Grammar-Based Evolutionary Fuzzing for JSON-RPC APIs

Veldkamp, L.S.; Olsthoorn, Mitchell; Panichella, A.

DOI
10.1109/SBFT59156.2023.00026

Publication date
2023

Document Version
Final published version

Published in
The 16th International Workshop on Search-Based and Fuzz Testing

Grammar-Based Evolutionary Fuzzing for JSON-RPC APIs

1st Lisette Veldkamp
Delft University of Technology
Delft, The Netherlands
L.S.Veldkamp@student.tudelft.nl

2nd Mitchell Olsthoorn
Delft University of Technology
Delft, The Netherlands
M.J.G.Olsthoorn@tudelft.nl

3rd Annibale Panichella
Delft University of Technology
Delft, The Netherlands
A.Panichella@tudelft.nl

2023 IEEE/ACM International Workshop on Search-Based and Fuzz Testing (SBFT) | 979-8-3503-0182-3/23/$31.00 ©2023 IEEE | DOI: 10.1109/SBFT59156.2023.00026

***Abstract*—Web Application Programming Interfaces (APIs) allow systems to be addressed programmatically and form the backbone of the internet. RESTful and RPC APIs are among the most common API architectures used. In the last decades, researchers have proposed various techniques for automated testing of RESTful APIs, however, to the best of the authors' knowledge there exists no work on testing JSON-RPC (one of the two data formats supported by RPC) APIs. To address this limitation, we propose a grammar-based evolutionary fuzzing approach for testing JSON-RPC APIs that uses a novel black-box heuristic. Specifically, we use a diversity-based fitness function based on hierarchical clustering to quantify the differences in API method responses. Our hypothesis is that responses that are unlike previously seen ones are an indication that new uncovered code paths are reached. We evaluate our approach on the XRP ledger, a large-scale industrial blockchain system that uses JSON-RPC APIs. Our results show that the proposed approach performs significantly better than the baseline (grammar-based fuzzer) and covers an additional 240 branches.**

***Index Terms*—Search-based software engineering, Fuzzing, Test Case Generation, API testing, Hierarchical Clustering**

I. Introduction

Evolutionary Algorithms (EAs) have been widely applied in literature to automate the process of designing and executing tests. While early research efforts focused on unit-level testing, later studies focused on generating tests at different granularity levels, such as integration, simulation-based, and system-level tests [1]. The latter type of tests allows for reaching high coverage more quickly and prevents false positives due to implicit pre-conditions and constraints [2]. System-level tests can be generated by using various search algorithms, including grammar-based fuzzing [3], and evolutionary approaches [1].

To generate valid test inputs, testing approaches require a grammar or some specifications of the APIs (e.g., OpenAPI specification for RESTful APIs). A grammar allows one to focus the search on the available methods without having to explore every possible string to guess the accepted input types. Such a grammar can be constructed from a provided specification of the API operations.

To the best of our knowledge, all state-of-the-art system-level test generators focus on RESTful APIs (e.g., [1], [3], [4]), which are resource-oriented and are used by the majority of web APIs nowadays. These generators use either white-box or black-box heuristics. The former requires access to the source code and promotes test cases based on their ability to cover (or be close to) uncovered code elements (*e.g.,* branches). Instead, black-box heuristics do not require instrumenting the source code but merely rely on input/output test data. Researchers have proposed both black-box and white-box approaches and tools for RESTful APIs, such as EvoMaster [1] and RESTler [3].

RPC APIs differ from RESTful APIs as they are action-oriented rather than resource-oriented. Despite their common use over the years in enterprise systems, there is no approach or tool that aims to generate test cases for RPC APIs. In this paper, we aim to fill this research gap by proposing a black-box approach for testing JSON-RPC APIs. To generate valid test inputs, our approach uses the OpenRPC specification, which is a standard programming language-agnostic interface description [5]. Our contributions are as follows:

- We present a prototype tool that generates system-level tests for JSON-RPC APIs. The tool implements three different grammar-based black-box strategies, namely (1) random search, (2) mutation-based fuzzing, and (3) evolutionary fuzzing.
- We introduce a novel black-box fitness function based on hierarchical clustering methods applied to responses returned by the APIs under test.
- We describe the results of a preliminary study on Ripple's XRP ledger, a large-scale enterprise blockchain application for global payments.

