Random Testing and Evolutionary Testing for Fuzzing GraphQL APIs

ASMA BELHADI, Kristiania University College, Norway
MAN ZHANG[*], Kristiania University College, Norway
ANDREA ARCURI, Kristiania University College and Oslo Metropolitan University, Norway

The Graph Query Language (GraphQL) is a powerful language for APIs manipulation in web services. It has been recently introduced as an alternative solution for addressing the limitations of RESTful APIs. This paper introduces an automated solution for GraphQL APIs testing. We present a full framework for automated APIs testing, from the schema extraction to test case generation. In addition, we consider two kinds of testing: white-box and black-box testing. The white-box testing is performed when the source code of the GraphQL API is available. Our approach is based on evolutionary search. Test cases are evolved to intelligently explore the solution space while maximizing code coverage and fault-finding criteria. The black-box testing does not require access to the source code of the GraphQL API. It is therefore of more general applicability, albeit it has worse performance. In this context, we use a random search to generate GraphQL data. The proposed framework is implemented and integrated into the open-source EvoMaster tool. With enabled white-box heuristics, i.e., white-box mode, experiments on 7 open-source GraphQL APIs and 3 search algorithms show statistically significant improvement of the evolutionary approach compared to the baseline random search. In addition, experiments on 31 online GraphQL APIs reveal the ability of the black-box mode to detect real faults.

CCS Concepts: • **Software and its engineering → Software verification and validation**; **Search-based software engineering**.

Additional Key Words and Phrases: GraphQL, EvoMaster, Evolutionary Algorithms, Automated Testing, Random, SBST, SBSE, fuzzing

ACM Reference Format:
Asma Belhadi, Man Zhang, and Andrea Arcuri. 2023. Random Testing and Evolutionary Testing for Fuzzing GraphQL APIs . 1, 1 (August 2023), 42 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Web services are very common in industry, especially in enterprise applications using microservice architectures [64]. They are also becoming more common with the appearing of smart city technologies, where microservices are largely exploited in industrial internet-of-things settings [37, 40]. The investigation of automating techniques for generating test cases for web service APIs has become a research topic of importance for practitioners [17].

[*] Corresponding author

Authors' addresses: Asma Belhadi, Kristiania University College, Prinsens gate 7-9, 0152, Oslo, Norway, asma.belhadi@kristiania.no; Man Zhang, Kristiania University College, Prinsens gate 7-9, 0152, Oslo, Norway, man.zhang@kristiania.no; Andrea Arcuri, Kristiania University College and Oslo Metropolitan University, Prinsens gate 7-9, 0152, Oslo, Norway, andrea.arcuri@kristiania.no.

XXXX-XXXX/2023/8-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Due to the high number of possible configurations for the test cases, evolutionary techniques have been successfully used to address different software testing problems [13, 53]. Common examples are EvoSuite for unit test generation for Java programs [47], Sapienz for mobile testing [61] and EvoMaster for REST API testing [16].

The Graph Query Language (GraphQL) is a powerful language of web-based data access, created in 2012 and open sourced by Meta/Facebook in 2015 [8]. It addresses some of the RESTful API limitations, like the possibility of specifying what to fetch on a graph of interconnected data with a single query [54, 72]. Different companies have started to provide web APIs using GraphQL[1], like for example Facebook, GitHub, Atlassian, and Coursera.

To the best of our knowledge, only three recent approaches in the scientific literature exist which explore the automated testing for GraphQL APIs, dealing with ''deviation testing'' [73], ''property-based testing'' [56] and ''harvesting production queries'' [79]. To deal with this gap in the scientific literature [67], this paper presents the first white-box and black-box test generation approach for automating GraphQL APIs testing. We developed a full framework for automate GraphQL APIs testing, from the schema extraction to the generation of the test cases in executable test suite files (e.g., using JUnit and Jest). The proposed framework has been implemented as an extension to the open-source EvoMaster tool [16, 27], and it is freely available online.

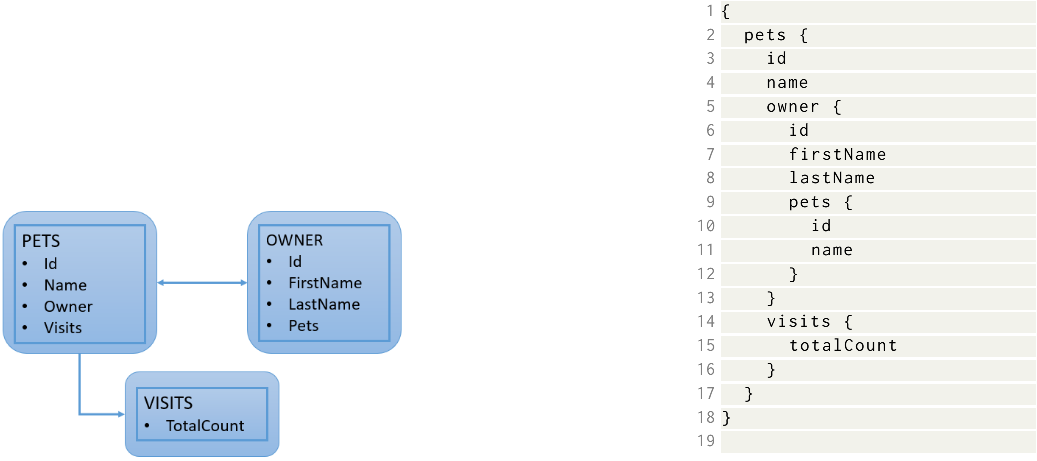
The main contributions of this research work are as follows:

