

MINER: A Hybrid Data-Driven Approach for REST API Fuzzing

Chenyang Lyu[†], Jiacheng Xu[†], Shouling Ji[†], Xuhong Zhang[†], Qinying Wang[†], Binbin Zhao^{*}, Gaoning Pan[†], Wei Cao[‡], Peng Chen[‡], and Raheem Beyah^{*}

[†]Zhejiang University, ^{*}Georgia Institute of Technology, [‡]Ant Group

Abstract

In recent years, REST API fuzzing has emerged to explore errors on a cloud service. Its performance highly depends on the sequence construction and request generation. However, existing REST API fuzzers have trouble generating long sequences with well-constructed requests to trigger hard-to-reach states in a cloud service, which limits their performance of finding deep errors and security bugs. Further, they cannot find the specific errors caused by using undefined parameters during request generation. Therefore, in this paper, we propose a novel hybrid data-driven solution, named MINER, with three new designs working together to address the above limitations. First, MINER collects the valid sequences whose requests pass the cloud service’s checking as the templates, and assigns more executions to long sequence templates. Second, to improve the generation quality of requests in a sequence template, MINER creatively leverages the state-of-the-art neural network model to predict key request parameters and provide them with appropriate parameter values. Third, MINER implements a new data-driven security rule checker to capture the new kind of errors caused by undefined parameters. We evaluate MINER against the state-of-the-art fuzzer RESTler on GitLab, Bugzilla, and WordPress via 11 REST APIs. The results demonstrate that the average pass rate of MINER is 23.42% higher than RESTler. MINER finds 97.54% more unique errors than RESTler on average and 142.86% more reproducible errors after manual analysis. We have reported all the newly found errors, and 7 of them have been confirmed as logic bugs by the corresponding vendors.

1 Introduction

Cloud services have developed rapidly over the last decade. These cloud services are programmatically accessed through *REST APIs* [23], and provide software-as-a-service applications without installing software locally. Based on the architecture of cloud services, multiple users can access a shared application through different processes. In this manner, incorrect REST API access may result in a crash of a process,

which can finally lead to cloud service collapse, broken access control, and private data leakage.

Therefore, to explore potential errors of cloud services via REST APIs, early works [1, 6, 7] leverage previously-captured API traffic and manually-defined rules to generate testing requests. Recently, to better automatically infer the dependencies among request types and construct request sequences, REST API fuzzing is proposed to test cloud services [12–14, 28]. Typically, 1) the generation strategies of a REST API fuzzer handle two main problems, i.e., how to construct a sequence template and how to generate each request in a template. After generation, the fuzzer sends the ready-to-use request sequences to test a target cloud service; 2) For each request in a sequence, the cloud service first checks whether the request conforms to the syntax and semantics. If not, it returns a response in $40 \times \text{Range}$. On the contrary, the cloud service performs a behavior according to the request, e.g., the cloud service deletes a resource when receiving a DELETE request. If the cloud service behaves normally, e.g., it successfully deletes the specified resource according to the request, it returns a response in $20 \times \text{Range}$. Otherwise, the cloud service behaves abnormally, e.g., it tries to delete a non-existent resource and results in an error state. Then, it returns a response in $50 \times \text{Range}$; 3) The fuzzer analyzes the response to each request and infers the triggered states of a cloud service; 4) The fuzzer collects the unique request sequences triggering error states, e.g., the sequences having responses in $50 \times \text{Range}$ or violating security rules. Following the above steps, a REST API fuzzer explores reachable cloud service states with generated request sequences.

However, we find that existing REST API fuzzers cannot efficiently generate long request sequences to test cloud services. As a result, existing fuzzers have trouble finding deep errors hidden in hard-to-reach states of cloud services. In particular, it is hard for existing fuzzers to find security bugs like use-after-free bugs and resource-hierarchy bugs as introduced in [14], which need to be triggered with a request sequence of no less than 3 in length. The underlying reason is that security bugs on cloud services often require complex execution

logic to trigger. We summarize two main reasons, corresponding to the sequence construction and request generation, that the state-of-the-art REST API fuzzers fail to generate long request sequences. 1) Existing fuzzers often fail to extend request sequence templates. Their sequence extension process frequently abandons the constructed sequence templates and starts with an empty template, making it challenging to construct sequences of larger lengths; 2) When generating a request in a sequence template, existing fuzzers randomly select values for the request’s parameters. They cannot figure out which parameter should be mutated or what parameter value should be assigned, i.e., the *key mutation* on a request. As a result, they have trouble generating valid requests that can pass a cloud service’s checking.

To overcome the above two limitations and improve error discovery, in this paper, we present a hybrid data-driven fuzzing solution named MINER with the following three important designs. 1) *Length-orientated sequence construction*: MINER leverages the historical data to guide the sequence generation. Specifically, MINER collects the sequence templates, whose requests have been successfully generated and have passed the cloud service’s checking of in the past fuzzing process, as the initial templates of the sequence extension process. Then, MINER leverages a probability function to assign more selection times to the sequence templates with large lengths in the extension process; 2) *Attention model-based request generation*: To improve request generation quality, for each valid request that passes the checking, MINER first collects its parameters and the used values as the key mutations. Second, MINER constructs an attention model, which is one of the state-of-the-art neural network models, to learn the implicit relationship among the key mutations. Then, MINER uses the model to generate more diverse combinations of desirable key mutations, which can construct high-quality requests to pass the checking and explore different states of a cloud service; 3) *Request parameter violation checking*: MINER implements an extra security rule checker to capture incorrect parameter usage errors. By using this checker, MINER generates a new request with an undefined parameter to test a cloud service. For instance, MINER can construct a PUT request containing the undefined parameter, which comes from a POST request. If the request gets a response in $50 \times \text{Range}$, MINER infers that an incorrect parameter usage error is triggered, and records the used request locally for further analysis. The above three core designs are compatible with each other, and work together in the fuzzing process to improve the fuzzing performance.

Our approach is a generic data-driven framework that does not require initial training data before fuzzing. Instead, it automatically collects training data during the fuzzing process. It can be applied to most REST API fuzzers, and improves their performance under the same experimental condition. In this paper, we implement a prototype of MINER based on the state-of-the-art fuzzer *RESTler* [12]. Then, we evaluate the performance of MINER and RESTler on 3 open-sourced

cloud services GitLab [5], Bugzilla [2] and WordPress [8] via 11 representative REST APIs. The average pass rate of MINER is 23.42% higher than RESTler, which indicates that MINER significantly improves the request generation to pass the cloud service’s checking. The evaluation also shows that our designs significantly improve the sequence construction of MINER, which assigns most executions to the sequences of length greater than 4. Furthermore, MINER discovers 142.86% more reproducible errors than RESTler, including 5 security bugs that access deleted resources.

In summary, we make the following main contributions.

- We find that REST API fuzzers have trouble generating long sequences containing high-quality requests and cannot find incorrect parameter usage errors. Motivated by our findings, we propose a new solution named MINER with three novel data-driven designs, i.e., 1) leveraging the history data to guide the sequence construction, 2) improving the request generation with desirable parameter values by constructing an attention model for prediction, and 3) exploring a new kind of errors by generating requests with undefined parameters.
- We evaluate MINER and RESTler on 3 open-sourced cloud services, and show the significant performance of MINER on sequence construction, request generation and error discovery. Furthermore, we utilize the published bugs of GitLab as the ground truth to demonstrate that MINER performs better than RESTler on serious bug discovery. The stepwise analysis shows the contribution of our designs to pass rate and error discovery. We conduct extensive analysis to show high code line coverage, low time overhead, and diverse execution distribution of requests achieved by MINER¹.

2 Background

2.1 REST API for Cloud Services

Most cloud services can be accessed via REST APIs to provide various functionalities. In implementation, different types of requests can be sent to trigger different behaviors of a cloud service. For instance, a client 1) can send a GET request to GitLab, in order to get a list of visible projects with the selected parameters after authentication; 2) can send a POST request to create a new project with the specified parameters; 3) is able to update the parameters of a specified project by sending a PUT request; and 4) can delete a project with a DELETE request. These behaviors can explore different states of a cloud service. Similarly, by automatically generating and sending request sequences via a cloud service’s REST API, a fuzzer can explore errors hidden in different states.

2.2 REST API Fuzzing

Two of the main challenges to fuzz cloud services are as follows: How to construct sequences composed of different

¹We will open source MINER at <https://github.com/puppet-meteor/MINER> to facilitate future research on REST API fuzzing.

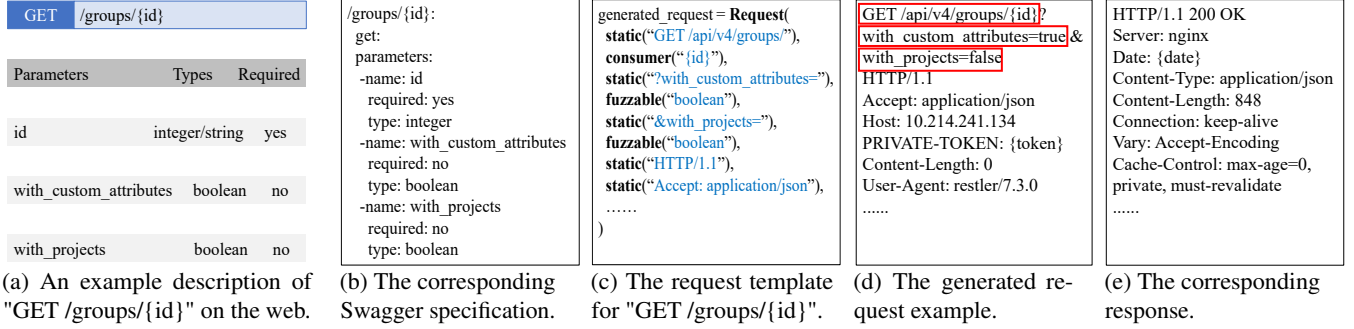


Figure 1: An example to convert the description of a request on the web into a request template, and what a generated request and its corresponding response look like.

requests to trigger a series of behaviors and explore deep states of a cloud service, and how to construct high-quality requests to pass the cloud service’s checking. Towards this, REST API fuzzing is proposed to explore errors hidden in the reachable execution states of a cloud service [12–14, 28]. A REST API fuzzer automatically generates request sequences to test a target cloud service, and is guided by the request responses. If a request in the generated sequence triggers a response in $50 \times \text{Range}$, the fuzzer considers that an error is triggered and stores the sequence for future analysis. The main modules of a REST API fuzzer are as follows.

Compiler Module: A REST API fuzzer implements the Compiler Module to generate a fuzzing grammar and construct request templates, whose procedures are as follows. First, the fuzzer requires the Swagger specification for each request type. As shown in Fig. 1, to obtain the Swagger specification, a user 1) can read the description for each request type published on the web page by vendors, and 2) manually transforms the description into the Swagger specification as shown in Fig. 1a and Fig. 1b. Then, based on the Swagger specification, the fuzzer performs a lightweight static analysis to construct *request templates* as shown in Fig. 1c. To be specific, it 1) infers the dependencies among different request types; 2) constructs a template for each request type; and 3) manually constructs a dictionary of alternative values for each parameter in a request template. For instance, as shown in Fig. 1b, a user can provide {"3", "true", "false"} as the dictionary for the parameter "with_projects", where "3" is the default value that does not match the parameter type.

