:2303.04926 [cs].
- [293] S. Vyas, C. Hicks, and V. Mavroudis. Mitigating Deep Reinforcement Learning Backdoors in the Neural Activation Space. In *2024 IEEE Security and Privacy Workshops (SPW)*, pages 76–86, May 2024. ISSN: 2770-8411.
- [294] D. Wang, Z. Zhang, H. Zhang, Z. Qian, S. V. Krishnamurthy, and N. Abu-Ghazaleh. {SyzVegas}: Beating Kernel Fuzzing Odds with Reinforcement Learning. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2741–2758, 2021.
- [295] J. Wang, B. Chen, L. Wei, and Y. Liu. Skyfire: Data-Driven Seed Generation for Fuzzing. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 579–594, San Jose, CA, USA, May 2017. IEEE.
- [296] J. Wang, B. Chen, L. Wei, and Y. Liu. Superion: Grammar-Aware Greybox Fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 724–735, Montreal, QC, Canada, May 2019. IEEE.
- [297] J. Wang, Y. Kang, C. Wu, Y. Hu, Y. Sun, J. Ren, Y. Lai, M. Xie, C. Zhang, T. Li, and Z. Wang. {OptFuzz}: Optimization Path Guided Fuzzing for {JavaScript} {JIT} Compilers. In *Proceedings of the 33rd USENIX Security Symposium (USENIX Security 24)*, pages 865–882, 2024.
- [298] J. Wang, C. Song, and H. Yin. Reinforcement Learning-based Hierarchical Seed Scheduling for Greybox Fuzzing. In *Proceedings 2021 Network and Distributed System Security Symposium*, page 17, Feb. 2021.
- [299] J. Wang, Z. Zhang, S. Liu, X. Du, and J. Chen. FuzzJIT: Oracle-Enhanced Fuzzing for JavaScript

- Engine JIT Compiler. In *Proceedings of the 32nd USENIX Security Symposium*. USENIX Association, 2023.
- [300] S. Wang, L. Huang, A. Gao, J. Ge, T. Zhang, H. Feng, I. Satyarth, M. Li, H. Zhang, and V. Ng. Machine/Deep Learning for Software Engineering: A Systematic Literature Review. *IEEE Transactions on Software Engineering*, 49(3):1188–1231, Mar. 2023.
- [301] X. Wang and H. Hu. Evading Web Application Firewalls with Reinforcement Learning. <https://openreview.net/pdf?id=m5AntlhJ7Z5>, 2020.
- [302] Y. Wang, P. Jia, L. Liu, C. Huang, and Z. Liu. A systematic review of fuzzing based on machine learning techniques. *PLOS ONE*, 15(8):e0237749, Aug. 2020.
- [303] Y. Wang, Z. Wu, Q. Wei, and Q. Wang. NeuFuzz: Efficient Fuzzing With Deep Neural Network. *IEEE Access*, 7:36340–36352, 2019.
- [304] Z. Wang, T. Schaul, M. Hessel, H. van Hasselt, M. Lanctot, and N. de Freitas. Dueling Network Architectures for Deep Reinforcement Learning. In *Proceedings of the 33rd International Conference on Machine Learning*, page 9, New York, NY, USA, 2016.
- [305] F. R. Ward, M. MacDermott, F. Belardinelli, F. Toni, and T. Everitt. The Reasons that Agents Act: Intention and Instrumental Goals, Feb. 2024. arXiv:2402.07221.
- [306] F. Weissberg, J. Möller, T. Ganz, E. Imgrund, L. Pirch, L. Seidel, M. Schloegel, T. Eisenhofer, and K. Rieck. SoK: Where to Fuzz? Assessing Target Selection Methods in Directed Fuzzing. In *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*, ASIA CCS ’24, pages 1539–1553, New York, NY, USA, July 2024. Association for Computing Machinery.
- [307] WordPress. Source code of WordPress/wp-includes/meta.php. <https://github.com/WordPress/WordPress/blob/master/wp-includes/meta.php#L434>, 2023.
- [308] Y. Wu, E. Mansimov, S. Liao, A. Radford, and J. Schulman. OpenAI Baselines: ACKTR & A2C. <https://openai.com/blog/baselines-acktr-a2c/>, Aug. 2017.
- [309] P. R. Wurman, S. Barrett, K. Kawamoto, J. MacGlashan, K. Subramanian, T. J. Walsh, R. Capobianco, A. Devlic, F. Eckert, F. Fuchs, L. Gilpin, P. Khandelwal, V. Kompella, H. Lin, P. MacAlpine, D. Oller, T. Seno, C. Sherstan, M. D. Thomure, H. Aghabozorgi, L. Barrett, R. Douglas, D. Whitehead, P. Dürr, P. Stone, M. Spranger, and H. Kitano. Outracing champion Gran Turismo drivers with deep reinforcement learning. *Nature*, 602(7896):223–228, Feb. 2022. Publisher: Nature Publishing Group.

