Effective REST APIs Testing with Error Message Analysis

LIXIN XU, State Key Laboratory for Novel Software Technology, Nanjing University, China
HUAYAO WU, State Key Laboratory for Novel Software Technology, Nanjing University, China
ZHENYU PAN, State Key Laboratory for Novel Software Technology, Nanjing University, China
TONGTONG XU, Huawei, China
SHAOHUA WANG, Central University of Finance and Economics, China
XINTAO NIU, State Key Laboratory for Novel Software Technology, Nanjing University, China
CHANGHAI NIE, State Key Laboratory for Novel Software Technology, Nanjing University, China

REST APIs are essential in building modern enterprise systems, while effectively examining their behaviors remains challenging due to the difficulty in inferring constraints from the specifications. To generate valid test inputs for REST APIs, existing approaches are typically feedback-driven, leveraging HTTP status codes received to guide further test input generation. However, these approaches overlook the potentially valuable information described in error messages accompanying HTTP status codes, leading to inefficiencies in exploring the input space of REST APIs. In this paper, we propose EmRest, a black-box testing approach that leverages error message analysis to enhance both valid and exceptional test input generation for REST APIs. For each operation under test, EmRest first identifies all possible value assignment strategies for each of its input parameters. It then repeatedly applies combinatorial testing to sample test inputs based on these strategies, and statistically analyzes the error messages (of 400-range status code) received to infer and exclude invalid combinations of value assignment strategies (i.e., constraints of the input space). Additionally, EmRest seeks to mutate valid value assignment strategies that are finally identified to generate test inputs for exceptional testing. The error messages (of 500-range status code) received are categorized to identify bug-prone operations, for which more testing resources are allocated. Our experimental results on 16 real-world REST APIs demonstrates the effectiveness of EmRest. It achieves higher operation coverage than state-of-the-art approaches in 50% of APIs, and detects 226 unique bugs that cannot be found by the other approaches.

CCS Concepts: • **Software and its engineering → Software testing and debugging**.

Additional Key Words and Phrases: REST APIs, OpenAPI, Feedback-Driven Testing

ACM Reference Format:
Lixin Xu, Huayao Wu, Zhenyu Pan, Tongtong Xu, Shaohua Wang, Xintao Niu, and Changhai Nie. 2025. Effective REST APIs Testing with Error Message Analysis. 1, 1 (April 2025), 24 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Authors' Contact Information: Lixin Xu, lxxu@smail.nju.edu.cn, State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China; Huayao Wu, hywu@nju.edu.cn, State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China; Zhenyu Pan, , State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China; Tongtong Xu, , Huawei, Shenzhen, China; Shaohua Wang, , Central University of Finance and Economics, Beijing, China; Xintao Niu, , State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China; Changhai Nie, , State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China.

ACM XXXX-XXXX/2025/4-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Given the widespread adoption of REST APIs in building modern enterprise systems [26], there is clearly a need to thoroughly test the correctness of such APIs to ensure system quality. Currently, OpenAPI Specification (OAS) [8] is the most well-known standard in documenting REST APIs, in which the operations that can be performed, and their input parameters required are explicitly described. The availability of OAS has then motivated researchers to develop many black-box REST APIs testing approaches [14, 26, 35, 61]. However, effective testing of these APIs remains challenging, not only due to the huge input space to explore, but more critically because the complex *constraints* involved in REST APIs cannot be easily identified from the given OAS.

In order to cater for the constraints in REST APIs, some studies proposed to extend OAS, so that developers can use formal languages to describe constraints [17, 43]. However, their applications require manual efforts and are hard to automate. To automatically extract constraints from the available OAS (in which constraints are often described in a description field using natural language), some testing approaches, like RestCT [60] and NLPtoREST [32], attempt to use Natural Language Processing (NLP). However, these approaches usually rely on pre-defined syntactic patterns [44] for constraint extraction, which might be limited to certain language styles. Moreover, in practice, developers might only describe parameters, but not their constraints, in such description fields [11]; the content in such fields could also be incorrect or outdated due to the changes of the server. These could further hinder the application of such NLP-based approaches.

In contrast to extracting constraints directly from OASs, many current testing approaches [16, 33, 37, 40–42] rely on feedback mechanisms to dynamically account for the constraints when generating test inputs. Despite that the concrete mechanisms developed could vary (e.g., the producer-consumer relationship exploited in RESTler [16], the attention model trained in MINER [42], and the reinforcement learning utilized in ARAT-RL [33]), these approaches all share a similar idea to generate constraints-satisfying test inputs for REST APIs. That is, if an input parameter value assignment yields a 200-range HTTP status code (i.e., a valid test input), this assignment is more likely to be reused in future tests. Conversely, if the assignment yields a 400-range HTTP status code (i.e., a violation of constraints), its likelihood of being reused decreases.

Although these feedback-driven testing approaches can help reach more operations when testing REST APIs [35], they only rely on HTTP status codes to generate future test inputs, and consequently, might take substantial time costs to find constraints-satisfying ones. This is because for operations with multiple input parameters, not *all* parameters are relevant to the 400-range status code triggered; but due to the lack of detailed feedback, these approaches could only treat all parameter assignments as equally responsible for the constraint violation. Worse more, in practice, each operation usually has multiple constraints among different input parameters, and violating any one of them will result in the same 400-range status code. In this case, adjusting test inputs to satisfy a particular constraint might result in test inputs that unintentionally violate some previously satisfied constraints.

Moreover, in addition to the exploration of valid input spaces (and reveal bugs triggered by such inputs) of REST APIs, it is also important to examine the ability of REST APIs on handling input values that mismatch the definition in specification, namely, exceptional testing [20]. To this end, current testing approaches [33, 42, 58] typically choose to mutate valid value assignments, hoping to use potentially invalid values to trigger bugs relevant to parameter validation codes (in such cases, the server is expected to return 400-range status code, but a 500-range code is received). However, in current approaches, the mutation operators are usually used in a rather random way. This might lead to an insufficient testing of operations that are more vulnerable to parameter validation-related bugs, and thus reduce bug detection effectiveness.

In order to implement a more directed and effective test input space exploration of REST APIs, in this paper, we propose EmRest, a novel black-box approach that leverages information in *error messages* included in HTTP response bodies to guide test input generation. In the context of REST APIs, error messages of 400-range status code typically describe the exact errors triggered, which can thus be exploited to identify constraints between input parameters. While error messages of 500-range status code (also referred to as *bug messages* in this study) usually provide bug localization information (e.g., stack traces), which can aid in categorizing detected bug.

Specifically, EmRest includes an *Inferring Constraints* phase and an *Exceptional Testing* phase, which primarily utilizes error messages of 400-range and 500-range status codes, respectively. In the *Inferring Constraints* phase, EmRest seeks to gradually construct and explore the valid input space for each available operation (different operations are tested in an order that considers operation dependencies). For each given operation, EmRest first initializes its input space by identifying all possible value assignment strategies for every input parameter. Then, EmRest repeatedly applies combinatorial testing to sample test inputs from the valid input space constructed so far, hoping to systematically examine the interactions between parameters, and at the same time, statistically assess how different combinations of value assignment strategies contribute to the triggering of each distinct error message. Here, if a combination of value assignment strategies appears highly frequently in test inputs that yield the same error message, then EmRest will treat this combination as an *invalid* combination (i.e., a constraint of the input space), and thereby exclude it in future test inputs sampling iterations. After constructing and exploring the valid input space, EmRest then proceeds to the *Exceptional Testing* phase, where potentially invalid test inputs are generated to examine the exception handling ability of APIs. To achieve this, EmRest first mutates value assignment strategies sampled from the valid input space of each operation at an initial round. Then, EmRest categorizes 500-range error messages (i.e., bug messages) received to identify operations from which the largest number of unique bugs are triggered. For complex operations, there may exist multiple ways to violate specifications. These include applying different mutation operators on various parameters, each potentially associated with distinct parameter validation code and bug messages. Consequently, EmRest prioritizes testing resources towards operations more prone to parameter validation issues, aiming to uncover as many distinct parameter validation-related bugs as possible.

To evaluate the effectiveness of EmRest, we conducted experiments on 16 real-world REST APIs [33, 60], and compared EmRest against six State-Of-The-Art (SOTA) REST APIs testing approaches: ARAT-RL [33], EvoMaster [15], Morest [41], RestCT [60], Schemathesis [28], and MINER [42]. The results demonstrate the superiority of EmRest in operation coverage, code coverage, and bug detection. Specifically, EmRest is able to cover more operations and codes than all baselines in 50% and 44% of APIs, respectively. The effectiveness of EmRest is more evident for APIs with a large number of input parameters and complex constraints, for which 15 operations can be exclusively covered. EmRest is also effective in bug detection, triggering 64% more unique bugs than the baselines across all APIs. Notably, 226 unique bugs can only be detected by EmRest, accounting for 26% of all bugs detected by all approaches.

In summary, the main contributions that this paper makes are as follows:

- We propose EmRest, a novel black-box approach that leverages statistical analysis of error messages to generate valid and exceptional test inputs for effective testing of REST APIs.
- We introduce a method to analyze 400-range error messages to infer invalid combinations of value assignment strategies, enabling a gradual construction and exploration of the valid input space for each operation. We also analyze 500-range error messages to categorize bugs,

for which operations that are more vulnerable to parameter validation can be identified and more thoroughly tested.

- We conducted extensive experiments on 16 real-world REST APIs to evaluate EmRest. The experimental results demonstrate the effectiveness of EmRest over SOTA approaches.

```yaml
paths:
  /orders:
    post:
      parameters:
        - in: body
          name: body
          required: true
          schema:
            $ref: '#/definitions/CreateOrder'
      responses:
        201:
          schema:
            type: object
            properties:
              id:
                type: integer
  /customers:
    get:
      responses:
        200:
          schema:
            type: array
            items:
              $ref: '#/definitions/Customer'
  /customers/{id}:
    get:
      parameters:
        - name: id
          in: path
          type: integer

definitions:
  CreateOrder:
    type: object
    properties:
      id:
        required: true
        type: integer
      type:
        required: true
        type: string
        enum: ['standard', 'express']
      address:
        type: string
        description: Required if 'type' is 'standard'
      priority:
        type: string
        enum: ['low', 'high']
        description: Required if 'type' is 'express'
  Customer:
    type: object
    properties:
      id:
        type: integer
      regionId:
        type: integer
```

Listing 1. An Example of the OpenAPI Specification (OAS)

2 Background

REST APIs refer to APIs that adhere to the REST architecture style [24]. Such APIs expose internal data as a collection of web resources, each uniquely identified by a Uniform Resource Identifier (URI). Clients interact with these resources by sending HTTP requests, which must specify the resource URI, the HTTP method (e.g., POST, GET, PUT, and DELETE), and input parameter values. Currently, the OAS is the most widely used standard for documenting REST APIs. It provides a machine-readable format (typically in JSON [5] or YAML [7]) to describe all the necessary information required for performing API operations.

Listing 1, for example, gives an OAS snippet of a shopping system. The *paths* field gives the URIs of two resources, and the corresponding HTTP methods that can be performed. In REST APIs testing, the combination of an HTTP method and a URI is referred to as an *operation* (e.g., there are three operations, *POST:/orders*, *GET:/customers*, and *GET:/customers/{id}*, in this example). The execution order of such operations is critical, as it reflects operation dependencies. According to REST principles, resources are structured hierarchically, as indicated by the forward slashes ('/') in their URIs. The number of segments separated by slashes defines the URI's *depth*. Typically, accessing a child resource (with a higher URI depth, e.g., *GET:/customers/{id}*) depends on the availability of its parent resource (with a lower URI depth, e.g., *GET:/customers*). Furthermore, HTTP methods provide additional guidance on the execution order of different operations of the same resource. Since the POST method is used to create a resource, it is generally prioritized, as subsequent operations, such as GET, PUT, and DELETE, rely on that newly created resource.

