foREST: A Tree-based Approach for Fuzzing RESTful APIs

Jiaxian Lin†, Tianyu Li†, Yang Chen†, Guangsheng Wei‡, Jiadong Lin‡, Sen Zhang‡, Hui Xu†.
†School of Computer Science, Fudan University
China
‡Huawei Cloud BU
China

ABSTRACT

Representational state transfer (REST) is a widely employed architecture by web applications and cloud. Users can invoke such services according to the specification of their application interfaces, namely RESTful APIs. Existing approaches for fuzzing RESTful APIs are generally based on classic API-dependency graphs. However, such dependencies are inefficient for REST services due to the explosion of dependencies among APIs. In this paper, we propose a novel tree-based approach that can better capture the essential dependencies and largely improve the efficiency of RESTful API fuzzing. In particular, the hierarchical information of the endpoints across multiple APIs enables us to construct an API tree, and the relationships of tree nodes can indicate the priority of resource dependencies, e.g., it’s more likely that a node depends on its parent node rather than its offspring or siblings. In the evaluation part, we first confirm that such a tree-based approach is more efficient than traditional graph-based approaches. We then apply our tool to fuzz two real-world RESTful services and compare the performance with two state-of-the-art tools, EvoMaster and RESTler. Our results show that foREST can improve the code coverage in all experiments, ranging from 11.5% to 82.5%. Besides, our tool finds 11 new bugs previously unknown.

ACM Reference Format:
Jiaxian Lin†, Tianyu Li†, Yang Chen†, Guangsheng Wei‡, Jiadong Lin‡, Sen Zhang‡, Hui Xu†. 2022. foREST: A Tree-based Approach for Fuzzing RESTful APIs. In Proceedings of under review (preprint). ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnn.nnmmm

1 INTRODUCTION

Nowadays, REST becomes a de facto architectural standard for web applications and cloud [11]. The architecture views each REST service as a storage system that can process data operation requests from clients and make responses. Similar to the classic CRUD (create, read, update, and delete) operations on storage systems, REST also defines four corresponding methods, GET, POST, PUT, and DELETE. The communication between REST clients and services is based on HTTP requests and responses, which brings the architecture excellent interoperability in nature. Due to such advantages, REST has been widely employed by industries. For example, both Microsoft Azure† and Google Cloud‡ adopt REST, and users can access their provided services via RESTful APIs.

To assure the quality of REST software, testing is an essential approach. While traditional unit test or integration test requires much human effort in writing test cases, fuzzing is a favorable testing approach that can be fully automated. Given the API specification of a target REST service as input, a fuzzing engine can generate test cases automatically and exercise each test case with the service. The main challenge of fuzzing lies in how to solve the dependencies among RESTful APIs in order to generate valid and diversified test cases. Existing work (e.g., RESTler [4] and RestTestGen[25]) on this direction mainly employs a straightforward API dependency graph. By traversing the graph via breadth-first search (BFS) or topological sort, a fuzzing engine should be able to generate valid test cases. However, the efficiency of such approaches is limited due to the explosion of edges on the dependency graph. As a result, there could be a huge number of possible paths for reaching an API. Furthermore, since some dependencies are inaccurate (discussed in Section 3.1.1), it is hard for such a tool to arbitrarily choose only one valid path and abandon the rest.

In this paper, we propose a novel tree-based approach for fuzzing RESTful APIs. Our approach can capture the essential dependencies among APIs via a tree structure and therefore improve the dependency complexity from quadratic (traversing a graph) to linear (traversing a tree) with respect to the number of APIs. Note that each RESTful API is uniquely defined with an endpoint and a method (GET, POST, PUT, or DELETE). Rather than modeling the dependencies of APIs based on their required parameters and response values, we extract API hierarchies according to the endpoint of each API. Our idea is inspired by the fact that the required parameters or resources for visiting a child node on an endpoint tree have a high chance to be fulfilled once its parent node has been visited. We can therefore pre-order traverse the tree to generate valid test cases. Moreover, the relationships among tree nodes can indicate the priority for acquiring the dependent resource. For example, it’s more likely that a node depends on its parent node than its descendant or siblings, although there could be dependencies among all of them according to the traditional dependency graph. In this way, our approach can improve the possibility of finding useful resources for generating valid requests and improve the efficiency of fuzzing.

