Deep Learning-Based Prediction of Test Input Validity for RESTful APIs

A. Giuliano Mirabella
SCORE Lab, I3US Institute
Universidad de Sevilla
Seville, Spain
amirabella@us.es

Alberto Martin-Lopez
SCORE Lab, I3US Institute
Universidad de Sevilla
Seville, Spain
alberto.martin@us.es

Sergio Segura
SCORE Lab, I3US Institute
Universidad de Sevilla
Seville, Spain
sergiosegura@us.es

Luis Valencia-Cabrera
SCORE Lab, I3US Institute
Universidad de Sevilla
Seville, Spain
lvalencia@us.es

Antonio Ruiz-Cortés
SCORE Lab, I3US Institute
Universidad de Sevilla
Seville, Spain
aruiz@us.es

***Abstract*—Automated test case generation for RESTful web APIs is a thriving research topic due to their key role in software integration. Most approaches in this domain follow a black-box approach, where test cases are randomly derived from the API specification. These techniques show promising results, but they neglect constraints among input parameters (so-called *inter-parameter dependencies*), as these cannot be formally described in current API specification languages. As a result, when testing real-world services, most random test cases tend to be invalid since they violate some of the inter-parameter dependencies of the service, making human intervention indispensable. In this paper, we propose a deep learning-based approach for automatically predicting the validity of an API request (i.e., test input) before calling the actual API. The model is trained with the API requests and responses collected during the generation and execution of previous test cases. Preliminary results with five real-world RESTful APIs and 16K automatically generated test cases show that test inputs validity can be predicted with an accuracy ranging from 86% to 100% in APIs like Yelp, GitHub, and YouTube. These are encouraging results that show the potential of artificial intelligence to improve current test case generation techniques.**

***Index Terms*—RESTful web API, web services testing, artificial neural network**

I. Introduction

RESTful web APIs [1] are heavily used nowadays for integrating software systems over the network. One common phenomenon occurring to RESTful APIs is that they contain *inter-parameter dependencies* (or simply *dependencies* for short), i.e., dependency constraints that restrict the way in which two or more input parameters can be combined to form valid calls to the service. For example, in the Google Maps API, when searching for places, if the `location` parameter is set, then the `radius` parameter must be set too, otherwise a 400 status code ("bad request") is returned. Likewise, when querying the GitHub API [2] to retrieve the authenticated user's repositories, the optional parameters `type` and `visibility` must not be used together in the same API request, otherwise an error will be returned. A recent study [3] revealed that these dependencies are extremely common and pervasive—they appear in 4 out of every 5 APIs across all application domains and types of operations. Unfortunately, current API specification languages like the OpenAPI Specification (OAS) [4] provide no support for the formal description of this type of dependencies, despite being a highly demanded feature by practitioners[1]. Instead, users are encouraged to describe dependencies among input parameters informally, using natural language, which leads to ambiguities and makes it hardly possible to interact with services without human intervention[2].

Automated testing of RESTful web APIs is an active research topic [5]–[10]. Most techniques in the domain follow a black-box approach, where the specification of the API under test (e.g., an OAS document) is used to drive the generation of test cases [6]–[8], [10]. Essentially, these approaches exercise the API using (pseudo) random test data. To test a RESTful API thoroughly, it is crucial to generate *valid* test inputs (i.e., successful API calls) that go beyond the input validation code and exercise the actual functionality of the API. Valid test inputs are those satisfying all the input constraints of the API under test, including inter-parameter dependencies.

Problem: Current black-box testing approaches for RESTful web APIs do not support inter-parameter dependencies since, as previously mentioned, these are not formally described in the API specification used as input. As a result, existing approaches simply ignore dependencies and resort to brute force to generate valid test cases, i.e., those satisfying all input constraints. This is hardly feasible for most real-world services, where inter-parameter dependencies are complex and pervasive. For example, the search operation in the YouTube API has 31 input parameters, out of which 25 are involved

[1]https://github.com/OAI/OpenAPI-Specification/issues/256
[2]https://swagger.io/docs/specification/describing-parameters/

in at least one dependency: trying to generate valid test cases randomly is like hitting a wall. This was confirmed in a previous study [11], where it was found that 98 out of every 100 random test cases for the YouTube search operation violated one or more inter-parameter dependencies.

To address this problem, Martin-Lopez et al. [11] devised a constraint-based testing technique, where valid combinations of parameters were automatically generated by analyzing the dependencies of the API expressed in Inter-parameter Dependency Language (IDL), a domain-specific language proposed by the own authors. Although effective, this approach requires specifying the dependencies of the API in IDL, which is a manual and error-prone task. Furthermore, sometimes dependencies are simply not mentioned in the API specification, not even in natural language, and thus they can only be discovered when debugging unexpected API failures.