To perform an operation, the client must also assign appropriate values to its input parameters based on their types, which are classified into six categories in the OAS: *integer*, *number*, *boolean*, *string*, *array*, and *object*. Among these, *integer*, *number*, *boolean*, and *string* are considered *basic types*, while *array* and *object* are *complex types* composed of multiple parameters. For example, in Listing 1, the *POST:/orders* operation has one object parameter, *CreateOrder* (as defined in the *definitions* field). This object contains one integer parameter, *id*, and three string parameters, *type*, *address*, and *priority*. In addition to input parameters, the OAS also specifies the format of responses for each operation. For example, if an execution of the *GET:/customers* operation returns a 200 status code, the response is expected to be an array of *Customer* objects, each of them contains two integer parameters, *id* and *regionId*.

Note that there are typically constraints that should be considered when determining input-parameter values of operations. Such constraints can stem from input parameters of a single operation, including value-specific rules to a specific parameter (e.g., the *type* parameter of *POST:/orders* can only take '*standard*' or '*express*' as its value), as well as inter-parameter dependencies [44] (e.g., for *POST:/orders*, the *address* parameter is required if the *type* is '*standard*'). The constraints can also stem from operation dependencies [16], e.g., for *POST:/orders*, the input parameter *id* must be assigned a valid *id* from responses of *GET:/customers*, which is not usually reflected through the URI hierarchy of resources [16].

When processing an HTTP request, the server returns a response that includes an HTTP status code and a response body (typically in JSON format). Status codes in the 200 range indicate a successful request, and the response body often contains details about the manipulated resource, e.g., *{"id": 10001, "regionId": '20010'}*. In contrast, status codes in the 400 and 500 ranges indicate request failures: A 400-range request fails when constraints between input parameter are violated; whereas a 500-range request fails because some internal bugs in the server prevent the request from being processed (i.e., a potential *bug*). In such cases, the response body typically contains error-related information (in string format). For example, when the *id* parameter of *POST:/orders* takes an invalid value, the 400-range error message might be: *{"message": "Invalid id: Must be a registered customer"}*; whereas a 500-range error message may include an error stack trace. According to the best practice of REST APIs development (e.g., Microsoft's Azure API guidelines [53]), modern REST APIs are expected to provide informative error messages that describe the potential causes of the failed requests. Some industry standards (e.g., RFC 7807 [49] and 9457 [50]) even force the inclusion of error messages in API responses. However, despite the widespread adoption of error messages in REST APIs, the meaningful diagnostic information in error messages is yet to be well exploited in current testing approaches. The aim of this study is exactly to leverage such error messages as a feedback mechanism to enhance REST APIs testing.

3 The EmRest Approach

In this study, we develop the EmRest approach to generate both valid and exceptional test inputs for REST APIs via analyzing error messages of both 400-range and 500-range status codes. Algorithm 1 gives the main algorithm of EmRest. Given an OAS as the input, EmRest first extracts all available operations O_{all}. Then, EmRest goes through an *Inferring Constraints* phase (lines 2–4) to construct and explore valid input space for every operation in O_{all}, allowing the evaluation of the APIs' behaviors under normal conditions. Here, EmRest especially categorizes O_{all} into two groups: O_{cur} (which contains POST, GET, and PUT operations) and O_{del} (which contains DELETE operations).

Algorithm 1 The Main Algorithm of EmRest

```
Require: the OpenAPI specification spec
 1: Extract operations $O_{all}$ from spec
 2: Categorize $O_{all}$ into $O_{cur}$ (POST, GET, and PUT operations) and $O_{del}$ (DELETE operations)
 3: InferringConstraints($O_{cur}$)
 4: InferringConstraints($O_{del}$)
 5: ExceptionalTesting($O_{all}$)
 6:
 7: function InferringConstraints($O$)                                        ▷ Section 4
 8:     $O$ ← Sort $O$ by URI hierarchy and HTTP method
 9:     while $O$ is not empty do
10:         op ← Select the head of $O$
11:         success ← SamplingAndAnalyzingErrorMessages(op)                    ▷ Algorithm 2
12:         if not success and no more than three attempts then
13:             Insert op to $O$
14:         end if
15:     end while
16: end function
17: function ExceptionalTesting($O$)                                          ▷ Section 5
18:     for op in $O$ do
19:         MutateAndCategorizingBugMessages(op)
20:     end for
21:     while within time budget do
22:         op ← Randomly select an operation from $O$ in proportion to the numbers of previously triggered unique bugs
23:         MutateAndCategorizingBugMessages(op)
24:     end while
25: end function
```

Operations in O_{cur} are tested earlier than those in O_{del} since DELETE operations remove resources, and these resources are often needed when testing operations in O_{cur}. After this, EmRest further goes through an *Exceptional Testing* phase (line 5), in which invalid inputs are deliberately introduced to test how well each operation in O_{all} handle exceptional scenarios.

The InferringConstraints(O) function (lines 7–16) describes the process to test each of the two operation sets, O_{cur} and O_{del} in the *Inferring Constraints* phase. To account for operation dependencies in REST style (as discussed in Section 2), EmRest sorts all operations in O based on URI hierarchies and CURD semantics (line 8). Here, for O_{cur}, operations with lower URI depths are executed earlier, ensuring that parent resources are created earlier than manipulating their child resources. While for O_{del}, operations with higher URI depths are executed earlier, ensuring that child resources are deleted before deleting their parent resources.

Once operations are sorted, EmRest repeatedly takes one operation *op* from O at a time for testing, and repeats until O is empty (lines 9–15). This is achieved through the SamplingAnd-AnalyzingErrorMessages(*op*) function (see Algorithm 2), in which EmRest repeatedly applies combinatorial testing to generate test inputs for *op*. During this process, EmRest manages to statistically analyze error messages of 400-range status code received to identify combinations of value assignments that frequently trigger error messages (i.e., *invalid* combinations for testing *op*), and thus exclude them from future iterations. For EmRest, an operation is considered *successfully tested* if at least one 200-range status code is observed in SamplingAndAnalyzingErrorMessages(*op*), namely, *success* = *true*. If *op* cannot be successfully tested and the number of retries does not exceed three, EmRest will add it back into O for further testing (line 13). Here, if *op* is a POST, GET, or PUT operation, it is inserted to the tail of O, and will thus be executed only when all preceding operations in O have been executed. This is because failures of non-DELETE operations are often caused by the lack of certain resources, and through testing such operations later, the resources required could be created by executing other operations (e.g., in Listing 1, the *id* parameter of *POST:/orders* is dependent on the response of *GET:/customers*, which is not reflected in the URI hierarchies of these two operations).

Once the valid input spaces of all operations are constructed, EmRest moves to the *Exceptional Testing* phase (line 5). In this phase, the MutatingAndCategoRizingBugMessages(*op*) function (see Algorithm 3) mutates the previously identified valid value assignments of each operation in O_{all}, hoping to generate potentially invalid inputs to test these operations' ability on handling exceptional scenarios. Specifically, EmRest first tests all operations in an initial round, and categorizes error messages of 500-range status code (i.e., bug messages) received to identify unique bugs triggered by each operation (lines 18–20). Then, as the inability to validate one parameter often suggests that the operation may also struggle to validate others, potentially leading to additional bugs with different causes and error messages, EmRest assigns each operation a selection probability proportional to its number of unique parameter validation-related bugs triggered before. In this way, the operations more prone to parameter validation issues are selected more frequently, allowing them to be tested more thoroughly with a more diverse set of value assignment mutations. (lines 21–24). The iteration process continues until the predefined time budget is exhausted.

Algorithm 2 SamplingAndAnalyzingErrorMessages(*op*)

```
Require: the operation to test, op
 1: Initialize input space $S = \{S_{p_1}, S_{p_2}, \ldots, S_{p_n}\}$ based on specification and historical responses    ▷ Section 4.1
 2: Initialize the set of invalid combinations of value assignment strategies $C = \emptyset$
 3: $\tau \leftarrow 1$, $LC \leftarrow 0$, $EC \leftarrow 0$
 4: while $LC \leq 10$ and $EC \leq 3$ do
 5:     $LC \leftarrow LC + 1$
 6:     Generate and execute a set of test inputs that cover all combinations of value assignment strategies of any $\tau$
        parameters based on $S$ and $C$    ▷ Section 4.2
 7:     Identify invalid combinations by statistically analyzing error messages, and update $C$ accordingly    ▷ Section 4.3
 8:     Increment $EC$ by one if the new error messages identified are identical to those from the previous iterations;
        otherwise, reset $EC$ to zero
 9:     Increment $\tau$ to two if $EC > 0$; otherwise, reset $\tau$ to one
10: end while
11: return true if at least one 200-range status code is observed
```

4 Sampling and Analyzing Error Messages

Algorithm 2 gives the implementation of the SamplingAndAnalyzingErrorMessages(*op*) function, which is designed to construct and explore the valid input space for the given operation *op*. Assume that *op* has *n* input parameters. EmRest first relies on the specification and historical HTTP responses to initialize a list of *value assignment strategies*, S_{p_i}, for each input parameter p_i ($1 \leq i \leq n$), where each strategy in S_{p_i} represents a distinct method for generating concrete values for p_i. The collection of value assignment strategy lists for all n input parameters, $S = \{S_{p_1}, S_{p_2}, \ldots, S_{p_n}\}$, exactly forms the initial *input space* of *op*. By selecting a value assignment strategy $s_{i,j} \in S_{p_i}$ for each input parameter p_i, we can sample an *abstract* test input, $t = \{s_{1,j_1}, s_{2,j_2}, \ldots, s_{n,j_n}\}$, from the input space. Then, by further applying the value assignment strategies specified in t, a *concrete* test input (i.e., an HTTP request) can be constructed and executed.

After the initial input space of *op* is constructed, EmRest then enters a loop (lines 4–10) to test *op*. At each iteration, EmRest first applies combinatorial testing [48] to sample a representative set of test inputs based on the value assignment strategy lists S and the set of constraints C obtained so far (initially, $C = \emptyset$), hoping to trigger diverse behaviors of *op*. Then, after transforming these abstract test inputs into concrete ones and executing them, EmRest collects all error messages of 400-range status codes received, and applies a statistical analysis method to identify invalid combinations of value assignment strategies (i.e., constraints) in S. By keeping adding these invalid combinations into C, EmRest can then avoid generating invalid test inputs in future iterations. The above loop repeats until either the maximum number of iterations LC is reached, or no new error messages are observed in the most recent EC iterations ($LC = 10$ and $EC = 3$ in this study).