To elaborate, our approach parses RESTful API specifications in Swagger or OpenAPI standard [13] and constructs endpoint trees...
for these APIs. To this end, it splits each endpoint URL (uniform resource locator) into several tokens by the slash symbol. Each token is a node on an endpoint tree, and we connect these nodes with edges such that an endpoint URL can be reconstructed as a path from the root to a descendent node. Meanwhile, each node may have several attributes indicating the supported methods of the endpoint, and a buffer to stores the resources of the node. When generating test cases for each node, we employ a template-based method that specifies an ad hoc order of HTTP request methods, e.g., GET, then several POST, followed by PUT and DELETE. Since PUT and DELETE may invalidate the created resource, we should execute POST more times than DELETE in order to leave valid resources for the decedent nodes to use. Note that there could be multiple trees for an application. Our approach should also work in this scenario by visiting each tree iteratively. The complexity is still linear to the number of APIs.

We have implemented a prototype for the tree-based approach, namely foREST. Besides the basic tree-based model for guiding request sequence generation, foREST has also realized the associated feature for resource management and retrieval. We have conducted a set of comparison experiments to study the efficiency of our approach. Our experimental result firstly verifies that such a tree-based approach is more efficient than traditional graph-based approaches in practice, including BFS and topological sort. Furthermore, we compare foREST with two state-of-the-art tools for RESTful API fuzzing, i.e., RESTler [4] and EvoMaster [1]. We fuzz WordPress and GitLab with these three tools, and each fuzzing experiment lasts for six hours. Results show that foREST can achieve the best code coverage in all experiments and has exceeded the second best one (EvoMaster) by 11.5% to 82.5% in different experiments. Besides, foREST has found 11 new bugs previously unknown, while RESTler and EvoMaster have found none.

In short, this article contains several major research contributions as follows.

- It proposes a novel tree-based RESTful API fuzzing approach that is more efficient than traditional graph-based approaches. To our best knowledge, it is the first systematic work in this direction. Our approach subtly models the relations of APIs with a tree structure which not only reduces the complexity of API dependencies but also captures the priority of resource dependencies.
- We have implemented a prototype tool, foREST, and released it as open source on GitHub\(^3\). Our fuzzing experimental results with real-world REST services show that foREST has achieved better performance than state-of-the-art tools. We believe our approach and the tool would be useful to advance the development of the community on the problem of RESTful API fuzzing.

The rest of the article is organized as follows. Section 2 firstly introduces the problem of RESTful API fuzzing and discusses the challenges of the problem. Section 3 then presents a motivating example for employing a tree-based approach and demonstrating our methodology. Section 4 evaluates the performance of our approach. Section 5 discusses related work, and finally Section 6 concludes the paper.

\(^3\)The link will be released soon.

---

**Figure 1: Sample APIs of the GitLab project.**

```
GET /projects
POST /projects
DELETE /projects/{id}
GET /projects/{id}
PUT /projects/{id}
```

(a) Five sample APIs.

(b) Definition of the API GET /projects in OpenAPI 3.0.
2.2 Challenges for Fuzzing RESTful APIs

Real-world RESTful services generally contain data dependencies among APIs. For example, Figure 2a is a dependency graph with detailed relationships among four APIs of Figure 1. To generate a valid test case for the API GET /projects/:id, the test case should pass a legal parameter id, which can be obtained from the response message of another API GET /projects or POST /projects. Therefore, the main challenge of RESTful API fuzzing lies in how to model and solve these data dependencies in order to generate valid test cases for each API.

Note that the problem is different from existing fuzz target generation problems for library APIs, such as [6, 14]. A library API generally takes one or several parameters and returns one value. In order to generate a valid API call as the fuzz target, the API parameter types should match the API signature. Most of these types are primitive ones (such as int or char) or abstract data types that can be returned by other APIs of the library. Therefore, there are also data dependencies among the parameters and return values of different APIs. Below, we discuss three typical differences between traditional fuzz target generation problems and RESTful API fuzzing.

M to N: Compared to a library API that returns only one value (i.e., \( f(x_1, \ldots, x_m) \rightarrow y \)), the response message of an RESTful API generally contains multiple values (i.e., \( f(x_1, \ldots, x_m) \rightarrow (y_1, \ldots, y_n) \)). Therefore, the dependencies should be more complicated, i.e., the parameter of an RESTful API depends on one particular field of the response from another API. Note that a library API may also return multiple values as a tuple, but it generally also contains another straightforward constructor for the required type.

