Black-Box Mutation Testing of Web APIs

***Abstract*—The key role of web Application Programming Interfaces (APIs) in software integration has led to a surge in the development of test case generation tools for web APIs. Most of these tools work in black-box mode, without access to the source code, and are primarily assessed based on their ability to detect crashes and violations of API specifications. As a result, their effectiveness in identifying other types of failures (e.g., incorrect outputs caused by functional bugs) remains largely unknown. In this article, we propose a black-box mutation testing approach to assess the fault-detection capabilities of testing tools for web APIs. Unlike traditional mutation testing, which introduces faults in the source code, our approach generates faulty but realistic variants of HTTP responses, simulating the observable effects of bugs in the underlying implementation. These mutated responses can then be used to evaluate the effectiveness of testing tools in detecting incorrect outputs, making our approach readily applicable to both closed- and open-source web APIs. The approach is implemented in an extensible tool, HTTPMUTATOR, which supports 23 mutation operators for HTTP responses, including mutations on status codes, headers, and JSON payloads. Evaluation results on 20 open-source and industrial APIs show that HTTPMUTATOR is effective in generating diverse and realistic erroneous outputs, with a positive correlation between black-box and white-box mutation coverage. More importantly, HTTPMUTATOR has already proven instrumental in assessing the effectiveness of test oracle generation tools for REST APIs, highlighting its practical relevance.**

***Index Terms*—IEEE, IEEEtran, journal, LATEX, paper, template.**

I. INTRODUCTION

Web Application Programming Interfaces (APIs) have become the backbone of modern software systems. They enable seamless communication between distributed components such as mobile, cloud, and Internet of Things (IoT) applications, fostering reuse and innovation across organizations. Today, both public and private APIs are widely adopted, with API directories such as *Postman Public API Network* [1], *APIs.guru* [2], and *Rapid API* [3] indexing tens of thousands of APIs from diverse domains including e-commerce, finance, social media, and and the rapidly developing LLM application ecosystem. Web APIs can be categorized into several types based on application design and communication protocols. Among them, Hypertext Transfer Protocol (HTTP) APIs, arguably the de-facto standard, use the HTTP protocol to interact—typically through CRUD (Create, Read, Update, and Delete) operations—with resources (e.g., a song in the Spotify API [] or a repository in the GitHub API []). HTTP APIs often follow the principles of the REpresentational State Transfer (REST) [4] architectural style for distributed systems and are commonly referred to as REST APIs []. More recently, GraphQL has emerged as an alternative approach, offering flexible and efficient data querying capabilities []. Henceforth, we will use the term web API to refer to HTTP-based APIs, including both modern REST and GraphQL APIs.

As the number and complexity of web APIs continue to grow, ensuring their reliability has become increasingly critical. A fault in a core API can quickly propagate and disrupt both internal and external services that depend on it. This concern has driven the development of numerous test case generation tools for web APIs, particularly REST APIs []. Most of these tools adopt a black-box approach, where test cases are automatically derived from the specification of the API under test, typically in the OpenAPI Specification (OAS) format []. Test cases are created by setting values to the input parameters and checking the validity of the returned responses by applying different test oracles. Common test oracles include the detection of crashes (responses with a 5XX HTTP status code) [], disconformities with the API specification (e.g., a missing output JSON property) [], regressions [], and violations of API best practices (e.g., checking that the results of multiple calls to idempotent operations are identical) []. More recent work has explored the automated generation of test oracles for REST APIs by analyzing the API specification and the corresponding set of API requests and responses [].

Test case generation tools for REST APIs are commonly evaluated based on their coverage and failure detection capabilities. Regarding coverage, prior work has employed traditional code-based metrics like line and branch coverage on open-source APIs []. Additionally, REST-specific black-box criteria, such as operation coverage, have been proposed []. In terms of failure detection, tools have been assessed using both commercial APIs and open-source alternatives, with the goal of assessing their ability to uncover real issues, mostly crashes [] and specification violations []. However, commercial and well-established open-source APIs rarely exhibit trivially-observable failures (e.g., 5XX HTTP status codes), if any at all. This lack of representative bugs presents a major obstacle for thoroughly evaluating the failure detection capabilities of existing tools. This is the challenge that motivates our work.

One potential solution to this challenge is to use mutation testing: assessing the ability of testing tools to uncover artificial faults automatically seeded in the source code of the web APIs under test []. In practice, however, source code is often unavailable. Furthermore, even when it is accessible, the technological diversity of open-source APIs and the considerable effort required to deploy them (e.g., populating databases) limit the feasibility of traditional mutation testing. This raises a key question: *Can faults be simulated in a web API without access to its source code?* This question is the seed that has driven our work.

In this article, we propose a black-box mutation testing approach for web APIs relying on the HTTP protocol. Unlike traditional mutation testing, which introduces faults in the source code, we propose to generate faulty but realistic variants of HTTP responses (e.g., removing a JSON property) simulating the observable effects of bugs. These output-level

mutants enable the evaluation of testing tools based on their ability to identify incorrect responses, thereby providing a practical and quantifiable measure of test oracle effectiveness in black-box settings.

We propose a catalog of X mutation operators for HTTP response messages. These operators introduce changes in the message status code (e.g.,), headers (e.g.,), and in the JSON body (e.g.,). They are derived from real-world bugs reported in academic literature and bug tracking systems, making them representative of actual faults. All operators have been implemented in an open-source, extensible tool, HTTPMUTATOR, that automates the generation of output-level mutants. This makes our approach readily applicable and compatible with existing test case generation tools for web APIs.

Evaluation results with... show that...

Our technique offers several advantages over traditional mutation testing. First, it is programming language-agnostic and applicable to any web API that communicates over HTTP. Second, it is highly efficient, as mutants are created at the output level by modifying HTTP messages rather than generating altered program versions. Third, black-box mutation naturally avoids the problem of equivalent mutants: since all modified responses differ by construction, they are always detectable by test cases. On the other hand, unlike white-box mutation, our black-box mutant space and score are bounded by observable test outputs and the strength of test oracles, rather than the program internals. Thus, test suites with limited output diversity can achieve high black-box mutation scores yet low code coverage. However, this limitation can be effectively addressed in practice by combining black-box mutation with existing coverage metrics to evaluate input and output quality (see Section **??**).

In summary, this paper first introduces background and related work on mutation testing and automated testing of web APIs (Section **??**). Then, the paper presents the following original contributions:

•

Section **??** discusses related work, Section **??** addresses threats to validity, and Section **??** concludes.

A preliminary version of this work appeared in XXX [**?**] . This paper extends our previous work in several directions. First, . Second, . Third, .

II. BACKGROUND AND RELATED WORK

This section provides the foundations and an overview of current approaches in mutation testing and automated testing of REST APIs.

A. Mutation Testing

Mutation testing is a fault-based technique for evaluating the effectiveness of test suites [5]. It applies small syntactic changes, known as *mutations*, to the source code or bytecode of the program in order to simulate potential faults. These changes are guided by a set of *mutation operators*, each specifying a type of syntactic modification, such as replacing arithmetic or relational operators, altering conditional expressions, or modifying return values. Applying a mutation operator produces a *mutant*, which is a modified version of the original program. Each mutant, along with the original program, is executed against the test suite. The outputs of the original and mutant versions are then compared to detect any differences. When a difference is found, it indicates that the injected fault was triggered by the test suite, and the mutant is therefore considered detected or *killed*. Conversely, when the output remains unchanged, the mutant is said to be *alive* and requires further analysis, as it can indicate a deficiency in the fault detection ability of the test suite. However, this is not always the case, since some mutants may be functionally identical to the original program. These are known as *semantically-equivalent* mutants. In practice, testers aim to kill as many mutants as possible to enhance the fault detection capability of the test suite. The ratio of killed mutants to the total number of non-equivalent mutants is referred to as the *mutation score* or *mutation coverage*.

The injection process of mutations is typically automated through the use of mutation tools that implement a variety of mutation operators. These operators are applied whenever a matching pattern is identified in the source code (for example, substituting an arithmetic operator such as + with −). According to a recent empirical study [6], over a hundred mutation testing tools have been developed to date, covering popular programming languages, including Java, C/C++, Python, and C#. Most tools operate in a white-box fashion, applying mutations either to the source code (e.g., *μJava* [7]) or to intermediate code representations such as bytecode (e.g., PIT [8] and Major [9]). In recent years, the number of available tools has continued to grow considerably, with several specialized variants emerging for specific domains such as Deep Learning [10] and Smart Contracts [11], [12].

B. Automated Testing of REST APIs

Modern web APIs are typically compliant, fully or partially, with the REpresentational State Transfer (REST) architectural style [4], being referred to as REST APIs or RESTful APIs [13]. REST APIs are usually composed of multiple web services, with each one of them implementing create, read, update, and/or delete (CRUD) operations on a resource (e.g., a video in the YouTube API [14]). These operations are typically invoked by sending HTTP requests to a Uniform Resource Identifier (URI) that represents a resource or a collection of resources.

Research on the automated generation of test cases for REST APIs has thrived in recent years. Most approaches follow a black-box approach, where test cases are automatically derived from the API specification, typically in the OpenAPI Specification (OAS) format [15]. An OAS document describes the API functionality in terms of the allowed inputs (HTTP requests) and outputs (HTTP responses), in a machine-readable format. Given an OAS document, test cases are created by setting values to the input parameters and checking the validity of the returned responses by applying different test oracles, i.e., tests of correctness [16]. Approaches mainly differ in the way in which they generate test cases and the types of test oracles used. Test case generation strategies

```
HTTP/1.1 200 OK
Content-Type: application/json


{
  "orderId": 12345,
  "customer": { "id": 678,
                "name": "Alice" },
  "total_items": 2,
  "items": [
    { "productId": 101,
      "name": "Laptop",
      "quantity": 1,
      "price": 1200.00 },
    { "productId": 202,
      "name": "Mouse",
      "quantity": 2,
      "price": 25.00 }
  ],
  "status": "shipped",
  "total": 1250.00
}
```

Listing 1: API response in JSON format.

encompass random testing (fuzzing) [17]–[20], property-based testing [18], [21]–[23], model-based testing [24], [25], and constraint-based testing [26]–[28], among others. More recently, several authors have explored the use of large language models to automate the generation of test cases [28]–[31]. White-box approaches—as those integrated into the popular tool EvoMaster [32], [33]—are less common because they depend on implementation access, yet they often achieve strong effectiveness by combining search-based testing with code coverage and database states guidance. Some approaches focus on individual API operations and generate isolated requests, while others construct sequences of API calls to enable stateful testing [20], [21], [28], [34]–[36].

In terms of failure detection, test oracles are primarily limited to detecting API crashes (e.g., 500 status codes), violations of the API specification [18], [21], [26], [33], [34], security properties [37], regressions [38], [39], or adherence to best design practices [37], [40]–[42]. Recent work has explored the automated generation of test oracles for REST APIs by analyzing the API specification and sets of API requests and responses [**?**], [43], [44]. Listing 1 shows a sample API response representing a shopping order that includes two products. A very basic test oracle would simply verify that the response status code is not 500, which would indicate a server error. A slightly more advanced oracle would validate the response against the API specification by checking the object structure, ensuring the presence of mandatory fields, and confirming type correctness (e.g., verifying that the value of the property `price` is a float). More sophisticated oracles can assert domain-specific test oracles, such as ensuring that the `quantity` value is a positive integer or that the size of the `items` array equals the value of the `total_items` property. Finally, a regression test oracle could verify that the output is identical to that of a previous API call, ensuring that recent modifications have not introduced unintended breaking changes.