In particular, some request templates require target object ids of target resources, e.g., the "{id}" as shown in Fig. 1d. If we provide static alternative values for these object ids, the fuzzer may suffer from the following two situations. 1) The fuzzer can only test the same resources on a cloud service, which limits the exploration space; 2) The resources can be deleted by DELETE requests, which hinders other requests for access to these resources. Therefore, the REST API fuzzer solves the problem by reading the target object id from the response of the previous request in a sequence. Specifically, it constructs special request templates to produce and consume target object ids. To produce a target object id, the fuzzer ana-

lyzes the response of a request like GET and POST, and stores the target id in memory, which will be freed when the test of the current request sequence ends. Then, if a latter request template needs a target object id, the fuzzer generates it by reading the stored id from memory. In general, a request that provides a target object id is named *producer*, and a request that requires a target object id is named *consumer*.

Sequence Module: A fuzzer combine different requests to construct different test sequences, e.g., it can send a sequence containing POST and PUT requests to create a resource and then modify this resource. To achieve this, it implements the Sequence Module to dynamically extend request sequence templates started from an empty one, whose details are as follows. 1) When entering the Sequence Module, a fuzzer starts with an empty sequence template; 2) It maintains a set of successfully extended sequence templates, and performs the extension process to construct new sequence templates based on these successfully extended templates. Specifically, it adds each candidate request at the end of the current sequence template and constructs new candidate sequence templates. Then, the fuzzer empties the set and waits for the new successfully extended sequence templates; 3) For each candidate sequence template, the fuzzer utilizes the Generation Module, which will be introduced later, to generate each request in the sequence template and then tests a target cloud service; 4) It analyzes the response of each request to infer the triggered behavior. For instance, a response of 400 Bad Request, 200 OK or 500 Internal Error represents that the cloud service performs the following behavior, respectively: The cloud service refuses to execute the request which does not pass the checking, the cloud service executes the request normally, or it executes the request abnormally which results in an unexpected error; 5) Generally, the fuzzer checks the response of the last request in each candidate sequence to infer whether the sequence is valid. If the last request gets a response in $20 \times \text{Range}$ from the cloud service, the fuzzer considers the candidate sequence successfully extended and adds it to the set of successfully extended sequence templates; 6) If the set is not empty, the fuzzer goes to step 2) to perform the next extension process based on the newly collected sequence templates. Otherwise, it goes to step 1) to restart

the extension process with an empty sequence template. Note that in the `Sequence Module`, the constructed sequence templates always satisfy the dependencies between producers and consumers.

Generation Module: A REST API fuzzer implements this module to generate each request in a sequence template. Specifically, the fuzzer assigns a value for each parameter in the request, and constructs a complete request that is ready to be sent to a cloud service. There are two methods to obtain a parameter value, which are 1) selecting an alternative value from the dictionary or 2) reading a target object id from a response of a previous request. The following instance shows how the fuzzer obtains parameter values using the above two methods. As shown in Fig. 1, the request template `GET /groups/{id}` contains 3 parameters needed to be set. To generate a ready-to-use request, the fuzzer selects "true" and "false" as the values of "*with_custom_attributes*" and "*with_projects*", respectively. As for the parameter "*id*", the fuzzer will read a target object id from the response of a producer like `POST`, which provides an object id after creation.

Checker Module: To improve error discovery of REST API fuzzing, Atlidakis et al. introduced several rule checkers to capture specific security rule violations [14], which are implemented as the `Checker Module` in the REST API fuzzer. For instance, a fuzzer implements a use-after-free rule checker to find the request sequences triggering use-after-free bugs, which access deleted resources in the cloud service scenario and can lead to private data leakage. An example process is as follows. The rule checker first sends a `POST` request to create a resource on a cloud service. Then, it sends a `DELETE` request to delete this resource. Finally, the fuzzer sends an access request, like `GET`, to access this deleted resource. If the cloud service does not return a response in $40 \times \text{Range}$, the fuzzer considers that the use-after-free violation is triggered and saves the used request sequence locally for future analysis. Therefore, a REST API fuzzer tries to trigger specific rule violations in the `Checker Module`.

By utilizing the above main modules, existing REST API fuzzers automatically generate request sequences to test a cloud service via its REST APIs, and record the request sequences locally that trigger unique errors hidden in different states. However, they still have limitations in sequence extension and request generation, resulting in slow state exploration progress. On the contrary, we observe that multiple kinds of historical data collected during the fuzzing process are valuable to guide REST API fuzzing and solve the above limitations, which is the main focus of this paper.

2.3 Neural Network Model

As marked by the red boxes in Fig. 1d, a request consists of an endpoint/name, several parameters and their values. Intuitively, we can regard these elements in the red boxes as *words* and transform a request into a word sequence. Then, a

request generation can be transformed into a sequence generation, which is one of the common applications for neural network models [21, 32]. Specifically, a neural network model leverages neurons to construct complex computational networks [18, 20]. During training, the model adjusts the weights between neurons to fit the implied relationship between words in each sequence. Given an input word, the model improves the prediction probability of the ideal output word. After training, the model predicts words one by one, which follows the implied relationship learned from the training set.

In our application, we collect the parameters and their values used in the valid requests, which pass the checking of a cloud service, as the key mutations. Then, we leverage a neural network model to learn the implied relationship between these key mutations, i.e., which key mutations should be used together on a request. After training, the model predicts the key parameters to be mutated and their appropriate values for request generation. More details are shown in Section 3.3.

3 Design of MINER

3.1 Why Valid Long Sequence Matters

As previously mentioned, different requests can be sent in a sequence to a cloud service to trigger a series of behaviors and further explore unique errors hidden in deep states. For instance, in a sequence, `POST`, `PUT`, and `DELETE` requests can be sent orderly via GitLab Projects API. Then, GitLab will create a new project, modify this project's parameters, and delete this project, respectively. We can conclude from the above instance that a long sequence covers the possible request combinations used in a short sequence, as does the range of states that can be reached. Therefore, to explore the errors hidden in hard-to-reach states, an ideal REST API fuzzer is expected to send long sequences more times, and should not always test a cloud service using short sequences.

To further motivate our research, we analyze the REST API-related issues on GitLab. To be specific, we collect the error issues related to the available resources for GitLab REST APIs, and count the number of issues triggered by a request sequence or a singular request, respectively. Then, we realize that the issues caused by a request sequence account for the majority. In all 56 issues, the proportion of the issues caused by request sequences is 76.79%. For instance, a published error is triggered by the following steps. 1) Create a GitLab project; 2) Delete this project; 3) Access this deleted project with a `GET` request. Since the last request achieves to access a deleted resource, this error is considered a use-after-free bug in the cloud service scenario and is confirmed by the vendors of GitLab. It takes at least 3 requests in a sequence to reproduce this published error. The case study demonstrates that a REST API fuzzer is expected to test a target cloud service with long request sequences.

However, we find that existing REST API fuzzers cannot

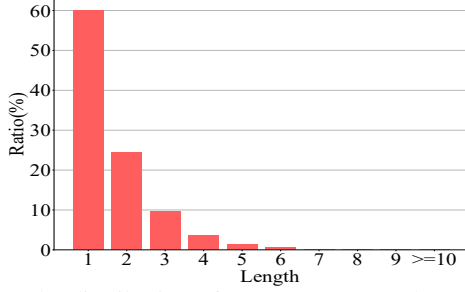


Figure 2: The distribution of request sequence length sent by RESTler when fuzzing GitLab Projects API.

effectively generate long sequences to test a cloud service. To illustrate this, we conduct the following case study to show the proportion of request sequence length executed in the fuzzing process. We utilize *RESTler* [12], which is one of the most famous REST API fuzzers and implements the main modules as shown in Section 2.2, to fuzz GitLab Projects API. Then, we construct the length distribution of the generated request sequences. The evaluation lasts for 12 hours, and the proportion of each sequence length is shown in Fig. 2, from which we can learn that the length of most request sequences generated by RESTler is less than 3. The proportion of the sequences with lengths of 1 and 2 is around 85%.

These results demonstrate that existing REST API fuzzers hardly generate long sequences to trigger series of behaviors on cloud services. Especially, it is hard for them to trigger security bugs explored by long request sequences as shown in [14]. We infer the reasons as follows. First, existing fuzzers cannot effectively generate valid requests that can pass a cloud service’s syntax and semantic checking. For each parameter in a request template, they cannot figure out the appropriate parameter value, but randomly select an alternative value from a preset dictionary, e.g., {"3", "true", "false"} for "*with_projects*" as exemplified in Section 2.2. Thus, their generated requests are more likely to receive responses in $40\times$ Range from the cloud service. This fact is also confirmed by the results in Section 4.2, which demonstrate the low pass rate of the requests generated by existing REST API fuzzers. Then, due to the lack of a proper mechanism to generate valid requests, existing fuzzers are difficult to extend a sequence template to form a long one. As shown in Section 2.2, the fuzzers frequently restart the sequence extension process with an empty sequence template, if the last request in a sequence fails to pass the checking. Moreover, the kinds of request combinations in a short sequence template are limited, which can be fully explored by existing fuzzers in the fuzzing process. Therefore, most of the time, existing REST API fuzzers tend to test a cloud service with short and already-used request sequence templates.

Motivation. Based on the above analysis, we discover that most published errors are triggered by request sequences. While existing REST API fuzzers are hard to generate long sequences with acceptable request generation quality. Most of the time, existing fuzzers test target cloud services with repeti-

tive short request sequences, limiting their performance of discovering unique errors. In particular, it is hard for them to explore security bugs triggered by long request sequences [14]. Therefore, to improve security bug discovery on cloud services, a new fuzzing solution is demanded to improve the sequence construction and request generation.

3.2 Framework of MINER

Motivated by the above findings, we propose a hybrid data-driven solution named MINER as follows. 1) MINER implements the *length-orientated sequence construction*, and leverages the used request sequences as seed sequence templates to guide sequence construction; 2) MINER implements the *attention model-based request generation*, and utilizes an attention model to provide proper values for the key parameters, which increases the pass rate to the cloud service’s checking; 3) To find errors caused by incorrect parameter usage, MINER implements the *request parameter violation checking*, and constructs a new security rule checker named *DataDriven Checker* to generate requests with undefined parameters. The above three designs work together to improve the fuzzer’s performance on cloud services.