(1) We investigate two kinds of testing: white-box and black-box testing. The white-box testing is performed when the source code of the GraphQL API is provided, whereas the black-box testing is employed when the source code of the GraphQL APIs is either missing or when users do not have access to it (e.g., remote services).
(2) We develop an evolutionary-based search for white-box testing. It intelligently explores the test case space in GraphQL APIs. Different types of genes are created which allow the complete data representation of the GraphQL schema. The genetic operators are used to explore the test case space while maximizing metrics such as code coverage and the number of detected faults. We also adopt a random search for black-box testing, where test cases are generated randomly from GraphQL schema without considering information from the source code of the GraphQL APIs.
(3) To validate the applicability of our presented framework, an empirical study has been carried out on 7 GraphQL APIs with source code, and 31 online GraphQL APIs. For white-box testing, we adapted and compared 3 existing search algorithms for test suite generation: MIO [18], MOSA [65] and WTS [48]. The results show a clear improvement of using the evolutionary algorithms compared with the random search baseline. Line coverage is improved by 7% on average (up to 30.8% in one API), and more errors were automatically found in the analyzed APIs (+71 new errors in total). For black-box testing, the results also reveal the detection of up to 641 endpoints with errors over 825 endpoints.

This paper is an extension of a short poster [36]. In [36] we provided an initial implementation to support a subset of GraphQL, with an empirical study of white-box testing on two artificial APIs. In this paper, we significantly extended such support (e.g., how to deal with nested function calls), and provided a much larger empirical study, including experiments on black-box testing as well as comparisons with different search algorithms.

The article is structured as follows. Section 2 provides background information on GraphQL APIs, EvoMaster and the employed search algorithms. Section 3 discusses related work. Section 4 gives a detailed explanation of the main components of the proposed framework. The details of our empirical study are presented in Section 5, followed by a discussion of the main findings of our

[1] https://graphql.org/users/



(a) Data graph

```
{
  pets {
    id
    name
    owner {
      id
      firstName
      lastName
      pets {
        id
        name
      }
    }
    visits {
      totalCount
    }
  }
}
```

(b) GraphQL query

Fig. 1. Example of GraphQL data graph and a query on it

research work in Section 6. Section 7 discusses the possible threats to validity. Finally, Section 8 concludes the paper.

2 BACKGROUND

This section provides important background information to better understand the rest of the paper, in particular regarding GraphQL APIs (Section 2.1), the EvoMaster tool (Section 2.2), and the compared search algorithms Random Search (Section 2.3), WTS (Section 2.4), MOSA (Section 2.5), and MIO (Section 2.6).

2.1 GraphQL

GraphQL is a query language and server-side runtime for application programming interfaces (APIs) [8]. Given a set of data represented with a graph of connected nodes, GraphQL enables to query such graph, specifying for each node which fields and connections to retrieve (and so recursively on each retrieved connected node). Figure 1a shows a simplified/reduced example of graph for a pet clinic, whereas Figure 1b shows a GraphQL query on it to retrieve the list of all pets that have been registered in the clinic with their owners and their total number of visits to the pet clinic.

A GraphQL web service would be typically listening on a TCP socket, expecting GraphQL queries as part of HTTP requests, on a given HTTP endpoint (typically `/graphql`). However, GraphQL is not tight to HTTP, and queries could be technically sent via other communication mechanisms. One advantage here is that a user can fetch all the data they need (and only that) in a single HTTP call.

A core concept in GraphQL is the schema, which is a collection of types and relationships between those types. It describes which kind of types and fields on those types a client can request. GraphQL is a strongly typed language and has its own language to write the schema.

Commonly used types definition in a GraphQL schema are as follows:

(1) **Object types**. The most frequent elements of a GraphQL schema are object types. It indicates which kind of object (e.g., a node in the graph) you can fetch and what fields it has.

(2) **Query type**. A Query type is a special type in GraphQL that defines entry points (equivalent to remote procedure calls) for fetching data from the graph. It is the same as an object type, but its name is always Query. Each field of the Query type describes the name and return type of a different entry point. Note that a GraphQL server has to define a query type.

(3) **Mutation type**. A GraphQL operation can either be a read or a write operation. A Query type is used to read data while the Mutation type is used to modify data. The Mutation type follows the same syntactical structure as queries, however it defines entry points for writing operations. Note that a GraphQL server may or may not have a mutation type.

(4) **Scalar types**. Scalar types are primitive types that resolve to concrete data. GraphQL has five default scalar types: Int, Float, String, Boolean and ID. Note that GraphQL allows creating custom scalar types for more specific usage.

(5) **Input types**. Input types are object types that allow passing complex objects as arguments to queries and mutations. An input type's definition is alike to an object type's, but it starts with the keyword `input` instead of type. Note that input types can only have basic field types (input types or scalar types) and can not have field arguments.

(6) **Enum types**. An Enum type is a special scalar type with a restricted set of allowed values specified in the schema.

(7) **Interface types**. An Interface type is an abstract type. An interface is composed of a set of fields held by multiple object types. When an object type implements an interface, it has to include all of that interface's fields. Thereby, interfaces enable returning any object type that implements that interface. Note that, we can query an interface schema type for any fields defined in the interface itself and we can also query it for fields that are not in the interface but in the object types implementing the interface.

