APIRL: Deep Reinforcement Learning for REST API Fuzzing

Myles Foley[1][2], **Sergio Maffeis**[1]

[1]Department of Computing, Imperial College London
[2]The Alan Turing Institute
m.foley20@imperial.ac.uk, sergio.maffeis@imperial.ac.uk

Abstract

REST APIs have become key components of web services. However, they often contain logic flaws resulting in server side errors or security vulnerabilities. HTTP requests are used as test cases to find and mitigate such issues. Existing methods to modify requests, including those using deep learning, suffer from limited performance and precision, relying on undirected search or making limited usage of the contextual information. In this paper we propose APIRL, a fully automated deep reinforcement learning tool for testing REST APIs. A key novelty of our approach is the use of feedback from a transformer module pre-trained on JSON-structured data, akin to that used in API responses. This allows APIRL to learn the subtleties relating to test outcomes, and generalise to unseen API endpoints. 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 also study how reward functions, and other key design choices, affect learnt policies with a thorough ablation study.

1 Introduction

REpresentational State Transfer (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 the Hyper Text Transfer Protocol (HTTP), and a series of associated parameters. Atlidakis et al. (2019) 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. Hence, it is crucial to test the robustness of REST APIs using sophisticated techniques.

Automated software testing finds bugs or vulnerabilities in an application by detecting abnormal behaviour. This automated process can be referred to as fuzz testing (fuzzing), security testing, or robustness testing. For REST APIs testing, this involves creating new HTTP request test cases or mutating existing templates. Often, this is done at random or following some predefined heuristics. A common criterion to practically estimate performance is code coverage. However, as only a small portion of code contains bugs, it can lead to a high number of executed tests.

Black-box testing has been applied to automatically generate test cases for REST APIs (Atlidakis et al. 2020; Liu et al. 2022). While this has lead to improvements in REST API testing, such approaches lack targeted search strategies, or do not harness contextual information. This limits the potential of frameworks due to the unique behaviour of endpoints, their diversity, and varying schema requirements. As a result software testing models can be inefficient, requiring a very large number of test cases to find bugs.

Recent research used attention-based neural networks to predict mutations in test cases (Lyu, Xu, and Ji 2023). This is a promising approach for REST API testing. Yet, current solutions often only use simplistic feedback from HTTP status codes to determine success, while the main body of HTTP responses is either discarded, or only used for populating data when testing (Corradini et al. 2022b; Liu et al. 2022).

Reinforcement Learning (RL) has shown potential in automated testing, and has been successful learning policies to test compilers (Li et al. 2022a) and web applications (Lee, Wi, and Son 2022; Zheng et al. 2021). These works have demonstrated that using off-the-shelf RL methods such as Deep Q-Networks (DQN) (2015) and Proximal Policy Optimisation (PPO) (2017) can harness feedback to find more bugs, and improve efficiency. However, similar to REST API testing, RL-based software testing often uses simple heuristics as feedback (Lee, Wi, and Son 2022; Li et al. 2022a), which may fail to fully capture the subtleties of the problem. While work from Kim et al. (2023) has shown that RL can find bugs in REST APIs, it suffers from the same issue: not harnessing contextual information for feedback. The key challenge to do so arises from the variation across different APIs and the diversity in individual endpoint responses. Yet, if the information-rich data structure can be used for test case generation, it could provide valuable feedback. To address this challenge, we develop a novel deep RL agent that mutates HTTP requests to find bugs in REST APIs.

Our RL problem formulation uses a transformer architecture we pre-train to harness JSON and natural language in the HTTP responses, providing feedback to facilitate adaptation to operations after training. We then introduce a Markov

Copyright © 2025, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

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 (Lee, Wi, and Son 2022).

In summary, the main contributions of this paper are:

- APIRL, a novel deep reinforcement learning based approach to testing REST APIs that learns how to manipulate varied HTTP requests to efficiently target bugs. We release APIRL at https://github.com/ICL-ml4csec/APIRL.
- A novel representation of the feedback obtained from the API, combining a direct functional representation of fixed outputs and a transformer-based embedding of variable-length responses. This enables our RL agent to benefit from richer information than in previous work.
- Insights on the subtleties of training an RL agent for real-world tasks via an ablation study of both design choices and 7 reward functions.
- The evaluation of APIRL across 26 REST APIs shows significant improvement over state-of-the-art methods in terms of bugs found, coverage, and test case efficiency.