II. Approach

In this section, we will present our automated black-box fuzzing approach that generates test cases for JSON-RPC APIs based on a grammar. The aim of this research is to evaluate the effectiveness of evolutionary fuzzing for JSON-RPC APIs compared to grammar-based fuzzing.

Our approach consists of multiple different components: (1) Grammar extraction, (2) Search algorithm, and (3) Fitness function, which are explained in the following subsections.

A. Grammar

The OpenRPC specification [5] contains the names of all API operations, as well as the names and schemas of the corresponding parameters. Additionally, it provides information on what the response from the API should look like. The schema of a parameter specifies the type and constraints (*e.g.,* range or

979-8-3503-0182-3/23/$31.00 ©2023 IEEE
DOI 10.1109/SBFT59156.2023.00026

enumeration values). Besides the type of each parameter, the *minimum* and *maximum* value can be specified in the schema for integer types, the *length* or *minlength* and *maxlength* can be defined for array types, and a regular expression *pattern*, as well as a list of predefined *enumeration* values, can be specified for string types. Furthermore, the *required* field allows for the specification of which parameters are required. Finally, the *allOf*, *oneOf*, and *anyOf* terms specify whether multiple parameters or schemas are allowed or even required.

In our approach, we parse the OpenRPC specification and construct a grammar. This grammar is used to generate API operation calls with inputs that match the signature of the target API. Different valid requests per API operation can be generated stochastically based on all the specified properties. Although these inputs can be considered valid from the syntactic point of view, certain inputs might still fail depending on the specific meaning of the parameters or the method call.

B. Evolutionary Grammar-Based Fuzzing

A test case is encoded as a sequence of HTTP requests, each including the HTTP method, the API operation, and the API operation parameters. We apply the (μ, λ)-Evolutionary Algorithm (EA), where μ is the number of parents while λ indicates the number of offspring solutions being generated at each iteration. As with any evolutionary algorithm, (μ, λ)-EA evolves an initial population of individuals (*i.e.,* test cases) based on their fitness function value until a predetermined termination criterion is met. In the following, we detail the search initialization, the variation operator, test execution, and termination. The fitness function used to select which test cases to evolve is discussed in Section II-C.

Search Initialization. The population is initialized by generating a pool of μ valid test cases. The validity is guaranteed by using the grammar described in the previous section and applying a random sequence of derivative rules.

Variation Operator. In every iteration, the search algorithm either generates new valid test cases or mutates previously generated ones. Like the state-of-the-art fuzzing tools for RESTful APIs that support mutation, our approach mutates test cases by adding, removing or mutating parameters (*e.g.,* changing the type or value). Note that there is no crossover in (μ, λ)-EAs. In our context, applying a crossover over solutions with incompatible derivative rules would lead to generating invalid/malformed requests.

Test Execution. Each generated test is executed against the System Under Test (SUT). The responses of the APIs (represented as JSON objects) are collected and later used for fitness function calculation. Notice that at this stage coverage is not collected as our approach is black-box (*i.e.,* we only collect input/output data). At the end of every execution, the state of the API system is reset to ensure that previously executed test cases do not affect the next test executions.

Selection. The new population of μ individuals is selected among the parent and offspring solutions based on their fitness value. This elitist selection is traditional in (μ, λ)-EA as it guarantees that the best solutions survive across the iterations until better solutions/tests are generated.

Termination Criterion. The search algorithm ends when the maximum number of executed test cases is reached.

C. Fitness

The fitness function plays a large role in guiding the evolutionary fuzzer toward optimal solutions. It assigns a numerical value (*i.e.,* the fitness) to an individual, which in our context represents the potential of how close a test case is to covering unexplored code paths or finding faults in a SUT. By evolving test cases that were able to uncover new code paths in the SUT, there is a high probability that new branches (linked to the previously discovered paths) in the code can be reached.

As a black-box approach, we do not require direct access to the code coverage obtained by each test case, allowing the fuzzer to be language-agnostic. Instead, we try to infer whether (additional) coverage was obtained by a test case by looking at the diversity in the responses returned by the API.