Approach: In this paper, we propose a deep learning-based approach for automatically inferring whether a RESTful API request is valid or not, i.e., whether the request satisfies all the inter-parameter dependencies. In contrast to the state-of-the-art methods, no input specification is needed. The model is trained with the API requests and their corresponding API responses observed in previous calls to the API. In effect, this allows to rule out potentially invalid test cases—unable to test the actual functionality of the API under test—without calling the actual API. This makes the testing process significantly more efficient and cost-effective, especially in resource-constrained scenarios where API calls may be limited. Preliminary evaluation results show that test inputs validity can be predicted with an accuracy ranging from 86% to 100% in APIs like Yelp, GitHub, and YouTube. These results are promising and show the potential of artificial intelligence techniques in the context of system-level test case generation.

The rest of the paper is organized as follows: Section II introduces the basics on automated testing of RESTful web APIs. Section III presents our approach for predicting the validity of test inputs for RESTful APIs. Section IV explains the evaluation process performed and the results obtained. The possible threats to validity are discussed in Section V. Related literature is discussed in Section VI. Lastly, Section VII mentions future lines of research and concludes the paper.

II. RESTful web APIs

Web APIs allow systems to interact over the network by means of simple HTTP interactions. A web API exposes one or more endpoints through which it is possible to retrieve data (e.g., an HTML page) or to invoke operations (e.g., create a Spotify playlist [12]). Most modern web APIs typically adhere to the REpresentational State Transfer (REST) architectural style [1], being referred to as *RESTful web APIs*. Such APIs are usually decomposed into multiple RESTful web services [13], each one allowing to manage one or more *resources*. A resource can be any piece of data exposed to the Web, such as a YouTube video [14] or a GitHub repository [2]. These resources can be accessed and manipulated via create, read, update, and delete (CRUD) operations, using specific HTTP methods such as `GET` and `POST`.

```
62  paths:
63    /user/repos:
64      get:
65        description: List repositories for the authenticated user.
66        parameters:
67          - name: type
68            in: query
69            type: string
70            enum:
71              - 'all'
72              - 'public'
73              - 'private'
74              - 'owner'
75              - 'member'
76            default: 'all'
77          - name: visibility
78            in: query
79            type: string
80            enum:
81              - 'all'
82              - 'public'
83              - 'private'
84            default: 'all'
85          - name: affiliation
86            in: query
87            type: array
88            items:
89              type: string
90              enum:
91                - 'owner'
92                - 'collaborator'
93                - 'organization_member'
94          - name: sort
95            in: query
96            type: string
97            enum:
98              - 'created'
99              - 'updated'
100             - 'pushed'
101             - 'full_name'
102           default: 'full_name'
103         - name: direction
104           in: query
105           type: string
106           enum:
107             - 'asc'
108             - 'desc'
109       responses:
110         '200':
111           description: OK
112           schema:
113             $ref: '#/definitions/repos'
114         '422':
115           description: Unprocessable entity
116         '403':
117           description: API rate limit exceeded.
```

Fig. 1. Excerpt of the OAS specification of the GitHub API.

RESTful APIs are commonly described with languages such as the OpenAPI Specification (OAS) [4], arguably considered the industry standard. An OAS document describes an API in terms of the allowed inputs and the expected outputs. Figure 1 depicts an excerpt of the OAS specification of the GitHub API. As illustrated, the operation `GET /user/repos` accepts five input parameters (lines 67, 77, 85, 94 and 103), three of which are involved in two inter-parameter dependencies, according to the documentation of the GitHub API [2]: 1) `type` and `visibility` cannot be used together; and 2) `type` and `affiliation` cannot be used together either. These dependencies must be satisfied in order to generate valid API inputs (i.e., HTTP requests). Upon valid API inputs, successful HTTP responses with a 200 status code will be returned (line 113); otherwise, an API error will be obtained, identifiable by a 4XX status code (lines 114 and 116).