Arbitrary matching: the dependencies of RESTful APIs are based on the names of values instead of the function signatures (i.e., \( t_{x_1}, \ldots, t_{x_m} \rightarrow t_y \)) for library APIs. Developers can arbitrarily define their names, and there is no strict rules similar to type checking. As a result, developers may choose different names among different APIs for the same resource, or using the same name for different resources. Such issues are very popular in real-world REST applications, leading to another challenge for fuzzing RESTful APIs.

Resource dependency: RESTful applications are generally stateful, and the API dependencies should also consider the state or resources maintained by the application. Otherwise, the generated data dependencies by simply matching the resource name could be fake. In particular, only POST or PUT creates new resources. The response of GET or DELETE may also contain resources, but these resources are not newly created.
Figure 3: Representing sample GitLab APIs as a tree. Each node may support multiple HTTP methods, and the endpoint URL can be reconstructed as a path from the root to each node.

3 TREE-BASED APPROACH

Since there are resource dependencies among APIs, we adopt a tree-based approach to model and solve such dependencies. This section first presents a motivating example showing the advantage of our approach, followed by our detailed methodology.

3.1 Motivating Example

3.1.1 Limitations of API Dependency Graph. We cannot adopt traditional signature-based API dependency because it is inaccurate for capturing the resource dependencies among RESTful APIs. As a result, fuzzing RESTful APIs via API dependency graph traversal would encounter severe issues.

We demonstrate our point with Figure 2. Figure 2a constructs a sample API dependency graph based on the request and response message format specified in the API definition of the GitLab project. We have manually matched all arbitrary names that denote the same resource. For example, path and id are two identifiers used in the response and request for the same thing, and we can denote the dependency edge as `path/id`. In order to traverse the graph, we should follow an order that can solve such data dependencies, e.g., via BFS or topological sort. Therefore, we should invoke `GET /projects` first because the API has no dependencies. Then we can extract the values of `name` and `path` from its response as the parameter values for composing requests for the rest three APIs.

The order apparently works but suffers several essential problems in practice.

- **Dense graph**: Since an RESTful API response generally contains multiple values, the dependency graph could be highly dense, i.e., many edges among API nodes. For example, all the responses of the four APIs in Figure 2a contain a path field, and three APIs require the value as the parameter value of `id`. Besides, many other GitLab APIs (such as those in Figure 2b) also require path/id. Traversing a dense graph is very challenging due to the path explosion problem and an efficient algorithm may neglect some important routes that should be explored.

- **Fake producer**: We should not treat all API dependencies equally because some dependencies are fake. For example, if no project has been created yet, the response of `GET /projects` could be null. In other words, `GET /projects` is not the real producer of the resources `name` and `path` required by other APIs.

- **Weak dependency**: There is another type of dependencies should be treated differently. For example, `POST /projects` depends on `GET /projects` for the parameters of `name` and `path`. However, such resources are conflicting because these two parameters should be uniquely initiated by users. One cannot create a project with an existing name or path. Imposing such a dependency is mainly useful for reaching the code for duplication check.

3.1.2 Hierarchy of RESTful APIs. We observe that the paths of well-designed RESTful APIs generally form a tree structure (or forest) in nature. Figure 3 contains dozens of GitLab APIs, which is a super set of the APIs in Figure 2. Each node on the graph indicates one component of the endpoint URL separated by `/`. Most nodes have one or several method attributes beneath them, denoting the request methods supported by an endpoint. The URL of each API can be reconstructed as a path from the root node (`projects`) to another node with method attributes. For example, `/projects`, `/projects/:id`, and `/projects/users` are all valid URLs. Based on the content of a node, we further divide them into two types: **token node** which contains a fixed path string, and **parameter node** which has a parameter to be specified.

Such API trees subtly capture the resource dependencies among APIs. In particular, if an API needs several parameters, it is likely that these parameter values can be produced by another endpoint along the same path but has shorter URL. For instance, the API `GET /projects/:id` requires one parameter `id`, which can be produced by `POST /projects`; `DELETE /projects/:id/repositories/branches/:branch` requires two parameters `id` and `branch`, which can be produced by `POST /projects` and `POST /projects/:id/repositories/branches` correspondingly. Such dependencies are based on the production and use (or consumption) of resources, which are essential for RESTful API testing.
3.2 Methodology

Figure 4 demonstrates the overall framework of our approach. There are two stages: a preparation stage and a fuzzing stage. In the preparation stage, we organize all the APIs into trees. Besides, we need to annotate some arbitrary resource names with unique identifiers. In the fuzzing stage, we generate test cases by traversing the API trees, and then execute each of them with the target REST service. Meanwhile, we analyze the response messages and extract useful resources for reuse based on a resource pool. Next, we discuss the detailed design of several key procedures.