As shown in Fig. 3, the framework of MINER contains 5 main components: Sequence Template Selection, Generation Module, Fuzzing Module, Collection Module, and Training Module. The basic workflow is that MINER continuously generates request sequences to test a cloud service via its REST API. Before fuzzing, like existing fuzzers, MINER constructs request templates for a target REST API with the Compiler Module as described in Section 2.2. For each test, MINER 1) utilizes the Sequence Template Selection to probabilistically select request sequence templates according to their lengths and then extend sequences; 2) uses the Generation Module to generate each request in sequence templates and constructs ready-to-use request sequences; and 3) utilizes the Fuzzing Module to test a cloud service with generated request sequences. During the execution of the above modules, the Collection Module collects the following historical data: First, MINER collects the valid request sequences, all of whose requests pass the cloud service’s checking, to enrich the sequence template set. Second, if a request passes the checking, MINER collects the mutated parameter values that are not the default ones. For clarity, we use "*param-value pair*" to represent the mutated parameter and its used value in request generation, e.g., <"*with_projects*", "false"> as shown in Fig. 1. Then, MINER periodically invokes the Training Module to train an attention model with the collected *param-value pairs*. After training, MINER uses the model to generate lists of desirable *param-value pairs* for each request template to enhance the request generation. In the following, we use "*param-value list*" to represent a list of generated *param-value pairs*. The details of each module are as follows.

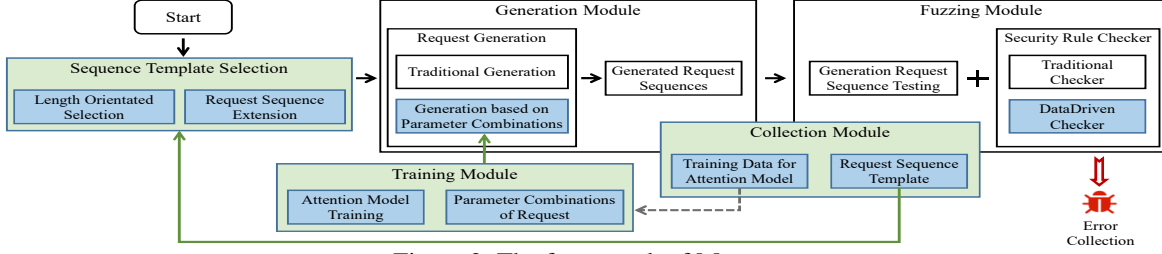


Figure 3: The framework of MINER.

Sequence Template Selection: In this component, MINER constructs candidate sequence templates based on the seed sequence templates collected from the Collection Module. Specifically, MINER first assigns a different selection probability to each seed sequence template according to its length. The longer a sequence template is, the larger its selection probability is. Thus, MINER obtains a set of selected sequence templates that tend to have larger lengths. Second, for each selected sequence template, MINER performs the extension process to construct a candidate sequence template. In other words, MINER adds a new request template at the end of the selected template. Based on the above design, MINER implements the *length-orientated sequence construction*. In implementation, we use $\log_{10}(l+1)$, where l is the length of a sequence template, as the probability function for the Sequence Template Selection.

After finishing all the extension processes, MINER obtains a set of candidate sequence templates and enters the Generation Module.

Generation Module: In this component, MINER generates each request in a candidate sequence template, i.e., MINER assigns a value for each parameter excluding a target object id in a request. As shown in Fig. 3, MINER implements two methods to generate requests. The first one is the traditional generation method as described in Section 2.2, i.e., randomly selecting an alternative value for each parameter. On the other hand, the procedure of the second method is as follows. To generate a request, MINER 1) selects the default value for each parameter; 2) finds the set of *param-value lists*, each of which contains a list of *param-value pairs* for mutation, generated by the attention model for this request template; 3) probabilistically selects a candidate *param-value list* from the set according to a uniform distribution; and 4) uses the specified values to modify the parameters according to the *param-value pairs* in the selected list. For the parameters that are not in the selected *param-value list*, MINER remains their default values in request generation. By performing the above procedure, MINER achieves the *attention model-based request generation*.

In implementation, to generate a sequence containing n requests, MINER uses *param-value lists* to generate the first $n-1$ requests, and uses the traditional generation method to generate the last request. The reasons are as follows. When generating the first $n-1$ requests, MINER would like to lever-

age *param-value lists* to improve the generation quality. This helps trigger normal executions of a target cloud service, and obtain a target object id if there is a producer in the request sequence. Then, MINER utilizes the traditional generation method to generate the last request, in order to increase the probability of triggering an abnormal behavior, e.g., tries to create a resource with the parameter value that does not conform to the defined type.

After generating all the requests, MINER obtains ready-to-use request sequences, and enters the Fuzzing Module.

Fuzzing Module: In this component, MINER performs the following process to explore unique errors. First, MINER sends a generated request sequence to a cloud service via its REST API, and analyzes the corresponding response to each request. If MINER receives a response in $50 \times \text{Range}$, MINER thinks that the corresponding request triggers an error, and stores the generated sequence among with their responses locally for future analysis. Second, MINER leverages each security rule checker to mutate the current request sequence, in order to trigger specific rule violations. In particular, we propose the *request parameter violation checking*, and implement a new security rule checker named *DataDriven Checker* to find the errors caused by incorrect parameter usage, whose target rule violation is as follows.

In general, vendors define specific parameters for each request to trigger specific behaviors of a cloud service. If a user adds an undefined parameter to a request and sends it to a cloud service, generally the cloud service ignores the incorrect usage of the undefined parameter, and performs behaviors according to the values of the defined parameters. However, a cloud service may perform unexpected behaviors according to the undefined parameters due to unexpected conditions, e.g., parameter definition update and incorrect code implementation. The unexpected behaviors can trigger security-related exceptions, e.g., a cloud service may access a non-existent resource. As a result, the process of a cloud service can crash and return a response in $50 \times \text{Range}$ due to undefined parameters.

To capture this kind of rule violation, the *DataDriven Checker* adds an undefined *param-value pair* to a request during the generation process. Specifically, for the last request in a sequence, MINER randomly selects an undefined *param-value pair* collected in the Collection Module. Then, MINER adds the selected pair into the last request of the se-

quence, e.g., adds `<"min_access_level", "1">` in the request shown in Fig. 1d. Finally, MINER sends the sequence with the newly constructed request to a cloud service via its REST API. If the cloud service returns a response in $50 \times \text{Range}$, MINER considers that the new request triggers an incorrect parameter usage error, and stores the sequence locally for further analysis.

After executing the above process for all the generated request sequences, MINER enters the Sequence Template Selection to start the next iteration.

Collection Module: In the fuzzing process, MINER first collects the valid request sequences, in which all the requests pass the cloud service’s checking in the Fuzzing Module. These sequences are used as the seed sequence templates in the Sequence Template Selection to guide the sequence construction. Second, MINER analyzes the response of each generated request from a cloud service, and extracts the *param-value pairs* used in the valid requests that pass the checking. Specifically, if a request gets a response in $20 \times$ and $50 \times \text{Range}$, which means that the request passes the checking and triggers a behavior of the cloud service, MINER analyzes the used value for each parameter in the request. If the used value is not the default one for a parameter, MINER regards the value as a key mutation on this parameter, and stores them as a *param-value pair*. Thus, MINER collects a list of *param-value pairs* for each valid request, which contains the valid mutations to help pass the cloud service’s checking. For instance, MINER will collect [`<"with_custom_attributes", "true">`, `<"with_projects", "false">`] for the example in Fig. 1d. The collected lists are used to 1) train the attention model in the Training Module, and 2) provide undefined *param-value pairs* for the DataDriven Checker to trigger incorrect parameter usage errors.

Training Module: MINER periodically invokes the Training Module to train an attention model with the *param-value pairs* collected in the Collection Module. After training, MINER utilizes the model to generate the *param-value lists* for each request template, which will be used in the Generation Module to generate requests. In the following, we describe the detailed usage of the attention model.

3.3 Attention Model used in MINER

In this paper, we utilize a customized attention model, which is one of the most powerful machine learning models for sequence generation, to learn the implicit relationship between *param-value pairs*. Then, we utilize the model to explore more ideal combinations of *param-value pairs*, which are denoted as *param-value lists*. To achieve this, in the Training Module, MINER trains the attention model using the lists of *param-value pairs* as the training data, which are collected from the requests triggering responses in $20 \times \text{Range}$. The process is as follows.

- 1) MINER adds the corresponding request name in the

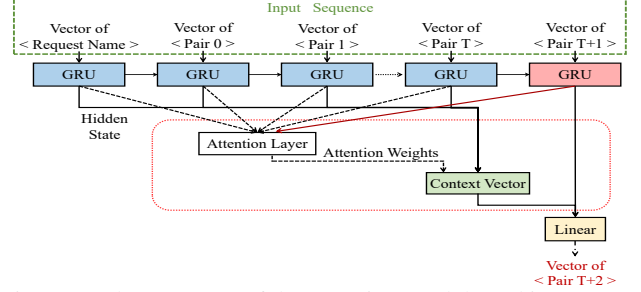


Figure 4: The structure of the attention model used in MINER.

front of each list of *param-value pairs*, and regards each list as a word sequence, e.g., [`"GET /api/v4/groups/{id}"`, `<"with_custom_attributes", true>`, `<"with_projects", false>`] for the example as shown in Fig. 1. Then, we follow the common processing of the machine learning model and embed all the words into vectors, e.g., `<"with_projects", false>` is represented by a unique vector in matrix form. Thus, the lists collected in the Collection Module are transformed into a set of vector sequences (similar to the input sequence as shown in Fig. 4); 2) MINER randomly divides the vector sequences into the training set and the validation set, which are used to train the model and verify the prediction performance of the model, respectively; 3) In each set, MINER utilizes a vector sequence to train/verify the attention model. Specifically, MINER uses the previous n vectors in the sequence as the input and the $n + 1$ th vector as the expected output as shown in Fig. 4. During the training process, the model adaptively adjusts the weights between neurons and increases the prediction probability of the expected output vector. For instance, assuming the ideal output is `<pair T+2>` as shown in Fig. 4, the model will adjust the weights to improve the prediction probability of `<pair T+2>` and lower others’ prediction probabilities; 4) After MINER finishes a certain number of training epochs and improves the prediction accuracy to a stable level, the training process ends.

After the training process, MINER utilizes the attention model to generate the *param-value lists* for the request templates, which provide the *param-value pairs* as the training data. To be specific, MINER employs the vector of a request name as the first input of the attention model, and predicts a *param-value pair* as the output. Next, the predicted pair is appended to the previous input as the new input to predict the next pair. The end conditions for the generation of *param-value pairs* are as follows. 1) The attention model thinks that the sequence generation finishes and outputs a terminator (generally the terminator is a null vector that does not represent any word); 2) The list of *param-value pairs* reaches the preset maximum length, which is significantly greater than the sequence length in the training set. Then, the generated list is stored locally, and will be used as a *param-value list* in the Generation Module as described in Section 3.2.