Test adequacy criteria define stopping rules that help determine when testing is sufficient. In the context of REST APIs, common criteria include code coverage (when the source code is available) [32], [33] or predefined budgets expressed in terms of time [20] or number of generated test cases [24]. To quantify coverage in black-box settings—where source code is not available—Martin-Lopez et al. introduced black-box coverage criteria for REST APIs, specifically the so-called Test Coverage Levels (TCLs) [45]. TCLs measure how thoroughly a test suite exercises an observable input–output space of an API. The framework defines eight levels (TCL0–TCL7), ranging from basic request exploration to complex multi-operation flows. However, while coverage metrics reveal how much of an API is exercised by input requests, they offer no assessment of the ability of test oracles to identify faulty outputs. To address this limitation, this article presents a complementary adequacy criterion for web APIs that evaluates oracle effectiveness by mutating API responses. This criterion provides measurable and actionable insights into the ability of test suites to reveal incorrect outputs, in both black-box and white-box settings.

III. Black-Box Mutation of Web APIs

We propose a black-box mutation testing approach for web APIs that operates at the level of HTTP responses. Instead of injecting faults into the service source code—often not available—our method generates faulty but realistic variants of HTTP output messages, for example, replacing a 200 status code with a 400. These output-level mutants emulate the observable effects of real faults and allow the evaluation of testing tools based on their ability to detect incorrect responses. This provides a practical and quantifiable means of assessing test oracle effectiveness in black-box mode, applicable to both open-source and closed APIs. The approach is intended to complement existing coverage-based techniques for measuring the quality of test inputs (HTTP requests), together offering a more comprehensive assessment of testing effectiveness in the context of web APIs. The following sections introduce the proposed mutation operators, mutation strategies, and the tool HttpMutator.

A. Mutation Operators

We introduce 23 novel mutation operators for generating variants of HTTP responses, summarized in Table I. These operators systematically modify the three primary elements of an API response: the status code, the response headers, and the JSON body payload. We focus on JSON as the de-facto standard exchange format in modern APIs, used by the vast majority of specifications and accounting for more than 90% of API traffic in practice [**?**], [**?**].

The design of these operators is based on two complementary sources. First, we analyzed around 130 documented bugs extracted from the issue trackers of two large-scale commercial REST APIs: Spotify[1] and YouTube[2], classifying them into common fault patterns such as incorrect status codes, malformed headers, and structurally or semantically invalid JSON payloads. Second, we relied on curated catalogues of real REST API bugs reported in recent empirical studies [**?**],

[1]https://github.com/spotify/web-api/issues
[2]https://issuetracker.google.com/issues?q=issues

TABLE I: Mutation operators

Response Element	Context	Mutation Operator	Description
Status Code		R2XX	Replaces an error status code (4XX/5XX) with a success code (e.g., 200, 201).
		R4XX	Replaces a successful status code (2XX) with a client error code (e.g., 400, 403).
		R5XX	Replaces a status code with a server error code (e.g., 500, 502).
Headers	Content-Type	CTC	Changes the media type part of the Content-Type header value.
		CTD	Removes the Content-Type header entirely.
	Charset	CPC	Changes the Charset property value.
		CPD	Removes the Charset property from the header's value.
	Location	LHC	Changes the Location header value.
		LHD	Removes the Location header entirely.
JSON Payload	Array	AEA	Adds a new element to an array.
		AER	Removes an element from an array.
		AEE	Exchanges an element for another in an array.
		EAS	Sets an array to empty.
	Object	OPA	Adds a property to an object.
		OPR	Removes a property from an object.
		OTPR	Removes an object-type property from an object.
	Property	PTC	Changes the data type of a property.
		BPR	Reverses the value of a boolean-type property.
		NPS	Sets a property's value to null.
		NPR	Replaces the value of a numeric-type property.
		SCA	Adds special characters to a string-type property.
		SPR	Replaces the entire string value of a property.
		SRE	Reduces or extends a string's length to a boundary value.

[**?**], [26], [44]. The complete bug classification underlying our operator design is available online for reproducibility [].

The following subsection introduces the mutation operators including their rationale, illustrative examples, and references to real bugs showing their occurrence in practice.

1) Status Code Mutation Operators: These operators replace the HTTP original status code with a value from a different, semantically distinct category to simulate common server-side logic errors. Specifically, we propose the following three operators:

Replacement with status code 2XX (R2XX). This operator replaces the status code of an HTTP message with one of the following 2XX success codes: 200 (OK), 201 (Created), 202 (Accepted), or 204 (No Content). For example, it may mutate an invalid input request (e.g., `"status": 400`) into one incorrectly labelled as successful (e.g., `"status": 200`). This behaviour mirrows real bugs observed in practice. For instance, in the Yelp API the parameters `open_now` and `open_at` are documented as mutually exclusive. Nevertheless, a request including both parameters returned a successful response instead of an invalid one [26].

Replacement with status code 4XX (R4XX). This operator replaces the original status code with one of the following 4XX client-error codes: 400 (Bad Request), 401 (Unauthorized), 403 (Forbidden), 404 (Not Found), or 409 (Conflict). For example, it may transform a successful response (e.g., `"status": 200`) into one incorrectly indicating an invalid request (e.g., `"status": 400`). Such issues—valid responses returning unexpected errors—are common in practice, revealing +1.2K failures across six different APIs in the study by Martin-Lopez et al. [26].

Replacement with status code 5XX (R5XX). This operator replaces the original status code with one of the following 5XX server-error codes: 500 (Internal Server Error), 502 (Bad Gateway), 503 (Service Unavailable), or 504 (Gateway Timeout). For example, it may transform a successful response (e.g., `"status": 200`) into one incorrectly indicating a server failure (e.g., `"status": 503`), thereby simulating a temporary outage. These types of issues challenge the ability of testing tools to detect unexpected server-errors (i.e., crashes).

2) Header Mutation Operators: HTTP headers carry essential metadata that determines how clients process API responses. We introduce mutation operators for two critical headers in the context of web APIs: the `Content-Type` header (including both its `media type` and `charset` property,

which govern body parsing and decoding) and the `Location` header (which specifies redirection after resource creation). By injecting malformed values (e.g., unsupported media types, invalid character sets, broken URIs) and by omitting these headers altogether, our operators challenge the ability of a test suite to detect and handle misconfigured responses. Specifically, we propose the following six mutation operators for HTTP headers.

Content-Type Change (CTC). This operator replaces the media type value of the `Content-Type` header with another one selected from common media types: *application/json*, *application/xml*, *text/plain*, *text/html*, *text/css*, *text/javascript*, or *application/x-www-form-urlencoded*. This operator simulates a misconfigured server that returns a valid payload but declares it using the wrong format. As an example, the following Listing shows the use of the CTC operator, where the `Content-Type` is changed from `application/json` to `text/plain`.

(a) Original

```
header('Content-Type: application/json; charset=utf-8');
```

(b) Mutant

```
header('Content-Type: text/plain; charset=utf-8');
```

Listing 2: Content-Type header change.

Content-Type Deletion (CTD). This operator removes the media type portion of the `Content-Type` header. It simulates a fault in which the server fails to declare the response format, thereby creating ambiguity. For example, `application/json;charset=utf-8` may be mutated to `charset=utf-8`.

Charset Property Change (CPC). This operator modifies the `charset` property within the `Content-Type` header, replacing it with a non-standard or incorrect value to simulate subtle encoding mismatches. As an example, this resembles an issue reported in the Spotify bug tracker, documented as "*Incorrect charset specified in response header for audio-features (UTF8 vs utf-8).*" (Bug ID 291).

Charset Property Deletion (CPD). This operator removes the `charset` property from the `Content-Type` header value, for example, transforming `application/json; charset=UTF-8` into `application/json`.

Location Header Change (LHC). This operator modifies the URL in the `Location` header to an incorrect path. This simulates a broken redirection for a newly created or moved resource. The example in Listing 3 shows the use of the LHC operator, where the `Location` header is modified.

(a) Original

```
header('Location: https://www.example.org/index.php');
```

(b) Mutant

```
header('Location: https://www.example.org/home');
```

Listing 3: Location header change.

Location Header Deletion (LHD). This operator deletes the `Location` header simulating a fault in which the server completes an action but fails to provide the client with the required navigational information. For example, a response containing `Location: https://www.example.org/index.php` may be mutated to an empty location header (`Location:`).

3) JSON Mutation Operators: JSON mutation operators introduce faults into the JSON payload of an API response, targeting its structure and data values. They simulate a wide range of common bugs, including missing or extraneous data [], incorrect data types [], and violations of value constraints [], among others. The operators are organized into three groups based on the JSON element they target: arrays, objects, and properties. Specifically, we propose the following 14 mutation operators for JSON:

Array Element Addition (AEA). This operator inserts a new structurally valid element into an existing array, simulating responses that incorrectly include additional items. Listing 4 presents an example of this mutation, where an extra object is inserted into the *items* array (highlighted in red color). This type of mutation mirrors real-world bugs such as the one reported in Spotify, where the operation XXX returned unexpected albums that should be hidden (ID 960).

(a) Original

```
"items": [
  {
    "position": 23,
    "title": "Hello",
    "author":"Adele"
  }
]
```

(b) Mutant

```
"items": [
  {
    "position": 23,
    "title": "Hello",
    "author":"Adele"
  },
  {
   "position": 31,
   "title": "Letter",
   "author":"Bob Dylan"
  }
]
```

Listing 4: Addition of an element to an array.

Array Element Removal (AER). This mutation operator deletes an existing element from an array, emulating responses with missing or incomplete data. Listing 5 shows a sample mutant generated by this operator by removing an object from the *items* array. This type of fault has appeared in real systems. For instance, bug ID 130331391[3] in YouTube describes the problem as "*API Search request announces*

[3]https://issuetracker.google.com/issues?q=issues

300+ results, delivers only 130".

(a) Original

```
"items": [
  {
    "position": 23,
    "title": "Hello",
    "author":"Adele"
  },
 {
   "position": 15,
   "title": "Remedy",
   "author":"Adele"
 }
]
```

(b) Mutant

```
"items": [
  {
    "position": 23,
    "title": "Hello",
    "author":"Adele"
  }
]
```

Listing 5: Removal of an element from an array.

Array Elements Exchange (AEE). This mutation operator swaps the positions of two elements within an array, simulating incorrect sorting or ordering logic. Listing 6 presents an example of this operator, where two object type elements in the *items* array exchange their positions. This type of fault is also common in real systems. For instance, the Spotify issue tracker includes bug ID 305 describing the issue as "*Search: Results not consistently ordered when using different limit arguments*".

(a) Original

```
"items": [
  {
    "position": 23,
    "title": "Hello",
    "author":"Adele"
  },
 {
    "position": 15,
    "title": "Remedy",
    "author":"Adele"
 },
 {
    "position": 4,
    "title": "My Same",
    "author":"Adele"
 }
]
```

(b) Mutant

```
"items": [
  {
    "position": 4,
    "title": "My Same",
    "author":"Adele"
 },
 {
    "position": 23,
    "title": "Hello",
    "author":"Adele"
 },
 {
    "position": 23,
    "title": "Hello",
    "author":"Adele"
 }
]
```

Listing 6: Exchange of two array elements.

Empty Array Setting (EAS). This mutation operator clears all elements from a non-empty array, representing a failure to return expected results. Listing 7 shows an example of this operator, where the *items* array is emptied. This type of behaviour mirrors faults observed in practice, such as issue ID 141100401 in YouTube, which states "*For several days the API search list of live events has not worked. I get an empty array, but I have a live stream on YouTube*".