4.1 Initializing Value Assignment Strategy List

For each operation, EmRest treats each parameter of basic types as a distinct input parameter. While for the parameter of *object* type, EmRest takes a fine-grained strategy that treats each of its leaf parameters (i.e., internal parameters located at the end of the nested structure of the object) as a distinct input parameter. Then, for each distinct input parameter p_i, EmRest follows the following four categories to determine its list of value assignment strategies, $S_{p_i} = \{s_{i,1}, s_{i,2}, \ldots, s_{i,m}\}$:

- **Null Value Assignment Strategy** (denoted as $NS()$): this strategy assigns a *NULL* value to the input parameter, indicating the parameter will not be included in the test input.
- **Fixed Value Assignment Strategy** (denoted as $FS(v)$): this kind of strategies assigns a single fixed value v that is considered important for testing to the parameter.
- **Random Value Assignment Strategy** (denoted as $RS(v)$): this kind of strategies generates a random value v that satisfies the parameter's type, format, and other specified requirements (e.g., maximum and minimum for *integer* parameters).
- **Resource-Based Value Assignment Strategy** (denoted as $RBS(op, field)$): this kind of strategies retrieves a runtime value from the *field* in the previously obtained HTTP responses of operation *op*, and assigns it to the input parameter.

For each p_i, if p_i is optional, a $NS()$ strategy is added to S_{p_i}. Next, if p_i has an *enum* field, EmRest adds one $FS()$ strategy for each of the enumerated values specified into S_{p_i}, and completes the initialization process (as these explicitly define all valid assignments for p_i). If no *enum* field is present in p_i, EmRest then extracts all fixed values from p_i's fields in the specification, and employs each value to define a $FS()$ strategy. Specifically, each individual value listed in *example* and *default* fields is extracted. Each character sequence enclosed in single or double quotes within the *description* field (as they are likely example values [32]) is also extracted. For parameters of *string*, *number*, and *integer* types, EmRest further defines $FS()$ strategies for boundary values: a zero-length string for *string* typed parameters, and the values 0, 1, and -1 for *integer* and *number* typed parameters.

Next, EmRest adds one $RS()$ into S_{p_i}. Note that it can be difficult to generate valid values for parameters of *string* type, as valid strings usually need to conform to specific formats. So, if a *string* typed parameter has no format specified, EmRest further follows the built-in string formats of OAS [1] to add three $RS()$ strategies into S_{p_i}: one that generates binary strings, another that generates base64-encoded byte strings, and a third that generates password strings (combinations of uppercase, lowercase, digits, and punctuation).

Then, EmRest adds a series of $RBS()$ strategies into S_{p_i}, in order to take operation dependencies into consideration (this kind of strategies will only be added when there are testing results). For this, EmRest maintains a pool of 200-range HTTP responses for each operation, each of which is associated with a response schema definition. Then, for each response pool, EmRest employs the *token_set_ratio* method of the *FuzzyWuzzy* library [3] to measure the token set similarity score $sim \in [0, 100]$ between the name of p_i and each response schema field. If sim exceeds 60 (which is a relatively low threshold, as we would like to include as many relevant fields as possible), EmRest collects all values under this field name from the response pool to construct a RBS strategy. This strategy will randomly select one of these collected values for assigning p_i.

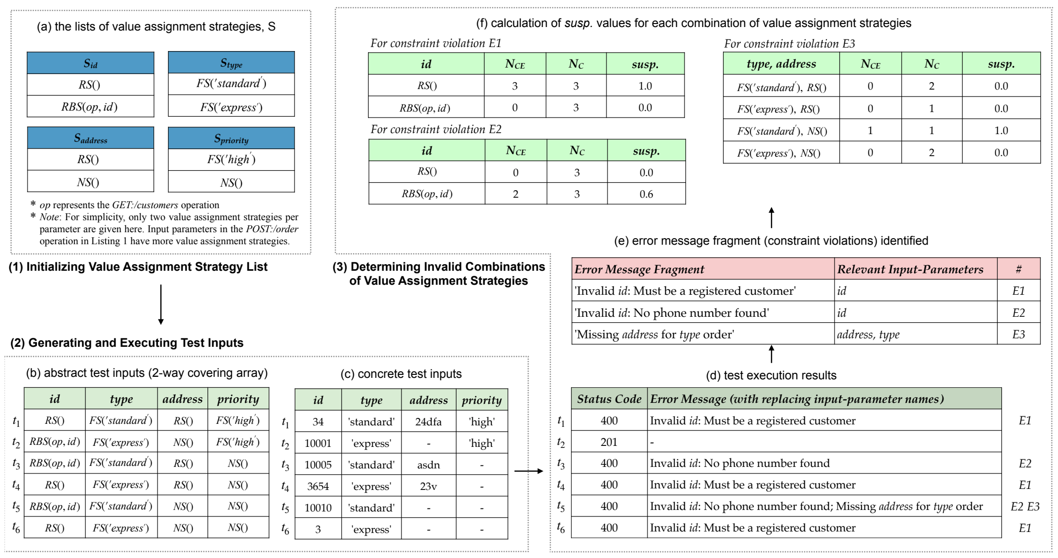
For example, in the *POST:/orders* operation from Listing 1, it has one *object* parameter, *CreateOder*, which expands into four distinct input parameters: *id*, *type*, *address*, and *priority*. For the input parameter *type*, since it has an *enum* field, we have S_{type} = {*FS*(‘*standard*’), *FS*(‘*express*’)}, which contains two fixed value assignment strategies only (as specified by the two values in the *enum* field). For the input parameter *address*, since it is optional, a null value assignment strategy, *NS*(), is added. Then, there are two character sequences ‘*type*’ and ‘*standard*’ enclosed in single quotes within its *description* field, two fixed value assignment strategies, *FS*(‘*type*’) and *FS*(‘*standard*’), are added. Furthermore, as a *string* parameter, another fixed value assignment strategy *FS*(‘’) is added (generating a zero-length string). Then, since no format is specified for this *string* parameter, four random value assignment strategies are initialized: *RS*(*string*), *RS*(*binary*), *RS*(*byte*), and *RS*(*password*). Now, suppose that the *GET:/customers* operation returns a response containing a *Customer* object with two output parameters: *id* and *regionId*. If *address* is determined to be sufficiently similar to these output parameter names, two resource-based value assignment strategies are added: *RBS*(‘*GET:/customers*’, ‘*id*’) and *RBS*(‘*GET:/customers*’, ‘*regionId*’). Finally, we have $S_{address}$ = {*NS()*, *FS*(‘*type*’), *FS*(‘*standard*’), *FS*(‘’), *RS*(*string*), *RS*(*binary*), *RS*(*byte*), *RS*(*password*), *RBS*(‘*GET:/customers*’, ‘*id*’), *RBS*(‘*GET:/customers*’, ‘*regionId*’)} as its value assignment strategy list.

4.2 Generating and Executing Test Inputs

At each iteration of Algorithm 2, EmRest applies combinatorial testing (specifically, the PICT [9] tool) to generate test inputs based on the value assignment strategy lists of the operation, S, and the collection of invalid combinations of value assignment strategies, C. Combinatorial testing is a popular test input space sampling technique [21, 55, 59, 60], which uses a τ-way *covering array* as the test suite to systematically cover all possible combinations of any τ parameter values. For each operation, EmRest first generates a 1-way covering array as the set of abstract test inputs, and then constructs one concrete test input for each abstract test input. This ensures that every value assignment strategy in S will be tested at least once, and accordingly, helps to collect error messages related to the violation of constraints on a single input parameter. Next, if the error message observed in this iteration is the same as that from the most recent previous round ($EC > 0$ in Algorithm 2), EmRest increases the value of τ to two, so the combinations between value assignment strategies of any two input parameters can be further tested. This helps not only detect interaction-related bugs in REST APIs, but also collect error messages related to violations of inter-parameter constraints

Fig.1 uses an example to illustrate the workflow of EmRest for testing the *POST:/orders* operation in Listing 1 (with $\tau = 2$). For simplicity, we consider a reduced list of value assignment strategies for each of the four input parameters, *id*, *type*, *address*, and *priority* (with $|S_{id}| = |S_{type}| = |S_{address}| = |S_{priority}| = 2$), as shown in Fig.1 (a). Fig 1 (b) gives a 2-way covering array generated (without constraints), where each row indicates an abstract test input. Note that these six test inputs cover all possible combinations of any two parameters (e.g., for *id* and *type*, all $2\times2 = 4$ strategy combinations appear at least once in the array). Then, by applying value assignment strategies specified in each abstract test input, one concrete test input is generated, as shown in Fig. 1 (c). After executing these concrete test inputs, the HTTP responses, including status codes and error messages, as shown in Fig 1 (d), are collected for further analysis.

Note that prior studies [36, 48] indicate that most software failures arise from combinations of a few parameters, with 2-way combinations accounting for the majority. Additionally, a large-scale analysis of REST APIs [44] found that only 14% of cases involve inter-parameter constraints with more than two parameters. While for complex operations, a large number of input parameters and *object*-typed parameters, can lead to a huge input space, prior studies [38, 39] have demonstrated that small-sized 2-way covering arrays can be efficiently generated, even for systems with hundreds of parameters. As EmRest uses up to 2-way covering arrays, its application can be scale to complex operations with a large number of input parameters.



(1) Initializing Value Assignment Strategy List

(a) the lists of value assignment strategies, S

S_{id}
RS()
RBS(*op*, *id*)

S_{type}
FS('*standard*')
FS('*express*')

$S_{address}$
RS()
NS()

$S_{priority}$
FS('*high*')
NS()

* *op* represents the *GET:/customers* operation
* *Note*: For simplicity, only two value assignment strategies per parameter are given here. Input parameters in the *POST:/order* operation in Listing 1 have more value assignment strategies.

(2) Generating and Executing Test Inputs

(b) abstract test inputs (2-way covering array)

	id	*type*	*address*	*priority*
t_1	*RS*()	*FS*('*standard*')	*RS*()	*FS*('*high*')
t_2	*RBS*(*op*, *id*)	*FS*('*express*')	*NS*()	*FS*('*high*')
t_3	*RBS*(*op*, *id*)	*FS*('*standard*')	*RS*()	*NS*()
t_4	*RS*()	*FS*('*express*')	*RS*()	*NS*()
t_5	*RBS*(*op*, *id*)	*FS*('*standard*')	*NS*()	*NS*()
t_6	*RS*()	*FS*('*express*')	*NS*()	*NS*()

(c) concrete test inputs

	id	*type*	*address*	*priority*
t_1	34	'standard'	24dfa	'high'
t_2	10001	'express'	-	'high'
t_3	10005	'standard'	asdn	-
t_4	3654	'express'	23v	-
t_5	10010	'standard'	-	-
t_6	3	'express'	-	-

(3) Determining Invalid Combinations of Value Assignment Strategies

(d) test execution results

	Status Code	Error Message (with replacing input-parameter names)	
t_1	400	Invalid *id*: Must be a registered customer	E1
t_2	201	-	
t_3	400	Invalid *id*: No phone number found	E2
t_4	400	Invalid *id*: Must be a registered customer	E1
t_5	400	Invalid *id*: No phone number found; Missing *address* for *type* order	E2 E3
t_6	400	Invalid *id*: Must be a registered customer	E1

(e) error message fragment (constraint violations) identified

Error Message Fragment	*Relevant Input-Parameters*	#
'Invalid *id*: Must be a registered customer'	*id*	*E1*
'Invalid *id*: No phone number found'	*id*	*E2*
'Missing *address* for *type* order'	*address, type*	*E3*

(f) calculation of *susp.* values for each combination of value assignment strategies

For constraint violation E1

id	N_{CE}	N_C	*susp.*
RS()	3	3	1.0
RBS(*op*, *id*)	0	3	0.0

For constraint violation E2

id	N_{CE}	N_C	*susp.*
RS()	0	3	0.0
RBS(*op*, *id*)	2	3	0.6

For constraint violation E3

type, address	N_{CE}	N_C	*susp.*
FS('*standard*'), *RS*()	0	2	0.0
FS('*express*'), *RS*()	0	1	0.0
FS('*standard*'), *NS*()	1	1	1.0
FS('*express*'), *NS*()	0	2	0.0

Fig. 1. The workflow of *Sampling and Analyzing Error Messages* in EmRest.