In implementation, we use a lightweight attention model, which contains a Gated Recurrent Unit (GRU) neural net-

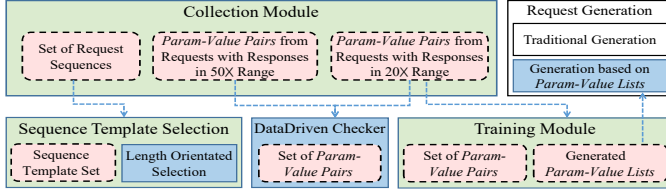


Figure 5: The data flow of the three new designs in MINER.

work [18, 20], an attention layer [38] and a linear layer as shown in Fig. 4, to handle our generation problem. MINER invokes the Training Module every two hours and retrain the model from scratch based on the newly discovered training data. The training and generation process of the attention model runs in parallel with the fuzzing process, which will not affect the fuzzing performance of MINER. Contributed by the above settings, the time cost to train our model is less than 500 seconds as shown in Section 5.5.

3.4 Data Flow of MINER

In this subsection, we illustrate how the three core data-driven designs work together in MINER from the perspective of data flow. As shown in Fig. 5, in the Collection Module, MINER mainly collects three datasets: 1) the set of request sequences whose requests successfully pass the checking of a cloud service; 2) the *param-value pairs* from the requests with responses in $50\times$ Range; and 3) the *param-value pairs* from the requests with responses in $20\times$ Range. In the following, we illustrate which design in MINER utilizes these datasets.

For the set of request sequences, MINER uses it to guide sequence construction in the Sequence Template Selection. Since MINER mainly deals with sequence template construction in this design, the other two datasets are not used.

The *param-value pairs* from the valid requests, which pass the cloud service’s checking and have responses in $50\times$ and $20\times$ Range, are used in the *DataDriven Checker* to explore incorrect parameter usage errors. Since a request should pass the checking before triggering a potential error, in the *DataDriven Checker* MINER does not use the *param-value pairs* from the requests having the responses in $40\times$ Range, which may cause the generated request to be rejected by a cloud service and reduces the error discovery efficiency.

The *param-value pairs* from the requests, whose responses are in $20\times$ Range, are also used in the Training Module as the training data of the attention model. Thus, the attention model tends to generate the *param-value lists* that help pass the checking of a cloud service. Since 1) the requests having responses in $50\times$ Range are pretty rare in the fuzzing process and 2) not all types of requests can get the responses in $50\times$ Range, their number is not sufficient as the training data of a model. Thus, we do not include their *param-value pairs* as the training data in our application scenario. On the contrary, if we collect sufficient requests with responses in $50\times$ Range in the application scenarios like continuous fuzzing for a cloud

Table 1: The target cloud services and 11 REST APIs used in the experiments.

Cloud Service	REST API	# Request Templates	# Request Parameters	Description
GitLab 14.1.0-ce.0.	Projects	29	304	API to interact with projects
	Groups	13	293	API related to groups
	Issues	23	89	API to interact with issues
	Commits	14	59	API related to commits
	Branches	8	28	API to interact with branches
Bugzilla 5.0.4	Comments	8	26	API to maintain comments
	Bugs	11	89	API to maintain bug reports
	Groups	6	29	API to maintain user groups
WordPress 5.8.1	Categories	5	20	API to maintain categories
	Posts	10	42	API to interact with posts
	Comments	8	40	API to maintain comments

service, we can utilize a machine learning model to learn how to generate a request triggering a response in $50\times$ Range, and utilize the model to generate requests on demand.

Based on the above illustration from the perspective of data flow, we can learn that all the datasets for MINER can be automatically collected during the fuzzing process. All the three designs rely on the datasets from the Collection Module, and are able to work together. Our hybrid data-driven approach is compatible with the fuzzing process, which can be deployed in general REST API fuzzers.

4 Evaluation

4.1 Experiment Setup

In implementation, in order to measure the error discovery performance of our checker, we construct two prototypes without and with the *DataDriven Checker*, which are named MINER_PART and MINER, respectively.

Compared fuzzer. We evaluate MINER_PART and MINER against the state-of-the-art open-sourced fuzzer *RESTler* [12], which is the first REST API fuzzer proposed in 2019 to explore errors on a cloud service automatically. Since another REST API fuzzer Pythia is not open-sourced, we cannot compare our fuzzers with it [13].

Target cloud services. We evaluate the above fuzzers on GitLab [5], Bugzilla [2] and WordPress [8] via 11 REST APIs as shown in Table 1, and the reasons to select these targets are as follows. First, GitLab, Bugzilla and WordPress are widely-used cloud services. Evaluating their security is meaningful for vendors and users. Second, the request templates for the REST APIs of GitLab contain many parameters to support a wide variety of functionalities. On the contrary, the request templates of Bugzilla and WordPress are relatively simple ones with fewer parameters, which limits the place that can be mutated. Thus, in the evaluation, we can analyze the performance of fuzzers on the targets with different functional complexity. Third, each REST API interacts with different kinds of resources on a cloud service. Fuzzing via these REST APIs, the evaluated fuzzers can execute different code lines

Table 2: The fuzzing performance of RESTler, MINER_PART and MINER on 3 cloud services.

Cloud Service	REST API	# Total Request Templates	RESTler			MINER_PART			MINER		
			Pass Rate	# Unique Request Templates	Errors	Pass Rate	# Unique Request Templates	Errors	Pass Rate	# Unique Request Templates	Errors ^a
GitLab	Projects	29	72.01%	22	7	95.77%	26	17	95.78%	26	21 (9)
	Groups	13	65.01%	10	21	92.37%	12	26	92.28%	12	33 (10)
	Issues	23	86.56%	21	7	96.13%	21	7	95.42%	21	15 (6)
	Commits	14	53.52%	12	16	86.65%	12	20	86.70%	12	37 (12)
	Branches	8	81.10%	8	2	89.91%	8	3	89.31%	8	9 (6)
Bugzilla	Comments	8	88.79%	7	8	89.74%	8	8	90.23%	8	8 (0)
	Bugs	11	45.03%	4	7	91.05%	4	7	93.16%	4	14 (6)
	Groups	6	54.88%	4	5	74.01%	5	6	72.85%	5	11 (5)
WordPress	Categories	5	75.33%	4	8	91.96%	4	10	92.64%	4	13 (4)
	Posts	10	94.06%	10	4	95.13%	10	5	95.56%	10	7 (2)
	Comments	8	96.61%	4	0	99.65%	4	0	99.43%	4	0 (0)
Average			73.90%	9.64	7.73	91.12%	10.36	9.91	91.21%	10.36	15.27 (5.45)

^aThe number of unique errors found by MINER is presented in two parts: the total number of all the unique errors as shown in front of the parentheses, and the number of unique errors found by the *DataDriven Checker* as shown in the parentheses.

and explore different states of a cloud service, which can examine their fuzzing performance more comprehensively.

To construct the targets, we deploy an open-sourced version of each cloud service on our server, and construct request templates with the *Compiler Module* for all the REST APIs, which is a one-shot effort and costs us about 1 hour.

Experiment settings. Each evaluation lasts for 48 hours on a docker container configured with 8 CPU cores, 20 GB RAM, Python 3.8.2, and the OS of Ubuntu 16.04 LTS. We run evaluations on 3 servers, each of which has two E5-2680 CPUs, 256GB RAM and a Nvidia GTX 1080 Ti graphics card.

Evaluation metrics. To measure the performance of each fuzzer on request generation and error discovery, we evaluate the fuzzers with the following three metrics. First, to evaluate the request generation quality, we measure the pass rate of each fuzzer to the syntax and semantic checking of a cloud service, which is calculated by dividing the number of the responses in $20\times$ and $50\times$ Range by the number of total responses. Second, to measure how many unique request templates are successfully generated by each fuzzer, we count the types of generated requests that get responses in $20\times$ Range. The more unique request templates are successfully generated and sent to a cloud service, the more kinds of behaviors are triggered by a REST API fuzzer. Third, we count the number of unique errors, which trigger the responses in $50\times$ Range or violate the defined security rules, reported by each fuzzer.

4.2 Fuzzing Performance Analysis

In this subsection, we analyze the fuzzing results reported by each fuzzer. The details are shown in Table 2, from which we have the following conclusions.

- As shown in Table 2, the new designs implemented in our fuzzers significantly improve the pass rate on a cloud service. For instance, the average pass rate of MINER_PART and MINER is 28.65% and 28.28% higher than RESTler on GitLab, respectively. The average pass rate of MINER_PART is 35.03% higher than RESTler on Bugzilla. Although the request templates of several APIs, e.g., WordPress Posts API,

are easy-to-construct ones containing few parameters, our fuzzers still achieve higher pass rate compared to RESTler. The experimental results confirm the pass rate improvement contributed by our hybrid data-driven approach.

- MINER_PART and MINER can generate high-quality requests for more unique request templates than RESTler. For instance, the average number of unique request templates covered by MINER_PART is 8.22% larger than RESTler on GitLab. The number of unique request templates covered by MINER is 20.00% larger than RESTler on GitLab Groups API. We analyze the executions of each request template and have the following conclusions. Our hybrid data-driven approach significantly improves the complexity of generated request sequences. First, the *length-orientated sequence construction* implemented in MINER increases the executions of long request sequences, and increases the attempts on the consumers which require target object ids. Second, MINER’s *attention model-based request generation* significantly improves request generation quality. In particular, MINER improves the producers’ pass rate and obtains target object ids, which can serve as the key parameter values for consumers. As a result, MINER successfully sends more types of requests to a cloud service. Since the only difference between MINER_PART and MINER is that MINER contains a new security rule checker *DataDriven Checker*, they perform the same on the number of covered request templates.

- Both MINER_PART and MINER find more unique errors than RESTler on most targets. For instance, MINER_PART finds 37.74% and 25.00% more unique errors than RESTler on GitLab and WordPress, respectively. We analyze the reasons as follows. As indicated in Section 3.1, the kinds of request combinations generated by RESTler are limited due to the short generated request sequences. On the contrary, contributed by our approach, MINER_PART and MINER can generate long sequences containing high-quality requests to test a cloud service. The kinds of request combinations cover and far exceed those generated by RESTler. Thus, they can trigger more kinds of the cloud service’s behaviors in a sin-

Table 3: The real errors found by RESTler, MINER_PART and MINER on GitLab, Bugzilla and WordPress.