Object Property Addition (OPA). This mutation operator adds a new property to a JSON object, simulating a response that includes unexpected data not defined in the API specification. Listing 8 illustrates this operator, where the property *duration* is inserted into the object.

Object Property Removal (OPR). This operator removes an existing property from a JSON object, simulating cases where an API fails to return the required information. Listing 9 shows an example of this operator, where the *author* property is removed. This behavior was observed in the GitHub API, where the absence of the `template_repository` field revealed that it was never used in practice despite appearing in the documentation [44].

(a) Original

```
"items": [
  {
    "position": 23,
    "title": "Hello",
    "author":"Adele"
  },
  {
    "position": 15,
    "title": "Remedy",
    "author":"Adele"
  },
]
```

(b) Mutant

```
"items": []
```

Listing 7: Setting an array to empty.

(a) Original

```
"items": [
  {
    "position": 23,
    "title": "Hello",
    "author":"Adele"
  }
]
```

(b) Mutant

```
"items": [
  {
    "position": 23,
    "title": "Hello",
    "author":"Adele",
    "duration": 1.2
  }
]
```

Listing 8: Adding a property to an object.

(a) Original

```
"items": [
  {
    "position": 23,
    "title": "Hello",
    "author":"Adele"
  }
]
```

(b) Mutant

```
"items": [
  {
    "position": 23,
    "title": "Hello"
  }
]
```

Listing 9: Removing a property from an object.

Object-Type Property Removal (OTPR). This operator removes a property whose value is a nested JSON object, simulating omissions that may occur due to conditional serialisation, skipped null values, or schema evolution. Listing 10 shows an example of this operator, where the object *details* is omitted. Bug 632 in Spotify resembles this behaviour, reported as "*Playlist owner is missing from response when creating a playlist*".

Property Type Change (PTC). This mutation operator changes the data type of a property value, simulating faults caused by incorrect data serialisation. Listing 11 shows an example of this operator, where the value of the property *position* is changed from a number to a string.

Boolean Property Reverse (BPR). This operator flips the value of a boolean property, simulating faults that arise from flawed server-side conditional logic or misconfigured feature flags. A similar issue was reported in Spotify (ID 623), for example, where the boolean property `is_playing` returned

(a) Original

```
"items": [
  {
    "position": 23,
    "title": "Hello",
    "author":"Adele"
    "details":
      {
        "duration": 1.5,
        "collaborations": 0,
        "awards": true
      }
  }
]
```

(b) Mutant

```
"items": [
  {
    "position": 23,
    "title": "Hello",
    "author":"Adele"
  }
]
```

Listing 10: Removing an object.

(a) Original

```
"items": [
  {
    "position": 20,
    "title": "Hello",
    "author":"Adele"
  }
]
```

(b) Mutant

```
"items": [
  {
    "position":"twenty",
    "title": "Hello",
    "author":"Adele"
  }
]
```

Listing 11: Changing the type of a property value.

true instead of false when no music was playing.

Null Property Setting (NPS). This mutation operator replaces a propertys value with `null`, simulating cases where data is unexpectedly missing or has not been properly initialised on the server. This operator resembles issues found in practice such as Spotify bug ID 930 titled "*Get a Playlist endpoint returns NULL on description when getting a Spotify playlist*".

Numeric Property Replacement (NPR). This mutation operator replaces a numeric value with another, simulating faults caused by incorrect calculations or erroneous data retrieval. This type of behaviour reflects common issues identified in commercial APIs, including hotel rooms with zero beds in the Amadeus Hotel API [44], countries with negative areas in RESTCountries [46], [47], comic books with zero pages in the Marvel API [44], and venues located outside the user specified search radius in the Foursquare API [44], among others.

String Property Replacement (SPR). This mutation operator replaces the value of a string property with arbitrary text, simulating incorrect or malformed textual data. Examples of these types of issues are well documented, including the GitHub API creating repositories with licences different from those requested [44], the OMDb API supporting undocumented media types such as videogames [44], and the Vimeo API returning a user type ("custom") not documented in its specification [**?**].

Special Characters Addition (SCA). This operator inserts special characters, such as "/" or "*", into the value of a string property. Incorrectly managed special characters is a common source of bugs in practice, such as malformed URLs in the GitLab API [48] or invalid timestamp formats in the Deutschebahn API [44].

String Length Reduction/Extension (SRE). This operator changes the length of a string property by shortening or extending it, simulating truncation or padding errors observed in practice. For example, YouTube bug 141319505 reports: "*I am currently experiencing an issue where the video descriptions returned by the YouTube API are truncated to 150–160 characters.*" A related issue was observed in the New York Times Books API, which returned ISBN10 values of invalid length [44].

B. Mutation Strategies

Mutation operators are typically characterised by three components: the application condition, the transformation rule, and the set of possible replacements. The *application condition* specifies the domain of applicability of the operator, that is, the syntactic pattern or structural element that must be present for the operator to be applied. For example, the application condition of the AER (Array Element Removal) operator is the presence of at least one array element in the JSON payload of the response. The *transformation rule* defines the type of modification the operator introduces, usually falling into broad categories such as adding, deleting, or changing an element. Finally, the *replacement set* enumerates the concrete alternatives that may substitute the original construct when the operator is applied. For instance, the R2XX operator replaces the status code with one of four predefined values (200, 201, 202, or 204), with each alternative producing a distinct mutant. A full specification of each mutation operator, including an explicit definition of its application condition, transformation rule, and replacement set, is provided in the supplementary material [].

Based on the proposed operators, we introduce two strategies for mutant generation, each offering a different balance between thoroughness and cost:

- *Exhaustive generation.* This strategy selects all applicable mutation operators for each element of the HTTP response and applies each of them once using a randomly chosen replacement. As it triggers every applicable operator, this strategy yields a strict oracle adequacy assessment but also leads to a potentially high number of mutants.
- *Random generation.* This strategy randomly selects one applicable mutation operator for each element of the HTTP response and applies it once using a randomly chosen replacement. By applying only a subset of possible operators—similar to selective mutation in classical mutation testing []—it provides a lightweight alternative that significantly reduces the number of generated mutants.

C. Tooling

To make our approach readily applicable, we present HTTPMUTATOR, a tool that implements all HTTP response mutation operators together with the two generation strategies described in Sections III-A and III-B, respectively. HTTPMUTATOR is open-source, written in Java (XX lines of code), available

on GitHub [], and can be imported via Maven. It supports mutating HTTP responses using a configurable set of operators as well as computing the mutation score. Input and output HTTP messages (mutants) are processed in XXX format, enabling straightforward integration with existing testing tools and workflows. The tool is designed to be easily extensible, allowing new mutation operators to be added with minimal effort.

HTTPMUTATOR has proved instrumental in previous studies evaluating the quality of automatically generated test oracles for REST APIs, both in those using dynamic invariants [] and in those based on large language models [].

IV. EVALUATION

Our evaluation seeks to answer the following research questions:

- **RQ1:** *How effective and efficient is* HTTPMUTATOR *in generating meaningful and diverse mutants for REST APIs?* We investigate the quantity and diversity of mutants generated by HTTPMUTATOR across test suites of varying sizes and from different APIs, as well as the time taken to generate them.
- **RQ2:** *To what extent does black-box mutation serve as an effective test adequacy criterion for evaluating REST API test suites?* The goal of our approach is to serve as a proxy of the fault-detection capability of test suites in a black-box setting, where white-box mutation testing is not applicable. We assess the correlation between our proposed black-box mutation score and traditional white-box mutation score.
- **RQ3:** *What black-box mutation score do state-of-the-art REST API test case generators achieve?* We empirically evaluate the fault-detection capability of state-of-the-art REST API test case generators by applying HTTPMUTATOR to their generated test suites and reporting the black-box mutation score. This investigation aims to shed light on the true effectiveness of these tools in detecting faults in REST APIs, and whether their reported effectiveness in current benchmarks is consistent with the results obtained through our black-box mutation testing approach.

In what follows, we first detail the selected APIs subject of our investigation and then answer the research questions, explaining the particular experimental setup for each experiment and the results obtained.

A. Subject APIs

To evaluate HTTPMUTATOR, we curated a benchmark set comprising 20 REST API operations: 10 open-source and 10 industrial (closed-source). This design ensures both diversity and realism, and supports comparison with prior work.

Open-Source APIs. We selected 10 open-source operations from two widely used REST API testing benchmarks: WFD [49] (formerly EMB) and REST-GO [50]. These benchmarks span a range of domains and API complexities. To enable comparison with white-box mutation testing using PIT [8], we restricted our selection to Java-based APIs. From the available pool, we excluded operations that: (i) lacked a defined success response schema (i.e., missing structured responses for HTTP 2XX codes), (ii) were unimplemented, or (iii) duplicated the logic of other operations. We then ranked the remaining candidates by number of input parameters, retained one operations per API for diversity, and selected the top 140.

Industrial APIs. We applied the same criteria to select 10 industrial operations from AGORA+ [44], the largest curated benchmark of industrial REST APIs. AGORA+ includes realistic APIs from services such as iTunes, Spotify, and YouTube. We excluded operations that were deprecated, inaccessible, or had fewer than two input parameters (limiting input coverage variation). This ensured parity in evaluation methodology across open and industrial APIs.

B. Test suites

For each API operation, we constructed three test suites with increasing sizes and coverage strength, labeled as Small, Medium, and Large. To this end, we adopted the black-box test coverage levels for REST APIs proposed by Martin-Lopez et al. [45] and used by different groups of authors [51], [52]. Also, we used Microsoft's combinatorial tool PICT [53], which generates t-way *covering arrays* – minimal sets of test cases that cover all t-way combinations – for the systematic generation of input combinations. Next, we describe the procedure followed for the generation of each test suite.

- **Small.** This test suite targets the TCL 4 coverage level by ensuring that all input parameters and status classes (success and error) are covered at least once. To achieve this, we first defined a set of potential values for each input parameter based on the API specification and its implementation. Parameters of enumerated or boolean types were assigned their complete set of possible values, while other types (e.g., integers, strings) were manually assigned three representative valid values per parameter. Optional parameters additionally included a NULL option to represent their absence. Next, we generated a 1-way covering array so that every value in each parameter's domain was exercised at least once. Then, we applied a greedy selection algorithm to remove redundant tests while still ensuring that each parameter was exercised with at least one non-NULL value. Finally, we added a test case specifically targeting a 4XX status class to fulfill the output coverage requirement.
- **Medium** Using a 1-way covering array, we ensured every value in each parameter's domain was exercised. We augmented the suite with dedicated test cases for all uncovered status codes (e.g., 401, 404).
- **Large.** A 2-way covering array ensured all value pairs across parameters were tested. As with Medium, we included test cases for missing status codes and added further cases to trigger uncovered response properties based on the response schema.