3.2.1 API Tree Construction. This step parses the specification document of RESTful APIs and organizes them into trees. As discussed in Section 3.1.2, an API tree can be defined as \( T(V, E) \), where each node \( v_i \) is a component of an URL (without the domain name or IP address) separated by "/", and an edge \( e_{ij} \) that connects \( v_i \) and \( v_j \) exists if and only if \( v_i/v_j \) appears in an URL. Each node \( v_i \) may have one or several method attributes, denoting the API methods supported by the URL from the root node \( v_0 \) to \( v_i \), i.e., the path can be restored from the tree as \(/v_0/.../v_i\).

3.2.2 Resource Annotation and Fuzzy Matching. We design a resource annotation table to deal with those arbitrary names of resources. The index of the table are unified resource identities, and each resource may have several arbitrary names in the specification. Our resource annotation table is in nature an array of name sets, i.e.,

\[
\bigcup_{i=1}^{n} \{R_i | R_i = \{name_1, ..., name_m\}\}
\]

Since filling in the table manually by developers would be labour intensive, we employ another fuzzy matching strategy to automatically recognize some potential name pairs commonly used in RESTful APIs. Our matching rule ignores the capitalization of letters, spaces and underlines between letters, etc. In this way, developers only need to annotate a limited number of names that are very different. Besides, we also employ a scoring mechanism to automatically filter falsely matched pairs, i.e., if the response code based on a fuzzy pair is 4XX, we give the pair a low score and tend to abandon it in the next round.

3.2.3 Resource Pool. We employ a resource pool to buffer and reuse dependent resources. The structure of our resource pool is consistent with the hierarchical relationship of APIs. In other words, we create a sub pool for each token node of the API tree. Since only particular resource combinations could satisfy the business requirement of the service, we do not record each resource separately but document them together as a tuple if they occur in the same successful request or response. Note that it is unnecessary to create a sub pool for the parameter node which may have different values, because such node generally indicates a specific resource that has already been included in its parent token node. For example, the token node /projects contains all project resources, and /projects/id contains only one particular project indicated by the project id.

Besides, the hierarchy of our resource pool can also indicate the priority of dependency. When generating a request for an API, we first search resources from the current node for the required parameters, and then its parent node, followed by other ancestor nodes.

3.2.4 Test Case Generation. The tree structure provides us with an essential reference for generating valid test cases. Our basic idea is that we can traverse each tree via either a depth-first or breadth-first order and generate valid test cases for each node. Since a node may support multiple HTTP methods, we adopt template-based order for exercising the methods of each node, where a template is a heuristic order of HTTP methods, e.g., GET, and then \( k \) times of POST, and then PUT, followed by DELETE, where \( k \) is a small random integer generated during runtime for the fuzzing purpose. The order is based on the fact that only POST can create resources, which is the prerequisite for executing GET, PUT, and DELETE. Also, DELETE and PUT may delete or alternate the resource, so we should create multiple resources via POST to benefit the successor nodes which may require the resource. We force GET to be the first test case because either a node (e.g., root node in Figure 3) requires no parameter for launching a GET request or we could retrieve its required parameters from the parent node. Again, note that the order in our current template is ad hoc for fuzzing purposes, there could be other orders which might be more efficient.

Algorithm 1 demonstrates our basic test case generation algorithm. The outer for loop iterates over the trees of an application.
For each tree, we traverse the tree via a depth-first order and recover the endpoint URLs. For each method of an endpoint, we obtain its required parameters and retrieve values for each parameter from the resource pool. There are two key procedures in this process: `Search()` and `FuzzyMatching()` for searching candidate resources, and `Retrieve()` for retrieving resource values. `Search()` is based on the annotation table. It returns the unique resource identifier if such an entry exists in the annotation table, or returns the original name otherwise. `FuzzyMatching()` searches for candidate resources along the tree path backward with the heuristics discussed in Section 3.2.2 until a candidate is found. Since there could be multiple matched resource identifiers, it selects a random one with matching score above a threshold. In order to further empower the fuzzing ability of our approach, we also enable `FuzzyMatching()` to return none no matter whether a matched resource is found or not. If `FuzzyMatching()` returns none, the `Retrieve()` function will use an arbitrary value generated by a simple fuzz engine, i.e., a random value of string, integer, or UTF-8 depending on the type of the parameter.