Target	No.	Endpoints of Used Requests	Responses	Description	RESTler	MINER_PART	MINER
GitLab	1	1. POST /api/v4/projects 2. GET /api/v4/projects/:id/fork	1. 201 Created 2. 500 Internal Error	Use <"namespace_id", "-1"> pair in the second request.	●	●	●
	2	1. POST /api/v4/projects 2. PUT /api/v4/projects/:id	1. 201 Created 2. 500 Internal Error	The second request updates a project with the undefined parameter "initialize_with_readme".			●
	3	1. POST /api/v4/projects 2. GET /api/v4/projects/:id/custom_attributes 3. DELETE /api/v4/projects/:id 4. GET /api/v4/projects/:id/custom_attributes	1. 201 Created 2. 200 OK 3. 202 Accepted 4. 500 Internal Error	Use after free ^a . The fourth request tries to access the attributes of a deleted project.		●	●
	4	1. GET /api/v4/projects/:id/custom_attributes	1. 500 Internal Error	Using string or invlaid number as :id.	●	●	●
	5	1. POST /api/v4/projects 2. POST /api/v4/projects/:proj_id/hooks 3. PUT /api/v4/projects/:proj_id/hooks/:hook_id	1. 201 Created 2. 200 OK 3. 500 Internal Error	The third request updates a hook with 3 undefined parameters "requirements_access_level", "auto_cancel_pending_pipelines" and "initialize_with_readme".			●
	6	1. GET /api/v4/groups	1. 500 Internal Error	List groups with 2 parameters "statistics" and "min_access_level".	●	●	●
	7	1. GET /api/v4/groups	1. 500 Internal Error	Use <"per_page", "0"> pair in the request.	●	●	●
	8	1. POST /api/v4/groups	1. 500 Internal Error	Error when the parameter "parent_id" is set to be a special number like 2, -1, -2.	●	●	●
	9	1. POST /api/v4/groups	1. 500 Internal Error	Create group with the undefined parameter "shared_runners_setting".			●
	10	1. POST /api/v4/groups 2. DELETE /api/v4/groups/:id 3. GET /api/v4/groups/:id/descendant_groups	1. 201 Created 2. 202 Accepted 3. 200 OK	Use after free. The third request successfully accesses the descendant groups of a deleted group. Private data leakage.		●	●
	11	1. POST /projects/:id/issues 2. PUT /projects/:id/issues/:issue iid	1. 201 Created 2. 500 Internal Error	Update issue with an extra-long label.	●	●	●
	12	1. POST /api/v4/groups 2. DELETE /api/v4/groups/:id 3. GET /api/v4/groups/:id	1. 201 Created 2. 202 Accepted 3. 200 OK	Use after free. The third request lists a group with the parameters "with_custom_attributes" and "with_projects". Private data leakage.		●	●
	13	1. POST /projects/:id/repository/branches 2. DELETE /projects/:id/repository/branches/:Br 3. GET /projects/:id/repository/branches/:Br 4. POST /projects/:id/repository/branches	1. 201 Created 2. 204 No Content 3. 404 Not Found 4. 201 Created	Use after free. The fourth request successfully creates a branch while its "ref" refers to the name of the deleted branch. Private data leakage.		●	●
	14	1. POST /api/v4/projects/:proj_id/repository/branches	1. 500 Internal Error	Create a new branch with the undefined parameter "sort".			●
Bugzilla	15	1. POST /rest/bugs	1. 500 Internal Error	Use invalid datetime value in the request.	●	●	●
WordPress	16	1. POST /categories ⇒ Category id: (cid) 2. POST /categories 3. DELETE /categories(cid) 4. POST /categories	1. 201 Created 2. 201 Created 3. 200 OK 4. 500 Internal Error	Use after free. The parent of both the second and forth requests refers to the same id of the category (cid). However, the fourth request triggers an error after (cid) gets deleted.		●	●
	17	1. POST /wp/v2/posts	1. Timeout	Connections get closed after timeout.		●	●
Total					7	13	17

^aA use-after-free error on cloud services means that the request accesses a deleted resource, which leads to bypassing resource quotas and corrupting service state.

gle request sequence test, followed by finding more errors than RESTler. In addition, since MINER_PART and MINER can successfully generate requests for more unique request templates as shown in Table 2, they can trigger more and different behaviors of a target cloud service compared to RESTler. Thus, they can trigger more unique errors based on these unique request templates. Due to the randomness in fuzzing, the number of unique errors found by MINER_PART and MINER is slightly different, if we do not count the errors found by the *DataDriven Checker* for MINER.

- The *DataDriven Checker* of MINER finds many unique errors caused by misuse of undefined parameters. For instance, MINER finds 43 unique errors in total with the *DataDriven Checker* on GitLab via the 5 REST APIs. MINER also finds 6 unique errors that misuse undefined parameters on Bugzilla Bugs API. The results demonstrate that 1) incorrect parameter usage errors commonly exist in various cloud services, and 2) MINER can effectively discover this kind of unique errors with our data-driven designs.

4.3 Real Error Analysis

To further evaluate the performance of each fuzzer on real error discovery, we manually analyze all the errors on GitLab, Bugzilla and WordPress reported by each fuzzer. Specifically, we filter out the errors that cannot be reproduced due to the change of server states as discussed in Section 6.2. Then, we deduplicate the errors that are caused by the same reason. The details of real errors on three cloud services found by each fuzzer are shown in Table 3.

- The exploration space of real errors triggered by RESTler is fully covered and exceeded by MINER_PART and MINER. After error filtering and deduplicating, MINER_PART finds all the errors discovered by RESTler, and finds 6 more unique errors missed by RESTler, from which we can have the following conclusions. 1) Since RESTler's possible request combinations are covered by MINER_PART, the real errors triggered by RESTler also can be found by MINER_PART; 2) Thanks to the *length-orientated sequence construction* implemented in MINER_PART, MINER_PART assigns more execution times to long request sequences. As a result, MINER_PART finds 5 new unique errors triggered by the request sequences of

Table 4: The pass rate and the number of unique errors found by RESTler, RESTler+Seq, RESTler+Rec1, RESTler+RecList, RESTler+Model, and MINER_PART, respectively.

Cloud Service	REST API	RESTler		RESTler+Seq		RESTler+Rec1		RESTler+RecList		RESTler+Model		MINER_PART	
		Pass Rate	Errors	Pass Rate	Errors	Pass Rate	Errors	Pass Rate	Errors	Pass Rate	Errors	Pass Rate	Errors
GitLab	Projects	69.48%	7.00	89.57%	7.50	66.10%	7.50	65.66%	7.25	87.11%	9.00	94.70%	9.75
	Groups	60.28%	16.00	94.67%	19.50	65.22%	19.00	65.56%	17.00	94.53%	19.00	94.66%	19.50
	Issues	86.32%	6.50	89.06%	6.50	88.14%	7.00	90.38%	7.00	91.00%	7.50	94.31%	7.00
Bugzilla	Comments	86.19%	8.00	89.05%	8.00	87.05%	8.00	89.05%	8.00	87.94%	8.25	89.36%	8.00
WordPress	Categories	81.35%	9.00	86.40%	8.75	81.03%	9.00	81.44%	9.00	82.41%	9.50	90.53%	9.50

length greater than 2; 3) We find that RESTler cannot find the No. 16 error triggered by 4 requests, but all four requests had been successfully generated by RESTler, as the number of unique request templates successfully generated by RESTler is the same as MINER_PART and MINER on WordPress Categories API as shown in Table 2. Thus, either RESTler fails to construct a sequence of sufficient length, or RESTler does not generate high-quality requests to pass the checking of WordPress. This case study demonstrates the motivation of our paper, i.e., a new solution is demanded to improve the sequence construction and request generation.

- The *DataDriven Checker* implemented in MINER can efficiently find the unique errors caused by incorrect parameter usage. For instance, as shown by the No. 2 error, MINER adds an undefined parameter "*initialize_with_readme*" to the PUT request, which is the parameter used in other requests like POST. Violating the standard API specification, GitLab executes the behavior of "*initialize_with_readme*". However, this behavior is not compatible with the behavior of PUT, and causes the response of 500 Internal Error. For another example shown by the No. 9 error, MINER sends a POST request with the undefined parameter "*shared_runners_setting*". When creating a group, GitLab accesses a non-existent resource because of this parameter, which causes the crash of the creation process. As a result, GitLab returns the response of 500 Internal Error. Although MINER finds several incorrect parameter usage errors on Bugzilla and WordPress, we find that their root causes are the same as the shown errors in Table 3. Thus, these errors are not shown in Table 3.

Security bugs. Our fuzzers find 5 security bugs that allow attackers to access resources that have been deleted, which may cause the leakage of private user data. For instance, as shown by the No. 13 error, the fourth request creates a new branch referring to the deleted one, which copies the resource data from the deleted one. Therefore, attackers can leverage the No. 13 error to restore the deleted private data of users. These cases demonstrate the significant performance of MINER in constructing long request sequences to explore head-to-reach states and find security bugs. On the contrary, RESTler hardly generates a request sequence of length greater than 2, and cannot trigger any of the 5 security bugs.

We have reported all the discovered unique errors on GitLab, Bugzilla and WordPress, and 7 of them have already been confirmed as logic bugs by vendors.

5 Further Analysis

5.1 Stepwise Analysis

In order to evaluate the contribution of our data-driven designs to the pass rate and error discovery, we construct the following 4 fuzzers with different designs. To be specific, we construct RESTler+Seq by implementing the *length-orientated sequence construction* in RESTler; RESTler+Rec1 is the one that generates requests by randomly using a *param-value pair* recorded in the *Collection* Module and using default values for other parameters; RESTler+RecList randomly uses a list of recorded *param-value pairs* to generate requests, i.e., it reproduces and sends the same requests to trigger the same behaviors of cloud services; and we combine RESTler with the *attention model-based request generation* to construct RESTler+Model. Then, we can measure the contribution of the *length-orientated sequence construction* by comparing the results of MINER_PART and RESTler+Model, and the results of RESTler+Seq and RESTler. We can measure the contribution of the *attention model-based request generation* from the results of MINER_PART and RESTler+Seq, and the results of RESTler+Model and RESTler. Furthermore, we can evaluate the contribution of different request generation algorithms on pass rate and error discovery by comparing the performance of RESTler+Rec1, RESTler+RecList and RESTler+Model. Since the evaluation in Section 4 demonstrates the significant performance of the *DataDriven Checker* on the discovery of incorrect parameter usage error, we do not evaluate its performance in this subsection. Each evaluation lasts for 12 hours and is repeated 4 times. The experiment settings are the same as in Section 4.1, and the results are shown in Table 4, from which we have the following conclusions.

- The *length-orientated sequence construction* significantly improves the pass rate of a REST API fuzzer, while it cannot improve the error discovery on most evaluations. For instance, the pass rate of MINER_PART is 9.85% higher than RESTler+Model on WordPress Categories API, and the average pass rate of RESTler+Seq is 26.48% higher than RESTler on GitLab. On the other hand, RESTler+Seq performs closely to RESTler on error discovery. To figure out the reason, we analyze the sequence templates constructed by RESTler+Seq, and have the following conclusion. The sequence templates of RESTler+Seq are mainly composed of easy-to-construct request templates, e.g., GET, which contain few parameters and

Table 5: The published bugs found by RESTler and MINER on GitLab.