(8) **Union types**. Like interface types, union type belong to the GraphQL abstract types. It allows to define a schema type that belong to multiple types. In its definition, a union type will determine which object types are included. In this case, the schema field can return any object type that is described by the union. Note that, all union's included types should be object types (e.g., not Input types).

(9) **Non-Nullable type**. All types in GraphQL are nullable by default, i.e., the server can return a null value for all the previous types. To override this default and specify that `null` is not a valid response, an exclamation mark (!) following the type is added indicating that this field is required. The Non-Null type can also be used in arguments. Note that, in queries a field is always optional, i.e., one can skip a non-nullable field and the query would be still valid. However, if a field is required (declared as non-nullable) the server must never return the null value if the query fetches such field. With regard to input arguments, by default they are optional. However, if a type is declared as non-null, besides not taking the value null, it also does not accept omission (i.e., the input argument must be present).

The following is a fragment of a GraphQL schema extracted from one of the SUTs used in our empirical study in Section 5 (i.e., *petclinic* [10]). The schema describes the entry point `pets` that returns a list of all pets that have been registered in the pet clinic. It is defined as a non-nullable object array type named `Pet`. The object type `Pet` has as a field a non-nullable integer scalar type that represents the `id` of the pet. It also defines a non-nullabe object type `Owner` which implements an interface named `Person`. The field `VisitConnection` is an object type specifying all the visits to the pet clinic of this pet. It has an non-nullable integer field `totalCount` that reports the total number of visits for this pet.

```
type Query {
    pets: [Pet!]!
}
```

```

type Pet {
    id: Int!
    name: String!
    owner: Owner!
    visits: VisitConnection!
}

type Owner implements Person {
    id: Int!
    firstName: String!
    lastName: String!
    pets: [Pet!]!
}

type VisitConnection {
    totalCount: Int!
}

interface Person {
    id: Int!
    firstName: String!
    lastName: String!
}
```

As a response to a query, a GraphQL API will return a JSON object with two fields: `data` that contains the result of the query, and `errors` if there was any error with the query (and in this case the `data` field would not be present). Notice that GraphQL makes no distinction between user errors (e.g., a wrongly formatted query or an input does not satisfy a business logic constraint) and server errors (e.g., internal crash in the business logic of the API, like a null-pointer exception). However, it might support it in the future[2]. Furthermore, as GraphQL is independent from HTTP, a query with errors could still have a HTTP response 200 (i.e., OK), and this is a common behavior among GraphQL framework implementations.

Another limitation of GraphQL is that currently it has no standarized way to express constraints on the fields of the graph (e.g., an integer within a specific numeric range, or a string that should satisfy a given regular expression). Constraints could be added with ‘‘directives’’, which are decorators used to extend the semantics of the schema. But those would be custom, and unique for each different implemented API.

GraphQL has received an increase interest in the research community in the last few years [67]. This includes for example performance comparisons with REST [74, 76], tradeoff studies in architectural design [57, 71], code/schema generation [46, 68], and formal analyses [39, 42].

2.2 EvoMaster

EvoMaster [16, 27] is an open-source tool that aims at system test generation, currently targeting REST web services [19]. EvoMaster is a search-based approach which has been integrated with a set of search algorithms (i.e., the Many Independent Objective (MIO) [15], the Many-Objective Sorting Algorithm (MOSA) [65], the Whole Test Suite (WTS) [48], and a Random Search Algorithm) to evolve test cases. Internally, by default it uses the MIO [15] evolutionary algorithm enhanced with *Adaptive Hypermutation* [80], and can handle both black-box [20] and white-box testing [25]. For white-box testing, it uses established heuristics like the *branch distance* [15, 19, 60], it employs testability transformations to smooth the search landscape [25], and can also analyze all interactions with SQL databases to improve the fitness function [24]. For black-box testing, EvoMaster employs

[2] https://github.com/graphql/graphql-spec/issues/698

a random test generation. The tool produces random inputs, but still syntactically valid with respect to the OpenAPI/Swagger schema of the SUT.

Two different studies [58, 81] compared EvoMaster with other fuzzers for RESTful APIs, showing that EvoMaster gives the best results. EvoMaster is open-source, hosted on GitHub [5], with each release automatically published on Zenodo for long-term storage. The extension presented in this paper is available to practitioners since EvoMaster version 1.5.0 [31].

EvoMaster currently targets REST APIs running on the JVM and NodeJS [85] (albeit for black-box testing it can be applied on any kind of REST web service), and it outputs test suite files in the JUnit and Jest format. Each generated test case is composed by one or more HTTP calls, and SQL data to initialize the database (if any).

When targeting RESTful APIs, EvoMaster analyzes their schema, and create a chromosome representation with a rich gene system to represent all possible needed types (from integers and strings to full JSON objects). The resulting phenotype will represent complete HTTP requests, where each gene would represent the different decisions that need to be made in these HTTP requests (e.g., query/path parameters in the URLs, and body payloads in POST/PUT/PATCH requests). EvoMaster evolves test cases based on different metrics, like statement and branch coverage, as well as coverage of the HTTP status codes for each endpoint in the API. To detect potential faults, it considers the 500 HTTP status code (i.e., server error) and possible mismatches between the schema and the concrete responses [62].