3.2.5 Response Analysis and Resource Extraction. We execute each test case immediately after it has been generated and analyze the response message. If the response code is 2XX, which implies a successful request, we can then extract the resources and save them into the corresponding node of the resource pool. Meanwhile, a successful request also implies the fuzzing pairs generated by `FuzzyMatching()` is useful, and we should increase the probability of using these pairs for composing new requests later via `IncreaseMatchingScore()`. If the response code is 4XX, which implies a bad request, we should decrease the probability of using the pairs via `DecreaseMatchingScore()`. Finally, a response code of 5XX implies a bug has been detected.

Figure 5: Demonstrating the test cases generated by our approach considering both required parameters and optional parameters with a toy API tree.
3.3 Overall Fuzzing Strategy

Our basic method in Algorithm 1 only considers required parameters. The number of generated test cases is linear to the number of APIs, i.e., \( \leq (k + 3) \times \text{API}\# \). Therefore, it can finish execution in a short period. However, besides required parameters, an API may also support dozens of optional parameters. Generating test cases with optional parameters supplied should be useful for achieving better code coverage.

Our overall fuzzing strategy is based on the Algorithm 1 but considers both required and optional parameters. The idea is straightforward, i.e., we firstly employ the basic method for each API with no optional parameters, and gradually increase the number of optional parameters. Generating test cases for optional parameters should be useful for achieving better code coverage.

Our overall fuzzing strategy is based on the Algorithm 1 but considers both required and optional parameters. The idea is straightforward, i.e., we firstly employ the basic method for each API with no optional parameters, and gradually increase the number of optional parameters. Generating test cases for optional parameters should be useful for achieving better code coverage.

Figure 5 presents an toy example for better demonstrating our overall fuzzing strategy. Suppose the API tree in Figure 5a contains three nodes, N1 and N3 which are token nodes, and :n2 which is a parameter node. We create two sub resource pools for N1 and N3. Then we generate test cases for N1: :n2, and N3 one by one. For each node, we first generate the test case of a GET request with required parameters only. Then we generate test cases with optional parameters for the API. Next, we generate \( k \) test cases for POST requests with required parameters only, followed by such requests with optional parameters. We continue the test case generation process for PUT and DELETE. After all test cases for the API tree have been executed, we can continue the fuzzing process and restart from N1. Note that our second fuzzing round can generate new test cases different from the first round, e.g., due to the randomness of the FuzzingMatching() function and updated matching scores of dependency pairs by IncreaseMatchingScore() or DecreaseMatchingScore().

4 EVALUATION

This section presents our evaluation experiments based on a prototype implementation of foREST. In particular, we are interested in two questions.

- **RQ1**: Compared to traditional approaches based on API dependency graphs, can our tree-based approach be more efficient in practice?
- **RQ2**: Will foREST perform better than other existing tools, such as RESTler [4] and EvoMaster [1]?

Next, we present our experimental setting and experimental results to answer these two questions.

4.1 Implementation and Experimental Setting

We have implemented a prototype tool for the proposed tree-based fuzzing approach with python 3.8, including both the tree-based structure for API modeling and the corresponding resource management and retrieving strategies. It contains 2K+ lines of python code. The tool supports RESTful API specifications in either Swagger 2.0 or Open API standards. We release our tool as open source, and it is available online.