Automated test case generation for RESTful APIs is an active research topic [5]–[10]. Most approaches in this domain exploit the OAS specification of the service to generate test cases, typically in the form of one or more HTTP requests. An HTTP request is identified by a method (e.g., `GET`), a path (e.g., `/user/repos`) and a set of parameters (e.g., `type` and `sort`). In some cases, parameter values can also be extracted from the API specification, as in the example shown in Figure 1: all parameters are defined as `enum` strings, i.e., with a finite set of values. Crafting an HTTP request would just be a matter of assigning values to random parameters within their domain, resulting in a test input such as `GET /user/repos?type=all&sort=pushed`. In order to generate a *valid* request, all inter-parameter dependencies of the API operation should be satisfied. Valid requests are essential for thoroughly testing APIs, since they exercise their inner functionality, going beyond their input validation logic. According to the REST best practices [13], valid requests should return successful responses (2XX status codes), while invalid requests should be gracefully handled and return "client error" responses (4XX status codes).

III. Approach

In this paper, we propose an artificial neural network for the automated inference of test input validity in the context of RESTful APIs (or simply APIs henceforth). The goal is to automatically infer whether an API call is valid or not without calling the actual API. This would be extremely helpful for the automated generation of cost-effective test suites, especially in resource-constrained scenarios where stressing the API with thousands or even millions of random API calls is not an option. The approach is divided in three steps, described below.

A. Data collection

The first step consists in the collection of a dataset of API calls properly labeled as valid or invalid. We consider an API call as valid if it returns a successful response (2XX status codes), and invalid if it returns a "client error" response (4XX status codes). As a precondition, the API calls must meet all the individual parameter constraints indicated in the API specification regarding data types, domains, regular expressions, and so on. For example, if a parameter is defined as an integer between 0 and 20, the value 42 would be invalid. Therefore, it can be assumed that all the invalid API calls in the dataset are invalid due to the violation of one or more inter-parameter dependencies, and not due to errors in individual parameters. A dataset of these characteristics (a set of valid and invalid API calls) can be automatically generated using state-of-the-art testing tools like RESTest [11] or RESTler [15], or collected directly from the users activity.

Table I depicts a sample dataset for the operation to browse user's repositories from the GitHub API (described in Figure 1). The dataset, in tabular format, contains five rows, one for each API call. It contains $n + 1$ columns, n being the number of input parameters of the API operation. Each cell $[i, j]$ represents the value of the parameter in column j for the API call in row i. The last cell of each row (column "faulty") states whether the API call is valid or not—the value to be predicted in our work. For example, row 2 in the dataset represents a request to the GitHub API (operation `GET /user/repos`) with the following key-value pairs: <`visibility=all`, `type=private`>. As stated in the *faulty* column, such call is *invalid* because, as explained in the API documentation [2], the parameters `type` and `visibility` cannot be used together. By contrast, row 4 is *valid*, because <`direction=desc`, `type=all`> does not violate any dependency.

TABLE I
Dataset example.

sort	direction	visibility	type	affiliation	faulty
full_name	-	public	public	-	True
-	-	all	private	-	True
-	-	private	-	collaborator,owner	False
-	desc	-	all	-	False

B. Data preprocessing

The raw dataset is not ready to be fed into the network, as it contains many empty values, not supported by standard deep learning frameworks. In order to fill those empty values, different techniques could be used. For instance, empty values of numeric variables could be filled with the mean or the median value of the column, or zero values. Unfortunately, none of these strategies can be applied in this domain because the sole presence of an input parameter in an API call can be crucial for the violation or fulfillment of a dependency. Additionally, artificial neural networks do not support string inputs, only numeric values. To address these problems, we propose enriching and processing the dataset as described below.

1) Data enrichment: we propose extending the original data as follows.

- For each `number` and `string` column, an auxiliary `boolean` column will be created informing whether the value is empty or not. Columns representing free string parameters will be deleted since dependencies typically constrain their presence or absence exclusively [3].
- For each `enum` column, a `string fake_value` will be assigned to empty values (e.g., "None0"), making sure that such values do not exist in the dataset already.

2) Data processing: next, we propose processing the dataset as follows.

- Numerical variables will be *normalized* in order to even the influence of variables of different orders of magnitude. For example, the operation for creating a coupon in the Stripe API includes, among others, the numerical parameters `redeem_by` and `duration_in_months`. While the latter is $\sim 10^2$, the former is typically $\sim 10^9$, which makes its influence to the network output much

stronger. The normalization step guarantees that the values of both parameters are reduced to the same order of magnitude, so that there is equity among inputs.

– `enum` values will be processed using *one-hot encoding* [16], which creates an alphabet of the k possible values found in dataset, and then transforms the `string` parameter into a binary vector of k elements. For example, possible values for the parameter *direction* of GitHub API are "asc", "desc", or `fake_value` (if it is missing). Those three possibilities are encoded as vectors $(1, 0, 0)$, $(0, 1, 0)$, $(0, 0, 1)$ respectively, where each position of the vector corresponds to a specific value and only one element is equal to 1. One-hot encoding technique provides better performance than simple integer encoding when there is no logical order among `enum` values [16].