2.3 Random Search

Random search (RS) is a method to generate solution(s) of an optimization problem at random. It is easy to implement, as it does not need gradient to guide the search or any special heuristics. It simply keeps sampling new random solutions from the search space, checking if they give better fitness than the best solutions sampled so far.

EvoMaster implements a RS to produce the tests in the context of test case generation, i.e., randomly sample a sequence of test actions with test data generated randomly according to the schema of the SUT. Each time a newly sampled test covers a new target, it is stored in an archive. At the end of the search, RS generates the best set of test cases based on the defined fitness function.

In the context of REST API testing, RS is the default algorithm for black-box mode of EvoMaster. Regarding white-box mode of EvoMaster, the RS can be considered as a grey-box testing approach as it still employs white-box heuristics to produce the best test cases at the end of the search (i.e., tests that cover more code are saved in the archive).

Such an approach can be used as a baseline [28] to assess the effectiveness of white-box approaches [15, 83, 85], e.g., in terms of effectiveness of evolving tests with the white-box heuristics (such as branch distance). In the literature, RS is often used as baseline to evaluate novel techniques [13]. A novel technique might have a non-trivial computational cost, which could make it slower and provide worse results than a naive RS (an example is un-constrained combinatorial testing [22]).

2.4 WTS

WTS was introduced by Fraser and Arcuri [48] for unit test generation of Java classes. The approach reformulates the test suite generation as a search problem, and employs a Genetic Algorithm (GA) to solve it, with originally the main focus on maximizing branch coverage in the context of white-box testing.

With the GA, an individual in a population is a test suite which is composed of a set of test cases. The search starts with a random population. At each iteration, a new generation (i.e., Z) is evolved, with a subset (e.g., 10%) composed of the best individuals from the current population (known as *elitism*). Then, two individuals are selected based on the rank selection from the population as parents for the next evolution (i.e., parents P_1 and P_2). The two parents will be modified by crossover with a given probability, and mutation to generate two offspring (i.e., offspring Q_1 and Q_2). The modification performed by crossover is at the test suite level, while the mutation can modify the test cases (like changing test data) or test suite (like removing or inserting new tests). The four individuals (i.e., P_1, P_2, Q_1 and O_2) will be evaluated by the fitness function, which calculates the sum of all the heuristics/optimization targets by the test suite as a single objective [69]. Based on the calculated fitness and constraints (e.g., maximum number of test cases), the best individuals from these four will be added into the new generation (i.e., Z). Like in a GA, this process is repeated until the new population Z is full (i.e., it reaches its maximum size). Then, a new generation is created following the same process, and so on. The search will terminate once the specified coverage criterion is satisfied or the specified search budget is used up, then produces the best test suite evolved so far in the final generation. WTS is further extended by using an archive to store the best tests [70]. More details on WTS can be found in [48, 69, 70].

2.5 MOSA

MOSA was introduced by Panichella *et al.* [65]. Unlike single objective handled by WTS, MOSA reformulates the test generation as many objectives. MOSA is also a white-box testing approach, i.e., each coverage target (such as branch and line) is defined as an objective to optimize. An individual in MOSA is a test, and not a test suite as in WTS. In MOSA, the population is composed of tests for an objective. Once the objective is achieved by a test, the test will be added into an archive that tracks the best tests during the search, and the objective will also not be involved in the fitness function in the following optimization steps.

MOSA is inspired by Non-Dominated Sorting Genetic Algorithm II (NSGA-II) [41], which is a widely applied algorithm in the scientific literature. To better tackle test case generation problem and make the search focus on uncovered targets, MOSA defines a *preference criterion* and a new rank algorithm based on such a criterion (i.e., *preference sorting*). Similar with NSGA-II, MOSA starts with a random population. At each iteration, crossover and mutation are used to generate new tests, and selection is used for next generation based on ranks. In MOSA, the main difference from NSGA-II is that its selection employs *preference sorting* which calculates ranks with a further consideration of the *preference criterion* (e.g., test cases for uncovered targets will be assigned with the best rank in order to have them survive in the next generation with a higher chance). The search will terminate based on specified search budget, and the best tests in the archived will be outputted at the end.

2.6 MIO

MIO is the default algorithm in EvoMaster, introduced by Arcuri [15]. It is a genetic based evolutionary algorithm inspired by the (1+1) Evolutionary Algorithm (EA) [45] , designed specifically for handling system test case generation. Here, we briefly discuss how it works, but for the full details of MIO we refer to [15]. MIO is a multi-population algorithm with one population for each testing target. Similarly to MOSA, MIO evolves individuals that are test cases, and outputs test suites. MIO has been employed for testing REST API [19, 85, 87] and RPC-based API [83].

At the beginning of the search, one single population is randomly initialized, based on the chromosome templates constructed from the schema. For testing RESTful APIs, several kinds of testing targets are taken into account, like for example statements coverage in the SUT, branch coverage and returned HTTP status codes. Next, at each step, the MIO either samples a new test at random or selects one existing test from one population that includes yet to covered targets, and mutate such a test. Different strategies are used to select which population to sample from.