Hierarchical Clustering. Clustering is a technique commonly used in data analysis to group a set of similar objects [6]. The purpose of clustering is to minimize the intra-cluster distance and maximize the inter-cluster distance. Most clustering methods belong to either partitioning or hierarchical methods [7]. Partitioning methods require the number of clusters as input. Since we are working in a black-box setting (and we do not know the optimal number of clusters), partitioning methods cannot be applied. Instead, a hierarchical clustering algorithm was chosen as it does not have this limitation.

Hierarchical methods can be either divisive (top-down) or agglomerative (bottom-up). An agglomerative method is less complex, particularly effective, and a popular approach for clustering data [6]. Agglomerative clustering is a bottom-up approach, where each object starts in its own cluster, and pairs of clusters are merged as one moves up the hierarchy. Similar clusters are sequentially combined until only one cluster is obtained. By observing the similarity between clusters at each step, the best number of clusters can be identified.

Feature Vector Representation. Hierarchical clustering uses the distance between feature vectors to determine which clusters should be merged. We represent response parameter values as a feature vector that contains all parameter values (*i.e.,* string, boolean, number, array, and JSON object types). However, only primitive types (string, boolean, and number types) can be stored in the feature vector directly. Feature vectors have to be fixed-length. This is why for arrays, only the first value of the array (which is a string, boolean, or number) is embedded in the feature vector. For JSON objects, all parameter values are extracted from the object and put in the feature vector. To account for the fact that these parameters are part of a JSON object, we also compute a weight vector. The weight of a value is defined as 1 divided by the depth of the parameter. For example, a parameter in a JSON object a has a weight of $\frac{1}{2}$ since this parameter is nested in a JSON

object. If this same JSON object a contains another JSON object b, the parameters of b have a weight of $\frac{1}{3}$. It is assumed that parameters with a higher depth are less important to differentiate between responses.

Computation of Distance. To calculate the distance between two vectors, we first compute the distance for each pair of parameters in two feature vectors separately using a suitable distance metric for that type. For boolean values, the distance is either 0 (values are equal) or 1 (values are not equal). For numbers, the distance is defined as the absolute difference between the numbers. For string values, the Levenshtein distance is often used [8]. This distance metric (also called the edit distance) is the most promising metric to compare strings by various edit operations (*e.g.,* deletion, insertion, and substitution of characters) [9]. It is defined as the minimum cost (always an absolute value) of transforming one string into another. We then multiply the distance of each parameter by the corresponding weight.

Finally, we use the Manhattan distance (as it is the most commonly used distance metric) to compute the total distance between two vectors. As part of future work, we aim to investigate the impact of using different distance metrics (*e.g.,* Euclidean distance, Mahalanobis distance, maximum distance, and the cosine similarity).

Fitness Function. The fitness function is defined as the distance of the individual's response object to previously encountered response objects. We call this fitness function *Diversity-Based Fitness*. Individuals that result in a (relatively) unique response, form a new cluster and are given a high fitness value. Individuals with responses that are very similar to those of other individuals are given a low fitness value. The hypothesis is that responses that are unlike what was seen before, are an indication that new code paths are reached. Evolving such individuals could then potentially lead to discovering other new code paths. The fitness is calculated as:

$$F_{DB} = \frac{1}{1 + maxSimilarity} \tag{1}$$

where $maxSimilarity$ is the similarity of the individual's response object to the closest response object in the clustering instance. $maxSimilarity$ is computed based on the distance to the closest object and the parameter weight.

The hierarchical clusters are recomputed every three iterations of the search algorithm as this will allow us to collect more data for clustering. Additionally, it is also worth noting that the cost of running the hierarchical clustering is negligible compared to the cost of running a system-level test case as resetting the state of the SUT is expensive.

D. Non-Evolutionary Grammar-Based Fuzzing

In addition to the algorithm presented in Section II-B, we implemented two alternative test case generation algorithms: (1) *grammar-based fuzzer* and (2) *grammar-based mutational fuzzer*. The grammar-based fuzzer only generates requests based on an API specification and produces completely new μ individuals in every iteration of the search algorithm. Therefore, no test case selection, fitness function, or mutation is applied/computed to existing tests.