2 Overview

APIRL is a new testing approach based on deep Q-learning that mutates HTTP requests to find bugs in REST APIs, indicated by 5XX response codes. We represent REST API testing as an MDP for a deep RL agent: Figure 2 shows the process and agent architecture. At a high level, an agent takes actions to mutate HTTP *request templates*, which the environment implements as API operations, receiving feedback 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 3.3). Using diverse and variable length features such as HTTP headers and body data, a neural network selects a corresponding action a_t from Table 1 to mutate the HTTP request.

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 3.4 and 4.5). 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.

3 Design

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

```
"/users/v1/{username}/email":
    put:
      parameters:
        - name: username
          in: path
          required: true
          type: string
      requestBody:
        content:
          application/json:
            schema:
              type: object
              properties:
                email:
                  type: string
        required: true
      responses:
        '204':
          content: {}
```

Figure 1: Part of the OpenAPI specification for VAmPI

3.1 Preprocessing

The OpenAPI Specification is the standard for describing REST APIs. It specifies the URL and HTTP method (`POST`, `GET`, `PUT`, `PATCH`, and `DELETE`) to form an *operation*. This specification is also is the standard starting point for REST API fuzzing frameworks (Liu et al. 2022). 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 1 would have a request template such as: (`PUT`, `/users/v1/{username}/email`, `{name:username, in:path, required:true, type:string}`). In Figure 1 the HTTP response code specified of ‘204’, indicates a successful request. However, the response could be in the range 400-499 (4XX) codes which handle bad requests gracefully, or 500-599 (5XX) indicating a server error or bug. In line with prior work, we define bugs as 5XX responses which result in unique traces. (Kim, Sinha, and Orso 2023; Corradini et al. 2022a; Arcuri 2021).

We also find related parameters to form valid requests, as recommended by the literature (Martin-Lopez et al. 2021). For more on this see Appendix A.

3.2 Actions

We provide the RL agent with a fixed action space of 23 actions a_t used in prior work (Atlidakis, Godefroid, and Polishchuk 2019; Barabanov et al. 2022; Corradini et al. 2022b). We reimplement actions in the APIRL framework using distinct values where possible, otherwise we provide concrete definitions (Table 1). Actions are performed on requests, independently 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 1, and can be broadly devided 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

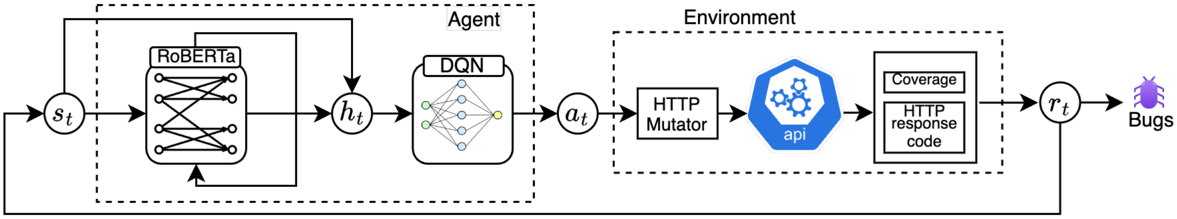

Figure 2: The REST API testing process using APIRL.

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 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 3.4). 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.

3.3 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 (Liu et al. 2019) transformer model. APIRL's transformer takes as input the HTTP response from test cases to produce a latent representation. APIRL can then 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), h_t in Figure 2. 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 vectors are combined (o_t in Figure 2) into a single vector of 772 (768 + 4) features and passed to the DQN. See Section 4.1 for further details on pre-training the transformer.

3.4 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 fuzzing, 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. 1). We will consider alternative rewards, including coverage, in an ablation study (Section 4.5).

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

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 (Sutton and Barto 2018). 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.

3.5 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 (DQN) (Mnih et al. 2015) with *Prioritised experience replay* (Schaul et al. 2016). We implement 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 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 B.

4 Evaluation

4.1 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. This form of curriculum learning prevents over-fitting by dividing training equally across diverse operations (Wahaibi, Foley, and Maffeis 2023). Specifically, APIRL is trained using an open-source REST API containing known bugs: Generic University. APIRL trains on each of Generic University's operations for 10,000 episodes, with the maximum steps per episode of 10 (Appendix B).