No.	Endpoints of Used Requests	Responses	Description	RESTler	MINER
1	1. POST /api/v4/projects 2. GET /api/v4/projects/:id/custom_attributes 3. DELETE /api/v4/projects/:id 4. GET /api/v4/projects/:id/custom_attributes	1. 201 Created 2. 200 OK 3. 202 Accepted 4. 500 Internal Error	Use after free ^a . The fourth request tries to access the attributes of a deleted project.		●
2	1. POST /api/v4/projects 2. POST /api/v4/projects/:id_1/fork 3. POST /api/v4/projects 4. POST /api/v4/projects/:id_2/fork 5. POST /projects/:id_1/fork/:id_2	1. 201 Created 2. 201 Created 3. 201 Created 4. 201 Created 5. 500 Internal Error	Error when creating a forked relationship between existing projects that already have forks.		●
3	1. GET /api/v4/groups	1. 500 Internal Error	List groups with 2 parameters " <i>statistics</i> " and " <i>min_access_level</i> ".	●	●
4	1. GET /api/v4/groups	1. 500 Internal Error	Use <" <i>per_page</i> ", "0"> pair in request.	●	●
		Total		2	4

^aA use-after-free error on a cloud service means that the request accesses a deleted resource.

Table 6: The average time spent by RESTler and MINER to trigger the published bugs on GitLab, where N/A means that the fuzzer fails to trigger the bug in 48 hours.

No.	REST API	# Request Templates	Minimum Length to Trigger Bugs	RESTler	MINER
1	Projects	3	3	N/A	66.5 mins
2	Projects	4	5	N/A	142.6 mins
3	Groups	1	1	27.6 mins	26.6 mins
4	Groups	1	1	13.7 mins	14.2 mins

are easy to pass the cloud service’s checking. Because of the *length-orientated sequence construction*, RESTler+Seq tends to generate sequences based on long sequence templates, in which all the requests have passed the checking. However, due to low request generation quality, RESTler+Seq rarely stores hard-to-construct request templates in its sequence templates, and its sequence templates are full of easy-to-construct request templates. As a result, the executions of easy-to-construct requests increase by dozens of times, which results in a high pass rate. On the contrary, MINER_PART improves the pass rate of hard-to-construct requests with the help of the *attention model-based request generation*, which improves the diversity of sequence templates and error discovery.

- The *attention model-based request generation* can improve the error discovery and pass rate of a REST API fuzzer. For instance, RESTler+Model finds more errors than RESTler on all 5 evaluations. MINER_PART also performs better than RESTler+Seq on pass rate and error discovery with the help of the *attention model-based request generation*. The results demonstrate the contribution of our new design to the fuzzing performance. Furthermore, MINER_PART performs the best on pass rate and error discovery on most evaluations, which implies that the two data-driven designs can be combined to improve each other.

- The *attention model-based request generation* can effectively provide key *param-value lists* for request generation and improve the fuzzing performance compared to straightforward algorithms. For instance, the pass rate and error discovery of RESTler+Model is better than RESTler+Rec1 on most

evaluations. The error discovery of RESTler+Model is significantly better than RESTler+RecList on all 5 evaluations. Thus, neither randomly selecting a mutation nor simply reproducing a well-constructed request can improve the performance of a REST API fuzzer. The results demonstrate that our design learns the implicit relations between key *param-value pairs*, and generates diverse yet valid mutations as *param-value lists*. Our design can serve as a new direction to improve request generation quality for REST API fuzzing.

5.2 Performance on Reproducing Serious Bugs

To evaluate the performance of RESTler and MINER on serious bug discovery, we manually check the published issues of GitLab, and collect 4 serious bugs that a REST API fuzzer can reproduce as the ground truth. Then, we construct the fuzzing grammar that only contains the relevant request templates for each unique bug, as shown in Table 6. For instance, to reproduce the No. 2 bug, a fuzzer needs to correctly generate 5 requests in a reasonable order, which are based on 4 different request templates in the grammar. Each evaluation lasts up to 48 hours and is repeated 5 times to calculate the average trigger time. The results are shown in Table 5 and Table 6, from which we have the following observations.

- MINER can reproduce the serious bugs published on GitLab more effectively and efficiently compared to RESTler. MINER can reproduce all 4 bugs, while RESTler can reproduce two. Furthermore, it takes 142.6 mins on average for MINER to construct a sequence with 5 requests in the correct order as shown in Table 5, while RESTler cannot construct the sequence within 48 hours. The results demonstrate the significant performance of MINER on serious bug discovery, and we analyze the reasons as follows. First, MINER can construct long request sequences with the *length-orientated sequence construction*, which increases the execution times on complicated request combinations compared to RESTler. Second, the *attention model-based request generation* implemented in MINER improves the pass rate of hard-to-construct requests to GitLab, which reduces the trigger time of bugs.

- Our designs will not prevent MINER from triggering

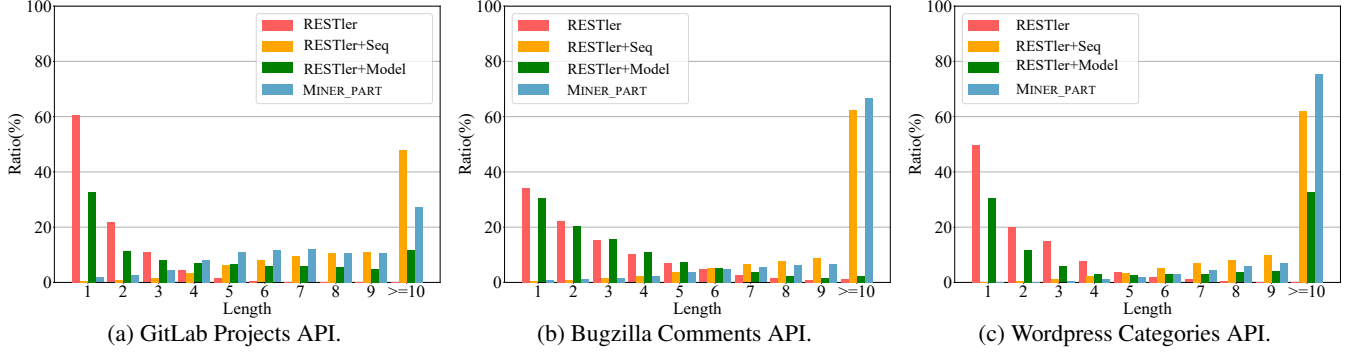


Figure 6: The distribution of sequence length sent by 4 fuzzers, where RESTler+Seq only implements the *length-orientated sequence construction* and RESTler+Model only implements the *attention model-based request generation*.

relatively simple bugs. As shown in Table 5, the No. 3 and No. 4 bugs are triggered by one request with incorrect parameters and values. Both MINER and RESTler trigger the two bugs by using the traditional generation method to generate the corresponding request, and the average time spent by the two fuzzers is close. In conclusion, MINER performs closely with RESTler on discovering this kind of bugs, which are triggered by one incorrectly constructed request.

5.3 Sequence Length Analysis

To analyze the contribution of different designs to sequence length, we conduct the experiments as follows. We evaluate RESTler, RESTler+Seq, RESTler+Model, and MINER_PART on GitLab Projects API, Bugzilla Comments API, and Wordpress Categories API. Each evaluation lasts for 12 hours, during which we count the executions of sequences with different lengths. Based on the above settings, we can measure the contribution of the *attention model-based request generation* to sequence length from the results of RESTler and RESTler+Model, and the results of RESTler+Seq and MINER_PART. We can measure the contribution of the *length-orientated sequence construction* to sequence length from the results of RESTler and RESTler+Seq, and the results of RESTler+Model and MINER_PART. Since the *DataDriven Checker* does not affect the sequence extension, we omit MINER in the evaluation. The results are shown in Fig. 6, from which we have the following conclusions.

- As shown in Fig. 6, the *attention model-based request generation* can increase the executions of long request sequences. For instance, RESTler+Model assigns more executions on longer request sequences than RESTler on all 3 targets. MINER_PART also generates more request sequences with larger lengths compared to RESTler+Seq. The results demonstrate the contribution of the *attention model-based request generation* to sequence extension. By improving the pass rate of requests with our design, a REST API fuzzer can construct request sequences with larger lengths.

- The *length-orientated sequence construction* significantly increases the executions of request sequences with large lengths. Both MINER_PART and RESTler+Seq assign most

executions to the request sequences whose lengths are greater than 10, demonstrating our design’s effectiveness.

5.4 Coverage Performance Analysis

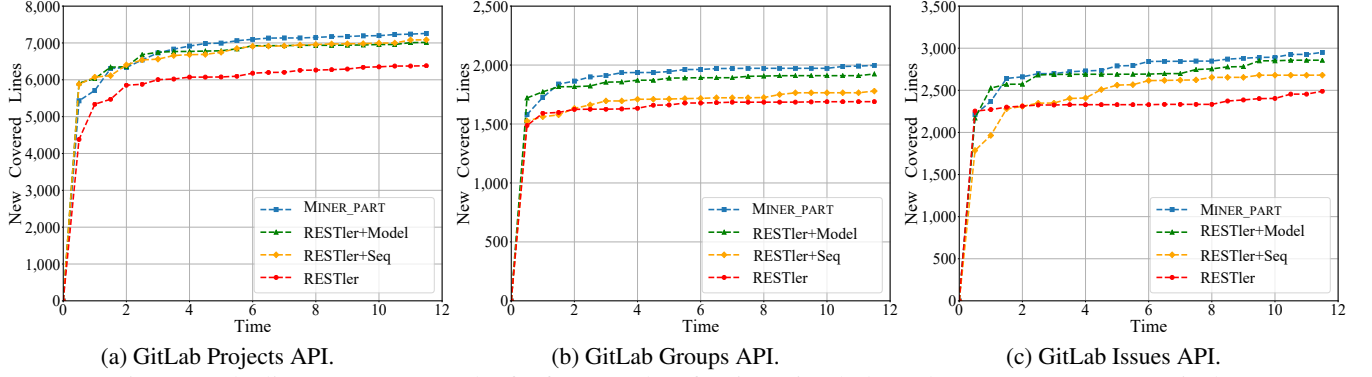
To evaluate the coverage performance of each fuzzer, we hook the source code of GitLab to trace the number of covered code lines triggered by the incoming requests. Specifically, we leverage a popular ruby gem named *Coverband* [3] to trace the code lines executed in GitLab’s *service/lib* and *service/app* folders. We also exclude the coverage statistics for irrelevant behaviors like service boot. Then, we conduct the line coverage evaluation for RESTler, RESTler+Seq, RESTler+Model, and MINER_PART, each of which lasts for 12 hours. Since the *DataDriven Checker* reuses the *param-value pairs* collected from the used requests and triggers the repeated code lines, it does not affect line coverage. Thus, we omit MINER in the evaluation. The line coverage growth of each fuzzer is shown in Fig. 7, from which we have the following observations.

- Both the *attention model-based request generation* and *length-orientated sequence construction* can improve line coverage. For instance, the line coverage of RESTler+Model and RESTler+Seq is better than RESTler on all the evaluations. Furthermore, the line coverage of RESTler+Model is better than RESTler+Seq on most targets, which implies that a sequence with well-constructed requests triggers deeper states of a cloud service than a long sequence with easy-to-construct requests.