We conduct our evaluation experiment in a local network environment, i.e., we deploy foREST on a PC and test it against local RESTful services. Our experimental applications include two
Table 1: Detailed experimental data of our comparison experiment with RESTler and EvoMaster in six hours. The LoC increase is calculated as (LoC of foREST - LoC of EvoMaster)/LoC of EvoMaster.

| Project     | API Group | # APIs | RESTler          |       |       |       |       |       |       |       |       |       |
|-------------|-----------|--------|------------------|-------|-------|-------|-------|-------|-------|-------|-------|-------|
|             |           |        | # requests       | LoC   | # bugs| # requests | LoC   | # bugs| # requests | LoC   | # bugs| LoC increase |
| WordPress   | all       | 39     | 6249             | 15111 | 0     | 6972     | 16183 | 0     | 7338     | 18217 | 3     | 12.5%     |
| GitLab      | projects  | 33     | 281526           | 1838  | 2     | 107297   | 7298  | 0     | 90470    | 10151 | 3     | 39.1%     |
|             | groups    | 23     | 377780           | 1407  | 0     | 89212    | 6668  | 2     | 89865    | 7433  | 2     | 11.5%     |
|             | commits   | 18     | 375126           | 1900  | 0     | 107200   | 8406  | 0     | 109533   | 15342 | 3     | 82.5%     |
| Total       | -         | 113    | 1040681          | 20256 | 2     | 310681   | 38533 | 2     | 297206   | 51143 | 11    | 32.7%     |

Figure 7: Experimental result of comparing foREST with RESTler and EvoMaster in fuzzing WordPress.

4.2 Efficiency of Tree-based Approach

In order to study the efficiency of our novel tree-based approach, we also implement another version with traditional graph-based approaches using the same foREST framework. For fair comparison, we implement two representative graph parsing algorithms, BFS and topological sort. The BFS algorithm directly employs the approach proposed in RESTler [4]; for topological sort, we employ a resource pool to buffer the result generated in previous request. Therefore, we have three configurations of fuzzing tools that only differ in the test case generation algorithms.

We fuzz WordPress for six hours with these three configurations and measure the covered lines of code. Our result is presented in Figure 6. Figure 6a demonstrates their coverage growths in the first 30 minutes with more observable differences. Each coverage upsurge on the graph generally implies a new API has been successfully triggered. We can observe that the three approaches perform comparably in the first ten minutes, and they all cover about 14000 lines of code. However, the code coverage of our tree-based approach becomes prominent after that. Figure 6b presents the coverage information of all the six hours. We can observe that our tree-based approach finally has covered most lines of code (18217), which is better than 16998 achieved by topological sort, and 14649 achieved by BFS.

Note that our experiment employs the same request frequencies for the three configurations, and their final request numbers are comparable. Therefore, we may conjecture that our tree-based approach is generally more efficient than traditional graph-based approaches. Meanwhile, it does not undermine the fuzzing effectiveness in code coverage. Our comparison experiments with RESTler in the next section can also coincide with this conjecture.

4.3 Comparison with Existing Tools

Now, we compare the performance of foREST with two state-of-the-art fuzzing tools for RESTful APIs, i.e., RESTler [4], [8] and EvoMaster [1]. RESTler is a graph-based black-box fuzzing tool collaborated by Columbia University and Microsoft. It adopts the same problem setting as we do, which only requires the API specification. EvoMaster is another powerful tool that supports both white-box and black-box mode. For fair comparison, we use its black-box mode (–blackBox true) with a request rate 300 per minute (–ratePerMinute 300) in our experiment.
we apply these three tools to fuzz two open source applications, WordPress and GitLab. Figure 7 shows our comparison result for WordPress in 30 minutes and six hours separately. We can observe that the coverage of foREST grows faster that RESTler and EvoMaster in the begining. Although EvoMaster outperforms foREST for a little while, foREST exceeds EvoMaster after the first five minutes, and the leading superiority becomes more obvious sooner after the
According to the detailed statistics in Table 1, foREST has sent 7298 lines of code with no bugs found. The improvement is about 4, i.e., a sequence of requests in order to be stateful, analyzing API specifications. Each test case generated by foREST is a graph-based approach to model the dependencies among APIs by our approach are more accurate. To study the performance of RESTler [12].

RestTestGen [25] is another tool similar to RESTler. It also models API dependencies as graphs but should be more powerful in the capability of dependence inference. Instead of strict string matching, RestTestGen employs several fuzzy matching strategies, such as case insensitive and stemming. Furthermore, it also introduces abnormal case generation strategies, such as mutation, missing required parameters, or incorrect input type. Corradini et al. [8] have conducted an experiment to compare the performance of RestTestGen with RESTler, as well as two other tools, bBOXRT [18] and RestTest [21], and their result shows that RESTler performs the best in the benchmark and it is the most robust tool.