C. Experiment 1: Mutant Generation

In this experiment, we run HTTPMUTATOR and HTTPMUTATOR-RANDOM on the Small, Medium, and

TABLE II: The subject API operations

Source	API / Operation	#Params	Description[lixin: TODO: Confirm whether it is correct]
Web Fuzzing Dataset []	ScoutAPI–CreateActivities	152	Creates activity records in the Scout demo service.
	ProjectSwagger–AssignTask	48	Assigns an existing task to a user.
	UserOpenAPI–UpdateUser	22	Updates profile attributes for a registered user.
	Market–RegisterUser	14	Registers a new marketplace user.
	PersonOpenAPI–CreatePerson	14	Creates a new person entity with contact fields.
	ProxyPrint–RegisterRequest	12	Submits a print job registration request.
	LanguageTool–CheckText	11	Grammar/style checking for input text.
	CatWatch–ListProjects	8	Lists OSS projects for a given organization.
	GestaoHospital–CreateHospital	7	Creates a hospital entry with administrative data.
	GenomeNexus–Annotate	4	Annotates genetic variants with curated knowledge.
Industrial APIs []	Stripe–CreateProduct	38	Creates a product object in Stripe's commerce API.
	Foursquare–SearchPlaces	19	Searches venues by text and geo filters.
	Yelp–SearchBusinesses	14	Returns businesses matching query and location.
	AmadeusHotel–GetOffers	13	Retrieves hotel offer availability and pricing.
	YouTube–GetVideos	12	Fetches video metadata by IDs or queries.
	DHL–FindByAddress	11	Address-based location/point-of-service lookup.
	FDIC–ListInstitutions	10	Lists insured financial institutions.
	ohsome–GetElements	10	Extracts OSM elements via spatial/temporal filters.
	DeutscheBahn–ListStations	8	Lists railway stations and metadata.
	iTunes–Search	8	Searches the iTunes/Apple media catalog.

Large test suites for all 20 API operations, which vary in size and thoroughness, to answer RQ1 by assessing the number and diversity of the mutants generated by HTTPMUTATOR.

Table III reports the results for all API operations, organized into three subtables corresponding to the Small, Medium, and Large test suites. Each row corresponds to one API operation and shows the number of test cases in the suite (#TC), the total number of response fields observed across all responses (#RF), and the number of mutants generated by HTTPMUTATOR. The mutants are further broken down into status code operators, header operators, and JSON payload operators. The final row of each table aggregates the totals across all API operations, reporting the overall number of test cases, response fields, and mutants for each operator category.

Across the three subtables, the number of mutants produced by HTTPMUTATOR primarily grows with the size of the test suite. Figure 1 shows a clear upward trend from the Small to the Medium and Large suites for almost all API operations, with Large suites often yielding substantially more mutants than Small ones. This is consistent with our black-box design, as mutants are generated from concrete responses observed during test execution, so adding more tests naturally exposes more mutation sites. In addition, the tables indicate that, for comparable numbers of test cases, operations whose responses contain more distinct fields (#RF) tend to produce somewhat more mutants.

We observe a highly consistent pattern in how HTTPMUTATOR uses its mutation operators, for almost all API operations. For the Large test suite, Figure 2 shows that HTTPMUTATOR relies mainly on the same small subset of payload-related operators (e.g., PTC, NPS, SPR, SCA, SRE), which together account for around 77.3% of all mutants on average across API operations. In contrast, status-code operators represent only about 2.8% of mutants, and header operators contribute less than 3.3%. Figure 3 aggregates operator usage across all 20 APIs under the Large test suites and confirms the same picture in absolute terms: the vast majority of mutants stem from JSON payload transformations, while header- and status-code mutations play a comparatively minor role. This behavior is expected, as header and status-code operators are fewer in number and applicable at fewer locations, whereas JSON payloads typically contain many fields, heterogeneous types, and nested structures, thereby exposing far more mutation sites. Overall, these results indicate that HTTPMUTATOR is particularly effective at producing diverse mutants for rich JSON payloads, which are common in web APIs.

Beyond the number of mutants and their distribution across mutation operators, HTTPMUTATOR aims to generate a diverse set of mutants while, by design, avoiding equivalent or duplicate ones. Each mutation operator introduces an unique and observable change to the HTTP response, and the implementation avoids applying the same operator configuration twice to the same response field. As a sanity check, we inspected all generated mutants and confirmed that no operator configuration was reapplied to the same field or produced an equivalent mutant. Only a few identical HTTP responses do appear due to random value collisions. For example, in the GenomeNexus-Annotate API operation, the field `"variant_allele"` often contains a single-character nucleotide (e.g., `"T"`); when mutation operator SPR replaces this field with a boundary string of length one, the randomly generated character may coincidentally match the original value, yielding identical responses. Such cases are rare and do not affect any aggregate results. Overall, the changes introduced by HTTPMUTATOR are small and localized, consistent with the fine-grained perturbations expected in mutation testing.

We further measured the time required for HTTPMUTATOR to generate all mutants for each API operation. Across the 20 API operations, the generation time ranges from 0.33 seconds (Market-RegisterUser) to 2491.51 seconds (iTunes-Search). Most APIs complete mutant generation within a few seconds (11 out of 20 operations require less than 3 seconds), whereas a small number of APIs that return very large and complex JSON payloads (e.g., YouTube-

TABLE III: Number of mutants generated by HTTPMUTATOR across API operations

(a) Small Test Suite

Operation	#TC	#RF	STATUS CODE			HEADERS				JSON PAYLOAD													
			R2XX	R4XX	R5XX	CTC	CTD	CPC	CPD	AEA	AER	AEE	EAS	OPA	OPR	OTPR	PTC	BPR	NPS	NPR	SCA	SPR	SRE
ScoutAPI-CreateActivities	4	170	4	4	4	4	4	4	0	15	0	0	11	22	22	7	158	4	158	70	51	51	51
ProjectSwagger-AssignTask	2	67	2	2	2	2	2	2	0	0	0	0	0	11	11	5	61	2	58	13	34	34	34
UserOpenAPI-UpdateUser	3	68	3	3	3	3	3	3	0	4	2	2	4	7	7	2	59	4	48	2	34	34	34
Market-RegisterUser	2	21	2	2	2	2	2	2	0	1	0	0	1	6	4	2	15	0	15	0	10	10	10
PersonOpenAPI-CreatePerson	2	35	2	2	2	2	2	2	0	2	0	0	2	6	6	2	29	2	25	5	12	12	12
ProxyPrint-RegisterRequest	2	22	2	2	2	2	2	2	2	0	0	0	0	2	2	0	16	0	16	4	12	12	12
LanguageTool-CheckText	4	91	4	4	4	3	3	3	0	4	1	1	2	22	22	8	81	7	78	11	37	37	37
CatWatch-ListProjects	2	11	2	2	2	2	2	2	2	1	0	0	0	1	1	0	5	0	5	1	4	4	4
Gestaohospital-CreateHospital	2	16	2	2	2	2	2	2	2	0	0	0	0	2	2	0	10	0	10	3	7	7	7
GenomeNexus-Annotate	2	1159	2	2	2	1	1	1	0	175	11	28	78	145	145	18	1155	5	1146	38	784	784	784
Stripe-CreateProduct	4	131	4	4	4	4	4	4	0	9	5	5	6	34	31	4	119	8	116	8	61	61	61
Foursquare-SearchPlaces	4	304	4	4	4	3	3	3	3	13	6	6	11	68	68	38	293	0	293	29	187	187	187
Yelp-SearchBusinesses	3	24	3	3	3	3	3	3	0	2	0	0	0	8	8	5	15	0	15	6	2	2	2
AmadeusHotel-GetOffers	3	940	3	3	3	3	3	3	0	41	20	20	41	298	298	121	931	4	931	63	528	528	528
YouTube-GetVideos	4	128	4	4	4	4	4	4	4	6	2	2	5	39	39	14	116	5	116	7	63	63	63
DHL-FindByAddress	2	10	2	2	2	2	2	2	0	1	0	0	0	2	2	0	4	0	4	1	2	2	2
FDIC-ListInstitutions	2	34	2	2	2	2	2	2	2	2	0	0	1	12	12	5	28	0	28	3	13	13	13
Ohsome-GetElements	4	44	4	4	4	4	4	4	0	3	0	0	3	11	11	3	32	0	32	5	17	17	17
DeutscheBahn-ListStations	3	11618	3	3	3	3	3	3	3	619	337	337	619	2183	2175	531	11609	2452	11609	1695	4663	4663	4663
iTunes-Search	2	245	2	2	2	2	2	2	2	13	13	13	13	9	9	1	239	0	239	44	175	175	175
Total	56	15138	56	56	56	53	53	53	20	911	397	414	797	2888	2875	766	14975	2493	14942	2008	6696	6696	6696

(b) Medium Test Suite

Operation	#TC	#RF	STATUS CODE			HEADERS				JSON PAYLOAD													
			R2XX	R4XX	R5XX	CTC	CTD	CPC	CPD	AEA	AER	AEE	EAS	OPA	OPR	OTPR	PTC	BPR	NPS	NPR	SCA	SPR	SRE
ScoutAPI-CreateActivities	6	226	6	6	6	6	6	6	0	20	0	0	14	29	29	10	209	6	209	93	66	66	66
ProjectSwagger-AssignTask	9	331	9	9	9	8	8	8	0	0	0	0	0	52	52	26	306	10	283	68	161	161	161
UserOpenAPI-UpdateUser	10	266	10	10	10	10	10	10	0	16	9	9	16	26	26	8	236	16	199	8	143	143	143
Market-RegisterUser	4	43	4	4	4	4	4	4	0	1	0	0	1	14	8	6	31	0	31	0	20	20	20
PersonOpenAPI-CreatePerson	4	69	4	4	4	4	4	4	0	4	0	0	4	12	12	4	57	3	50	11	24	24	24
ProxyPrint-RegisterRequest	7	92	7	7	7	7	7	7	7	0	0	0	0	7	7	0	71	0	70	19	51	51	51
LanguageTool-CheckText	6	133	6	6	6	5	5	5	0	6	1	1	2	32	32	12	117	11	112	15	53	53	53
CatWatch-ListProjects	15	140	15	15	15	15	15	15	15	24	1	1	1	6	6	0	95	0	65	31	19	19	19
Gestaohospital-CreateHospital	5	43	5	5	5	5	5	5	5	0	0	0	0	5	5	0	28	0	28	9	19	19	19
GenomeNexus-Annotate	9	5421	9	9	9	8	8	8	0	965	53	181	481	533	533	59	5396	31	5365	250	3594	3594	3594
Stripe-CreateProduct	18	656	18	18	18	18	18	18	0	48	25	25	29	169	153	18	602	44	576	52	281	281	281
Foursquare-SearchPlaces	14	407	14	14	14	13	12	13	12	22	6	6	11	114	114	74	366	0	366	56	188	188	188
Yelp-SearchBusinesses	7	2242	7	7	7	7	7	7	0	131	94	94	125	402	402	38	2221	261	2150	336	1027	1027	1027
AmadeusHotel-GetOffers	6	1429	6	6	6	6	6	6	0	70	32	32	70	433	433	170	1411	16	1411	106	792	792	792
YouTube-GetVideos	45	11665	45	45	45	45	45	45	45	250	226	226	248	2562	2386	782	11530	526	11530	1141	7096	7096	7096
DHL-FindByAddress	27	6014	27	27	27	27	27	27	0	530	221	221	309	1329	1329	202	5933	0	5933	305	3796	3796	3796
FDIC-ListInstitutions	17	931	17	17	17	17	17	17	17	18	2	2	7	134	134	58	880	0	880	287	458	458	458
Ohsome-GetElements	10	118	10	10	10	10	10	10	0	9	0	0	9	31	31	9	88	0	88	13	45	45	45
DeutscheBahn-ListStations	7	15442	7	7	7	7	7	7	5	828	446	446	828	2881	2872	703	15421	3277	15421	2270	6172	6172	6172
iTunes-Search	31	12821	31	31	31	31	31	31	31	529	311	311	400	378	378	1	12728	141	12728	2022	9689	9689	9689
Total	257	58489	257	257	257	253	252	253	137	3471	1427	1555	2555	9149	8942	2180	57726	4342	57495	7092	33694	33694	33694