C. Network design

The discipline of deep learning is a common choice when facing a classification problem. Specifically, the proposed model is the multilayer perceptron [17], [18]: it can easily handle numerical data and learn knowledge representation; moreover, its implementation is relatively straightforward, and it offers many possible configurations and architectures to experiment with.

The system is a classifier of $\mathbb{R}^n$ into 2 classes: valid or faulty; it accepts the parameters of an API call as input, and returns the predicted validity as output. While the number of classes is already known, the number of inputs n is variable, and it depends on the number of API parameters. For this reason, the network automatically adapts to the number of inputs of the corresponding API operation. More specifically, it has the following structure:

– Input layer: an input layer will be automatically built fitting to the dataframe number of columns, so as for the accepted shape to be flexible.
– Inner layers: we have experimented with several depths and different neuronal units per layer.
– Output layer: one neuron, emitting the probability of the instance of belonging to the valid class. The class will be predicted as faulty or valid depending on the value of the output being greater than 0.5 or not.

In order to achieve better results, we experimented with different combinations of values for the key *hyperparameters* driving the learning process. The *optimizer* is the computation method used to update network weights [19]. The *batch size* refers to the number of instances the system is fed with before each weight update happens. Finally, the *learning rate* determines how much the gradient of the error influences the weights update. The best configuration found is *optimizer="Adam"*, *batch size=8*, and *learning rate=0.002*.

Figure 2 depicts a visual representation of the proposed architecture, consisting in: n-units input layer, five inner layers with 32, 16, 8, 4 and 2 units, respectively, and the final one-unit output layer. As usual in these feed-forward networks, all layers are densely connected to the preceding and following ones. Besides, the first, second, and third layers are followed in this case by three dropout layers, with the dropout coefficient equal to 0.3.

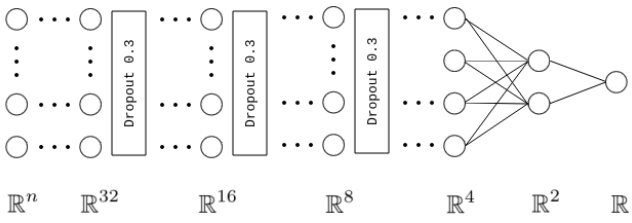

Fig. 2. Network architecture.

D. Training and automated inference

For each target API, the network must be trained with a dataset of previous calls to the API before using it as an effective predictor. Training the network excessively is counterproductive as it may results on the network memorizing instances rather than learning; a phenomena called *overfitting*. To avoid overfitting, we propose the following strategies:

– L1 and L2-regularization factors in each layer [20], which prevents the entropy among weights from increasing excessively. A lower entropy guarantees the uniform distribution of the information, in this case the knowledge, among weights.
– Several Dropout layers along the architecture [21], which randomly silence neurons and prevent the network from activating them, and therefore from improving their weights.
– Early stopping of the training process [20]. Along the training process several metrics are controlled and when they stop improving the training is forced to stop.

Once the system has been properly trained, the validity of new API calls can be predicted by simply applying the network over new instances of the same API (i.e., introducing the instance as an input and applying the forward step only, thus getting the output of the network). In the GitHub example, for instance, when running the network with the API call `GET /user/repos?sort=created`, not included in the original dataset depicted in Table I, the system should ideally predict it as valid, since it satisfies all the input constraints, including the inter-parameter dependencies described in Section II.

IV. Evaluation

For the evaluation of the approach, we implemented a neural network in Python and we assessed its prediction capability on an automatically generated dataset. Next, we describe the research questions, experimental setup and the results obtained.

A. Research questions

We address the following research questions (RQs):

– **RQ1:** *How effective is the approach in predicting the validity of test inputs for RESTful APIs?* This is the main goal of our work. To answer it, we will study the

TABLE II
RESTful services used in the evaluation.

API	Operation	#Parameters	#Dependencies
GitHub	Get user repositories	5	2
LanguageTool	Proofread text	11	4
Stripe-CC	Create coupon	9	3
Stripe-CP	Create product	18	6
Yelp	Search businesses	14	4
YouTube-GCT	Get comment threads	11	5
YouTube-GV	Get videos	12	5
YouTube-S	Search	31	16

prediction accuracy of the approach using a large dataset of calls to industrial APIs.

– **RQ2:** *What is the impact of the size of the dataset in the accuracy of the network?* The accuracy of the approach largely depends on the size of the dataset. To investigate its impact, we will study the accuracy achieved along different sizes of the dataset to provide further insights on the applicability of the approach.