- [310] P.-S. Xie, S. Wang, Y.-W. Zhao, W.-J. Shao, W. Li, and T. Feng. Security Situation Prediction Method of Industrial Control System Based on Self-Attention and GRU Neural Network. *International Journal of Network Security*, 25(5):729–735, 2023.
- [311] Y. Yao, J. He, T. Li, Y. Wang, X. Lan, and Y. Li. An Automatic XSS Attack Vector Generation Method Based on the Improved Dueling DDQN Algorithm. *IEEE Transactions on Dependable and Secure Computing*, 21(4):2852–2868, July 2024.
- [312] M. Yousefi, N. Mtetwa, Y. Zhang, and H. Tianfield. A Reinforcement Learning Approach for Attack Graph Analysis. In *2018 17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/ 12th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*, pages 212–217, New York, NY, USA, Aug. 2018. IEEE.
- [313] J. Yu, W. Guo, Q. Qin, G. Wang, T. Wang, and X. Xing. AIRS: Explanation for Deep Reinforcement Learning based Security Applications. In *Proceedings of the 32nd USENIX Security Symposium (USENIX Security 23)*, 2023.
- [314] T. Yue, P. Wang, Y. Tang, E. Wang, B. Yu, K. Lu, and X. Zhou. EcoFuzz: Adaptive Energy-Saving Greybox Fuzzing as a Variant of the Adversarial Multi-Armed Bandit. In *Proceedings of the 29th USENIX Security Symposium*, pages 2307–2324, 2020.
- [315] M. Zalewski. American Fuzzy Lop. <https://lcamtuf.coredump.cx/afl/>, 2007.
- [316] C. Zhang, O. Vinyals, R. Munos, and S. Bengio. A Study on Overfitting in Deep Reinforcement Learning, Apr. 2018. arXiv:1804.06893.
- [317] G. Zhang, X. Zhou, Y. Luo, X. Wu, and E. Min. PTfuzz: Guided Fuzzing With Processor Trace Feedback. *IEEE Access*, 6:37302–37313, 2018.
- [318] L. Zhang, D. Zhang, C. Wang, J. Zhao, and Z. Zhang. ART4SQLi: The ART of SQL Injection Vulnerability Discovery. *IEEE Transactions on Reliability*, 68, 2019.
- [319] Z. Zhang, B. Cui, and C. Chen. Reinforcement Learning-Based Fuzzing Technology. In L. Barolli, A. Poniszewska-Maranda, and H. Park, editors, *Innovative Mobile and Internet Services in Ubiquitous Computing*, pages 244–253, Cham, 2021. Springer International Publishing.
- [320] J. Zhao, T. Dong, Y. Cheng, and Y. Wang. CMM: A Combination-Based Mutation Method for SQL Injection. In *Structured Object-Oriented Formal Language and Method*, 2020.
- [321] J. Zhao, W. Zhou, T. Zhao, Y. Zhou, and H. Li. State Representation Learning For Effective

- Deep Reinforcement Learning. In *2020 IEEE International Conference on Multimedia and Expo (ICME)*, pages 1–6, July 2020. ISSN: 1945-788X.
- [322] Y. Zhao, X. Wang, L. Zhao, Y. Cheng, and H. Yin. Alphuzz: Monte Carlo Search on Seed-Mutation Tree for Coverage-Guided Fuzzing. In *Proceedings of the 38th Annual Computer Security Applications Conference, ACSAC '22*, pages 534–547, New York, NY, USA, Dec. 2022. Association for Computing Machinery.
- [323] Y. Zheng, Y. Liu, X. Xie, Y. Liu, L. Ma, J. Hao, and Y. Liu. Automatic Web Testing Using Curiosity-Driven Reinforcement Learning. In *Proceedings of the 43rd International Conference on Software Engineering, ICSE '21*, pages 423–435, Madrid, Spain, Nov. 2021. IEEE Press.
- [324] D. Zhou, L. Li, and Q. Gu. Neural contextual bandits with UCB-based exploration. In *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *ICML '20*, pages 11492–11502, 2020.