(c) Large Test Suite

Operation	#TC	#RF	STATUS CODE			HEADERS				JSON PAYLOAD													
			R2XX	R4XX	R5XX	CTC	CTD	CPC	CPD	AEA	AER	AEE	EAS	OPA	OPR	OTPR	PTC	BPR	NPS	NPR	SCA	SPR	SRE
ScoutAPI-CreateActivities	31	1521	31	31	31	31	31	31	0	135	0	0	93	194	194	71	1429	38	1429	642	450	450	450
ProjectSwagger-AssignTask	44	1926	44	44	44	43	43	43	0	0	0	0	0	303	303	154	1796	60	1661	404	937	937	937
UserOpenAPI-UpdateUser	52	1622	52	52	52	52	52	52	0	100	57	57	100	152	152	50	1466	100	1245	50	895	895	895
Market-RegisterUser	15	164	15	15	15	15	15	15	0	1	0	0	1	58	30	28	119	0	119	0	75	75	75
PersonOpenAPI-CreatePerson	35	587	35	35	35	35	35	35	0	32	0	0	32	102	102	35	482	22	428	94	213	213	213
ProxyPrint-RegisterRequest	28	386	28	28	28	28	28	28	28	0	0	0	0	28	28	0	302	0	295	82	213	213	213
LanguageTool-CheckText	30	1117	30	30	30	29	29	29	0	45	12	12	27	294	294	90	1029	74	1000	138	478	478	478
CatWatch-ListProjects	155	1298	155	155	155	155	155	155	155	246	4	4	25	47	47	0	833	0	557	277	142	142	142
Gestaohospital-CreateHospital	19	169	19	19	19	19	19	19	19	0	0	0	0	19	19	0	112	0	112	37	75	75	75
GenomeNexus-Annotate	41	27562	41	41	41	40	40	40	0	4546	244	871	2333	2703	2703	367	27441	149	27286	1900	18028	18028	18028
Stripe-CreateProduct	228	8966	228	228	228	228	228	228	0	678	374	374	414	2303	2077	228	8282	604	7875	620	3898	3898	3898
Foursquare-SearchPlaces	130	9902	130	130	130	129	128	129	128	410	105	105	302	2394	2151	1186	9513	0	9513	1135	5704	5704	5704
Yelp-SearchBusinesses	49	32826	49	49	49	49	49	49	0	1963	1323	1323	1873	5754	5754	480	32679	3721	31394	4865	15140	15140	15140
AmadeusHotel-GetOffers	30	29350	30	30	30	30	30	30	0	1807	747	747	1807	8949	8944	2895	29252	359	29252	1652	16515	16515	16515
YouTube-GetVideos	333	144546	333	333	333	333	333	333	333	3000	2799	2799	2992	29624	27461	8440	143547	6141	143547	14577	90538	90538	90538
DHL-FindByAddress	186	29169	186	186	186	186	186	186	0	2584	1137	1137	1468	6475	6475	960	28611	0	28611	1442	18296	18296	18296
FDIC-ListInstitutions	201	73472	201	201	201	201	201	201	201	507	33	33	363	3259	3259	1408	72869	0	72869	23491	45813	45813	45813
Ohsome-GetElements	47	589	47	47	47	47	47	47	0	46	0	0	46	158	158	46	448	0	448	66	225	225	225
DeutscheBahn-ListStations	27	87367	27	27	27	27	27	27	25	4708	2584	2584	4708	16307	16245	4002	87286	18558	87286	12888	34852	34852	34852
iTunes-Search	678	436212	678	678	678	678	678	678	678	17526	10954	10954	14027	10882	10882	1	434178	4410	434178	62606	339432	339432	339432
Total	2359	888751	2359	2359	2359	2355	2354	2355	1567	38334	20373	21000	30611	90005	87278	20441	881674	34236	879105	126966	591919	591919	591919

GetVideos, FDIC-ListInstitutions, DeutscheBahn-ListStations, iTunes-Search) form a long tail with higher generation times. These results indicate that HTTPMUTATOR is fast for typical API workloads and remains practical even for APIs with extremely large responses, where the higher cost simply reflects the larger number of mutation sites. In contrast to white-box mutation testing, which invokes the API system under test for each program mutant and thus incurs substantial computational overhead, HTTPMUTATOR operates purely on previously recorded HTTP responses. Mutant generation requires no further interaction with the API and consists only of in-memory transformations, keeping its cost low. Furthermore, mutant generation on different responses can be easily parallelized, allowing large numbers of mutants to be produced with modest resources.

We also compare HTTPMUTATOR with its variant HTTPMUTATOR-RANDOM to assess the effect of HTTPMUTATOR's mutant generation strategy. Fig. 4 plots, for each API operation, the ratio HTTPMUTATOR-RANDOM / HTTPMUTATOR in terms of the number of mutants (Fig. 4a) and the generation time (Fig. 4b); the operations on the x-axis are sorted by the number of mutants produced by HTTPMUTATOR

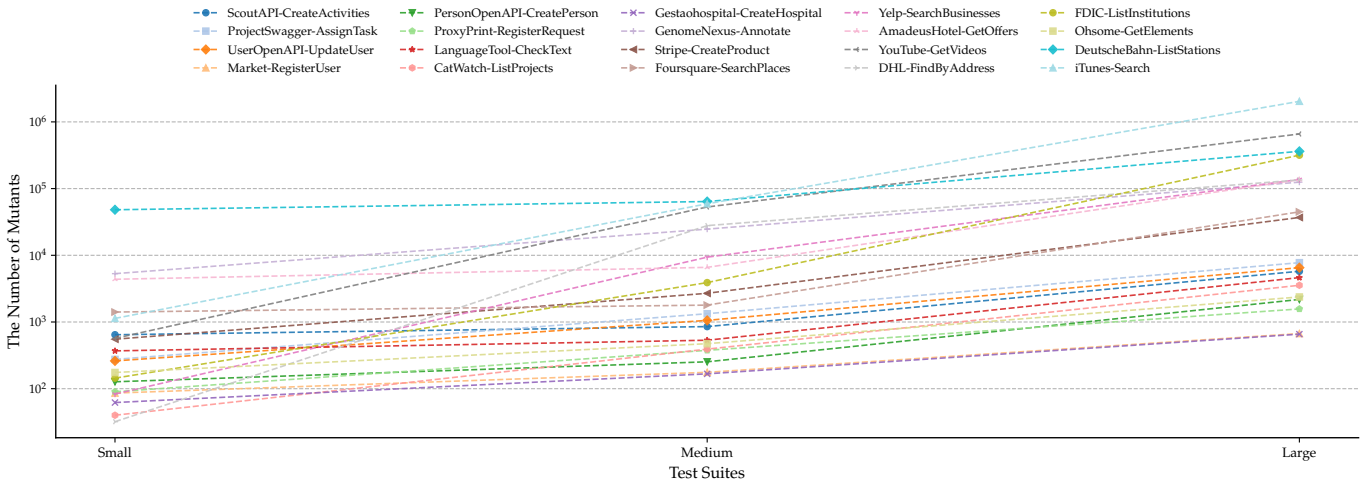


Fig. 1: Number of mutants generated by HTTPMUTATOR for each API operation under the Small, Medium, and Large test suites.

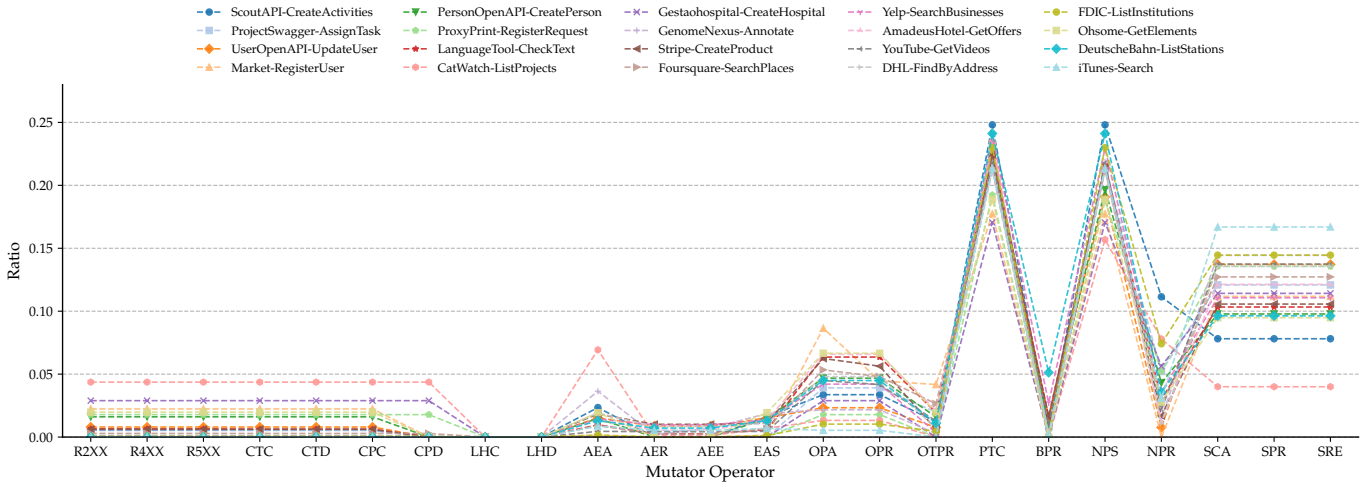


Fig. 2: Proportion of HTTPMUTATOR mutation operators used per API operation in the Large test suite.

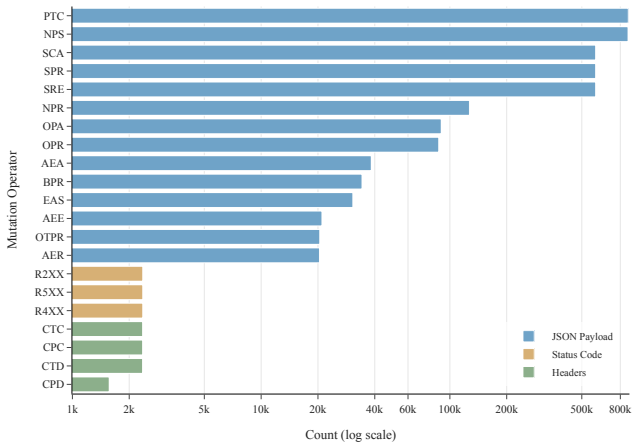
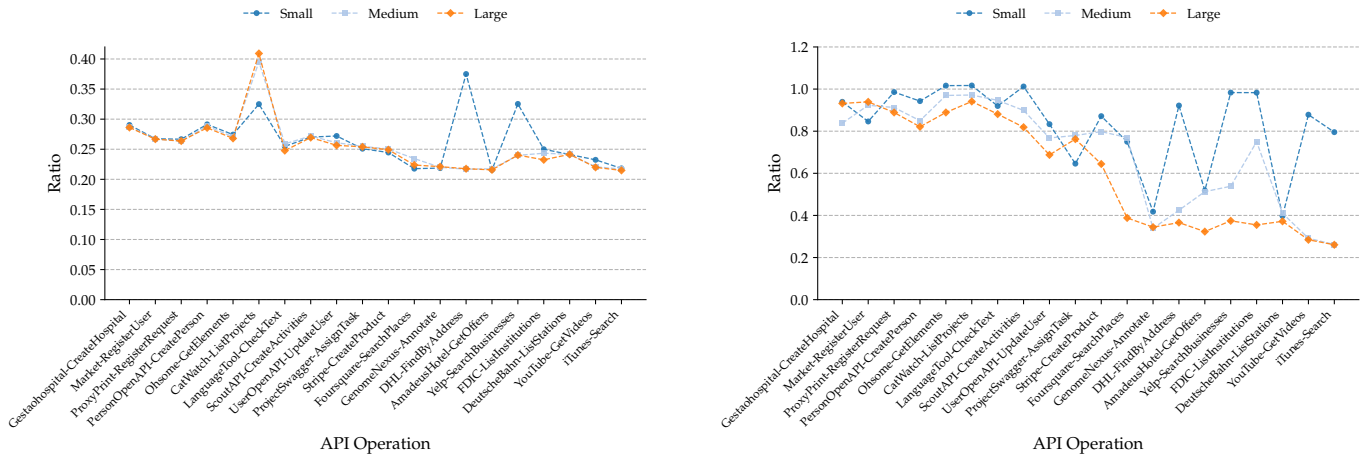