Individuals are manipulated through only one operator, which is the mutation operator (i.e., no crossover). Two types of mutation are applied: either a structure mutation or internal mutation. A test case is composed by one or more "action" (i.e., an HTTP call in the case of testing of web services). In contrast to the internal mutation, which affects only the values of the genes in the actions, for instance flipping the value of a boolean gene from `True` to `False`, the structure mutation acts on the structure itself, such as adding and removing actions (e.g., HTTP calls). Each time a new test is sampled/mutated, its fitness is calculated. If it achieves any improvement on any target (regardless of the population it was sampled from), it will be saved in the corresponding populations (and the worst individuals in such populations are deleted). In this context, if a target is covered by a test, it is saved in an archive, the corresponding population is shrunk to one single individual, and it will never expand again nor used for sampling. If new targets are reached (but not fully covered) during the evaluation of a test, a new population is created for each such newly discovered targets. At the end of the search, MIO does output a test suite (i.e., a set of test cases) based on the best tests in the archive for each testing target.

3 RELATED WORK

3.1 Testing of GraphQL APIs

Automated testing of GraphQL APIs is a topic that has been practically neglected in the research literature [67]. To the best of our knowledge, so far only three approaches have been investigated regarding the automated testing of GraphQL APIs [56, 73, 79] (besides the poster version of this extended paper [36]).

Vargas *et al.* [73] proposed a technique called "Deviation Testing". It consists of three steps. In the first step, an already existing test case is taken as input. This test constitutes a base to seed and compare the newly generated tests. The second step is the test case variation, where variations of the initial seeded test case are generated using deviation rules (where a deviation consists of a small modification). Four types of deviation rules are defined: 1) field deviation consists of adding and deleting the selection of fields in the original query; 2) not null deviation consists of replacing a declared non null argument with null; 3) type deviation consists of changing an argument type by another type; 4) empty fields deviation consists of deleting all fields and sub fields of the original query. The third step is the test case execution where the input test and its variations are executed. The last step consists of comparing the results between the input test and its variation (e.g., wrong inputs should lead to a response containing an error message).

Karlsson *et al.* [56] proposed a black-box property based testing method. The method consists of the following steps. First, all specifications of the types and their relations are extracted from the schema. Data is generated at random according to the schema, with customized "data generators" provided by the user. In addition, the authors suggest two strategies to use as automated oracles: the first one aims to check the returned HTTP status codes, and the second one verifies that the resulting data returned conform to the given schema.

Zetterlund *et al.* [79] presented a technique to capture the HTTP calls done by users in production (e.g., when interacting with web frontend), and generate test cases for the GraphQL API in the backend. These generated tests can then be used for regression testing.

Our novel solution does not require any pre-existing test case (like in [73]), nor it requires the user to write customized input generators (like in [56]), nor it requires users interacting with a frontend [79]. Due to those limitations, no empirical comparisons with those techniques could be performed on our case study. Our framework benefits from the EvoMaster tool, which is able to do advance white-box testing (e.g., using testability transformations [25] and SQL interaction analysis [24]), and also be able to perform black-box testing when source code of the SUT is not available. In this paper, we provide a complete testing pipeline, and an intelligent genetic-based exploration for the possible test case configurations.

3.2 Testing of REST APIs

In recent years, there has been an increasing interest in the research community about the automation of testing web services, where RESTful APIs are the currently the most common type [51].

For example, Godefroid et al. [50] introduced the differential regression testing for avoiding breaking changes in the REST APIs. They analyzed both types of regressions in APIs, regression in the contract rules between the client and the server, and regression in the server itself, the different changes in the server versions. The differential testing is performed to automatically identify abnormal behavior in both kinds of regressions. It consists of comparing different versions of the server and also the different versions of the contracts with the client.

Viglianisi et al. [75] proposed an approach for automatically generating black-box test cases for REST APIs. It takes as input the Swagger/OpenAPI specification of the API and consists of three modules: It first analyzes the schema and computes its corresponding operation dependency graph which is a graph that represents the data dependencies. It then automatically generates test cases of the REST API by reading both the graph created in the previous step and the schema in order to test the nominal scenarios. It finally applies mutation operators to the nominal tests which violate data constraints for testing error scenarios. In order to decide whether a test is successful or not, two oracles are established based on the returned status codes and on the compliance with the schema.

Lopez et al. [63] presents a new formulation of the automated test APIs problem using CSP (Constraint Satisfaction Problem). The IDL (Inter-parameter Dependency Language) is first introduced to formally describe the different relations among input parameters of the REST API. Then, the CSP is used to automatically analyse the IDL specification. Finally, a catalogue of analysis is constructed in order to extract helpful information such as checking whether an API call is valid or not

So far, all approaches presented in the literature for testing RESTful APIs are black-box [51]. EvoMaster (Section 2.2) is currently the only tool that can do both white-box and black-box testing. It uses evolutionary techniques for white-box testing, and random search for black-box testing. Furthermore, recent comparisons of tools [59, 81, 82] show that EvoMaster gives the best results on the selected APIs used in those tool comparisons.

Note that empirical comparisons with existing fuzzers targeting RESTful APIs would not be particularly useful, as they would fail to generate any test case for GraphQL APIs (as for example the API schema definitions are different). Likewise, comparing with other testing tools targeting different domains such mobile apps (e.g., Sapienz [61]) or data parsers (e.g., AFL [1]) would provide no useful information either.

4 GRAPHQL TEST GENERATION

This section presents our novel proposed framework for automated test case generation of GraphQL APIs, built on top of the EvoMaster tool. As sketched in Figure 2, the proposed framework targets both white-box and black-box testing. The white-box testing is performed when the information related to the schema is provided and have access to source code of the GraphQL API. In case of only having the schema, black-box testing is rather used.