- MINER_PART performs the best on line coverage on all the evaluations. For instance, MINER_PART covers the most code lines at the end of each evaluation. The line coverage triggered by MINER_PART grows rapidly on GitLab Groups API and GitLab Issues API. The results demonstrate that our data-driven designs can be combined to improve the coverage performance of a REST API fuzzer.

5.5 Schedule of the Training Module

To evaluate the impact of different schedules of the Training Module on fuzzing performance, we measure the average



(a) GitLab Projects API.

(b) GitLab Groups API.

(c) GitLab Issues API.

Figure 7: The line coverage growth of 4 fuzzers when fuzzing GitLab through 3 REST APIs, respectively.

Table 7: The average time overhead of the attention model and the pass rate of requests sent by MINER_PART when the iteration duration of the Training Module is 1 hour, 2 hours and 3 hours, respectively.

Target	1 Hour		2 Hours		3 Hours	
	Time Overhead	Pass Rate	Time Overhead	Pass Rate	Time Overhead	Pass Rate
GitLab Projects API	346.20s	92.78%	380.80s	91.54%	989.30s	89.55%
GitLab Groups API	318.90s	93.49%	386.90s	92.80%	1,002.67s	91.92%
GitLab Issues API	463.40s	90.90%	498.67s	90.67%	1,001.60s	90.23%
Bugzilla Comments API	97.60s	88.89%	117.30s	86.47%	143.70s	85.37%
WordPress Categories API	305.93s	91.02%	323.25s	90.84%	402.40s	90.14%

time overhead and pass rate for MINER_PART when using 1 hour, 2 hours and 3 hours as the iteration duration of the Training Module, respectively. Each evaluation lasts for 12 hours. Thus, the invocation times of the Training Module are 12, 6 and 4 when using the above iteration durations, respectively. For each evaluation, we 1) calculate the average time overhead for each invocation of the Training Module to train the attention model and generate the *param-value lists*, and 2) calculate the pass rate of requests. The results are shown in Table 7, from which we have the following conclusions. Note that due to the different number of request templates and the different number of parameters in each request template, the number of the collected *param-value pairs* (i.e., the training data for the attention model) is different when using MINER_PART to fuzz a cloud service via different REST APIs. Thus, the average time overhead to train the attention model and generate *param-value lists* can be different when fuzzing different targets.

- The average time overhead to train the attention model and generate *param-value lists* increases when using a larger iteration duration of the Training Module. For instance, as shown in Table 7, the average time overhead of the Training Module increases by 185.76% when using 3 hours as the iteration duration to fuzz GitLab Projects API compared to 1 hour. The average time overhead increases by 100.85% when using 3 hours as the iteration duration to fuzz GitLab Issues API compared to 2 hours. We analyze the reasons for the results as follows. When the iteration duration of the Training Module increases, the training data collected in the Collection Module increases correspondingly. Thus, the training set of the attention model increases, which re-

quires more time to finish the training.

- The pass rate of MINER_PART slightly decreases when using a longer iteration duration to invoke the Training Module. For instance, the pass rate of MINER_PART decreases from 90.90% to 90.23% when using larger iteration durations to fuzz GitLab Issues API. We infer the reasons for the results as follows. The slight difference in pass rate is caused by the request generation in the first period, in which MINER_PART does not have the *param-value lists* generated by the attention model and uses the traditional generation method to generate requests with a low pass rate. Thus, the larger the iteration duration is, the more times MINER_PART generates requests with a low pass rate in the first period. After the first time invoking the Training Module and obtaining *param-value lists*, MINER_PART improves the generation quality of requests and significantly increases the pass rate. As a result, when using 3 hours as the iteration duration compared to 1 and 2 hours, MINER_PART generates more requests with the traditional generation method, which results in a slightly lower average pass rate.

5.6 Execution Distribution of Requests

In this subsection, we conduct the analysis to count the executions of different request templates sent by RESTler and MINER. Each evaluation lasts for 12 hours, and the distribution of the executions of different request templates is shown in Fig. 8, from which we have the following conclusion. MINER assigns more executions to diverse request templates than RESTler. For instance, as shown in Fig. 8, MINER assigns most executions to 17 templates of requests when fuzzing GitLab Projects API. On the contrary, RESTler

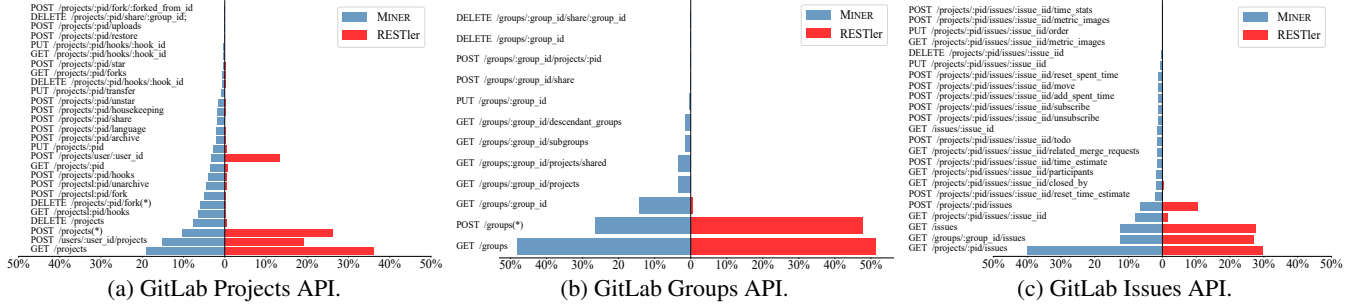


Figure 8: The distribution of executions of different request templates sent by RESTler and MINER, where the request template with (*) means that it has multiple versions to provide different target object ids or it contains different parameters.

mainly tests GitLab with 4 request templates. When fuzzing GitLab Groups API, MINER assigns significantly more executions to 5 extra request templates compared to RESTler. Therefore, based on more different request templates, MINER can generate more diverse requests with different parameter values compared to RESTler. Then, MINER triggers more kinds of behaviors of a target cloud service with these requests, which can explore more and deeper states. Thus, the aforementioned conclusion can serve as one of the reasons that MINER performs better than RESTler on line coverage and error discovery.

6 Discussion and Limitation

6.1 Data-Driven Sequence Generation

In this paper, MINER utilizes the attention model to learn and generate *param-value lists* for each request, which improves the request generation quality. In addition, the model can also be used to explore the implicit relations across different requests and provide key mutation strategies for a request sequence. Due to not having enough training data collected in the fuzzing process, it is hard for us to train an attention model for sequence generation. However, there can be sufficient training data for sequence generation in different scenarios like continuous fuzzing and parallel fuzzing. Therefore, utilizing a machine learning model to provide key mutation strategies for a request sequence can be an interesting topic in future work.

6.2 Error Reproduction Across Sequences

Due to the change in the server states, some unique errors can only be triggered in a specific state of a cloud service, while cannot be reproduced in future analysis. It is a common problem in error reproduction for REST API fuzzing. For instance, a fuzzer can trigger an error by accessing a resource, which is created a few hours ago by other request sequences. However, the fuzzer deletes this resource in the following fuzzing process, making the error irreproducible. Thus, how to analyze the correlation between request sequences that are far apart and how to reproduce this kind of errors can be an

interesting research direction for REST API fuzzing.

7 Related Work

7.1 REST API Fuzzing

In 2019, Atlidakis et al. presented the first stateful REST API fuzzer named *RESTler* to automatically fuzz a cloud service via its REST API [12]. Then, Atlidakis et al. implemented several security rule checkers in RESTler that can automatically detect violations of these security rules [14]. Godefroid et al. defined differential regression testing for REST APIs, which leveraged RESTler to construct network logs on different versions of the target REST APIs and detected service and specification regressions by comparing these network logs [28]. Godefroid et al. also studied how to generate data payloads learned from REST API specifications and found data-processing bugs in cloud services [26]. Based on RESTler, Pythia is presented to leverage a machine learning model to decide different mutation strategies on different positions of a request. Pythia also leverages the code coverage information to guide the fuzzing process, which needs manual cost before fuzzing [13].

Different from existing REST API fuzzers, MINER utilizes an attention model to locate the parameters to be mutated and provide the appropriate values, which is implemented by using the predicted *param-value lists* in request generation. Furthermore, MINER leverages the other two data-driven designs to increase the execution times on long request sequences and explore incorrect parameter usage errors.

7.2 Generation-based Fuzzing

Similar to REST API fuzzing, most generation-based fuzzers generate test cases with specific input formats to fuzz a target program with syntax and semantic checking [4, 22, 27, 33]. Wang et al. presented a novel data-driven seed generation approach named *Skyfire* to learn the syntax and semantic rules and generate the inputs with specific input formats, which are used as the initial seeds for fuzzers [49]. Han et al. presented a novel generation algorithm named *semantics-aware assembly* to generate test cases with semantical and syntactical

correctness [31]. *Nautilus* combines the usage of grammars with code coverage feedback, which generates the test cases with a higher probability of having semantical and syntactical correctness [10]. Lee et al. presented a neural network language model-guided fuzzer named *Montage* to explore JavaScript engine vulnerabilities [36]. *Fuzzilli* utilizes a designed Intermediate Representation (IR) to build syntactically and semantically correct test cases [30].

7.3 Machine Learning-based Fuzzing

Multiple works focus on improving fuzzers with machine learning techniques [15, 17, 29, 39, 42, 43, 46]. *NEUZZ* employs a neural network model to learn the real-world program’s branching behaviors, and then utilizes the program smoothing technique to locate the bytes in a test case that influence the branching behaviors [45]. *MTFuzz* presented by She et al. utilizes a multi-task neural network to learn a compact embedding of program input spaces. The neural network guides the mutation process by predicting which input bytes with the highest likelihood to impact code coverage [44].

7.4 Mutation-based Fuzzing

Mutation-based fuzzing focuses on exploring unique bugs on a target program without the input format requirement, which is different from REST API fuzzing. However, the following development directions of mutation-based fuzzing can still inspire future REST API fuzzing.

Several studies make use of symbolic execution [19, 34, 35, 47, 52, 53] to solve difficult path constraints and improve code coverage. Profuzzer includes a lightweight mechanism to discover the relationship between input bytes and program behaviors [50]. Aschermann et al. presented *Redqueen* to solve magic bytes and checksums automatically with input-to-state correspondence [11]. Gan et al. presented a lightweight data flow-sensitive fuzzing solution named *GREY-ONE*, which contains a taint-guided mutation strategy to mutate a seed on appropriate positions with appropriate values and a conformance-guided evolution solution to collect better seeds in the queue [24].

Energy allocation strategies used in mutation-based fuzzing assign more energy to explore low-frequency paths, untouched branches and effective mutations [16, 25, 40, 51]. Several studies allocate energy based on multi-objective metrics [9, 37, 41, 54]. *AFL++HIER* leverages a multi-armed bandit model to allocate energy to different clusters of seeds with multi-level coverage metrics [48].