EvoMaster [1, 2] is a white-box testing tool but also supports black-box mode. Similar to our work, it has no API dependency graph. In order to generate a valid test case for a particular endpoint, it heuristically searches the antecedent requests or preconditions that should be fulfilled along the path of the endpoint URL. EvoMaster is a state-of-the-art tool, and has attracted follow-up work, such as [26] that enhances the verification ability of EvoMaster for particular RESTful services.

Besides inferring the dependencies based on the default API specification. There are also other papers that resorts to extra input, such as manual dependency specification [19, 20], or historical traffic data with machine learning [22]. These approaches are orthogonal to other work.

To summarize, our work tackles the RESTful API fuzzing problem, which has also been studied by RESTler [4], RestTestGen [25], and EvoMaster [1, 2]. The novelty of our paper lies in that it serves as the first attempt to propose a systematic tree-based approach for RESTful API fuzzing. Our approach has been shown more efficient leveraging several subtle design choices, such as tree-based API modeling and resource management, fuzzing matching and dynamic dependency pair evaluation strategies, etc.

5 RELATED WORK
As REST surges into popularity, there are dozens of papers working on RESTful API testing. A large portion of these work investigates on how to define test-oriented specifications, including [7, 10, 15–17, 23, 24]. Similar to traditional software testing, one major problem for RESTful test case specification lies in how to provide an oracle for test case verification, such as via property-based [15, 16, 23] and metamorphic testing [24]. Such problems and investigations are not directly related to our work.

Our work focuses on RESTful API fuzzing, and there are several papers (e.g., [1–4, 8, 9, 25]) work on the same problem. As we have studied in this paper, the critical problem for RESTful API fuzzing lies in how to model the dependency relationships for generating valid requests. Existing representative work that studies the problem and supports stateful fuzzing includes RESTler [4], EvoMaster [1, 2], and RestTestGen[25]. Next, we mainly compare our work with these three tools.

RESTler [4] is a stateful RESTful API fuzzing tool. It employs a graph-based approach to model the dependencies among APIs by analyzing API specifications. Each test case generated by RESTler is a sequence of requests in order to be stateful, i.e., the front requests

6 CONCLUSION
This work has studied the problem of black-box RESTful API fuzzing based on the API specifications in Swagger or OpenAPI formats. The challenge mainly lies in how to model the relationships of different APIs in order to solve resource dependencies and generate valid requests. To tackle this problem, this work proposes a novel tree-based approach to effectively capture the relationships among APIs. On one hand, it can largely simplify the dependency relationships among APIs employed by traditional graph-based approaches. On the other hand, the resource dependencies modeled by our approach are more accurate. To study the performance of our approach, we have implemented a prototype and conducted several groups of comparison experiments with widely-employed real-world REST services. Our experimental results first verified the efficiency of the tree-based approach compared to traditional graph-based approaches, and then showed that foREST can achieve a higher code coverage than state-of-the-art tools. We believe our

https://docs.gitlab.com/ee/api/api_resources.html
proposed approach and prototype would be useful to the community in advancing the development of RESTful API fuzzing.

REFERENCES

[1] Andrea Arcuri. 2019. RESTful API automated test case generation with EvoMaster. ACM Transactions on Software Engineering and Methodology (TOSEM) 28, 1 (2019), 1–37.

[2] Andrea Arcuri. 2020. Automated Black-and-White-Box Testing of RESTful APIs With EvoMasters. IEEE Software 38, 3 (2020), 72–78.

[3] Vaggelis Attilakis, Roxana Geambasu, Patrice Godefroid, Marina Polichshuk, and Baishakhi Ray. 2020. Pythia: grammar-based fuzzing of REST APIs with coverage-guided feedback and learning-based mutations. arXiv preprint arXiv:2005.11498 (2020).

[4] Vaggelis Attilakis, Patrice Godefroid, and Marina Polichshuk. 2019. RESTler: Stateful REST API fuzzing. In 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). IEEE, 748–758.

[5] Vaggelis Attilakis, Patrice Godefroid, and Marina Polichshuk. 2020. Checking security properties of cloud service rest apis. In 2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST). IEEE, 387–397.

[6] Domagoj Babic, Stefan Bucur, Yaohui Chen, Franjo Ivančić, Tim King, Markus Kuuzno, Caroline Lemieux, László Seekeres, and Wei Wang. 2019. Fudge: fuzz driver generation at scale. In Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 975–985.