Algorithm 1 presents the pseudo-code of the GraphQL test generation algorithm. The input data is the entry point to the GraphQL API (e.g., a URL). In case of white-box testing, we also provide the source code as input. The results will be a set of generated tests (e.g., in JUnit or Jest format). The process starts by extracting the schema from the graph data in line 3. In case of having the source code of the GraphQL API, the white-box solution is performed. It first determines the corresponding

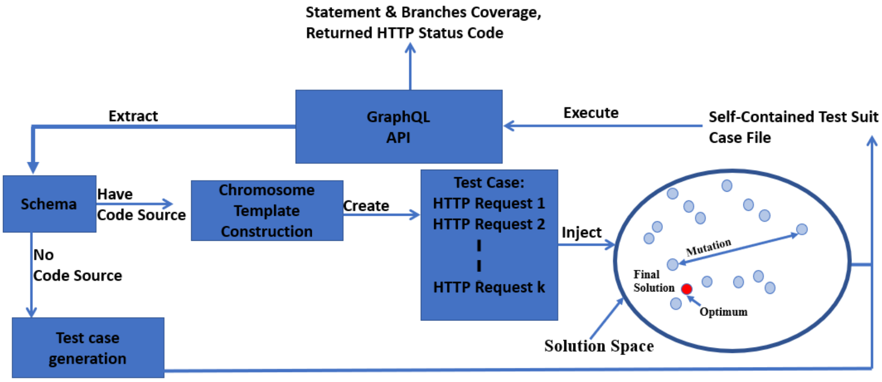


Fig. 2. Framework for GraphQL test generation

Algorithm 1 GraphQL Test Generation Algorithm

```
 1: Input: $G = \{N_1, N_2, \ldots, N_n\}$: the set of $n$ data in the GraphQL API.
 2: Output: Tests: the set of generated tests.
 3: Schema ⟵ Extraction(G)
 4: if code source in G then
 5:     Gene ⟵ ChromosomeTemplateConstruction(Schema)
 6:     TestCase ⟵ Create(Gene)
 7:     Tests ⟵ EvolutionaryAlgorithm(TestCase)
 8: else
 9:     Tests ⟵ Random(Schema)
10: end if
11: return  Tests
```

genes, creates the chromosome template, and applies an evolutionary algorithm process to generate the tests (from line 4 to line 7). In case of missing the source code of the GraphQL API, the black-box solution is established by generating random tests based on the schema, illustrated in line 9. The algorithm returns the set of generated tests as shown in line 11. In the following, both kinds of testing are explained in more details.

4.1 White-Box Testing

When carrying out scientific research to apply evolutionary computation to solve a new engineering or scientific problem for the first time in the literature, many decisions need to be made and empirically evaluated. This includes defining the problem representation for the evolving individuals (Section 4.3), the search operators (Section 4.4), the fitness function (Section 4.5), and the final output format (Section 4.6). All these decisions based on scientific research can have major impact on the final results. Therefore, proper empirical evaluations with a large case study is needed.

Our process for automated search-based testing for GraphQL APIs starts by extracting the schema from the tested API (Section 4.2). The chromosome template is then constructed from the schema. Test cases are represented by a sequence of HTTP requests, instantiated from the chromosome template (Section 4.3). The test cases are evolved using a search algorithm, where each test case will contain genes representing how to build the GraphQL queries based on the given schema. The evolutionary search is performed by applying mutation operators to evolve the test cases, e.g., one to change the queries/mutations in each HTTP calls, and the other to add/remove HTTP calls in a test case (Section 4.4). Test cases are rewarded based on their achieved code coverage and found faults (Section 4.5). From the final evolved solution, a self-contained test suite file (e.g., in JUnit format) is generated as output of the search (Section 4.6).

In the following, we describe the main components of the white-box testing in more details.

4.2 API Schema

At a high level, a GraphQL API can be seen a server opening a TCP socket, processing HTTP requests, with body payloads written in a specific format (i.e., using the GraphQL language) for the different functionalities provided by the API. Sending random bytes on such TCP connection would unlikely lead to any meaningful message that would be immediately discarded by the API. Likewise, sending properly formatted GraphQL messages would result only in errors if those messages are not based on the actual entry points and expected input types of the API.

To send meaningful GraphQL messages that would execute the business logic of the API, such messages must be based on the *schema* of the API (recall Section 2.1). Each GraphQL API must have a schema definition, which can be retrieved online from the API itself (unless such option is disabled due to security reasons).

In order to fetch the whole schema from a GraphQL API, an *introspective* query is used. Given an entry point to the GraphQL API (e.g., typically a `/graphql` HTTP endpoint), GraphQL enables a standard way to fetch a schema description of the API itself. The schema specifies all the information about the available operation types, such as queries, mutations and all available data types on each of them. As a result, the GraphQL schema is returned in JSON format.