Fig. 3: Mutation operator usage by HTTPMUTATOR in the Large test suites across 20 API operations.

under the Large test suite. For 17 out of 20 operations, the mutant-count ratios are consistent across the three test suites and lie between 20% and 30%. The remaining three operations show higher and less stable ratios because their test suites contain many `null` values or empty lists, for each of which HTTPMUTATOR produces at most a single mutant, so HTTPMUTATOR-RANDOM cannot reduce the number of mutants in those cases. For the Large test suite, the time-cost ratios exhibit a similar trend: when HTTPMUTATOR completes in only a few seconds, the ratios are close to 1, but as the number of mutants (and thus the absolute generation time) increases, the ratio decreases and approaches about 0.2.



(a) Ratio for mutant count

(b) Ratio for time cost.

Fig. 4: Ratios of HTTPMUTATOR-RANDOM / HTTPMUTATOR for each API operation across test suites. The operations on the x-axis are sorted by the number of mutants generated by HTTPMUTATOR under the Large test suite.

Answer to RQ1

HTTPMUTATOR is highly effective at generating large and diverse sets of mutants of web API responses. Over a set of test suites, HTTPMUTATOR generated XXXXX mutants in total, XXX mutants per test suite on average, taking an average of XX seconds per test suite. The generated mutants are non-equivalent, unique and diverse, including instances that resemble real-world bugs.

D. Experiment 2: Effectiveness as a Test Adequacy Criterion

This experiment assesses whether the black-box mutation score produced by HTTPMUTATOR can function as a practical test adequacy criterion in settings where traditional white-box metrics are unavailable. We measure the association between HTTPMUTATOR 's mutation scores and established white-box adequacy measures (i.e., line coverage and white-box mutation scores) computed over the *same* test suites. As the baseline white-box mutation tool, we use PITEST [8] due to its maturity, maintenance, and widespread adoption. Strong positive correlation would indicate that HTTPMUTATOR 's black-box score can substitute for white-box adequacy when source access is not feasible.

1) Experimental Setting: We applied this experiment to the 10 open-source APIs in our benchmark. For each API operation, we used the comprehensive Large test suite (Section IV-B), selected for its high input–output coverage and representativeness, to study the association between HTTPMUTATOR and white-box adequacy measures across different oracle levels.

All experiments were conducted on a dedicated server equipped with an Intel(R) Xeon(R) Gold 5118 CPU @ 2.30 GHz (48 cores), 128 GB RAM, and running CentOS 7. To ensure reproducibility and test isolation, for APIs backed by a database, we reinitialized the database before every test execution. This procedure ensured a clean and consistent system state for those APIs across all runs.

Oracles. We implemented four cumulative test oracles for REST APIs, aligned with the categories in Section II-B: the *Crash oracle* (CR), the *Specification oracle* (SC), the *Semantic oracle* (SE), and the *Regression oracle* (RG). Each oracle contains all checks performed by the preceding ones and detects deviations in API responses triggered by mutant execution.

- **Crash oracle (CR).** The CR oracle flags a test case as failing if the response status code indicates a server error (5XX). Detection is performed by parsing HTTP responses and checking the status code.
- **Specification oracle (SC).** The SC oracle extends CR by additionally validating responses against the API specification. Concretely, we use swagger-request-validator-core [54] to check the response object structure, presence of mandatory fields, and type correctness against the OpenAPI specification. To reduce false positives from underspecified or incorrect schemas, we first executed Large test suite on the *unmutated* API and reconciled the specification with observed correct behavior (e.g., optional fields, value ranges, formats). These edits were minimal and evidence-based, and applied once before mutation analysis. Under SC, a test is marked as failing if it either (i) triggers a 5XX response or (ii) violates the reconciled specification.
- **Semantic oracle (SE).** The SE oracle extends SC with semantic checks based on *invariants*—properties describing stable input–output relations of the API operation. Conceptually, these invariants capture domain-level semantics such as value monotonicity, field dependencies, collection-size constraints, or cross-field consistency (e.g., the size of an `items` array matching a `total_items` field). Under the SE oracle, a test is marked as failing if it (i) triggers a 5XX response, (ii) violates the reconciled API specification, or (iii) violates any semantic invariant.

Realizing SE with AGORA+. To operationalize these invariants, we use AGORA+'s [44] pipeline, which automatically mines invariants from observed API executions and turns them into executable assertions. AGORA+ integrates (i) Beet [55] to infer invariants over request parameters and response fields from execution traces, and (ii) PostmanAssertify [56] to translate each mined invariant into an assertion script applicable to arbitrary tests.

Trace collection and invariant mining. Following AGORA+'s recommended configuration, we used RESTest [24] to automatically generate $10k$ requests per API operation. Executing these requests on the unmutated API produced traces from which Beet inferred candidate invariants. PostmanAssertify instantiated these invariants into executable assertions.

Stability filtering. To avoid flaky oracles, we validated all candidate invariants using the Large suite: any invariant not consistently satisfied by all Large executions on the unmutated API was discarded. The remaining stable invariants constitute the SE oracle used during mutation analysis.

- **Regression oracle (RG).** The RG oracle extends SE with regression checking via output comparison. We use JSONassert [57] to compare JSON responses. We configure JSONassert in non-strict mode so that differences in array order are not treated as failures as long as the array contains the same items. We observed this behavior in our benchmark: the Yelp-SearchBusinesses operation can return the same set of transactions items in different orders for identical requests. To address other sources of non-determinism, each request was executed twice on the original API to identify systematically varying fields (e.g., timestamps, UUIDs), which were then excluded from further comparisons. During mutation analysis, a test case is marked as failing under RG if it (i) triggers a 5XX response, (ii) violates the reconciled specification, (iii) violates any retained invariant, or (iv) exhibits any other field difference between the mutated and baseline response.

White-box analysis via PITEST. We used PITEST [8] to obtain white-box mutation scores and line coverage. Because our tests exercise only one API operation rather than the entire backend, substantial portions of the source code are never executed. To avoid polluting the mutation scores with mutants in such unreachable code, we first ran PITEST on the Large suite in coverage-gathering mode to identify which classes and methods are not executed when invoking the selected endpoints. Based on this coverage information, we then manually configured `excludedClasses` and `excludedMethods` in PITEST to exclude these unexecuted classes and methods from mutant generation. This prevents PITEST from generating mutants in code never reach, and therefore restricts the white-box analysis to mutants in the parts of the code that are involved in handling the tested API operation.

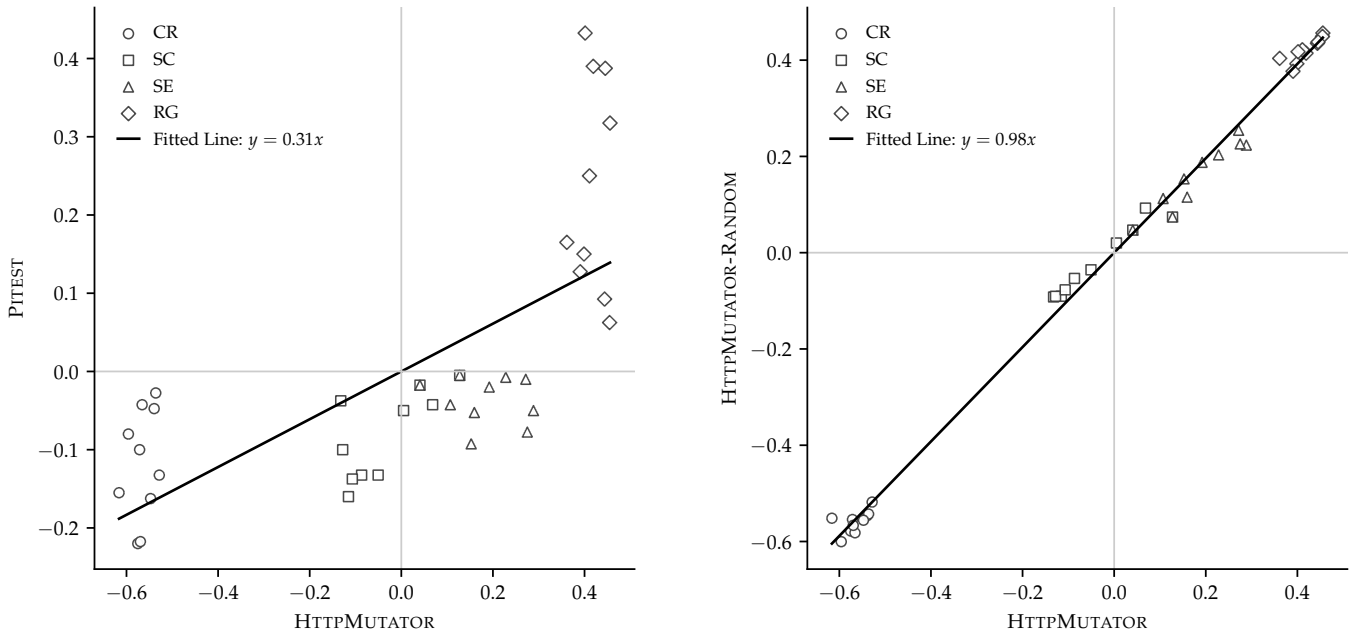
After this filtering step, we ran PITEST to obtain mutation scores and line coverage for each of our four oracle configurations (CR, SC, SE, and RG) on the same Large suite. To speed up execution, we enabled `auto_threads` in PITEST, allowing it to parallelize workers based on the available CPU cores.

2) Experimental Results: Figure 5 visualizes the relationship between mutation scores produced by HTTPMUTATOR, PITEST, and HTTPMUTATOR-RANDOM on the Large suite. For each API operation, we first compute the average mutation score of each tool across the four oracle configurations (CR, SC, SE, and RG), and subtract this per-operation average from the operation's four scores. The plotted values therefore represent how much each oracle-level score deviates from the tool's operation-specific mean, allowing us to compare relative adequacy trends between tools independently of differences in their absolute score scales. We then flatten these centered scores across operations and oracle levels to compute a global Spearman's rank correlation coefficient (ρ) and fit a linear regression.