4.3 Determining Invalid Combinations of Value Assignment Strategies

Once the concrete test inputs are executed and the testing results are obtained, EmRest applies statistical analysis to infer constraints from the received 400-range error messages. For EmRest, a *constraint* in the input space S is expressed as a combination of k value assignment strategies (selected from k different strategy lists), denoted as $\{p_{i_1} = s_{i_1,j_1}, p_{i_2} = s_{i_2,j_2}, \ldots, p_{i_k} = s_{i_k,j_k}\}$ where $s_{i,j}$ is the j-th strategy selected from S_{p_i}. Such a combination is also referred to as an *invalid combination of value assignment strategies*, as its application in generating concrete values will produce 400-range status codes. For example, for the *POST:/orders* operation in Listing 1, {*type* = *FS*('*standard*'), *address* = *NS()* } is an invalid combination, as the OAS specifies that *address* is required if *type* is '*standard*'.

4.3.1 Identifying Error Message Fragments. Given a 400-range HTTP response body and its corresponding concrete test input, EmRest extracts each field value from the response body (except for timestamp, status code, path, and URI that are unlikely to contain error related information) as a single error message (a string of text). Then, EmRest relies on the concrete input parameter values found in each error message to associate it with its relevant input parameters. On this basis, EmRest further replaces each concrete value with its corresponding parameter name to distinguish distinct errors. For example, in the *POST:/orders* operation from Listing 1, if *id* is assigned an invalid value, such as 34, the server may return the error message: "Invalid 34: Must be a registered customer". EmRest replaces 34 with the corresponding parameter name, transforming the message into: "Invalid *id*: Must be a registered customer." As such, error messages describing the same error can be categorized (e.g., assigning either 34 or 35 to *id* triggers the same error, even though the original error messages received are different). For simplicity, the error messages displayed in Fig. 1 (d) are those after replacing concrete input parameter values.

Note that an error message might describe multiple constraint violations when the test input violates multiple constraints. So, to track each distinct violation, EmRest further identifies *error message fragments*, where each fragment is a snippet text of the error message that describes a single constraint violation and does not contain any other error message fragment. Specifically, given a set of previously identified error message fragments (initially empty when EmRest starts) and a set of newly observed error messages, EmRest first treats each new error message as a new error message fragment. Then, EmRest iteratively compares every pair of fragments to determine whether one fragment contains another, and keeps splitting and replacing the longer fragment with the shorter one. This process repeats until no two fragments contain each other. At this moment, all fragments remaining are error message fragments.

For example, in Fig. 1 (d), EmRest initially collects a set of three error messages: "Invalid *id*: Must be a registered customer." (from t_1, t_4, and t_6), "Invalid *id*: No phone number found." (from t_3), and "Invalid *id*: No phone number found; Missing *address* for *type* order" (from t_5). Since the third message contains the second, so EmRest splits the third into two messages: "Invalid *id*: No phone number found" and "Missing *address* for *type* order". After removing the original third message and add the two newly identified messages, EmRest confirms the final three messages as the three error message fragments identified (because they do not contain each other). Fig. 1 (e) shows these three identified error message fragments, E_1, E_2, and E_3, and their relevant input parameters.

4.3.2 Determining Invalid Combinations. After identifying the set of error message fragments (i.e., distinct constraint violations), EmRest determines the invalid combinations of value assignment strategies that contribute to each constraint violation. To achieve this, a straightforward approach is to directly mark the combinations used in the test inputs that trigger any error message fragment as invalid. However, this tends to overestimate constraints in the input space, as some value assignment strategies do not always produce values that violate constraints. For example, in Fig. 1 (b) and (d), the *RBS*(*op*, *id*) strategy for parameter *id* is only invalid when the retrieved customer lacks a phone number. Simply discarding this strategy may overly restrict the input space, and if it's the only strategy producing valid values, *id* may fail to obtain a valid value in further testing.

To accurately infer constraints, EmRest adopts a statistical approach to assess the likelihood of each combination of value assignment strategies triggering each error message fragment. The core idea behind this step is that if the use of a certain combination of value assignment strategies causes test inputs triggering the same error message more frequently, then this combination is more likely to be invalid. For each error message fragment E, EmRest first identifies its relevant input parameters and collects all combinations of value assignment strategies used for these parameters across the set of test inputs. Then, for each strategy combination c, EmRest calculates the number of test inputs where c is used, denoted as N_C, and the number of times the use of c leads to E, denoted as N_{CE} (all previously executed test inputs are also considered in this step to enable a more accurate estimation) Finally, the likelihood of strategy combination c triggering error message fragment E is computed as $susp. = N_{CE}/N_C$. If the suspicious value *susp.* exceeds a predefined threshold, then c is identified as an invalid combination of value assignment strategies.

For example, Fig. 1 (f) illustrates the calculation of *susp.* values for the three error message fragments. For E_1 and E_2, they are relevant to parameter *id* only, which corresponds to two strategy combinations[1] for calculation. While for E_3, it is relevant to two parameters, and there are thus four strategy combinations for calculation (note that these four combinations ensure to appear in Fig. 1 (b), because a 2-way covering array is used). Let us then take E_3 as the example. For the first strategy combination, {*address* = *RS*(), *type* = *FS('standard')*}, we have $N_C = 2$ because this

[1]For consistency, in this study, we refer to each single value assignment strategy as also a strategy *combination* (i.e., 1-way combination [48]).

combination is used twice in Fig. 1 (b) (t_1 and t_3); and $N_{CE} = 0$ because none of these uses results in error message fragment E_3 (as t_1 and t_3 trigger E_1 and E_2, respectively). As such, the *susp.* value of this combination is as $0/2 = 0$. Similarly, we can calculate the *susp.* value of $\{id = RS()\}$ for E_1 as $3/3 = 1$ (all three uses of this strategy, in t_1, t_4, and t_6, result in E_1). At this moment, if the threshold is set to 0.7, then $\{id = RS()\}$ will be identified as an invalid combination (i.e., a constraint).

Note that to ensure the accuracy of identifying invalid combinations, the threshold should be close to 1.0, maximizing confidence that the identified combinations are truly invalid. However, in practice, some errors may mask others, meaning that even if a test input contains two invalid combinations, the received error message may include only one error message fragment. In such cases, a relatively low threshold can help identify invalid combinations related to the masked error, as N_{CE} tends to be small. In this study, we set the threshold to 0.7 to balance accuracy and efficiency in identifying invalid strategy combinations.

Algorithm 3 MutateAndCategorizingBugMessages(*op*)

```
Require: the operation to mutate op
 1: Let S and C be the value assignment strategy lists and constraints obtained at the end of Inferring Constraints phase
 2: $LC \leftarrow 0$, $BC \leftarrow 0$
 3: while $LC \leq 10$ and $BC \leq 3$ and within time budget do
 4:     $LC \leftarrow LC + 1$
 5:     Generate a set of abstract test inputs $T_a$ based on S and C
 6:     for each abstract test input t in $T_a$ do
 7:         if the content type of t is not null and random.uniform(0, 1) < 0.5 then
 8:             Apply Content Type Mutation operator to the content type of t
 9:         end if
10:         for each input parameter p in t do
11:             if random.uniform(0, 1) < 0.5 then
12:                 Apply Input Type Mutation operator to replace value assignment strategy of p to an exceptional strategy
13:             end if
14:         end for
15:         Construct and execute a concrete test input of t
16:     end for
17:     Collect and categorize bug messages
18:     Increment BC by one if the newly identified error messages are identical to those from the previous iteration;
    otherwise, set BC to zero
19: end while
```

5 Mutating and Categorizing Bug Messages

Algorithm 3 outlines the implementation of the MutatingAndCategorizingBugMessages(*op*) function, which aims to generate invalid test inputs to uncover as many parameter validation-related bugs as possible (i.e., exceptional testing [33, 58]). To this end, EmRest first reuses all lists of value assignment strategies S and the set of constraints C obtained at the end of the *Inferring Constraints* phase as the valid input space of *op*. Then, similar to the SamplingAndAnalyzingErrorMessages(*op*) function, EmRest repeatedly applies combinatorial testing to sample a series of valid abstract test inputs, but here, these abstract test inputs will be further mutated before constructing concrete test inputs (line 5–16).

Specifically, for each constraints-satisfying abstract test input t, EmRest applies two kinds of mutation operators to generate exceptional test input. First, if t has a request body, EmRest applies the *Content Type Mutation* operator (with probability 50%) to randomly change its content type into a different choice, as suggested in [33] (lines 7–9). Second, for each input parameter p in t, EmRest applies the *Input Type Mutation* operator (with possibility 50%) to randomly assign p an exceptional strategy that generates random values of an incompatible type (e.g., a *string* value for an *integer* parameter). To this end, EmRest maintains a set of random value assignment strategies as candidate exceptional strategies, each corresponding to a unique parameter type defined by OAS. When the *Input Type Mutation* operator is applied to parameter p, EmRest randomly selects one strategy from these candidates that does not match p's type as its exceptional strategy. To encourage diverse and thorough testing of API input validation, EmRest initially assigns each exceptional strategy a weight of 1.0 per parameter. Once an exceptional strategy is selected, its weight is reduced by a factor of 0.9, thus increasing the chance of selecting other strategies to trigger different parameter validation-related bugs. During the execution of the generated concrete test inputs, EmRest continuously collects error messages of 500-range status code (i.e., bug messages), and applies the same fragment identification process (see Section 4.3.1) to identify distinct bug message fragments. This sampling and mutation process repeats until either a predefined iteration limit LC is reached, or no new bug message fragments appear in the most recent BC iterations (LC = 10 and BC = 3 in this study).

6 Experiment

This section presents the experiment conducted to assess the effectiveness and efficiency of EmRest. In particular, we focus on the following three research questions:

RQ1: How effective is EmRest in achieving operation and code coverage for REST APIs?
RQ2: How effective is EmRest in detecting bugs in REST APIs?
RQ3: How does the operation selection strategies impact the effectiveness of EmRest?

6.1 Experiment Setup

Table 1. The Subject APIs

REST APIs (*abbr*)	Description	# Operations	# Input Parameters
Features Service (*FeatSrv*)	Manages product features for product-related functionality.	18	1.9
Genome Nexus (*GenomeNex*)	Provides tools for annotating and interpreting genetic variants in cancer.	23	2.2
Language Tool (*LangTool*)	Open-source proofreading tool that supports multiple languages.	2	5.5
Market Service (*MktSrv*)	A project focused on Spring and web technologies for backend development.	13	3.6
Person Controller (*PersCtrl*)	Java project using MongoDB for easy data management.	12	5.0
Project Tracking System (*ProjTrack*)	Software for managing and tracking project progress.	60	5.2
REST Countries (*RestCtry*)	Get information about countries.	22	1.5
User Management Microservice (*UserMgmt*)	Manages user data for online systems.	22	2.9
NCS (*NCS*)	An artificial software for mathematical and statistical computations.	6	2.3
SCS (*SCS*)	Tool for calculations and text data processing.	11	2.4
GitLab Branch (*GLBranch*)	Manages branches in Git repositories.	9	10.7
GitLab Commit (*GLCommit*)	Tracks commits in Git repositories.	15	10.0
GitLab Groups (*GLGroups*)	Manages teams within GitLab.	17	8.9
GitLab Issues (*GLIssues*)	Tracks project issues and bug reports.	27	10.1
GitLab Project (*GLProject*)	Manages GitLab projects throughout their lifecycle.	31	10.0
GitLab Repository (*GLRepo*)	Manages data and versions in Git repositories.	10	10.9

6.1.1 Subject APIs. In this study, we used a total of 16 representative real-world REST APIs as the experimental subjects. Table 1 gives their names, descriptions, number of operations and average number of input parameters per operation. Specifically, the first ten APIs come from a SOTA REST API testing approach, ARAT-RL [33]. According to [33], these APIs are selected from a previous study [35] that includes a diverse set of 20 APIs. However, since some of these APIs require domain-specific knowledge, or rely on outdated external dependencies for execution, they are excluded in [33]. The study [33] also excludes APIs that have authentication issues, or restricted rate limits, as these might influence the evaluation of automated test inputs generation approaches. Finally, there are ten APIs used in [33]. To enable a fair comparison, we directly used the same specifications of these ten APIs in our experiments.