B. Experimental setup

Next, we describe the dataset, the network implementation, and the evaluation procedure.

1) Dataset: we used the tool RESTest for the automated generation of the dataset. RESTest [11] is a black-box test case generation framework for RESTful web APIs. RESTest implements a novel constraint-based testing approach that leverages the manually enriched specification of APIs (using the IDL language to specify inter-parameter dependencies) to automatically generate valid or invalid calls to a given API operation [11]. Specifically, we generated test cases for the eight real-world API operations depicted in Table II. For each API service, the table shows its name, operation under test, number of input parameters and number of inter-parameter dependencies. A detailed description of the dependencies is given in Appendix A. For each of such API operations, we used RESTest to automatically generate 1,000 valid and 1,000 invalid API calls, creating eight datasets with 2,000 API calls each, 16,000 API calls in total. RESTest was configured to satisfy all the individual parameter constraints (e.g., using data dictionaries) ensuring that invalid calls are caused by the violation of one or more dependencies. As a sanity check, we ran all the generated calls into the actual APIs to confirm that they were correctly labeled as valid or invalid. Also, we manually confirmed that each of the dependencies is violated at least once in the dataset.

2) Network implementation: the proposed approach was implemented using common deep learning packages for Python. The preprocessing step was based entirely on `pandas 1.2.0`, which offers general dataframe manipulation tools, and `scikit-learn 0.23.2` which provides *normalization*, and *one-hot encoding* functionalities. The network was built using `tensorflow 2.3.0` and `keras 2.4.3`; among others, most relevant tools were: `Dense` class to build inner layers; `Dropout` class, and `regularizers` method to implement regularization techniques; finally, `Sequential` class was used to build the model.

TABLE III
Prediction accuracy for each service.

API	Accuracy (%)	
	Cross Validation	Test
GitHub	99.8	100
LanguageTool	85.6	86.0
Stripe-CC	97.8	100
Stripe-CP	98.4	98.8
Yelp	94.8	94.1
YouTube-GCT	98.2	100
YouTube-GV	99.4	100
YouTube-S	98.3	99.3
Mean	96.5	97.3

3) Network evaluation: In order to evaluate the performance of each configuration and choose the best option, we used the stratified 5-fold cross validation (CV) technique, which makes possible biases highly unlikely. Stratified K-fold CV is a variant of K-fold CV which guarantees that each fold has a balanced presence of instances of each class (*faulty* and *valid*, in this case), and it is supported by `scikit-learn` package.

The fundamental metric considered to choose the best architecture during the cross validation is the *accuracy*, which measures the fraction of correctly predicted instances. One common practice when facing classification problems is to consider also the *area under the ROC curve* (AUC). AUC measures the homogeneity of the system accuracy among different classes, and it is especially useful when classes are not equally represented in the dataset, and the accuracy score can easily mislead the intuition. In our case, since the dataset is completely balanced, AUC plays a less relevant role as a metric. However, we were controlling its values as well, to guarantee they were always considerably high, thus increasing the confidence that the good results in accuracy were similar in the different classes represented (*valid* and *faulty*). Both metrics were calculated using `keras`.

The database was randomly split into test and train datasets (20% and 80%, respectively). Training data were exploited by CV training in order to choose the best system architecture and tune hyperparameters, while the test data were used to evaluate the final accuracy of the predictor. Such division of the dataset guarantees the measured scores are not only adjusted to the specific CV training data, but also legitimate in predicting completely new instances. Just like in cross validation, the fundamental metric to ponder is the *accuracy*.

C. Experimental results

Table III shows the predictor accuracy for each API operation both in CV training and test data. With 99% confidence CV and test accuracy are $(96.5 \pm 3.97)\%$ and $(97.3 \pm 4.23)\%$, respectively. The results show that the overfitting phenomena has been controlled successfully along training process, as

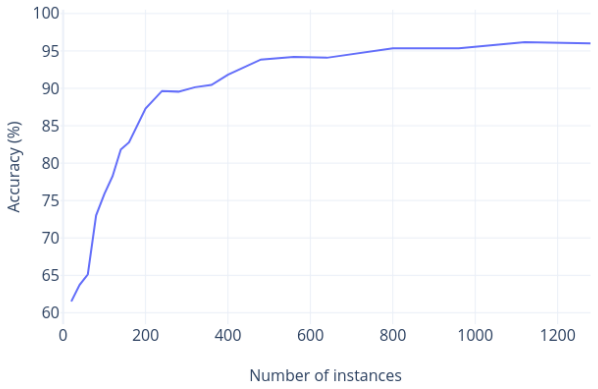

Fig. 3. Accuracy evolution by dataset size.

there is a strong similarity between CV and test scores for each API operation.