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	`Int`, `String`, `Bool`, `Array`, `Object`	Randomly change the parameter value type to a different one.
Duplicate Parameter	5	`{parameter1:[value1, value1]}`	Duplicate the parameter value.
Remove Parameter	6	`{}`	Remove the parameter from the request template.
Extension	7- 9	`.txt`, `.pdf`, `.doc`	Append a file extension to the value of the parameter.
Append	10	`{parameter1:[value1, newValue]}`	Convert the parameter to a JSON array and append an additional value from observed values.
Request Method	11	`POST` → `PUT`, `PUT`→ `POST`	Change HTTP method from `PUT` to `POST` or `POST` to `PUT`.
Add Parameter	12	`{parameter1:value1, newParameter:newValue}`	Add an additional parameter to the request template from parameters related to the parameter.
Wildcard	13-15	`*`, `.*`, `%`	Append a wildcard to the value of the parameter.
Change ID number	16-17	`1`, `-1`	Increment the `Int` value of the value by 1.
Set Parameter value	18-21	`'admin'`, `-1`, `999999999` ,`''`	Set the parameter value to a default value.
Set Existing Value	22	`{parameter1:value2}`	Set the parameter value to a related value from the API.
Set admin	23	`{parameter1:value1, admin:TRUE}`	Set `admin: True` in an `Object`.

Table 1: Mutation actions that can be applied by APIRL to the current operation.

We pre-train a RoBERTa transformer using HTTP responses from 103 different REST APIs, comprising 1283 operations. This has several advantages: limiting the bias from HTTP responses in training, potential for overfitting, and reduces instability in training the RL agent (Parisotto et al. 2020). API specifications were taken from a public OpenAPI specification platform (see Appendix C). A vocabulary of 52,000 tokens is formed via Byte-Pair Encoding (Gage 1994). The transformer is trained by masked language modeling (2019). Through training the transformer learns relationships between parameters, to provide a meaningful, generalised, embedding for the agent (Adolphs and Hofmann 2019). A gridsearch is used to select hyperparameters as in Appendix B.

As the state-action space is target agnostic, the trained policy 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. As such we run APIRL in black-box fashion.

4.2 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%. This was seen to balance runtime and bug finding by reducing wasted requests on true negatives or invalid actions. Experiments are run on Ubuntu Linux, with 16GB RAM and Intel core i7 8700k processor. To mitigate the intrinsic stochasticity of approaches, we repeat each experiment five times.

REST APIs. We evaluate our framework on smaller REST APIs, including: VAmPI, vAPI v1.3, and c{api}tal. 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 with such time limits is in line with other studies (Liu et al. 2022; Kim et al. 2022). Additionally, broader studies in fuzzing, such as by Böhme et al. (2016) have shown that most fuzzers find the majority of coverage early in testing, after which they asymptomatically converge. Due to ethical concerns over data privacy, integrity, and availability, *we do not test on live REST APIs*, but instead deploy them locally. 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* (2023) builds call sequences using feedback from executed requests and an attention based neural network to generate mutational parameters. *ARAT-RL* (2023) is a tabular-Q-learning based REST API tester. *EvoMaster* (2021) is based on evolutionary algorithms and heuristics. *RestTestGen* (2022a) 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. We also provide each tool with the required authentication token.

Evaluation Criteria. We use several different criteria in our evaluation. *Line coverage* (LoC) measures precise coverage performance. Details on how this is collected can be seen in Appendix F. Note that coverage cannot be collected from BitBucket as it is closed-source (Liu et al. 2022), and Spree Commerce does not support coverage collection across its multi-server architecture. We use the *request volume* as recommended by Caturano et al. (2021). It represents how 'intelligent' an approach is, where better scanners achieve similar results in fewer requests. The *number of bugs* found in

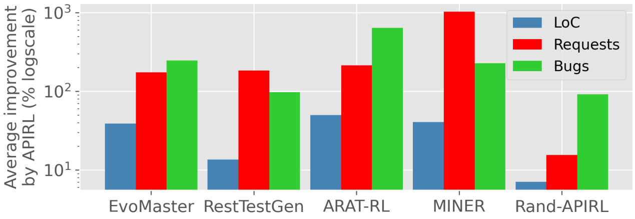

Figure 3: Improvements of APIRL compared to baselines.