The grammar-based mutational fuzzer either generates new test cases or mutates existing ones. The test cases to mutate are randomly selected from the previous population at random and without applying any fitness function. Grammar-based (mutational) fuzzing may be inefficient at exploring the space of API inputs, which is typically very large for complex systems like blockchain applications.

III. Evaluation

This section details the empirical evaluation of our approach guided by the following research question:

RQ1 *How effective is evolutionary fuzzing with regard to structural coverage in comparison to grammar-based fuzzing for JSON-RPC APIs?*

Benchmark System. We evaluate our approach on *Rippled* (v1.6.0). Rippled is a large industrial peer-to-peer software system that runs the XRP Ledger, a decentralized cryptographic ledger. XRP is one of the most popular cryptocurrencies in the market today. As of January 2023, Ripple's network has over 1.29 million transactions per day. Users interact (*e.g.,* manage accounts and create transactions) with Rippled through a JSON-RPC API. In our experiments, we run Rippled in stand-alone mode, which means it is isolated from the rest of the peer-to-peer network and cannot contact other (blockchain) nodes. This is done so that the experiment does not interfere with the global network and vice versa. As Rippled does not have a OpenRPC specification, we created a basic specification based on the documentation available on Ripple's website [10]. All non-admin API operations and corresponding parameters (and constraints) are included. The specification has 34 unique API operations.

Experimental Protocol. All experiments are conducted on a server with an Intel® Xeon® processor E5-2650 v3 @ 2.30 GHz (20 cores, 40 threads) and Ubuntu 20.04, running 10 simultaneous executions.

In the evaluation, we compare the following black-box configurations: (1) a grammar-based fuzzer (*i.e.,* no mutations, each generation has new individuals) which acts as a baseline, (2) a grammar-based mutational fuzzer (GB-MUT), and (3) a grammar-based evolutionary fuzzer (GB-EVO). We configured the GB-MUT and GB-EVO with a population of 100 individuals and a mutation rate of 80 %. All algorithms were set with a search budget of 10K fitness evaluations.

After running the configurations, we instrument Rippled to compute structural coverage over the generated tests to quantify the performance of the approach. The instrumentation is not used during the fuzzing process. Additionally, we want to ensure that the server state (that may have been impacted by previous test cases) is consistent during all tests. To this end, data files are automatically reset and the server is restarted before each test case.

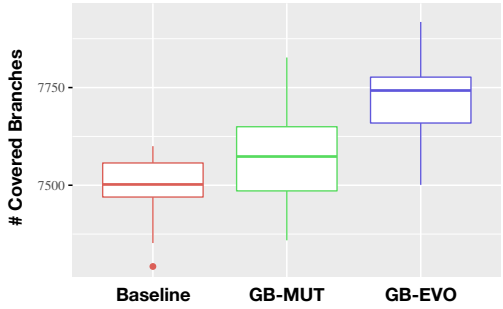

Fig. 1: Branches covered after 10K fitness function evaluations ($\approx$ 2200 min running time).

Configurations	Covered branches		Statistics	
	Median	IQR	p-value	$\hat{A}_{12}$
Baseline	7502	87.25	-	-
GB-MUT	7573.5	164.25	0.557	0.67 (medium)
GB-EVO	7742.5	117.50	0.006	0.91 (large)

TABLE I: Left: Median and Inter-Quartile Range (IQR) of the number of covered branches achieved by the three approaches. Right: p-value and $\hat{A}_{12}$ metrics of GB-MUT and GB-EVO compared to the baseline.

Since all algorithms are randomized, we can expect some variability in our results. To mitigate this, the experiments are repeated 10 times and we reported the median results. We use the Wilcoxon rank-sum test with a threshold of 0.05 to determine the significance of the results. We combine this with the Vargha-Delaney statistic to measure the effect size, which determines the magnitude of the differences.