The comparison between HTTPMUTATOR and PITEST (Fig. 5a) reveals a clear positive correlation relationship between HTTPMUTATOR and PITEST. While the fitted line exhibits a relatively small slope (0.31), the rank correlation remains strong ($\rho = 0.81$, $p < 0.001$), suggesting that HTTPMUTATOR tends to assign higher-or-lower mutation score to operations in broadly the same order as PITEST. This alignment is noteworthy because the two tools operate under fundamentally different assumptions: PITEST perturbs internal program structures, whereas HTTPMUTATOR manipulates externally visible response behaviors. The fact that they nevertheless yield correlated patterns indicates that black-box mutation retains meaningful information about the underlying fault-detection capability of a test suite, supporting that HTTPMUTATOR can act as an effective test adequacy measure when white-box mutation is not applicable.

The relationship between HTTPMUTATOR and its variant HTTPMUTATOR-RANDOM (Fig. 5b) is almost perfectly linear. After centering, the points fall tightly around the diagonal with an estimated slope close to one (0.98) and a near-ideal rank correlation ($\rho = 0.99, p < 0.001$).It shows that the reductions introduced in HTTPMUTATOR-RANDOM– primarily aimed at lowering the number of generated mutants and improving efficiency – do not alter the relative adequacy trends across API operations. Consequently, HTTPMUTATOR-RANDOM provides a more efficient replacement for HTTPMUTATOR when test resource is a concern.

Table IV summarizes the corresponding execution time of mutation testing. Overall, HTTPMUTATOR completes mutation analysis within only a few seconds for most API operations across all oracle configurations. The primary exception is GenomeNexus-Annotate, where HTTPMUTATOR requires noticeably more time. This behavior is expected: the Large test suite for this operation contains 27562 response fields, which results in 124870 generated mutants, as shown in Table IIIc, far exceeding the scale of the other operations. The increased number of mutants directly translates into longer execution time for HTTPMUTATOR, while the remaining operations remain lightweight. Even in this extreme case, however, HTTPMUTATOR is still substantially faster than PITEST under the CR, SC, and RG configurations.



(a) HTTPMUTATOR vs PITEST (b) HTTPMUTATOR vs HTTPMUTATOR-RANDOM

Fig. 5: Correlation between HTTPMUTATOR, PITEST and HTTPMUTATOR-RANDOM mutation scores after removing operation-level mean differences. Spearman rank correlations: (a) $\rho = 0.81$ ($p < 0.001$), (b) $\rho = 0.99$ ($p < 0.001$).

TABLE IV: Execution time (seconds) of mutation testing on TCL6 test suites with HTTPMUTATOR (HM), HTTPMUTATOR-RANDOM (HM′), and PITEST (PIT), across four oracle levels.

Operation	CR			SC			SE			RG		
	HM	HM′	PIT	HM	HM′	PIT	HM	HM′	PIT	HM	HM′	PIT
ScoutAPI-CreateActivities	1.27	1.04	7386.00	4.33	2.39	6586.00	141.40	38.33	10630.00	2.55	1.55	6020.00
ProjectSwagger-AssignTask	1.01	0.77	639.00	4.83	2.45	615.00	369.51	89.95	1860.00	1.34	0.81	1648.00
UserOpenAPI-UpdateUser	1.12	0.77	12119.00	4.28	2.33	11694.00	126.59	31.27	11071.00	1.83	1.03	3130.00
Market-RegisterUser	0.33	0.31	498.00	1.77	1.16	358.00	5.70	4.04	679.00	0.35	0.31	320.00
PersonOpenAPI-CreatePerson	0.67	0.55	223.00	1.75	1.07	233.00	31.25	10.14	1613.00	1.05	0.72	155.00
ProxyPrint-RegisterRequest	0.99	0.88	1265.00	1.71	1.13	1276.00	20.48	7.55	2177.00	0.96	0.89	1048.00
LanguageTool-CheckText	1.43	1.26	14963.00	3.77	2.16	15074.00	92.84	25.06	24244.00	2.32	1.55	11908.00
CatWatch-ListProjects	1.19	1.12	4230.00	2.65	1.85	4168.00	28.54	11.44	9905.00	1.34	1.20	1956.00
Gestaohospital-CreateHospital	0.44	0.41	922.00	0.90	0.59	941.00	8.70	4.27	1358.00	0.50	0.42	368.00
GenomeNexus-Annotate	40.29	13.87	5082.00	318.45	75.13	4902.00	13730.86	2866.10	13286.00	250.59	63.25	2815.00

When comparing HTTPMUTATOR with PITEST more broadly, we observe that PITEST requires several orders of magnitude more time than HTTPMUTATOR for most API operations across all oracle levels. This difference stems from the execution model of program-level mutation: each PITEST mutant must be compiled, loaded, and exercised by running the service and interacting with the API for that single mutant. As the number of mutants grows, this per-mutant execution pattern leads to long analysis times and sustained CPU usage during mutation testing. In contrast, HTTPMUTATOR applies mutations on recorded HTTP responses and can exercise many mutants without repeatedly restarting or reconfiguring the service, resulting in much lower time and resource consumption.

Execution time increases for all tools under the SE oracle. This slowdown is caused by AGORA+, which relies on Postman to execute assertions. When the number of mutants becomes large, Postman's oracle evaluation becomes a dominant cost, making mutation testing slower regardless of the underlying mutation engine.

Finally, HTTPMUTATOR-RANDOM further reduces time cost compared with HTTPMUTATOR due to the smaller number of generated mutants, while preserving the adequacy trends observed earlier. This makes HTTPMUTATOR-RANDOM a practical choice for large-scale or frequent mutation-based assessment of REST API test suites, where both effectiveness and efficiency are required.

Answer to RQ2

HTTPMUTATOR exhibits adequacy trends consistent with PITEST across varying oracle strengths, enabling effective adequacy assessment in black-box settings when source-based tools are not applicable. Since HTTPMUTATOR-RANDOM preserves these trends while being more efficient, it can be used as the representative instance of our black-box mutation testing approach.

E. Experiment 3: Fault Detection Capability

This experiment evaluates the effectiveness of existing REST API testing approaches using HTTPMUTATOR as a black-box adequacy criterion. We assess their effectiveness in both the process of fuzzing-based testing, which reflects a tool's ability to detect faults during execution, and the generated test suites, which represent its adequacy and reusability for continuous or regression testing. This two-aspects evaluation provides a comprehensive view of how black-box mutation testing captures the quality of automated REST API testing.

To study these two stages in practice, we selected SCHEMATHESIS and EVOMASTER, two representative and actively maintained REST API testing tools. We chose them based on their maturity, continued maintenance, and demonstrated performance in recent studies. SCHEMATHESIS provides a Python interface with flexible hooks that facilitate monitoring during fuzzing, capturing generated test cases, and accessing oracle during execution, which makes it suitable for studying the testing process stage. EVOMASTER is designed to generate executable test suites for regression and continuous testing. It supports multiple oracle types and produces reusable artifacts, which makes it appropriate for studying the post-generation stage.

1) Experimental Setup: Each selected tool was executed on every API operation in our benchmark to evaluate its testing effectiveness under our black-box mutation framework. For each API, we ran every configuration of each tool with a fixed request budget of 1000. In the case of SCHEMATHESIS, this corresponds to executing all 1000 fuzzing-generated requests, each treated as an individual test case during the fuzzing process. The evaluation for SCHEMATHESIS focuses on the collective effectiveness of these executions, reflecting how well SCHEMATHESIS detects faults and exercises diverse input–output behaviors during fuzzing. For EVOMASTER, the same budget 1000 determines the search effort for test generation, and the evaluation focuses on the final minimized and executable test suite produced after generation. For each run, we analyzed the resulting executions or test suites using three metrics:

- **Input and Output Coverage**: measured using RESTATS, capturing how thoroughly the test cases exercised request parameters and response structures.
- **Faults Detected**: defined as test executions that triggered unexpected or erroneous API behaviors (e.g., status codes inconsistent with the API specification or failed assertions). **[lixin: formally define what is a fault in this context]**
- **Black-Box Mutation Score**: computed using HTTPMUTATOR, quantifying the adequacy of each test suite or execution trace with respect to the mutants derived from the observed HTTP responses.

Because both tools expose multiple configuration options that directly affect their ability to detect faults and generate adequate tests, we executed each configuration independently on every API operation and compared their results across the three evaluation metrics.

EVOMASTER *Configurations* EVOMASTER was configured to generate executable test suites under different oracle strengths in order to study how the type and strength of response validation affect the adequacy of the resulting tests. All configurations used the same input-generation heuristics, with coverage-guided search enabled to maximize input diversity and the range of exercised API behaviors. We defined four configurations, corresponding to progressively stronger oracle levels:

- **EM-Base**: This configuration relies on EVOMASTER 's default functional testing mode, in which only basic behavioral checks are performed. For collection-type fields, only the size of collections is asserted.
- **EM-Schema**: It extends the baseline by enabling response validation against the OAS.
- **EM-Security**: Extends the schema configuration with an additional security testing phase that targets access control and authentication vulnerabilities. This phase examines forbidden operations, existence leakage, and authentication bypass issues by generating test cases with different user credentials and analyzing response patterns. It augments the suite with additional negative and privilege-based test cases.
- **EM-AssertAll**: Builds upon the EM-Security configuration by asserting all elements within collections.

SCHEMATHESIS *Configurations*

2) Experimental Results:

V. DISCUSSION

VI. THREATS TO VALIDITY (ABS)

VII. CONCLUSIONS

APPENDIX

REFERENCES

[1] Postman, Inc. (2025) Postman public api network. [Online]. Available: https://www.postman.com/explore
[2] APIs-guru. (2025) Openapi directory: Wikipedia for web apis. [Online]. Available: https://github.com/APIs-guru/openapi-directory
[3] RapidAPI, Inc. (2025) Rapidapi public api marketplace. [Online]. Available: https://rapidapi.com/public/
[4] R. T. Fielding and R. N. Taylor, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, 2000, aAI9980887.
[5] R. DeMillo, R. Lipton, and F. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, no. 4, pp. 34–41, 1978.

TABLE V: TCL-6 mutation testing results: number of test cases (#TC), line coverage (LC) on mutated classes measured by PITEST (PIT), mutants generated (#M), and mutation score by HTTPMUTATOR (HM), HTTPMUTATOR-RANDOM (HM′), and PITEST (PIT) across four oracle levels

Operation	#TC	LC (%)	CR			SC			SE			RG		
			HM	HM′	PIT	HM	HM′	PIT	HM	HM′	PIT	HM	HM′	PIT
ScoutAPI-CreateActivities	31	45	0.01	0.01	0.22	0.61	0.63	0.25	0.79	0.80	0.28	1.00	1.00	0.45
ProjectSwagger-AssignTask	44	78	0.01	0.01	0.07	0.47	0.50	0.13	0.85	0.84	0.28	1.00	1.00	0.68
UserOpenAPI-UpdateUser	52	52	0.01	0.01	0.02	0.42	0.47	0.03	0.78	0.76	0.06	0.99	0.99	0.16
Market-RegisterUser	15	64	0.02	0.04	0.40	0.77	0.67	0.55	0.77	0.67	0.55	1.00	1.00	0.72
PersonOpenAPI-CreatePerson	35	54	0.02	0.03	0.05	0.46	0.49	0.05	0.70	0.66	0.13	1.00	1.00	0.50
ProxyPrint-RegisterRequest	28	30	0.02	0.02	0.05	0.46	0.49	0.05	0.88	0.80	0.10	1.00	1.00	0.40
LanguageTool-CheckText	30	57	0.01	0.01	0.10	0.58	0.60	0.11	0.58	0.60	0.11	1.00	1.00	0.19
CatWatch-ListProjects	155	73	0.04	0.04	0.17	0.68	0.72	0.17	0.72	0.74	0.17	1.00	1.00	0.34
Gestaohospital-CreateHospital	19	44	0.03	0.02	0.15	0.49	0.51	0.23	0.87	0.81	0.29	1.00	1.00	0.80
GenomeNexus-Annotate	41	54	0.00	0.00	0.02	0.50	0.52	0.05	0.70	0.71	0.09	0.99	0.99	0.57

[6] A. B. Sánchez, P. Delgado-Pérez, I. Medina-Bulo, and S. Segura, "Mutation testing in the wild: findings from github," *Empirical Softw. Engg.*, vol. 27, no. 6, Nov. 2022. [Online]. Available: https://doi.org/10.1007/s10664-022-10177-8

[7] J. Offutt. (2024) Mujava: Mutation system for java. Accessed: 2025-05-29. [Online]. Available: https://cs.gmu.edu/ offutt/mujava/

[8] H. Coles. (2025) Pit mutation testing. Accessed: 2025-05-30. [Online]. Available: https://pitest.org/

[9] R. Just, "The Major mutation framework: Efficient and scalable mutation analysis for Java," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, San Jose, CA, USA, July 23–25 2014, pp. 433–436.