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. We define unique bugs as 5XX responses from each operation that originate from different lines of source code. Detecting bugs this way provides a simple and effective oracle within the scope of this work (Arcuri 2021).

The experiment results are shown in Table 2, and the average improvement of APIRL over the respective baselines is shown in Figure 3.

4.3 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 performance distributions differ between APIRL and Rand-APIRL, with 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 all episodes reaching maximum length. MINER consistently achieves low coverage; where a lack of targeted mutations is apparent from the significantly high number of requests in both Table 2 and Figure 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 requires 173.8% more requests than APIRL, resulting in 38.9% less coverage. 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.

4.4 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 3). APIRL finds these bugs with lower or competitive standard deviation. It has a higher standard deviation than its random counterpart only twice. 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, the best case performance of Rand-APIRL 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 117.6% and 99.2% fewer bugs than APIRL respectively. RestTestGen inconsistently finds bugs due to its heuristic matching of parameter names for creating the call graph ODG.

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. APIRL is configured to have a maximum of 3 episodes and 10 requests sent per episode, meaning that the upper limit of requests for the 518 operations is $3 \times 10 \times 518 = 15,540$. 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 to state-of-the-art as Rand-APIRL occasionally finds more bugs. We reason the APIRL test framework gives it an ‘edge’ over the other testing strategies. However, APIRL uses the state-action space effectively, always finding more bugs in fewer test cases. Highlighting the ability of RL to efficiently target mutations, even when testing on unseen REST APIs.

Of the 49 unique bugs 22.4% were unique to APIRL and no other scanner. EvoMaster alone found a unique bug. APIRL misses 7 bugs, 31.8%, as Rand-APIRL, and 68% less than EvoMaster. MINER, and ARAT-RL find 5 of the 7 bugs missed by APIRL, yet this has an increased cost in the number of requests required, and a lower overall number of bugs found. Such bugs can be found by including additional keys-values pairs in requests with default (and intentionally incorrect) values. APIRL can only include parameter keys from the schema so it cannot trigger this functionality.

APIRL can trigger handling errors by inserting `Int` and `String` objects, and removing required parameters. APIRL learns to cast objects to alternative types, causing bugs in 7 endpoints in Spree which are missed by other approaches. Furthermore, APIRL misses no bugs in WordPress, Spree, or vAPI, . In WordPress’s POST `wp/v2/posts/id`[1] operation APIRL triggers a `Type` error by inserting an additional `password` parameter into the body of the request. 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. 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`.

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

[1] https://core.trac.wordpress.org/ticket/61837

Test Service	Operations	Metric	EvoMaster	RestTestGen	ARAT-RL	MINER	Rand-APIRL	APIRL
Capital	16	Unique Bugs	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
		LoC	500.2 (21.7)	592.6 (9.9)	385.8 (74.2)	446 (0)	727.4 (11.3)	**751.3 (11.1)**
		Requests	1670.2 (310.7)	1074.6 (0.5)	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 (1.4)	2 (0)	1.8 (1.3)	**3 (0.7)**
		LoC	245.8 (83.2)	327.6 (42.8)	272.4 (10.4)	277 (0)	327.2 (51.4)	**382.8 (8.7)**
		Requests	4564.6 (1190.6)	3990.6 (874.6)	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)	2.2 (1.5)	6 (0)	7.6 (1.1)	**9.5 (0.6)**
		LoC	362.2 (1.6)	349 (0)	359.2 (17.9)	344 (0)	362.8 (16.5)	**385.7 (6.5)**
		Requests	629.75 (41.3)	1330.6 (68.5)	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)	0.3 (0.6)	1.75 (1.5)	2.25 (1.0)	**10.2 (0.8)**
		Requests	1657.0 (99.0)	2034.2 (1.0)	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)	3.4 (4.3)	6.3 (1.5)	5 (1)	**6.6 (1.1)**
		Requests	1315.8 (438.6)	1194 (0)	**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)	1 (1.0)	0.6 (0.5)	3 (2.8)	**5.2 (3.5)**
		LoC	5658.5 (514.1)	7843.8 (105.8)	5295 (0.0)	5639.8 (331.6)	7995 (159.5)	**8122.2 (117.5)**
		Requests	6327.6 (5138.8)	**2298.4 (35.3)**	5064.8 (889.7)	7373.6 (3380.3)	5685.3 (25.9)	5678.4 (55.4)

Table 2: Average unique bugs, LoC, and number of requests taken on each test service, with standard deviation shown in brackets.

does not exist. An example of the long term strategy of APIRL is shown by manipulating a series of `PATCH` requests in Spree Commerce. By bypassing input sanitisation APIRL causes an error by trying to access the attributes of the user. Interested readers may refer to Appendix D.