It would be sensible to think that a correlation should exist between goodness of results and complexity of the API operation, i.e., validity of services with less number of parameters and dependencies should be easily predictable. This theoretical intuition is confirmed by GitHub or Stripe-CC, for instance, where accuracy is 100%. However, Yelp scores are counter-intuitively lower, which could be due to its dependencies being arithmetic; similarly, LanguageTool validity is by far the hardest to predict despite having only 4 dependencies. By contrast, the search operation validity in YouTube is predicted very accurately (99.3%), despite being its parameters and dependencies the most numerous ones. This may due to the fact that most of its parameters are `enum`, with a fixed set of possible values, and therefore there are less combinations of parameters and values for which to predict input validity. All these facts suggest that the complexity of validity prediction does not lie only on the number of dependencies, but also on their type and the shape of the parameters involved.

In view of these results, RQ1 can be answered as follows:

> *The accuracy of the network ranges from 86% to 100% with a mean accuracy in the eight API operations under study of 97.3%.*

Figure 3 shows the evolution of the mean accuracy across all the API operations under study with respect to the size of the dataset. As illustrated, accuracy increases drastically as the size approaches to 400 instances, properly balanced. Then, the prediction accuracy increases slowly until the 800-1000 instances, after which the growth is asymptotic. As a result, RQ2 can be answered as follows:

> *The approach achieves an accuracy of 90% with balanced datasets of about 400 instances, reaching its top performance with 800 or more instances.*

V. Threats to validity

Next, we discuss the possible internal and external validity threats that may have influenced our work, and how these were mitigated.

A. Internal validity

Threats to the internal validity relate to those factors that might introduce bias and affect the results of our investigation. A possible threat in this regard is the existence of bugs in the implementation of the approach or the tools used. For the generation of the API requests, we relied on RESTest [11], a state-of-the-art testing framework for RESTful APIs. It could happen that, due to bugs on RESTest or errors in the API specifications, some valid requests were actually invalid, or vice versa. To neutralize this threat, we executed all the requests generated against each of the APIs, to make sure that all of them were correctly labeled as valid or invalid.

For the implementation of our approach, we used cross validation to select the best combination of hyperparameters that would yield the best possible results. Regarding the dataset used, it could be argued that it is not diverse enough, especially for the invalid requests: since these were generated with RESTest (which we used as a black box), it could happen that all of them were invalid due to violating the same dependency over and over again. To mitigate this threat, we manually confirmed that every single dependency was violated at least once in the dataset.

B. External validity

External validity concerns the extent to which we can generalize from the results obtained in the experiments. Our approach has been validated on five subject APIs, and therefore they might not completely generalize further. To minimize this threat, we resorted to industrial APIs with millions of users world-wide. Specifically, we selected API operations with different characteristics in terms of numbers of parameters (from 5 to 31), parameter types (e.g., strings, numeric, enums), number of inter-parameter dependencies (from 2 to 16), and type of dependencies. In this regards, it is worth mentioning that the selected API operations include instances of all the eight dependency patterns identified in web APIs [3].

VI. Related work

The automated generation of *valid* test inputs for RESTful APIs is a somehow overlooked topic in the literature. Most testing approaches focus on black-box fuzzing or related techniques [7], [8], [10], [15], where test inputs are derived from the API specification (when available) or randomly generated, with the hope of causing service crashes (i.e., 5XX status codes). Such inputs are unlikely to be valid, especially in the presence of inter-parameter dependencies like the ones discussed in this paper.

Applications of artificial intelligence and deep learning techniques for enhancing RESTful API testing are still in their infancy. Arcuri [5] advocates for a white-box approach where genetic algorithms are used for generating test inputs

that cover more code and find faults in the system under test. Atlidakis et al. [22] proposed a learning-based mutation approach, where two recurrent neural networks are employed for evolving inputs that are likely to find bugs. In both cases, the source code of the system is required, which is seldom available, especially for commercial APIs such as the ones studied in our evaluation (e.g., YouTube and Yelp).