[10] L. Ma, F. Zhang, J. Sun, M. Xue, B. Li, F. Juefei-Xu, C. Xie, L. Li, Y. Liu, J. Zhao, and Y. Wang, "Deepmutation: Mutation testing of deep learning systems," in *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*, 2018, pp. 100–111.

[11] Z. Li, H. Wu, J. Xu, X. Wang, L. Zhang, and Z. Chen, "Musc: A tool for mutation testing of Ethereum smart contract," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019, pp. 1198–1201.

[12] M. Barboni, A. Morichetta, and A. Polini, "Sumo: A mutation testing strategy for solidity smart contracts," in *2021 IEEE/ACM International Conference on Automation of Software Test (AST)*, 2021, pp. 50–59.

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

[14] "YouTube API," 2025, accessed November 2025. [Online]. Available: https://developers.google.com/youtube/v3/

[15] (2025) OpenAPI Specification - Version 3.1.0 | Swagger. Accessed: 2025-11-13. [Online]. Available: https://swagger.io/specification/

[16] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 507–525, 2015.

[17] H. Ed-douibi, J. L. C. Izquierdo, and J. Cabot, "Automatic generation of test cases for REST APIs: A specification-based approach," *2018 IEEE 22nd International Enterprise Distributed Object Computing Conference (EDOC)*, pp. 181–190, 2018.

[18] S. Karlsson, A. Causevic, and D. Sundmark, "Quickrest: Property-based test generation of openapi-described restful apis," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. Los Alamitos, CA, USA: IEEE Computer Society, oct 2020, pp. 131–141. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ICST46399.2020.00023

[19] D. Corradini, A. Zampieri, M. Pasqua, E. Viglianisi, M. Dallago, and M. Ceccato, "Automated black-box testing of nominal and error scenarios in restful apis," *Software Testing, Verification and Reliability*, vol. 32, no. 5, p. e1808, 2022.

[20] V. Atlidakis, P. Godefroid, and M. Polishchuk, "Restler: stateful rest api fuzzing," in *Proceedings of the 41st International Conference on Software Engineering*, ser. ICSE '19. IEEE Press, 2019, p. 748–758. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00083

[21] Z. Hatfield-Dodds and D. Dygalo, "Deriving semantics-aware fuzzers from web api schemas," in *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, ser. ICSE '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 345–346. [Online]. Available: https://doi.org/10.1145/3510454.3528637

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

[23] S. Segura, J. C. Alonso, A. Martin-Lopez, A. Durán, J. Troya, and A. Ruiz-Cortés, "Automated generation of metamorphic relations for query-based systems," in *2022 IEEE/ACM 7th International Workshop on Metamorphic Testing (MET)*, 2022, pp. 48–55.

[24] A. Martin-Lopez, S. Segura, and A. Ruiz-Cortés, "Restest: automated black-box testing of restful web apis," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 682–685. [Online]. Available: https://doi.org/10.1145/3460319.3469082

[25] Y. Liu, Y. Li, G. Deng, Y. Liu, R. Wan, R. Wu, D. Ji, S. Xu, and M. Bao, "Morest: model-based restful api testing with execution feedback," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1406–1417. [Online]. Available: https://doi.org/10.1145/3510003.3510133

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

[27] A. Martin-Lopez, S. Segura, C. Müller, and A. Ruiz-Cortés, "Specification and Automated Analysis of Inter-Parameter Dependencies in Web APIs," *IEEE Transactions on Services Computing*, 2021.

[28] T. Le, T. Tran, D. Cao, V. Le, T. N. Nguyen, and V. Nguyen, " KAT: Dependency-Aware Automated API Testing with Large Language Models ," in *2024 IEEE Conference on Software Testing, Verification and Validation (ICST)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2024, pp. 82–92. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ICST60714.2024.00017

[29] M. Kim, T. Stennett, D. Shah, S. Sinha, and A. Orso, "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*, ser. ICSE-NIER'24. New York, NY, USA: Association for Computing Machinery, 2024, p. 37–41. [Online]. Available: https://doi.org/10.1145/3639476.3639769

[30] M. Kim, T. Stennett, S. Sinha, and A. Orso, "A multi-agent approach for rest api testing with semantic graphs and llm-driven inputs," in *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering*, ser. ICSE '25. IEEE Press, 2025, p. 1409–1421. [Online]. Available: https://doi.org/10.1109/ICSE55347.2025.00179

[31] M. Kim, S. Sinha, and A. Orso, "Llamaresttest: Effective rest api testing with small language models," *Proc. ACM Softw. Eng.*, vol. 2, no. FSE, Jun. 2025. [Online]. Available: https://doi.org/10.1145/3715737

[32] D. Stallenberg, M. Olsthoorn, and A. Panichella, "Improving test case generation for rest apis through hierarchical clustering," in *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '21. IEEE Press, 2022, p. 117–128. [Online]. Available: https://doi.org/10.1109/ASE51524.2021.9678586

[33] A. Arcuri, "Restful api automated test case generation with evomaster," *ACM Trans. Softw. Eng. Methodol.*, vol. 28, no. 1, jan 2019. [Online]. Available: https://doi.org/10.1145/3293455

[34] E. Viglianisi, M. Dallago, and M. Ceccato, "RESTTESTGEN: Automated Black-Box Testing of RESTful APIs," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pp. 142–152.

[35] M. Kim, S. Sinha, and A. Orso, "Adaptive REST API Testing with Reinforcement Learning." IEEE Computer Society, pp. 446–458. [Online]. Available: https://www.computer.org/csdl/proceedings-article/ase/2023/299600a446/1SBGpzCcTGo

[36] D. Corradini, Z. Montolli, M. Pasqua, and M. Ceccato, "Deeprest: Automated test case generation for rest apis exploiting deep reinforcement learning," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '24. New York, NY, USA: Association for Computing Machinery, 2024, pp. 1383–1394. [Online]. Available: https://doi.org/10.1145/3691620.3695511

[37] V. Atlidakis, P. Godefroid, and M. Polishchuk, "Checking security properties of cloud service rest apis," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, 2020, pp. 387–397.

[38] L. Gazzola, M. Goldstein, L. Mariani, I. Segall, and L. Ussi, "Automatic ex-vivo regression testing of microservices," in *Proceedings of the IEEE/ACM 1st International Conference on Automation of Software Test*, ser. AST '20, New York, NY, USA, 2020, p. 11–20.

[39] P. Godefroid, D. Lehmann, and M. Polishchuk, "Differential regression testing for rest apis," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 312–323. [Online]. Available: https://doi.org/10.1145/3395363.3397374

[40] M. Zhang and A. Arcuri, "Adaptive hypermutation for search-based system test generation: A study on rest apis with evomaster," *ACM Trans. Softw. Eng. Methodol.*, vol. 31, no. 1, sep 2021.

[41] E. Barlas, X. Du, and J. C. Davis, "Exploiting input sanitization for regex denial of service," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22, New York, NY, USA, 2022, p. 883–895.

[42] T. Vassiliou-Gioles, "A simple, lightweight framework for testing restful services with ttcn-3," in *2020 IEEE 20th International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, 2020, pp. 498–505.

[43] J. C. Alonso, S. Segura, and A. Ruiz-Cortés, "AGORA: Automated Generation of Test Oracles for REST APIs," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2023. New York, NY, USA: Association for Computing Machinery, 2023, pp. 1018–1030. [Online]. Available: https://doi.org/10.1145/3597926.3598114

[44] J. C. Alonso, M. D. Ernst, S. Segura, and A. Ruiz-Cortés, "Test Oracle Generation for REST APIs," *ACM Trans. Softw. Eng. Methodol.*, Mar. 2025. [Online]. Available: https://doi.org/10.1145/3726524

[45] A. Martin-Lopez, S. Segura, and A. Ruiz-Cortés, "Test coverage criteria for restful web apis," in *Proceedings of the 10th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, ser. A-TEST 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 15–21. [Online]. Available: https://doi.org/10.1145/3340433.3342822

[46] "RESTCountries commit fixing country with negative area bug." 2025, accessed November 2025. [Online]. Available: https://gitlab.com/restcountries/restcountries/-/commit/ee498c74ad21c93b66a577d63d2c8eacefc58d42

[47] "RESTCountries GitLab issue. Country with negative area." 2025, accessed November 2025. [Online]. Available: https://gitlab.com/restcountries/restcountries/-/issues/219

[48] "GitLab issue. Invalid URLs in the getApiV4ProjectsIdBadges operation." 2025, accessed January 2025. [Online]. Available: https://gitlab.com/gitlab-org/gitlab/-/issues/473603

[49] A. Arcuri, M. Zhang, A. Golmohammadi, A. Belhadi, J. P. Galeotti, B. Marculescu, and S. Seran, "Emb: A curated corpus of web/enterprise applications and library support for software testing research," in *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*, 2023, pp. 433–442.

[50] M. Kim, Q. Xin, S. Sinha, and A. Orso, "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*, ser. ISSTA 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 289–301. [Online]. Available: https://doi.org/10.1145/3533767.3534401

[51] D. Corradini, A. Zampieri, M. Pasqua, and M. Ceccato, "Restats: A test coverage tool for restful apis," in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2021, pp. 594–598.

[52] H. Sartaj, S. Ali, and J. M. Gjøby, "Rest api testing in devops: A study on an evolving healthcare iot application," *ACM Trans. Softw. Eng. Methodol.*, Sep. 2025, just Accepted. [Online]. Available: https://doi.org/10.1145/3765744

[53] (2024) PICT. Accessed: 2024-07-04. [Online]. Available: https://github.com/microsoft/pict

[54] Atlassian, "Swagger request validator: A java library for validating rest api requests and responses against openapi/swagger specifications," https://bitbucket.org/atlassian/swagger-request-validator, 2024, version v2.44.1, accessed: 2025-11-18.

[55] J. C. Alonso, "Beet: A daikon front-end for invariant-based test oracle generation in rest apis," 2023, accessed: 2025-11-18. [Online]. Available: https://github.com/isa-group/Beet

[56] ——, "Postmanassertify: A tool for the automated generation of test assertions for rest api testing," https://github.com/juaaloval/PostmanAssertify, 2025, accessed: 2025-11-18.

[57] Skyscreamer, "Jsonassert: Write json unit tests in less code. great for testing rest interfaces," https://github.com/skyscreamer/JSONassert, 2024, version v2.0-rc1, accessed: 2025-11-18.