4.5 Ablation Study

APIRL displays impressive performance, finding bugs in unseen REST APIs, in a minimal number of requests. However, we wish to determine the extent to which core elements of APIRL contribute to performance. Thus, we conduct an ablation study, presenting the results in Table 3. In particular, we vary rewards based on coverage (APIRL-cov) and response code (APIRL), we ablate transformer embeddings (APIRL-m), and change the RL training algorithm to PPO (APIRL-p), see Appendix B for hyperparameters. Interested readers may see Appendix E for a feature importance study further confirming the utility of transformer embeddings.

APIRL Variant	Successful Requests	Error Requests	Invalid Requests	Coverage	LoC per Request ×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

Table 3: Ablations of both APIRL and APIRL-cov. Lighter colours represent poor performance, while darker colours represent good performance.

Reward Variations. Finding the correct reward function is fundamentally important as it provides the feedback for learning. To prevent agents from finding exploitative strategies in complex reward functions we design simple reward functions (Sutton and Barto 2018). Prior work from Li et al. (2022a) studied how reward functions alter behaviour of RL agents when generating code for compiler testing. Further work from Bates et al. (2023) 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. While both coverage and status code provide feedback for RL agents to test REST APIs, it is unclear how they will affect the learning. As such we develop several reward variants:

- *A coverage based reward (R_{cov}).* Similar to R_{sc}, R_{cov} rewards most for increasing coverage (10), giving a smaller reward for recovering the same code (1), and penalises otherwise (-1). We refer to this agent as APIRL-cov.
- *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.
- *A reward ratio.* We train APIRL-r with status code ratio of: $r = \sum(5XX + 2XX)/\sum(XXX)$. We also train APIRL-cov-r using $r = \sum(LoC_{new})/\sum(LoC)$.
- *An ARAT-RL style reward.* Similar to R_{sc} Kim et al. (2023) design a status code based reward that gives 1 for $4XX$ and $5XX$ responses, penalising -1 for $2XX$. 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 confirm APIRL-cov models *indirectly* finding bugs by increasing coverage. Compared to learnt policies, Rand-APIRL makes 108% more invalid requests (4XX), and has 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 low coverage and successful requests (2XX) in both reward ablations. This is likely due to diminishing returns, *i.e.* the delta of 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, resulting 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 the optimal outcome. Furthermore, coverage-based rewards achieve better coverage compared to request-based. APIRL is the best model in class, with more coverage, error requests, and LoC per request.

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 3, 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 using the structured HTTP responses for feedback in learning. As policies learns 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 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. 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 Related Work

Diverse techniques have been used to test REST APIs. Kim et al. (2023) enhance the OpenAPI specification via NLP techniques. MINER (2023) uses an attention based neural network to generate parameters. RestTestGen (2022a) traverses an ODG to develop call sequences. EvoMaster (2021) 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 (2023) 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 such less granular rewards can harm performance. Additionally, the deep architecture of APIRL learns which parameters to include, how to manipulate values, and how to alter the HTTP method and auth token.

Other RL approaches for fuzzing have used off-the-shelf architectures, to find bugs in software (Böttinger, Godefroid, and Singh 2018; Li et al. 2022b,a) These works use similar rewards based around code coverage, expressing it as a function of *how much* new coverage is achieved. As 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 reward functions.

RL has also been used for automation tasks, often using simple heuristics (Zheng et al. 2021; Li et al. 2022a; Böttinger, Godefroid, and Singh 2018; Li et al. 2022b) or manually defined features (Foley and Maffeis 2022; Lee, Wi, and Son 2022). RL has even been used to test GraphQL APIs for denial-of-service by McFadden et al. (2024). 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.