In addition, to cover more complex APIs in our experiments, we further included six APIs from GitLab, an enterprise level Git repository management system. GitLab APIs provide operations on core version control and project management resources, including branches, projects, groups, issues, commits, and repositories. These APIs are also widely used in recently REST APIs testing studies [16, 40, 42, 60]. From Table 1, we can see that the GitLab APIs generally involve more input parameters per operation than the first ten APIs, thus covering more complex test scenarios for evaluating constraints inference and input space exploration. We obtained the specifications of GitLab APIs from a previous study [60]. Given the fact that these specifications were manually created in [60], we further compared these specifications with GitLab's official documentation (v14.4.2, which is a more recent version) for validation, and made corrections when necessary. We provide our revised versions of these specifications in the supplementary data of this paper.

6.1.2 Evaluation Baselines. In this study, we selected six SOTA REST APIs testing approaches, ARAT-RL [33], EvoMaster (black-box version) [15], Morest [41], RestCT [60], Schemathesis [28], and MINER [42], as the baselines for comparison. Specifically, we first identified an initial set of ten approaches from a recent empirical study [35], and selected the top three black-box testing approaches in terms of code coverage and bug detection: EvoMaster [15], RESTler [16], and Schemathesis [29]. Then, to ensure a more comprehensive evaluation, we further identified four SOTA testing approaches that were more recently proposed: ARAT-RL [33], MINER [42], Morest [41], and RestCT [60]. These approaches were included to account for the latest advancements in REST APIs testing. We note that the MINER [42] approach is built upon RESTler [16], following the same main testing process while incorporating an attention model for exploring longer sequences with optimized value assignments. The testing effectiveness of RESTler can be inherently evaluated by evaluating MINER. Moreover, both MINER and ARAT-RL consistently outperform RESTler in all studied cases in prior studies [42] and [33], respectively. Therefore, we excluded RESTler from our experiments to avoid redundancy.

To further investigate the contribution of different components involved in EmRest, we further compared EmRest with its three variants, EmRest$_{Infer}$, EmRest$_{Random}$, and EmRest$_{NoRetry}$. In particular, EmRest$_{Infer}$ removes the *Exceptional Testing* phase, aiming to evaluate the impact of *Inferring Constraints* and *Exceptional Testing* phases on the overall performance of EmRest (*RQ1* and *RQ2*). EmRest$_{Random}$ modifies the operation selection strategy in the *Inferring Constraints* phase: instead of categorizing operations into O_{cur} and O_{del} and selecting operations based on URL hierarchy and HTTP method, it randomly selects operations from O_{all} (line 10 in Algorithm 1). Lastly, EmRest$_{NoRetry}$ limits that an operation is only tested once in the *Inferring Constraints* phase, that is, if the operation fails to obtain 200-range status code, it will not be inserted back to O and re-tested (line 13 in Algorithm 1). EmRest$_{Random}$ and EmRest$_{NoRetry}$ are designed to investigate the impact of operation selection strategies on the overall performance of EmRest (*RQ3*).

6.1.3 Experimental Procedure. For each subject API, we ran each testing approach and each variant of EmRest for one hour. This time budget is suggested by a recent study [35], which indicates that REST APIs testing approaches typically plateau in performance within one hour, with code coverage after one hour being comparable to that achieved after 24 hours. This one-hour budget also aligns with the experimental setup used in another recent study [33]. To account for the inherent randomness in all testing approaches, we repeated the testing process 30 times per subject. Before each run, we deleted and redeployed the subject API to ensure a clean and consistent testing environment. The experiments were conducted on a server equipped with an Intel(R) Xeon(R) Gold 5118 CPU @ 2.30GHz (48 cores) and 128GB of memory, running CentOS 7.

For data collection, we relied on Mitmproxy [6] to capture all HTTP requests generated and responses received during the entire testing process for each subject API. To collect code coverage data, we used JaCoCo [4] for the first ten subject APIs, as they are written in Java. For the GitLab APIs, we used a Docker image of GitLab with built-in code coverage support, as provided in [40].

Based on the raw data obtained, we calculated and compared *operation coverage*, *code coverage*, and *bug detection*, achieved by the different testing approaches. In this study, an operation is considered *covered* if at least one 200-range HTTP response is received [45], and we measure the number of covered operations for each approach. We measure the percentage of lines of code exercised by the test inputs generated by each approach. Regarding bug detection ability, we measure the number of unique bugs triggered by each approach. Since the same 500-range status code can result from different bugs, we follow a previous study [33] to categorize unique bugs based on the response messages received for each operation. Specifically, we first remove unrelated components such as timestamps from the response and then define a *unique* bug as a combination of an operation and a distinct response message instance.

6.2 Experiment Results

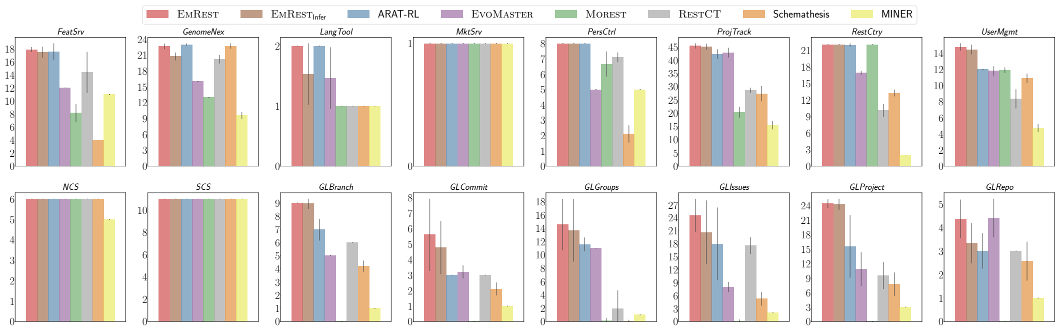


Fig. 2. Number of Operations Covered By Different Testing Approaches

6.2.1 RQ1 (Operation and Code Coverage). Fig. 2 and Fig. 3 show the average number of operations covered and line coverage achieved by different testing approaches across 30 runs, respectively. Each subplot here corresponds to a subject API, with error bars indicating standard deviation observed (a shorter bar indicates a more stable performance).

From Fig. 2, EmRest is generally the best approach in covering operations, followed by ARAT-RL, EvoMaster, RestCT, and Schemathesis. In particular, EmRest is able to cover more operations than all six baselines in eight out of 16 APIs, and in six APIs EmRest and the best baseline achieve the same coverage. Even if there are two APIs (*GenomeNex* and *GLRepo*) in which EmRest performs worse, the difference in the average number of operations covered by EmRest and the respective best baseline does not exceed 0.33. Especially, the advantage of EmRest is more evident for GitLab APIs, as for *GLProject*, it can cover up to 8.87 more operations than ARAT-RL. Further, for these APIs, some approaches like MoRest and MINER can only cover up to 22.9% of operations covered by EmRest (though these approaches often perform well for the first ten APIs). This indicate their potential shortcomings in generating valid test inputs that satisfy complex constraints.

Overall, across all APIs, EmRest covers a total of 30.3 more operations than the best baseline, ARAT-RL. The set of operations covered by EmRest is generally a superset of operations covered by other baselines (there are only four operations that other baselines can cover but not EmRest). EmRest especially exhibits a stable performance for the first ten APIs, as its average standard deviation is only 0.2. While for the more complex GitLab APIs, this value can increase to 1.9, indicating a greater impact of the randomness (in such cases, ARAT-RL yields a larger standard deviation, 2.9). In addition, regarding EmRest and EmRest$_{Infer}$ (without *Exceptional Testing* phase), these two approaches achieve nearly identical performance in covering operations for most APIs.

This is because the goal of *Exceptional Testing* is to use potentially invalid test inputs to examine the parameter validation ability of APIs. Since such test inputs are generated to trigger 500-range status codes (not 200-range status codes), they tend to have litter contribution to operation coverage.

Moreover, for all operations covered in all APIs, there are 15 operations (one in each of *ProjTrack* and *GLGroups*, three in *UserMgmt*, and ten in *GLCommit*) that can only be covered by EmRest. By analyzing these operations, we find that they typically contain multiple interleaving constraints, and these constraints are usually absent in the OAS. For example, for *UserMgmt*, the *POST:/users* operation has 21 input-parameters with many constraints involved: four parameters are mandatory and should always be included; *gender* can only be *male* or *female*; *username* and *email* must be unique; and *password* must conform to a specific format. Existing approaches might have the ability to find valid values for each specific parameter (e.g., relying on responses to find a valid *email*, or using a special strategy to generate a valid *password*). But, with the increased number and complexity of parameters, they tend to fail to cope with the multiple constraints together; even if the status codes-based feedback can help adjust value assignments, however, modifying one parameter value to satisfy one constraint might result in a test input that violates another constraint. In contrast, EmRest utilizes a statistical analysis method to infer invalid combinations from error messages. It does not depend on the availability of constraint descriptions in the OAS, and is able to identify multiple invalid combinations from the given constraint violation (see Section 4.3). Accordingly, EmRest tends to exhibit a better performance for such complex APIs.

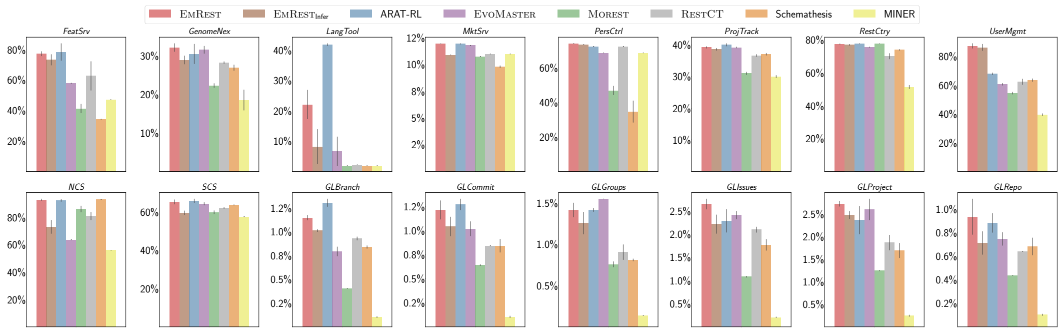


Fig. 3. Line Coverage Achieved By Different Testing Approaches

As far as the code coverage is concerned, from Fig. 3, EmRest achieves the highest line coverage in seven out of 16 subject APIs (with an average improvement of 10.5%). However, for the remaining eight APIs, EmRest performs slightly worse than the baselines, especially when comparing with ARAT-RL for *LangTool* (the two approaches achieve the same operation coverage in this case, as shown in Fig. 2). This is because for the *POST:/check* operation of *LangTool*, the different valid values assigned to the *language* parameter usually lead to different code execution paths, and there are some particular values, like '*ga*', resulting in the execution of more codes. In this case, the value re-use strategy in ARAT-RL tends to reuse some particular values multiple times, while EmRest tends to assign other valid values. Nevertheless, EmRest's stronger ability to cover operations remains beneficial for code coverage. In particular, for *UserMgmt* (where EmRest covers 2.8 more operations than ARAT-RL), EmRest can cover up to 28.7% more lines than ARAT-RL.