The handling of inter-parameter dependencies is key for ensuring the validity of RESTful API inputs, as shown in a recent study on 40 industrial APIs [3]. Two previous papers have addressed this issue up to a certain degree: Wu et al. [23] proposed a method for inferring these dependencies by leveraging several resources of the API. Oostvogels et al. [24] proposed a DSL for formally specifying these dependencies, but no approach was provided for automatically analyzing them. The most related work is probably that of Martin-Lopez et al. [11], [25], where an approach for specifying and analyzing inter-parameter dependencies was proposed, as well as a testing framework supporting these dependencies. However, the API dependencies must be manually written in IDL language, which is time-consuming and error-prone. The work presented in this paper is a step forward in automating this process, since we provide a way for predicting the validity of API inputs *without* the need of formally specifying the dependencies among their parameters, just by analyzing the inputs and outputs of the API.

VII. Conclusion and Future Work

In this paper, we proposed a deep learning-based approach for predicting the validity of test inputs for RESTful APIs. Starting from a dataset of previous calls labeled as valid or faulty, the proposed network is able to predict the validity of new API calls with an accuracy of $\sim 97\%$. In contrast to existing methods, relying on manual means or brute force, our approach is fully automated, leveraging the power of current deep learning frameworks. This is a small step, but promising, showing the potential of AI to achieve an unprecedented degree of automation in software testing.

Many plans remain for our future work: first and foremost, we plan to perform a more thorough evaluation including different datasets and more APIs. In addition, we are currently exploring the use of different machine learning techniques for the automated inference of inter-parameter dependencies in RESTful APIs.

Verifiability

For the sake of reproducibility, the code and the dataset of our work are publicly accessible in the following anonymous repository:

https://anonymous.4open.science/r/
8954c607-8d6c-4348-a23c-d57c920cdc22/

References

[1] R. T. Fielding, "Architectural Styles and the Design of Network-based Software Architectures," Ph.D. dissertation, 2000.

[2] "GitHub API," accessed January 2020. [Online]. Available: https://developer.github.com/v3/

[3] A. Martin-Lopez, S. Segura, and A. Ruiz-Cortés, "A Catalogue of Inter-Parameter Dependencies in RESTful Web APIs," in *International Conference on Service-Oriented Computing*, 2019, pp. 399–414.

[4] "OpenAPI Specification," accessed April 2020. [Online]. Available: https://www.openapis.org

[5] A. Arcuri, "RESTful API Automated Test Case Generation with EvoMaster," *ACM Transactions on Software Engineering and Methodology*, vol. 28, no. 1, pp. 1–37, 2019.

[6] V. Atlidakis, P. Godefroid, and M. Polishchuk, "Checking Security Properties of Cloud Services REST APIs," in *IEEE International Conference on Software Testing, Validation and Verification*, 2020, pp. 387–397.

[7] H. Ed-douibi, J. L. C. Izquierdo, and J. Cabot, "Automatic Generation of Test Cases for REST APIs: A Specification-Based Approach," in *IEEE International Enterprise Distributed Object Computing Conference*, 2018, pp. 181–190.

[8] S. Karlsson, A. Causevic, and D. Sundmark, "QuickREST: Property-based Test Generation of OpenAPI Described RESTful APIs," in *IEEE International Conference on Software Testing, Validation and Verification*, 2020, pp. 131–141.

[9] S. Segura, J. A. Parejo, J. Troya, and A. Ruiz-Cortés, "Metamorphic Testing of RESTful Web APIs," *IEEE Transactions on Software Engineering*, vol. 44, no. 11, pp. 1083–1099, 2018.

[10] E. Viglianisi, M. Dallago, and M. Ceccato, "RestTestGen: Automated Black-Box Testing of RESTful APIs," in *IEEE International Conference on Software Testing, Validation and Verification*, 2020, pp. 142–152.

[11] A. Martin-Lopez, S. Segura, and A. Ruiz-Cortés, "RESTest: Black-Box Constraint-Based Testing of RESTful Web APIs," in *International Conference on Service-Oriented Computing*, 2020, pp. 459–475.

[12] "Spotify Web API," accessed November 2016. [Online]. Available: https://developer.spotify.com/web-api/

[13] L. Richardson, M. Amundsen, and S. Ruby, *RESTful Web APIs*. O'Reilly Media, Inc., 2013.

[14] "YouTube Data API," accessed April 2020. [Online]. Available: https://developers.google.com/youtube/v3/

[15] V. Atlidakis, P. Godefroid, and M. Polishchuk, "RESTler: Stateful REST API Fuzzing," in *International Conference on Software Engineering*, 2019, pp. 748–758.

[16] F. N. Kerlinger and E. J. Pedhazur, *Multiple regression in behavioral research*. Holt, Rinehart and Winston of Canada Ltd, New York (N.Y.), 1973.

[17] F. Rosenblatt, *Principles of neurodynamics: perceptions and the theory of brain mechanisms*. Washington, DC: Spartan, 1962.