6 Conclusions and Future Work

Testing REST APIs is key to ensuring the continued functioning of web infrastructure. Thus, we propose a deep RL framework to test for bugs using a combined state representation from manual features and a pre-trained transformer. Our implementation, APIRL, leverages complex and varied responses from REST APIs as feedback for learning. We conduct an extensive ablation study of rewards and design choices showing how they affect behaviour. We show that APIRL consistently achieves higher code coverage and finds more bugs than the SOTA, in a lower request budget. Bugs we found have been reported and are either already fixed, or in the process of being fixed.

In future work, other approaches could add in targeted or guessable key-value pairs using known heuristics or generative methods (Lyu, Xu, and Ji 2023). Such approaches are not present in APIRL, but could be added with enough engineering effort. However, given the performance of APIRL in Section 4, we believe this functionality serves limited purpose. Furthermore, RL approaches could be trained and tailored to specific APIs *e.g.* GraphQL. We believe this to be interesting as we have already shown the potential for generalisation of APIRL in 26 different REST APIs.

Acknowledgements

Research partially funded by EPSRC grant EP/T51780X/1 and the Defence Science and Technology Laboratory (Dstl). Dstl is an executive agency of the UK Ministry of Defence providing world class expertise and delivering cutting-edge science and technology for the benefit of the nation and allies. The research supports the Autonomous Resilient Cyber Defence (ARCD) project within the Dstl Cyber Defence Enhancement programme.

References

Adolphs, L.; and Hofmann, T. 2019. LeDeepChef: Deep Reinforcement Learning Agent for Families of Text-Based Games.

Arcuri, A. 2021. Automated Black- and White-Box Testing of RESTful APIs With EvoMaster. *IEEE Software*.

Atlidakis, V.; Geambasu, R.; Godefroid, P.; Polishchuk, M.; and Ray, B. 2020. Pythia: Grammar-Based Fuzzing of REST APIs with Coverage-guided Feedback and Learning-based Mutations. *arXiv:2005.11498*.

Atlidakis, V.; Godefroid, P.; and Polishchuk, M. 2019. RESTler: Stateful REST API Fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*.

Barabanov, A.; Dergunov, D.; Makrushin, D.; and Teplov, A. 2022. Automatic detection of access control vulnerabilities via API specification processing. *arXiv:2201.10833*.

Bates, E.; Mavroudis, V.; and Hicks, C. 2023. Reward Shaping for Happier Autonomous Cyber Security Agents. In *Proceedings of the 16th ACM Workshop on Artificial Intelligence and Security*, 221–232. Copenhagen Denmark: ACM. ISBN 9798400702600.

Böhme, M.; Pham, V.-T.; and Roychoudhury, A. 2016. Coverage-based Greybox Fuzzing as Markov Chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, 1032–1043. New York, NY, USA: Association for Computing Machinery. ISBN 978-1-4503-4139-4.

Böttinger, K.; Godefroid, P.; and Singh, R. 2018. Deep Reinforcement Fuzzing. In *2018 IEEE Security and Privacy Workshops (SPW)*.

Caturano, F.; Perrone, G.; and Romano, S. P. 2021. Discovering reflected cross-site scripting vulnerabilities using a multiobjective reinforcement learning environment. *Computers & Security*.

Corradini, D.; Zampieri, A.; Pasqua, M.; and Ceccato, M. 2022a. RestTestGen: An Extensible Framework for Automated Black-box Testing of RESTful APIs. In *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*.

Corradini, D.; Zampieri, A.; Pasqua, M.; Viglianisi, E.; Dallago, M.; and Ceccato, M. 2022b. Automated black-box testing of nominal and error scenarios in RESTful APIs. *Software Testing, Verification and Reliability*.

Devlin, J.; Chang, M.-W.; Lee, K.; and Toutanova, K. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding.

Foley, M.; and Maffeis, S. 2022. HAXSS: Hierarchical Reinforcement Learning for XSS Payload Generation. In *IEEE 20th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*.

Gage, P. 1994. A New Algorithm for Data Compression. *C Users Journal*.

Kim, M.; Corradini, D.; Sinha, S.; Orso, A.; Pasqua, M.; Tzoref-Brill, R.; and Ceccato, M. 2023. Enhancing REST API Testing with NLP Techniques. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 1232–1243. Seattle WA USA: ACM. ISBN 9798400702211.