Answer to RQ1: EmRest is the most effective approach for covering operations, as it covers more operations than all six baselines in eight out of 16 APIs, and can cover up to 8.87 more operations for *GLProject*, a more complex GitLab API. Moreover, across all subjects, there are 15 operations that can only be covered by EmRest.

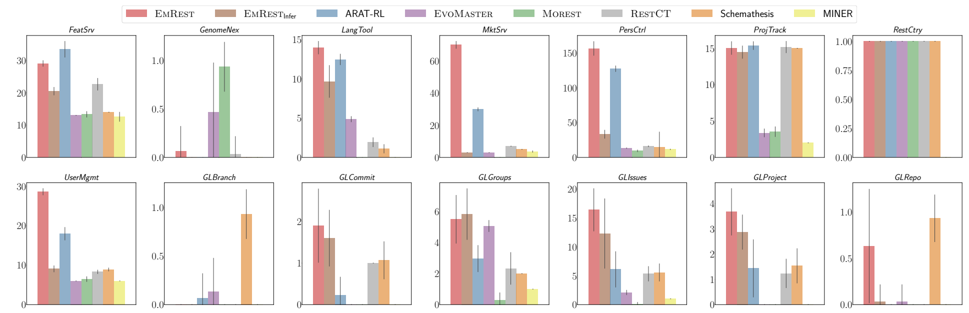


Fig. 4. Number of Unique Bugs Detected by Different Approaches (no bugs were detected in *NCS* and *SCS*)

6.2.2 RQ2 (Bugs Detection). Fig. 4 presents the average number of unique bugs detected by different approaches across 30 runs. Similar to Fig. 2 and Fig. 3, each subplot corresponds to a subject API, with error bars indicating standard deviation observed, Two APIs, *SCS* and *NCS*, are excluded here, because no bugs were detected (these two APIs are artificially developed to evaluate code coverage of unit testing). From Fig. 4, EmRest detects more unique bugs than all six baselines in eight out of 16 APIs (performs no worse in 11 APIs). The most notable improvement observed is in *MktSrv*, where EmRest detects, on average, 40 more unique bugs than the second-best approach, ARAT-RL (an improvement of 134%). Generally, ARAT-RL is the best baseline identified, followed by RestCT, Schemathesis, and MoRest, While operation coverage does not strongly correlate with bug detection, the standard deviation follows a similar trend, exhibiting stable performance in the first ten APIs and a relatively higher deviation in GitLab APIs.

In addition, from Fig. 4, EmRest detects significantly more unique bugs than EmRest$_{Infer}$, especially in the first ten APIs. This is mainly due to the critical role of *Exceptional Testing* in exposing parameter validation-related bugs, which account for the majority of bugs detected in the first ten APIs (as most of these APIs were particularly created as benchmarks, where parameter validation was not a core requirement). However, for the more mature enterprise system, GitLab APIs, the parameter validation bugs are rare and most bugs stem from business logic errors. As *Exceptional Testing* contributes little in these cases, EmRest and EmRest$_{Infer}$ show similar bug detection ability. Nevertheless, EmRest still outperforms all baselines in bug detection for GitLab APIs, primarily due to its use of combinatorial testing, which systematically examines inter-parameter interactions (e.g., for the *GET:/groups* operation in *GLGroups*, a bug occurs only when *per_page* > 0, *statistics* = *true*, and *min_access_level* is used).

Fig. 5 further visualizes the overlap of all unique bugs detected by all approaches in all 30 runs. Here, the leftmost horizontal bar shows the total number of unique bugs that each approach detects (e.g., EmRest detects 523 unique bugs in total). The upper vertical bar shows the number of unique bugs that are exclusively detected by a set of approaches, each of which is marked by a black dot in the figure (e.g. in the eighth column, EmRest and ARAT-RL are marked by dots, indicating that there are 125 bugs that can only be detected by these two approaches). From Fig. 5, EmRest identifies 523 unique bugs in total, accounting for 60% of all 867 bugs detected in the whole experiments.

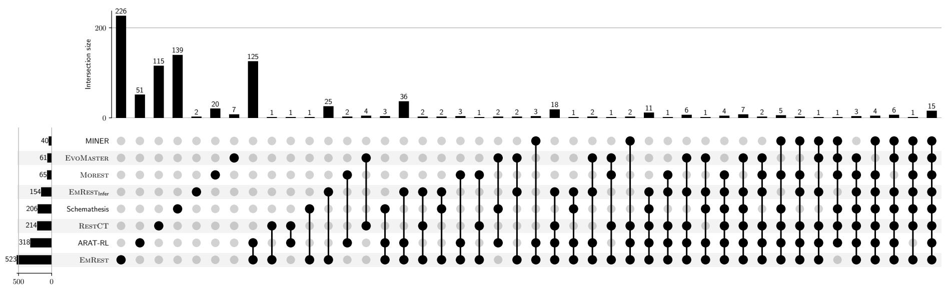


Fig. 5. Overlap of All Detected Unique Bugs Across 30 Runs

Notably, there are 226 bugs that can only be detected by EmRest, whereas the second and third best approaches, Schemathesis and RestCT, can only exclusively detect 139 and 115 unique bugs, respectively (though ARAT-RL detects more bugs than RestCTand Schemathesis, most of these bugs can be detected by the other approaches).

By further analyzing the bugs that are exclusively detected by EmRest, we find that they are usually parameter validation-related bugs, and highly concentrated within seven operations (of *PersCtrl* and *MktSrv* APIs) that a large number of input-parameters are involved (an average of 13 parameters, compared to 4.1 for operations in which EmRest cannot detect bugs). In such cases, randomly selecting operations for exceptional testing (as adopted in baselines) tends to be ineffective, while EmRest prioritizes operations that are more prone to parameter validation issues (see Algorithm 1). EmRest also prioritizes mutation operators that are less frequently used on each parameter (see Algorithm 3), ensuring a more diverse examination of its validation logic. This finding further demonstrates the effectiveness of the *Exceptional Testing* phase in detecting bugs.

Answer to RQ2: EmRest detects more unique bugs than all six baselines in eight out of 16 APIs (with a maximum improvement of 134%). Among all unique bugs found in the whole experiments, EmRest is able to detect 60% of them, with 26% being exclusively detected.

Table 2. Covered Operations, Line Coverage, and Detected Unique Bugs for EmRest, EmRest$_{Random}$, and EmRest$_{NoRetry}$ Across 16 Subject APIs Over 30 Runs

	Operation Coverage			Line Coverage			Unique Bugs		
	EmRest	EmRest$_{\text{Random}}$	EmRest$_{\text{NoRetry}}$	EmRest	EmRest$_{\text{Random}}$	EmRest$_{\text{NoRetry}}$	EmRest	EmRest$_{\text{Random}}$	EmRest$_{\text{NoRetry}}$
FeatSrv	17.9 ± 0.3	17.9 ± 0.3	18.0 ± 0.0	77.3% ± 1.4	77.6% ± 1.4	77.6% ± 0.8	29.1 ± 0.9	29.6 ± 1.2	29.2 ± 1.2
GenomeNex	22.6 ± 0.5	22.7 ± 0.6	22.5 ± 0.6	32.2% ± 1.0	30.5% ± 1.6	31.0% ± 1.3	0.1 ± 0.3	0.0 ± 0.2	0.0 ± 0.2
LangTool	2.0 ± 0.0	2.0 ± 0.0	2.0 ± 0.0	22.1% ± 4.7	23.0% ± 4.8	23.6% ± 4.5	14.0 ± 0.8	13.6 ± 1.1	14.5 ± 1.0
MktSrv	1.0 ± 0.0	1.0 ± 0.0	1.0 ± 0.0	11.9% ± 0.0	11.9% ± 0.0	12.1% ± 0.9	69.9 ± 2.2	70.7 ± 2.2	70.3 ± 2.2
PersCtrl	8.0 ± 0.0	7.5 ± 0.7	8.0 ± 0.0	74.0% ± 0.0	73.1% ± 1.3	74.0% ± 0.0	156.6 ± 10.0	166.8 ± 17.7	173.9 ± 9.9
ProjTrack	45.4 ± 0.9	42.9 ± 1.7	43.6 ± 1.2	39.3% ± 0.3	39.0% ± 0.4	39.0% ± 0.4	15.0 ± 0.9	11.3 ± 1.7	15.1 ± 1.0
RestCtry	22.0 ± 0.0	21.9 ± 0.6	22.0 ± 0.0	77.5% ± 0.0	77.5% ± 0.2	77.5% ± 0.0	1.0 ± 0.0	1.0 ± 0.0	1.0 ± 0.0
UserMgmt	14.8 ± 0.4	14.8 ± 0.4	13.6 ± 0.5	87.4% ± 1.6	87.6% ± 1.3	80.5% ± 3.1	28.6 ± 0.8	28.5 ± 1.1	28.1 ± 0.7
NCS	6.0 ± 0.0	6.0 ± 0.0	6.0 ± 0.0	92.9% ± 0.6	92.8% ± 0.1	92.8% ± 0.2	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
SCS	11.0 ± 0.0	11.0 ± 0.0	11.0 ± 0.0	65.3% ± 1.1	65.7% ± 0.9	65.2% ± 1.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
GLBranch	9.0 ± 0.0	7.6 ± 1.4	4.7 ± 2.5	1.1% ± 0.0	1.1% ± 0.1	0.9% ± 0.2	0.0 ± 0.0	0.1 ± 0.4	0.0 ± 0.0
GLCommit	5.6 ± 2.3	5.3 ± 2.0	2.6 ± 1.7	1.2% ± 0.1	1.2% ± 0.1	0.9% ± 0.2	1.9 ± 0.9	2.0 ± 0.8	1.0 ± 0.9
GLGroups	14.5 ± 3.8	5.1 ± 3.6	4.9 ± 5.6	1.4% ± 0.1	1.2% ± 0.1	1.2% ± 0.1	5.5 ± 1.5	3.2 ± 1.4	3.4 ± 2.1
GLIssues	24.6 ± 3.8	19.6 ± 7.4	8.8 ± 8.5	2.6% ± 0.1	2.5% ± 0.2	2.2% ± 0.3	16.4 ± 3.7	12.4 ± 6.0	5.9 ± 5.9
GLProject	24.4 ± 0.9	18.8 ± 3.7	13.7 ± 10.5	2.7% ± 0.1	2.5% ± 0.2	2.2% ± 0.5	3.7 ± 0.9	2.6 ± 1.2	1.8 ± 1.9
GLRepo	4.4 ± 0.8	3.9 ± 1.1	1.8 ± 1.3	0.9% ± 0.2	0.8% ± 0.1	0.6% ± 0.1	0.6 ± 0.6	0.6 ± 0.6	0.2 ± 0.4

6.2.3 RQ3 (Impact of Operation Selection Strategies). Table 2 presents operation coverage, line coverage, and the number of unique bugs detected by EmRest and its two variants, EmRest$_{Random}$ (operations are selected randomly) and EmRest$_{NoRetry}$ (each operation is tested once), across 30 runs (each data is formatted as *average ± standard deviation*). From Table 2, EmRest$_{Random}$ and

EmRest$_{NoRetry}$ achieve similar operation and code coverage to EmRest for the first ten APIs (there is only one exception, *ProjTrack*), but EmRest greatly outperforms the two variants for GitLab APIs. By analyzing the APIs where EmRest$_{NoRetry}$ and EmRest$_{Random}$ perform worse, we find that they typically involve operation dependencies that are not reflected by URI hierarchy (e.g., for *ProjTrack*, four operations of the same URI length must be executed in the following order: *POST:/authenticate*, *POST:/employees*, *POST:/projects*, *POST:/assignment*). In such cases, EmRest$_{NoRetry}$ will miss the opportunity to test an operation if its operation dependencies are not satisfied at first, and EmRest$_{Random}$ can only discover the correct order by chance, both resulting in a lower performance than EmRest.