[7] Sujit Kumar Chakrabarti and Prashant Kumar. 2009. Test-the-test: An approach to testing restful web-services. In 2009 Computation World: Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns. IEEE, 302–308.

[8] Davide Corradini, Amedeo Zampieri, Michele Pasqua, and Mariano Ceccato. 2021. Empirical comparison of black-box test case generation tools for RESTful API. In 2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM). IEEE, 226–236.

[9] Hamza Ed-Douibi, Javier Luis Cánovas Inquierdo, and Jordi Cabot. 2018. Automatic generation of test cases for REST APIs: A specification-based approach. In 2018 IEEE 22nd international enterprise distributed object computing conference (EDOC). IEEE, 181–190.

[10] Tobias Fertig and Peter Braun. 2015. Model-driven testing of restful apis. In Proceedings of the 24th International Conference on World Wide Web. 1497–1502.

[11] Antonio Gamez-Diaz, Pablo Fernandez, and Antonio Ruiz-Cortes. 2017. An analysis of RESTful APIs offerings in the industry. In International Conference on Service-Oriented Computing. Springer, 589–604.

[12] Patrice Godefroid, Daniel Lehmann, and Marina Polichshuk. 2020. Differential regression testing for REST APIs. In Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis. 312–323.

[13] OpenAPI Initiative et al. [n. d.]. OpenAPI specification. https://github.com/OAI/OpenAPI-Specification/blob/main/versions/3.0.0.md.

[14] Jianfeng Jiang, Hui Xu, and Yangfan Zhou. 2021. RULF: Rust library fuzzing via API dependency graph traversal. In Prof. of the 36th IEEE/ACM International Conference on Automated Software Engineering (ASE).

[15] Stefan Karlsson, Adnan Caušević, and Daniel Sundmark. 2020. QuickREST: Property-based test generation of OpenAPI-described RESTful APIs. In 2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST). IEEE, 131–141.

[16] Stefan Karlsson, Adnan Caušević, and Daniel Sundmark. 2021. Automatic property-based testing of graphql apis. In 2021 IEEE/ACM International Conference on Automation of Software Test (AST). IEEE, 1–10.

[17] Pablo Lamela SesIAS, Huiping Li, and Simon Thompson. 2013. Towards property-based testing of RESTful web services. In Proceedings of the twelfth ACM SIGPLAN workshop on Erlang. 77–78.

[18] Nuno Larameejo, João Ageno, and Jorge Bernardino. 2021. A Black Box Tool for Robustness Testing of REST Services. IEEE Access 9 (2021), 24738–24754.

[19] Alberto Martin-Lopez, Sergio Segura, Carlos Muller, and Antonio Ruiz-Cortés. 2021. Specification and automated analysis of inter-parameter dependencies in web APIs. IEEE Transactions on Services Computing (2021).

[20] Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. 2019. A catalogue of inter-parameter dependencies in RESTful web APIs. In International Conference on Service-Oriented Computing. Springer, 399–414.

[21] Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. 2020. RESTTest: Black-box constraint-based testing of RESTful Web APIs. In International Conference on Service-Oriented Computing. Springer, 459–475.

[22] A Giuliano Mirabella, Alberto Martin-Lopez, Sergio Segura, Luis Valencia-Cabrera, and Antonio Ruiz-Cortés. 2021. Deep Learning-Based Prediction of Test Input Validity for RESTful APIs. In 2021 IEEE/ACM Third International Workshop on Deep-Learning for Testing and Testing for Deep Learning (DeepTest). IEEE, 9–16.

[23] Pedro Victor Pontes Pinheiro, Andre Takeshi Endo, and Adenilso Simao. 2013. Model-based testing of RESTful web services using UML protocol state machines. In Brazilian workshop on systematic and automated software testing. Citeseer, 1–10.

[24] Sergio Segura, José A. Parejo, Javier Troya, and Antonio Ruiz-Cortés. 2017. Metamorphic testing of RESTful web APIs. IEEE Transactions on Software Engineering 44, 11 (2017), 1083–1099.

[25] Emanuele Vigliani, Michael Dallago, and Mariano Ceccato. 2020. RestTestGen: automated black-box testing of RESTful APIs. In 2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST). IEEE, 142–152.

[26] Man Zhang, Bogdan Marculescu, and Andrea Arcuri. 2019. Resource-based test case generation for RESTful web services. In Proceedings of the genetic and evolutionary computation conference. 1426–1434.