Kim, M.; Sinha, S.; and Orso, A. 2023. Adaptive REST API Testing with Reinforcement Learning. ArXiv:2309.04583 [cs].

Kim, M.; Xin, Q.; Sinha, S.; and Orso, A. 2022. 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.

Lee, S.; Wi, S.; and Son, S. 2022. Link: Black-Box Detection of Cross-Site Scripting Vulnerabilities Using Reinforcement Learning. In *ACM Web Conference*.

Li, X.; Liu, X.; Chen, L.; Prajapati, R.; and Wu, D. 2022a. ALPHAPROG: Reinforcement Generation of Valid Programs for Compiler Fuzzing. In *Proceedings of the AAAI Conference on Artificial Intelligence*.

Li, X.; Liu, X.; Chen, L.; Prajapati, R.; and Wu, D. 2022b. FuzzBoost: Reinforcement Compiler Fuzzing. In *Information and Communications Security: 24th International Conference, ICICS 2022, Canterbury, UK, September 5–8, 2022, Proceedings*.

Liu, Y.; Li, Y.; Deng, G.; Liu, Y.; Wan, R.; Wu, R.; Ji, D.; Xu, S.; and Bao, M. 2022. Morest: Model-based RESTful API Testing with Execution Feedback.

Liu, Y.; Ott, M.; Goyal, N.; Du, J.; Joshi, M.; Chen, D.; Levy, O.; Lewis, M.; Zettlemoyer, L.; and Stoyanov, V. 2019. RoBERTa: A Robustly Optimized BERT Pretraining Approach.

Lyu, C.; Xu, J.; and Ji, S. 2023. MINER: A Hybrid Data-Driven Approach for REST API Fuzzing. In *Proceedings of the 32nd USENIX Security Symposium*.

Martin-Lopez, A.; Segura, S.; Muller, C.; and Ruiz-Cortes, A. 2021. Specification and Automated Analysis of Inter-Parameter Dependencies in Web APIs. *IEEE Transactions on Services Computing*.

McFadden, S.; Maugeri, M.; Hicks, C.; Mavroudis, V.; and Pierazzi, F. 2024. WENDIGO: Deep Reinforcement Learning for Denial-of-Service Query Discovery in GraphQL. In *IEEE Workshop on Deep Learning Security and Privacy (DLSP)*.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Rusu, A. A.; Veness, J.; Bellemare, M. G.; Graves, A.; Riedmiller, M.; Fidjeland, A. K.; Ostrovski, G.; Petersen, S.; Beattie, C.; Sadik, A.; Antonoglou, I.; King, H.; Kumaran, D.; Wierstra, D.; Legg, S.; and Hassabis, D. 2015. Human-level control through deep reinforcement learning. *Nature*.

Parisotto, E.; Song, H. F.; Rae, J. W.; Pascanu, R.; Gulcehre, C.; Jayakumar, S. M.; Jaderberg, M.; Kaufman, R. L.; Clark, A.; Noury, S.; Botvinick, M. M.; Heess, N.; and Hadsell, R. 2020. Stabilizing Transformers for Reinforcement Learning. In *Proceedings of the 37 th International Conference on Machine Learning*.

Schaul, T.; Quan, J.; Antonoglou, I.; and Silver, D. 2016. Prioritized Experience Replay. *arXiv:1511.05952*.

Schulman, J.; Wolski, F.; Dhariwal, P.; Radford, A.; and Klimov, O. 2017. Proximal Policy Optimization Algorithms. *arXiv:1707.06347*.

Sutton, R. S.; and Barto, A. G. 2018. *Reinforcement learning: an introduction*. Adaptive computation and machine learning series. Second edition edition.

Wahaibi, S. A.; Foley, M.; and Maffeis, S. 2023. SQIRL: Grey-Box Detection of SQL Injection Vulnerabilities Using Reinforcement Learning. In *Proceedings of the 32nd USENIX Security Symposium (USENIX Security 23)*, 6097–6114. ISBN 978-1-939133-37-3.

Zheng, Y.; Liu, Y.; Xie, X.; Liu, Y.; Ma, L.; Hao, J.; and Liu, Y. 2021. Automatic Web Testing Using Curiosity-Driven Reinforcement Learning. In *Proceedings of the 43rd International Conference on Software Engineering*, ICSE '21.