In terms of bug detection, EmRest performs similarly to the two variants in the first ten APIs. In some cases, such as *PersCtrl*, EmRest$_{NoRetry}$ detects 17.3 more unique bugs than EmRest. This is because EmRest$_{NoRetry}$ lacks a retry mechanism, so that it completes the *Inferring Constraints* phase more quickly. Accordingly, it has more time for the *Exceptional Testing* phase, which help to uncover more parameter validation-related bugs (under the same test budget). However, in GitLab APIs, both EmRest$_{Random}$ and EmRest$_{NoRetry}$ detect substantially fewer bugs than EmRest. This is because most GitLab bugs result from parameter interactions (as also discussed in RQ2), and to detect these bugs, capturing complex operation dependencies is crucial. This finding demonstrates the effectiveness of the operation selection strategies of EmRest.

Answer to RQ3: The operation selection strategy used in EmRest indeed improves its test coverage and bug detection ability, especially for handling complex operation dependencies.

7 Threats To Validity

As far as internal threats to validity is concerned, the performance of EmRest might be influenced by its configuration settings (e.g, the threshold for field matching similarity (*sim*)). Currently, these settings are determined empirically, and their effectiveness is demonstrated by the superior performance of EmRest over the baselines. Similarly, the performance of other baseline approaches used for comparison may also be influenced by their implementations. We obtained the latest stable releases of these approaches from their respective supplementary material, and followed their default settings. We note that the performance of MINER can be further enhanced through manual configuration adjustments (e.g., by providing value dictionaries for some input-parameters). To ensure a fair comparison, we chose to not perform any manual configuration for all approaches, and ensure that they only rely on the target OpenAPI specification for execution. We also note that a certain degree of randomness is inherent in all testing approaches. To mitigate the impact, the execution of each approach was repeated 30 times for each subject API.

In addition, we acknowledge that the performance of EmRest might also be influenced by the availability of error messages. But, we believe this will not be a major obstacle to its real-world adoption, because according to the best practice and standard of REST APIs development [49, 50, 53], modern REST APIs are expected to provide such error messages (as discussed in Section 2). Nevertheless, even if the APIs under test do not provide error messages (or there are error messages, but no relevant input parameters are mentioned), EmRest can still treat each HTTP status code (or each combination of HTTP status code and error message) as a distinct error, and try to infer the combinations of values assignment strategies that are most likely to contribute to each error. We also acknowledge that all testing approaches involved in this study rely on the availability of OpenAPI specifications. Since there are automated tools that can help generate such specifications from the source code [56] or test cases [52], we believe that these approaches could also be extended and adopted for testing APIs without specifications.

The external threat to validity lies in the generalizability of our results, as our experiments were conducted on 16 subject APIs only. To mitigate this issue, we have selected well recognized benchmark APIs (the first ten APIs in Table 1) that are widely used in current studies [33, 35]. Additionally, we have further included six APIs from a real-world large-scale enterprise system, GitLab, to cover more complex test scenarios. We believe that these APIs can represent a diverse set of experimental subjects for evaluation, making our findings applicable across various APIs.

8 Related Work

Numerous approaches have been proposed [2, 10, 13, 15, 16, 19, 20, 23, 25, 28, 31, 33, 34, 37, 40–42, 46, 47, 54, 57, 58, 60] for testing REST APIs, while to thoroughly examine APIs' behaviors, the approach needs to take the complex constraints in REST APIs into a careful consideration, so that valid test inputs can be generated. Specifically, constraints in REST APIs stem from value-specific rules, inter-parameter dependencies, and operation dependencies. Current testing approaches typically extract value-specific rules from specific fields (e.g., *examples* and *default*) of the OpenAPI specification (OAS) [16, 33, 40, 58, 60]. While inter-parameter dependencies are often embedded in natural language descriptions. IDL [17, 27] offers a formal language to define such dependencies. NLPtoREST [32] and RestCT [60] utilize NLP techniques to extract constraints. Additionally, operation dependencies are often implicit and rarely documented. Existing approaches usually rely on heuristic strategies to associate input parameters with response for inferring them [33, 41, 58, 60].

To improve the testing effectiveness, current testing approaches [16, 33, 37, 40–42, 58] further use status codes based feedback mechanisms to dynamically account for the constraints. For instance, RESTler [16] identifies invalid operation sequences by detecting 400-range status codes in sequence execution. MINER [42] builds upon this strategy by assigning higher weights to sequences that yield 200-range responses, facilitating deeper exploration of system logic. Other approaches rely on graph-based techniques [40, 41, 58] to model the dependency and update the graph based on the obtained status codes. ARAT-RL [33] utilizes reinforcement learning to adjust parameter assignments to satisfy constraints based on the obtained status codes. Although these approaches can generate valid test inputs for many APIs, they tend to struggle to handle multiple constraints within an operation, as adjusting test inputs for one constraint may violate another.

In constraint to the above approaches, EmRest proposes to leverage the diagnostic information in error messages for constraints inference. It especially manages to identify error message fragments to distinguish distinct constraint violations, and then adopts a statistical approach to identify the combinations of value assignment strategies that most likely contribute to each constraint violation. Since the idea of error message-guided testing has also been utilized in various fuzzing studies, including database [30], network protocol [18, 51], and web vulnerability [22], it can be a promising direction to further exploit such error messages to enhance the testing effectiveness of REST APIs.

Note that EmRest and RestCT both employ combinatorial testing for efficient input space sampling, but EmRest leverages error messages instead of relying solely on HTTP status codes, allowing it to infer more precise invalid value combinations. Additionally, EmRest derives value assignment strategies from a broader range of sources and includes an *Exceptional Testing* phase to expose parameter validation bugs.

Recently, some studies use external knowledge sources like DBpedia [13] and large language models (LLMs) [34, 37] to generate realistic parameter values that meet value-specific rules and inter-parameter dependencies. While effective for specific domains, these approaches are limited in specific domains that are not covered by the knowledge base. Additionally, using LLMs may pose privacy and confidentiality risks in some sensitive areas. In contrast, EmRest only relies on the OAS and test execution results, allowing it to be used in a more diverse test scenarios.

9 Conclusion

This paper presents EmRest, the first black-box approach leveraging error message analysis for REST API testing. By analyzing 400-range errors, it infers and excludes invalid value assignment strategies. Additionally, it mutates valid strategies and uses 500-range errors to prioritize bug-prone operations. Experiments on 16 real-world REST APIs demonstrate EmRest's superiority in test coverage and bug detection over existing methods.

DATA AVAILABILITY

The source code of EmRest and all experimental data are publicly available at our GitHub repository: https://github.com/GIST-NJU/EmRest. A snapshot of the repository, including the exact version used in this paper, is also archived on Zenodo [12] for long-term preservation and reproducibility.

Acknowledgments

This work is supported in part by National Natural Science Foundation of China (No. 62472209), Natural Science Foundation of Jiangsu Province (No. BK20221439), and Primary Research and Development Plan of Jiangsu Province (No. BE2023025-2).

References

[1] 2024. *Built-in String Formats of OpenAPI.* https://swagger.io/docs/specification/data-models/data-types/#string Accessed: 2024-07-02.

[2] 2024. *Dredd.* https://github.com/apiaryio/dredd Accessed: 2024-10-30.

[3] 2024. *FuzzyWuzzy.* https://github.com/seatgeek/fuzzywuzzy Accessed: 2024-07-04.

[4] 2024. *JaCoCo.* https://www.eclemma.org/jacoco/ Accessed: 2024-07-02.

[5] 2024. *JSON.* https://www.json.org/json-en.html Accessed: 2024-07-04.

[6] 2024. *MITMProxy.* https://mitmproxy.org/ Accessed: 2024-07-04.

[7] 2024. *The Official YAML Web Site.* https://yaml.org/ Accessed: 2024-07-04.

[8] 2024. *OpenAPI Specification - Version 3.1.0 | Swagger.* https://swagger.io/specification/ Accessed: 2024-07-02.

[9] 2024. *PICT.* https://github.com/microsoft/pict Accessed: 2024-07-04.

[10] 2024. *Tcases.* https://github.com/Cornutum/tcases Accessed: 2024-10-30.

[11] 2024. *YouTube Data API Specification.* https://api.apis.guru/v2/specs/googleapis.com/youtube/v3/openapi.yaml Accessed: 2024-10-29.

[12] 2025. *Replication package.* https://zenodo.org/records/14940931 Accessed: 2025-02-28.

[13] Juan C. Alonso. 2021. Automated generation of realistic test inputs for web APIs. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021).* Association for Computing Machinery, New York, NY, USA, 1666–1668. doi:10.1145/3468264.3473491

[14] Andrea Arcuri. 2018. An experience report on applying software testing academic results in industry: we need usable automated test generation. *Empirical Softw. Engg.* 23, 4 (Aug. 2018), 1959–1981. doi:10.1007/s10664-017-9570-9

[15] Andrea Arcuri. 2019. RESTful API Automated Test Case Generation with EvoMaster. *ACM Trans. Softw. Eng. Methodol.* 28, 1, Article 3 (jan 2019), 37 pages. doi:10.1145/3293455

[16] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. 2019. RESTler: stateful REST API fuzzing. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19).* IEEE Press, 748–758. doi:10.1109/ICSE.2019.00083

[17] Saman Barakat, Ana Belén Sánchez, and Sergio Segura. 2023. IDLGen: Automated Code Generation for Inter-parameter Dependencies in Web APIs. In *Service-Oriented Computing: 21st International Conference, ICSOC 2023, Rome, Italy, November 28 – December 1, 2023, Proceedings, Part I* (Rome, Italy). Springer-Verlag, Berlin, Heidelberg, 153–168. doi:10.1007/978-3-031-48421-6_11

[18] Wu Biao, Tang Chaojing, and Zhang Bin. 2021. FFUZZ: A Fast Fuzzing Test Method for Stateful Network Protocol Implementation. In *2021 2nd International Conference on Computer Communication and Network Security (CCNS).* 75–79. doi:10.1109/CCNS53852.2021.00023

[19] Davide Corradini, Michele Pasqua, and Mariano Ceccato. 2023. Automated Black-Box Testing of Mass Assignment Vulnerabilities in RESTful APIs. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23).* IEEE Press, 2553–2564. doi:10.1109/ICSE48619.2023.00213

[20] Davide Corradini, Amedeo Zampieri, Michele Pasqua, Emanuele Viglianisi, Michael Dallago, and Mariano Ceccato. 2022. Automated black-box testing of nominal and error scenarios in RESTful APIs. *Software Testing, Verification and Reliability* 32, 5 (2022), e1808.