[18] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, *Learning Internal Representations by Error Propagation*. Cambridge, MA: MIT Press, 1986, p. 318–362.

[19] Keras. Optimizers. [Online]. Available: https://keras.io/api/optimizers/

[20] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016.

[21] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *Journal of Machine Learning Research*, vol. 15, no. 56, pp. 1929–1958, 2014. [Online]. Available: http://jmlr.org/papers/v15/srivastava14a.html

[22] 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," Tech. Rep., 2020.

[23] Q. Wu, L. Wu, G. Liang, Q. Wang, T. Xie, and H. Mei, "Inferring Dependency Constraints on Parameters for Web Services," in *World Wide Wed*, 2013, pp. 1421–1432.

[24] Oostvogels, N., De Koster, J., De Meuter, W., "Inter-parameter Constraints in Contemporary Web APIs," in *International Conference on Web Engineering*, 2017, pp. 323–335.

[25] A. Martin-Lopez, S. Segura, C. Müller, and A. Ruiz-Cortés, "Specification and Automated Analysis of Inter-Parameter Dependencies in Web APIs," *IEEE Transactions on Services Computing*, 2020, in press.

Appendix A
IDL dependencies from the APIs under study

Table IV shows the inter-parameter dependencies present in the eight API operations considered in our study, expressed in the IDL language.

TABLE IV
INTER-PARAMETER DEPENDENCIES INCLUDED IN THE API OPERATIONS UNDER STUDY DESCRIBED IN IDL.

API	Inter-parameter dependencies
GitHub	ZeroOrOne(type, visibility); ZeroOrOne(type, affiliation);
LanguageTool	OnlyOne(text, data); IF preferredVariants THEN language==’auto’; IF enabledOnly==true THEN NOT (disabledRules OR disabledCategories); IF enabledOnly==true THEN (enabledRules OR enabledCategories);
Stripe-CC	Or(amount_off, percent_off); IF amount_off THEN currency; AllOrNone(duration==’repeating’, duration_in_months);
Stripe-CP	IF caption THEN type==’good’; IF [deactivate_on[]] THEN type==’good’; IF [package_dimensions[height]] OR [package_dimensions[length]] OR [package_dimensions[weight]] OR [package_dimensions[width]] THEN type==’good’; AllOrNone([package_dimensions[height]], [package_dimensions[length]], [package_dimensions[weight]], [package_dimensions[width]]); IF shippable THEN type==’good’; IF url THEN type==’good’;
Yelp	Or(location, latitude AND longitude); ZeroOrOne(open_now, open_at); IF offset AND limit THEN offset + limit <= 1000; IF offset AND NOT limit THEN offset <= 980;
YouTube-GCT	OnlyOne(allThreadsRelatedToChannelId, channelId, id, videoId); ZeroOrOne(id, moderationStatus); ZeroOrOne(id, order); ZeroOrOne(id, pageToken); ZeroOrOne(id, searchTerms);
YouTube-GV	OnlyOne(chart, id, myRating); ZeroOrOne(maxResults, id); ZeroOrOne(pageToken, id); IF regionCode THEN chart; IF videoCategoryId THEN chart;
YouTube-S	ZeroOrOne(forContentOwner, forDeveloper, forMine, relatedToVideoId); IF forContentOwner==true THEN onBehalfOfContentOwner AND type==’video’ AND NOT (videoDefinition OR videoDimension OR videoDuration OR videoLicense OR videoEmbeddable OR videoSyndicated OR videoType); IF forMine==true THEN type==’video’ AND NOT (videoDefinition OR videoDimension OR videoDuration OR videoLicense OR videoEmbeddable OR videoSyndicated OR videoType); IF relatedToVideoId THEN type==’video’ AND NOT (channelId OR channelType OR eventType OR location OR locationRadius OR onBehalfOfContentOwner OR order OR publishedAfter OR publishedBefore OR q OR topicId OR videoCaption OR videoCategoryId OR videoDefinition OR videoDimension OR videoDuration OR videoEmbeddable OR videoLicense OR videoSyndicated OR videoType); IF eventType THEN type==’video’; AllOrNone(location, locationRadius); publishedAfter >= publishedBefore; IF videoCaption THEN type==’video’; IF videoCategoryId THEN type==’video’; IF videoDefinition THEN type==’video’; IF videoDimension THEN type==’video’; IF videoDuration THEN type==’video’; IF videoEmbeddable THEN type==’video’; IF videoLicense THEN type==’video’; IF videoSyndicated THEN type==’video’; IF videoType THEN type==’video’;