A. Results

Fig. 1 displays the distribution of the branches covered by the three fuzzing approaches: the baseline, GB-MUT, and GB-EVO. Additionally, the medians, Inter-Quartile Ranges (IQRs), and significance statistics are presented in Table I. The statistical significance of both GB-MUT and GB-EVO are compared to the baseline, with significant cases highlighted in gray color. We observe that the GB-EVO approach achieves a larger number of branches covered compared to the baseline and the GB-MUT approach. The median number of branches covered over the 10 experiment runs is 7742, 240 branches more than the baseline. The GB-EVO fuzzer performs significantly better than the baseline (p-value $= 0.006$), with a large effect size according to the $\hat{A}_{12}$ statistic. The GB-MUT fuzzer, on the other hand, does not achieve a significantly higher coverage than the baseline. In summary, the evolutionary fuzzing approach (GB-EVO) was able to achieve nearly 7750 branches covered in the Rippled blockchain system. This is a significant improvement from traditional grammar-based fuzzing (which covered 7500 branches).

B. Threats to Validity

External Validity: One threat to validity is the generalization of our study. Although only a single benchmark system was used to evaluate the approach, it is a large system part of a distributed blockchain network used in industry. To increase confidence in our results, we plan to evaluate our approach on more benchmark systems in future work.

Conclusion Validity: Both the mutation and evolutionary-based approaches make use of randomness to generate and mutate/evolve the individuals to search the problem space. To minimize the risk that the results were influenced by favourable randomness, we have performed the experiment 10 times with different random seeds. We have followed the best practices for running experiments with randomized algorithms as laid out in well-established guidelines [11].

IV. Conclusion

In this paper, we proposed an approach for black-box fuzzing JSON-RPC APIs using a grammar-based evolutionary algorithm. We implemented the approach in a prototype tool and evaluated it on an industrial distributed blockchain network. In future work, we plan to design a suitable selection procedure for retaining fewer test cases while maximizing the structural coverage. Additionally, several techniques can be applied to optimize the fuzzing process further. A dynamic mutation rate might improve the performance of the evolutionary fuzzer. Furthermore, the approach was evaluated with only one request to the API per individual. Related work shows that increasing this number is an effective way to reach deeper parts of the code. Finally, we plan to evaluate our approach on more benchmark systems to increase confidence in the generalizability of our results.

References

[1] A. Arcuri, "Restful api automated test case generation with evomaster," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 28, no. 1, pp. 1–37, 2019.

[2] A. Zeller, "Search-based testing and system testing: a marriage in heaven," in *2017 IEEE/ACM 10th International Workshop on Search-Based Software Testing (SBST)*. IEEE, 2017, pp. 49–50.

[3] V. Atlidakis, P. Godefroid, and M. Polishchuk, "Restler: Stateful rest api fuzzing," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 748–758.

[4] D. Stallenberg, M. Olsthoorn, and A. Panichella, "Improving test case generation for rest apis through hierarchical clustering," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 117–128.

[5] OpenRPC, "What is openrpc?" accessed: 2022-06-19. [Online]. Available: https://open-rpc.org/

[6] E. K. Tokuda, C. H. Comin, and L. da F. Costa, "Revisiting agglomerative clustering," *Physica A: Statistical Mechanics and its Applications*, vol. 585, p. 126433, 2022.

[7] M. Wallace, G. Akrivas, and G. Stamou, "Automatic thematic categorization of documents using a fuzzy taxonomy and fuzzy hierarchical clustering," in *The 12th IEEE International Conference on Fuzzy Systems, 2003. FUZZ '03.*, vol. 2, 2003, pp. 1446–1451 vol.2.

[8] K. Sasirekha and P. Baby, "Agglomerative hierarchical clustering algorithm-a," *International Journal of Scientific and Research Publications*, vol. 83, no. 3, p. 83, 2013.

[9] L. Yujian and L. Bo, "A normalized levenshtein distance metric," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 29, no. 6, pp. 1091–1095, 2007.

[10] X. Ledger, "Documentation," accessed: 2022-06-19. [Online]. Available: https://xrpl.org/public-api-methods.html

[11] A. Arcuri and L. Briand, "A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering," *Software Testing, Verification and Reliability*, vol. 24, no. 3, pp. 219–250, 2014.