[21] Swaroopa Dola, Rory McDaniel, Matthew B. Dwyer, and Mary Lou Soffa. 2024. CIT4DNN: Generating Diverse and Rare Inputs for Neural Networks Using Latent Space Combinatorial Testing. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 118, 13 pages. doi:10.1145/3597503.3639106

[22] Adam Doupé, Ludovico Cavedon, Christopher Kruegel, and Giovanni Vigna. 2012. Enemy of the state: a state-aware black-box web vulnerability scanner. In *Proceedings of the 21st USENIX Conference on Security Symposium* (Bellevue, WA) *(Security'12)*. USENIX Association, USA, 26.

[23] Hamza Ed-douibi, Javier Luis Cánovas Izquierdo, and Jordi Cabot. 2018. Automatic Generation of Test Cases for REST APIs: A Specification-Based Approach. *2018 IEEE 22nd International Enterprise Distributed Object Computing Conference (EDOC)* (2018), 181–190. doi:10.1109/edoc.2018.00031

[24] Roy Thomas Fielding and Richard N. Taylor. 2000. *Architectural styles and the design of network-based software architectures*. Ph. D. Dissertation. AAI9980887.

[25] Patrice Godefroid, Bo-Yuan Huang, and Marina Polishchuk. 2020. Intelligent REST API data fuzzing. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 725–736. doi:10.1145/3368089.3409719

[26] Amid Golmohammadi, Man Zhang, and Andrea Arcuri. 2023. Testing RESTful APIs: A Survey. *ACM Trans. Softw. Eng. Methodol.* 33, 1, Article 27 (nov 2023), 41 pages. doi:10.1145/3617175

[27] Henk Grent, Aleksei Akimov, and Maurício Aniche. 2021. Automatically Identifying Parameter Constraints in Complex Web APIs: A Case Study at Adyen. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 71–80. doi:10.1109/ICSE-SEIP52600.2021.00016

[28] Zac Hatfield-Dodds and Dmitry Dygalo. 2022. Deriving semantics-aware fuzzers from web API schemas. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 345–346. doi:10.1145/3510454.3528637

[29] Zac Hatfield-Dodds and Dmitry Dygalo. 2022. Deriving semantics-aware fuzzers from web API schemas. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 345–346. doi:10.1145/3510454.3528637

[30] Zu-Ming Jiang, Jia-Ju Bai, and Zhendong Su. 2023. DynSQL: Stateful fuzzing for database management systems with complex and valid SQL query generation. In *Proceedings of the 32nd USENIX Conference on Security Symposium* (Anaheim, CA, USA) *(SEC '23)*. USENIX Association, USA, Article 277, 17 pages.

[31] S. Karlsson, A. Causevic, and D. Sundmark. 2020. QuickREST: Property-based Test Generation of OpenAPI-Described RESTful APIs. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE Computer Society, Los Alamitos, CA, USA, 131–141. doi:10.1109/ICST46399.2020.00023

[32] Myeongsoo Kim, Davide Corradini, Saurabh Sinha, Alessandro Orso, Michele Pasqua, Rachel Tzoref-Brill, and Mariano Ceccato. 2023. Enhancing REST API Testing with NLP Techniques. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 1232–1243. doi:10.1145/3597926.3598131

[33] Myeongsoo Kim, Saurabh Sinha, and Alessandro Orso. 2023. Adaptive REST API Testing with Reinforcement Learning. IEEE Computer Society, 446–458. doi:10.1109/ASE56229.2023.00218

[34] Myeongsoo Kim, Tyler Stennett, Dhruv Shah, Saurabh Sinha, and Alessandro Orso. 2024. Leveraging Large Language Models to Improve REST API Testing. In *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results* (Lisbon, Portugal) *(ICSE-NIER'24)*. Association for Computing Machinery, New York, NY, USA, 37–41. doi:10.1145/3639476.3639769

[35] Myeongsoo Kim, Qi Xin, Saurabh Sinha, and Alessandro Orso. 2022. Automated test generation for REST APIs: no time to rest yet. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, South Korea) *(ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 289–301. doi:10.1145/3533767.3534401

[36] D.R. Kuhn, D.R. Wallace, and A.M. Gallo. 2004. Software fault interactions and implications for software testing. *IEEE Transactions on Software Engineering* 30, 6 (2004), 418–421. doi:10.1109/TSE.2004.24

[37] Tri Le, Thien Tran, Duy Cao, Vy Le, Tien N. Nguyen, and Vu Nguyen. 2024. KAT: Dependency-Aware Automated API Testing with Large Language Models . In *2024 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE Computer Society, Los Alamitos, CA, USA, 82–92. doi:10.1109/ICST60714.2024.00017

[38] Manuel Leithner, Andrea Bombarda, Michael Wagner, Angelo Gargantini, and Dimitris E. Simos. 2024. State of the CArt: evaluating covering array generators at scale. *Int. J. Softw. Tools Technol. Transf.* 26, 3 (May 2024), 301–326. doi:10.1007/s10009-024-00745-2

[39] Jinkun Lin, Shaowei Cai, Bing He, Yingjie Fu, Chuan Luo, and Qingwei Lin. 2021. FastCA: An Effective and Efficient Tool for Combinatorial Covering Array Generation. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 77–80. doi:10.1109/ICSE-Companion52605.2021.00040

[40] Jiaxian Lin, Tianyu Li, Yang Chen, Guangsheng Wei, Jiadong Lin, Sen Zhang, and Hui Xu. 2023. foREST: A Tree-based Black-box Fuzzing Approach for RESTful APIs. IEEE Computer Society, 695–705. doi:10.1109/ISSRE59848.2023.00023

[41] Yi Liu, Yuekang Li, Gelei Deng, Yang Liu, Ruiyuan Wan, Runchao Wu, Dandan Ji, Shiheng Xu, and Minli Bao. 2022. Morest: model-based RESTful API testing with execution feedback. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1406–1417. doi:10.1145/3510003.3510133

[42] Chenyang Lyu, Jiacheng Xu, Shouling Ji, Xuhong Zhang, Qinying Wang, Binbin Zhao, Gaoning Pan, Wei Cao, Peng Chen, and Raheem Beyah. 2023. MINER: a hybrid data-driven approach for REST API fuzzing. In *Proceedings of the 32nd USENIX Conference on Security Symposium* (Anaheim, CA, USA) *(SEC '23)*. USENIX Association, USA, Article 253, 18 pages.

[43] Alberto Martin-Lopez, Sergio Segura, Carlos Müller, and Antonio Ruiz-Cortés. 2022. Specification and Automated Analysis of Inter-Parameter Dependencies in Web APIs. *IEEE Transactions on Services Computing* 15, 4 (2022), 2342–2355. doi:10.1109/TSC.2021.3050610

[44] Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. 2019. A Catalogue of Inter-parameter Dependencies in RESTful Web APIs. In *Service-Oriented Computing: 17th International Conference, ICSOC 2019, Toulouse, France, October 28–31, 2019, Proceedings* (Toulouse, France). Springer-Verlag, Berlin, Heidelberg, 399–414. doi:10.1007/978-3-030-33702-5_31

[45] Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. 2019. Test coverage criteria for RESTful web APIs. In *Proceedings of the 10th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation* (Tallinn, Estonia) *(A-TEST 2019)*. Association for Computing Machinery, New York, NY, USA, 15–21. doi:10.1145/3340433.3342822

[46] Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. 2021. RESTest: automated black-box testing of RESTful web APIs. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, Denmark) *(ISSTA 2021)*. Association for Computing Machinery, New York, NY, USA, 682–685. doi:10.1145/3460319.3469082

[47] A. Giuliano Mirabella, Alberto Martin-Lopez, Sergio Segura, Luis Valencia-Cabrera, and Antonio Ruiz-Cortes. 2021. Deep Learning-Based Prediction of Test Input Validity for RESTful APIs. In *2021 IEEE/ACM Third International Workshop on Deep Learning for Testing and Testing for Deep Learning (DeepTest)* (Madrid, Spain, 2021-06). IEEE, 9–16. doi:10.1109/DeepTest52559.2021.00008

[48] Changhai Nie and Hareton Leung. 2011. A survey of combinatorial testing. *ACM Comput. Surv.* 43, 2, Article 11 (Feb. 2011), 29 pages. doi:10.1145/1883612.1883618

[49] Mark Nottingham and Erik Wilde. 2016. Problem Details for HTTP APIs. RFC 7807. doi:10.17487/RFC7807

[50] Mark Nottingham, Erik Wilde, and Sanjay Dalal. 2023. Problem Details for HTTP APIs. RFC 9457. doi:10.17487/RFC9457

[51] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. 2020. AFLNET: A Greybox Fuzzer for Network Protocols. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. 460–465. doi:10.1109/ICST46399.2020.00062

[52] Postman API Evangelist. 2024. Postman to OpenAPI Collection. https://www.postman.com/api-evangelist/artificial-intelligence/collection/txy0rdd/postman-to-openapi Accessed: 2025-02-11.

[53] RobBagby. 2023. Web API Implementation - Best Practices for Cloud Applications. https://learn.microsoft.com/en-us/azure/architecture/best-practices/api-implementation.

[54] Sergio Segura, José A. Parejo, Javier Troya, and Antonio Ruiz-Cortés. 2018. Metamorphic testing of RESTful web APIs. (2018), 882. doi:10.1145/3180155.3182528

[55] Sunny Shree, Krishna Khadka, Yu Lei, Raghu N. Kacker, and D. Richard Kuhn. 2024. Constructing Surrogate Models in Machine Learning Using Combinatorial Testing and Active Learning. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering* (Sacramento, CA, USA) *(ASE '24)*. Association for Computing Machinery, New York, NY, USA, 1645–1654. doi:10.1145/3691620.3695532

[56] SpringDoc Contributors. 2024. SpringDoc OpenAPI. https://springdoc.org/ Accessed: 2025-02-11.

[57] Dimitri Stallenberg, Mitchell Olsthoorn, and Annibale Panichella. 2022. Improving test case generation for REST APIs through hierarchical clustering. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering* (Melbourne, Australia) *(ASE '21)*. IEEE Press, 117–128. doi:10.1109/ASE51524.2021.9678586

[58] Emanuele Viglianisi, Michael Dallago, and Mariano Ceccato. 2020. RESTTESTGEN: Automated Black-Box Testing of RESTful APIs. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)* (2020-10). 142–152. doi:10.1109/ICST46399.2020.00024

[59] Xinyi Wang, Paolo Arcaini, Tao Yue, and Shaukat Ali. 2024. QuCAT: A Combinatorial Testing Tool for Quantum Software. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering* (Echternach, Luxembourg) *(ASE '23)*. IEEE Press, 2066–2069. doi:10.1109/ASE56229.2023.00062

[60] Huayao Wu, Lixin Xu, Xintao Niu, and Changhai Nie. 2022. Combinatorial testing of RESTful APIs. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 426–437. doi:10.1145/3510003.3510151

[61] Man Zhang and Andrea Arcuri. 2023. Open Problems in Fuzzing RESTful APIs: A Comparison of Tools. *ACM Trans. Softw. Eng. Methodol.* 32, 6, Article 144 (sep 2023), 45 pages. doi:10.1145/3597205

Received 20 October 2024; revised 27 February 2025; accepted 31 March 2009