Inspired by these state-of-the-art solutions, in future work, REST API fuzzing can be further developed by 1) locating request templates that are most relevant to error discovery and 2) assigning more mutation energy to the request sequences with better error discovery and code coverage.

8 Conclusion

To solve the limitations of REST API fuzzers and improve their error discovery performance, we present a hybrid data-driven approach with three new designs, which help find security bugs triggered by long request sequences and explore incorrect parameter usage errors. We implement the prototype MINER based on our approach, and evaluate MINER against RESTler on 3 open-sourced cloud services via 11 REST APIs. The results show that MINER performs much better than RESTler on request generation and error discovery. Based on manual analysis, MINER also finds 10 more real errors than RESTler, including 5 security bugs that try to access deleted resources. Furthermore, using the published bugs of GitLab as the ground truth, we demonstrate the significant performance of MINER on serious bug discovery. In addition, we conduct extensive analysis to demonstrate the outstanding performance of MINER on the sequence extension, line coverage and time overhead. Overall, our approach can serve as a new direction to improve the sequence extension, pass rate, and error discovery of REST API fuzzers.

Acknowledgments

We sincerely appreciate the guidance from the shepherd. We would also like to thank the anonymous reviewers for their valuable comments and input to improve our paper.

References

- [1] APIFuzzer: HTTP API Testing Framework. <https://github.com/KissPeter/APIFuzzer>.
- [2] Bugzilla. <https://www.bugzilla.org>.
- [3] Coverband. <https://github.com/danmayer/coverband>.
- [4] Funfuzz. <https://github.com/MozillaSecurity/funfuzz>.
- [5] GitLab. <https://gitlab.com/gitlab-org/gitlab>.
- [6] Qualys Web Application Scanning (WAS). <https://www.qualys.com/apps/web-app-scanning>.
- [7] TnT-Fuzzer: OpenAPI Fuzzer Written in Python. <https://github.com/Teebytes/TnT-Fuzzer>.
- [8] WordPress. <https://wordpress.org>.
- [9] Nadia Alshahwan, Xinbo Gao, Mark Harman, Yue Jia, Ke Mao, Alexander Mols, Taijin Tei, and Ilya Zorin. Deploying Search based Software Engineering with Sapienz at Facebook. In *International Symposium on Search Based Software Engineering*, pages 3–45, 2018.

- [10] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. NAUTILUS: Fishing for Deep Bugs with Grammars. In *Network and Distributed Systems Security Symposium*, 2019.
- [11] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. Redqueen: Fuzzing with Input-to-State Correspondence. In *Network and Distributed Systems Security Symposium*, 2019.
- [12] V. Atlidakis, P. Godefroid, and M. Polishchuk. RESTler: Stateful REST API Fuzzing. In *Proceedings of the 41st International Conference on Software Engineering*, pages 748–758, 2019.
- [13] Vaggelis Atlidakis, Roxana Geambasu, Patrice Godefroid, Marina Polishchuk, and Baishakhi Ray. Pythia: Grammar-based Fuzzing of REST APIs with Coverage-guided Feedback and Learning-based Mutations. *arXiv preprint:2005.11498*, 2020.
- [14] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. Checking Security Properties of Cloud Service REST APIs. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification*, pages 387–397, 2020.
- [15] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. Synthesizing Program Input Grammars. *ACM SIGPLAN Notices*, 52(6):95–110, 2017.
- [16] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based Greybox Fuzzing as Markov Chain. In *ACM SIGSAC Conference on Computer and Communications Security*, pages 1032–1043, 2016.
- [17] Konstantin Böttinger, Patrice Godefroid, and Rishabh Singh. Deep Reinforcement Fuzzing. In *2018 IEEE Security and Privacy Workshops*, pages 116–122, 2018.
- [18] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation. *arXiv preprint:1406.1078*, 2014.
- [19] Mingi Cho, Seoyoung Kim, and Taekyoung Kwon. Intriguer: Field-Level Constraint Solving for Hybrid Fuzzing. In *ACM SIGSAC Conference on Computer and Communications Security*, pages 515–530, 2019.
- [20] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling. *arXiv preprint:1412.3555*, 2014.
- [21] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *arXiv preprint:1810.04805*, 2018.
- [22] Kyle Dewey, Jared Roesch, and Ben Hardekopf. Language Fuzzing Using Constraint Logic Programming. In *International Conference on Automated Software Engineering*, pages 725–730, 2014.
- [23] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. University of California, Irvine, 2000.
- [24] Shuitao Gan, Chao Zhang, Peng Chen, Bodong Zhao, Xiaojun Qin, Dong Wu, and Zuoning Chen. GREYONE: Data Flow Sensitive Fuzzing. In *USENIX Security Symposium*, pages 2577–2594, 2020.
- [25] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. CollaFL: Path sensitive fuzzing. In *Symposium on Security and Privacy*, pages 679–696, 2018.
- [26] Patrice Godefroid, Bo-Yuan Huang, and Marina Polishchuk. Intelligent REST API Data Fuzzing. In *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 725–736, 2020.
- [27] Patrice Godefroid, Adam Kiezun, and Michael Y Levin. Grammar-based Whitebox Fuzzing. In *ACM Sigplan Notices*, 2008.
- [28] Patrice Godefroid, Daniel Lehmann, and Marina Polishchuk. Differential Regression Testing for REST APIs. In *International Symposium on Software Testing and Analysis*, pages 312–323, 2020.
- [29] Patrice Godefroid, Hila Peleg, and Rishabh Singh. Learn&Fuzz: Machine Learning for Input Fuzzing. In *International Conference on Automated Software Engineering*, pages 50–59, 2017.
- [30] Samuel Groß. FuzzIL: Coverage Guided Fuzzing for JavaScript Engines. *Department of Informatics, Karlsruhe Institute of Technology*, 2018.
- [31] HyungSeok Han, DongHyeon Oh, and Sang Kil Cha. CodeAlchemist: Semantics-Aware Code Generation to Find Vulnerabilities in JavaScript Engines. In *Network and Distributed Systems Security Symposium*, 2019.
- [32] Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. *Neural computation*, 9(8):1735–1780, 1997.

- [33] Christian Holler, Kim Herzig, and Andreas Zeller. Fuzzing with Code Fragments. In *USENIX Security Symposium*, pages 445–458, 2012.
- [34] Heqing Huang, Peisen Yao, Rongxin Wu, Qingkai Shi, and Charles Zhang. Pangolin: Incremental Hybrid Fuzzing with Polyhedral Path Abstraction. In *Symposium on Security and Privacy*, pages 1613–1627, 2020.
- [35] Kyungtae Kim, Dae R Jeong, Chung Hwan Kim, Yeongjin Jang, Insik Shin, and Byoungyoung Lee. HFL: Hybrid Fuzzing on the Linux Kernel. In *Network and Distributed Systems Security Symposium*, 2020.
- [36] Suyoung Lee, HyungSeok Han, Sang Kil Cha, and Soeul Son. Montage: A Neural Network Language Model-Guided JavaScript Engine Fuzzer. In *USENIX Security Symposium*, pages 2613–2630, 2020.
- [37] Yuekang Li, Yinxing Xue, Hongxu Chen, Xiuheng Wu, Cen Zhang, Xiaofei Xie, Haijun Wang, and Yang Liu. Cerebro: Context-Aware Adaptive Fuzzing for Effective Vulnerability Detection. In *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 533–544, 2019.
- [38] Minh-Thang Luong, Hieu Pham, and Christopher D Manning. Effective Approaches to Attention-based Neural Machine Translation. *arXiv preprint:1508.04025*, 2015.
- [39] Chenyang Lyu, Shouling Ji, Yuwei Li, Junfeng Zhou, Jianhai Chen, Pan Zhou, and Jing Chen. SmartSeed: Smart Seed Generation for Efficient Fuzzing. *arXiv preprint:1807.02606*, 2018.
- [40] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. MOPT: Optimized Mutation Scheduling for Fuzzers. In *USENIX Security Symposium*, pages 1949–1966, 2019.
- [41] Ke Mao, Mark Harman, and Yue Jia. Sapienz: Multi-objective Automated Testing for Android Applications. In *International Symposium on Software Testing and Analysis*, pages 94–105, 2016.
- [42] Nicole Nichols, Mark Raugas, Robert Jasper, and Nathan Hilliard. Faster Fuzzing: Reinitialization with Deep Neural Models. *arXiv preprint:1711.02807*, 2017.
- [43] Mohit Rajpal, William Blum, and Rishabh Singh. Not All Bytes are Equal: Neural Byte Sieve for Fuzzing. *arXiv preprint:1711.04596*, 2017.
- [44] Dongdong She, Rahul Krishna, Lu Yan, Suman Jana, and Baishakhi Ray. MTFuzz: Fuzzing with a Multi-task Neural Network. In *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 737–749, 2020.
- [45] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. NEUZZ: Efficient Fuzzing with Neural Program Smoothing. In *Symposium on Security and Privacy*, pages 803–817, 2019.
- [46] Suphannee Sivakorn, George Argyros, Kexin Pei, Angelos D Keromytis, and Suman Jana. HVLearn: Automated Black-box Analysis of Hostname Verification in SSL/TLS Implementations. In *Symposium on Security and Privacy*, pages 521–538, 2017.
- [47] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *Network and Distributed Systems Security Symposium*, 2016.
- [48] Jinghan Wang, Chengyu Song, and Heng Yin. Reinforcement Learning-based Hierarchical Seed Scheduling for Greybox Fuzzing. In *Network and Distributed Systems Security Symposium*, 2021.
- [49] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Skyfire: Data-Driven Seed Generation for Fuzzing. In *Symposium on Security and Privacy*, pages 579–594, 2017.
- [50] Wei You, Xueqiang Wang, Shiqing Ma, Jianjun Huang, Xiangyu Zhang, XiaoFeng Wang, and Bin Liang. ProFuzzer: On-the-fly Input Type Probing for Better Zero-day Vulnerability Discovery. In *Symposium on Security and Privacy*, pages 769–786, 2019.
- [51] Tai Yue, Pengfei Wang, Yong Tang, Enze Wang, Bo Yu, Kai Lu, and Xu Zhou. EcoFuzz: Adaptive Energy-Saving Greybox Fuzzing as a Variant of the Adversarial Multi-Armed Bandit. In *USENIX Security Symposium*, pages 2307–2324, 2020.
- [52] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *USENIX Security Symposium*, pages 745–761, 2018.
- [53] Lei Zhao, Yue Duan, Heng Yin, and Jifeng Xuan. Send Hardest Problems My Way: Probabilistic Path Prioritization for Hybrid Fuzzing. In *Network and Distributed Systems Security Symposium*, 2019.
- [54] Xiaoqi Zhao, Haipeng Qu, Wenjie Lv, Shuo Li, and Jianliang Xu. MooFuzz: Many-Objective Optimization Seed Schedule for Fuzzer. *Mathematics*, 9(3):205, 2021.