Let us consider the following example of the introspective query in which we query one of the SUTs used in our empirical study in Section 5 (i.e., *petclinic* [10]) to obtain the resources that are available:

```
query IntrospectionQuery {
  __schema {
    queryType {
      name
    }
    mutationType {
      name
    }
    types {
      kind
      name
      fields {
        name
        args {
          name
          type {
            kind
            name
            ofType {
              kind
              name
            }
          }
        }
        type {
          kind
          name
          ofType {
            kind
            name
          }
        }
      }
    }
  }
}
```

In this introspective query, we query the field `__schema`, which provides all information about the schema of a GraphQL service. It is considered as a meta-field used by GraphQL for the introspection system. Such field is accessible from the root type of a query operation, and its type is defined in the following:

```
type __Schema {
  queryType: __Type!
  mutationType: __Type
  types: [__Type!]!
}
```

By querying the fields `queryType` and `mutationType`, GraphQL *petclinic* server will return all queries and mutations available from the schema. In this case, both queries and mutations operations are available.

The field named `types` (of kind `__Type`) is at the core of the introspection system. It represents all types in the system: both named types (e.g., `OBJECT` kind) and type modifiers (e.g., `NON_NULL` kind). A reduced subset (due to its length) of the returned result of the introspective query for this example is shown in the following:

```
{
   "data": {
      "__schema": {
         "queryType": {
            "name": "Query"
         },
         "mutationType": {
            "name": "Mutation"
         },
         "types": [
            {
               "kind": "OBJECT",
               "name": "Query",
               "fields": [
                  {
                     "name": "owner",
                     "args": [
                        {
                           "name": "id",
                           "type": {
                              "kind": "NON_NULL",
                              "name": null,
                              "ofType": {
                                 "kind": "SCALAR",
                                 "name": "Int"
                              }
                           }
                        }
                     ],
                     "type": {
                        "kind": "NON_NULL",
                        "name": null,
                        "ofType": {
                           "kind": "OBJECT",
                           "name": "Owner"
                        }
                     },
                     "isDeprecated": false,
                     "deprecationReason": null
```

```
                        }
                    ]
                }
            ]
        }
    }
}
```

Here, the Query type is an object that has a field called owner. The owner field takes as argument a non-nullable integer named id and returns an object named Owner.

Like the software of the API itself, also the GraphQL schemas can have faults. And this is for example a common issue among RESTful APIs [62]. As the schema is the main source of information on how to prepare syntactically valid requests, issues in the schema can have negative impacts on the performance of the fuzzing sessions [81].

Most GraphQL frameworks (e.g., Apollo [3]) do validate the syntax of the API endpoints based on the defined schema, e.g., to check the presence and format of each endpoint (i.e., query and mutation methods). In case of errors and mismatches, they would return a response with an error message at each incoming request, or simply crash the server at start-up time. This kind of issue can be quickly identified, and corrected. However, a schema could be *under-specified*. For example, the API could have implementations of endpoints that are not declared in the schema. But it would not be possible to call any of these endpoints, as the GraphQL frameworks running the API would not be aware of those endpoints. Therefore, even in these cases such kind of issue would be easily detected by users without the need of using any fuzzer.

This means that for GraphQL APIs, in contrast to RESTful APIs (where typically the framework servers do not validate the schemas), problems in the schemas do not seem to be as serious for testing purposes. However, more research will be needed to evaluate this potential issue in more details.

4.3 Problem Representation

Once the schema of the tested API is fetched, this latter is then parsed in our EvoMaster extension and used to create a set of action templates, one for each query and mutation operation. Each action will contain information on the fields related to input arguments (if any is present) and return values. A chromosome template is defined for each action, which is composed of non-mutable information (e.g., field's names) and a set of mutable genes. In this context, each gene characterizes either an argument or a return value in the GraphQL query/mutation.

For objects as return values, a query/mutation must specify which fields should be returned (at least one must be selected), and so on recursively if any of the selected fields are objects as well. To represent the fact that a field is always optional for queries, a return gene is modeled by an object gene where all its fields are optional. However, we had to extend the mutation operator in EvoMaster with a post-processing phase, to guarantee that at least one field gene is selected during the search. In other words, if after a mutation of a gene, which represents a returned object value in the GraphQL query/mutation, all fields are de-selected, then the post-processing will force the selection of one of them (and so on recursively if the selected field is an object itself). On the other hand, if a return value is a primitive type, then there is no need to create any gene for it, as there is no selection to make. Furthermore, similar to functions calls, fields in the returned value can have input argument themselves. When a returned value for a parent field is executed, both input arguments and returned value are recursively selected to generate a child field value until it produces a scalar value whether in input arguments or in returned values. To model those function calls we introduced a new special type of gene called Tuple, discussed next.

To fully represent what is available from the GraphQL specification, the following kinds of gene types from EvoMaster have been re-used and adapted:

(1) `String`: It contains string variables which are defined by an array of characters. A minimum length of the string is zero which represents the empty string. Each string gene cannot exceed a predefined maximum number of characters (e.g., 100).
(2) `Enum`: This gene represents the enumeration type, where a set of possible values is defined, and only one value is activated at a given time. The elements in the set can be in different formats (e.g., enumerations of numbers or enumerations of strings).
(3) `Float/Integer/Boolean`: genes representing variables with simple data types. Boolean genes represent variables with true or false values. Integer and float genes represent integer and real-value variables, respectively.
(4) `Array`: This gene represents a sequence of genes with the same type. This gene has variable length, where elements can be added and removed throughout the search. In order to mitigate creating too large test cases, for instance with millions of genes, the size of an array gene should not exceed a given threshold.
(5) `Object`: This gene defines an object with a specific set of internal fields. Differently from the array gene, where the elements should be with the same type, an object gene may contain elements with different types. To do so, this gene is represented by a map, where each key in the map is determined by the field name in each element in the object.
(6) `Optional`: A gene containing another gene, whose presence in the phenotype is controlled by a boolean value. This is needed for example to represent nullable types in arguments and selection of fields in returned objects.
(7) `CycleObject`: This special gene is used as a placeholder to avoid infinite cycles, when selecting object fields that are objects themselves, which could be references back to the starting queried object. Once a test case is sampled, its gene tree-structure is scanned, and all `CycleObject` genes are forced to be excluded from the phenotype (e.g., if inside an `Optional` gene, that gets marked as non-selected, and the mutation is prevented to select it; if the `CycleObject` is the type for an Array gene, such array gets a fixed size of 0, and the mutation operator is prevented from adding new elements in it).
(8) `LimitObject`: GraphQL schemas are often very large and complex, and the levels of nesting fields can be potentially huge. We use this special gene as a placeholder when a customized depth limit is reached. The depth is the number of nesting levels of the object fields.
(9) `Tuple`: This gene is needed for example when representing the inputs of function calls. It is composed of a list of elements of possible different types, where the last element can be treated specially. This is for example the case of function calls when the return type is an object, on which we need to select what to retrieve (and these selected elements could be function calls as well, and so on this is handled recursively).

To make this discussion more clear, let us consider a small, simplified portion of the schema of *GitLab* (one of the SUTs used in our empirical study in Section 5).

```
type Query {
    currentUser: UserCore
}
type UserCore{
    groups(permissionScope: GroupPermission): GroupConnection
}
type GroupConnection{
    nodes: [Group]
}
type Group{
    fullName: String!
}
enum GroupPermission{
    CREATE_PROJECTS
    TRANSFER_PROJECTS
}
```

To send the query to the server, the user must follow the above representation of the schema. For instance, to query the field `fullName`, the user might send this query:

```
{
  currentUser {
    groups(permissionScope: CREATE_PROJECTS) {
      nodes {
        fullName
      }
    }
  }
}
```

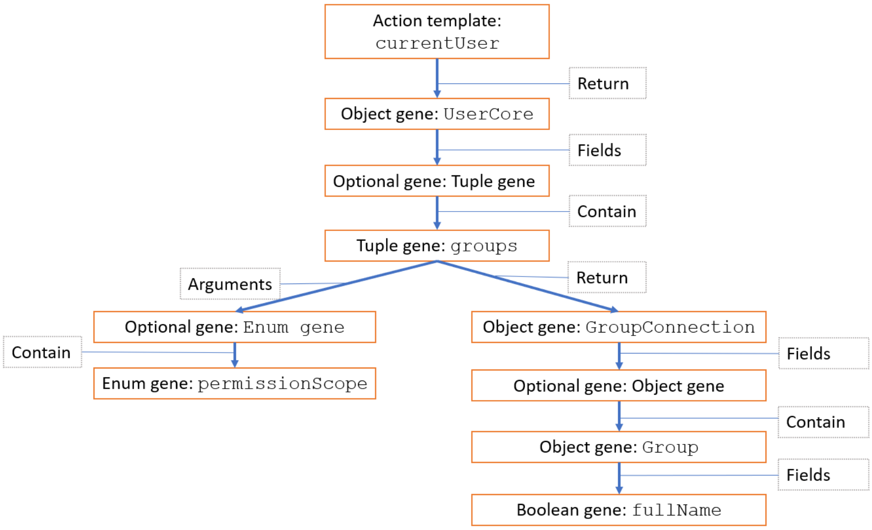


Fig. 3. A gene representation of a user query

This query is syntactically valid, conforming the schema represented above. But it is not the only possible query conforming such schema. A user could rather send a `permissionScope` with value `TRANSFER_PROJECTS`, or simply such optional input could be avoided altogether. So, a genotype representation needs to be able to express all possible queries which are valid for the given schema. The tree in Figure 3 shows a genotype structure (seen as a tree of genes) for such schema, using the previously discussed gene system used in our framework. For example, the action representing the GraphQL query `currentUser` has the `object` gene `UserCore`, which contains an `optional` `tuple` field `groups`. When calling `currentUser`, one needs to specify which fields of the returned object `UserCore` to include in the response. In this particular example, there is only one field called `groups`. As which field to return is optional, to represent this each of these fields is inside an `optional` gene. If an optional gene is deactivated, none of its internal genes is used in the phenotype of the test case. The field `groups` is itself a remote function call. It is represented with a `tuple` gene, having an input argument `permissionScope` and return value `GroupConnection`. The argument is represented as an optional gene containing an enum gene (for the 2 possible values defined in the schema). The return value GroupConnection is represented with an object gene. For each field of an object, we need a gene to represent it. A field can be yet another object, or array of them, like the case of nodes. So, this process is applied recursively. The non-method/non-object fields are represented with a boolean gene (to check whether it will be part of the returned object or not), like the case of fullName. In this simplified example, the choices that need to be made are for example whether the optional genes should be active or not, the boolean values of the boolean genes, and values for the enum genes. The evolutionary process will make modifications to these values throughout the search. From this genotype, then the phenotype will represent syntactically valid queries.

For this subset of schema, the search space of all possible queries is small. However, it would increase exponentially when dealing with more inputs, in particular for strings.

To fully support the whole specification of GraphQL, there are several special cases that need to be handled, like the use of *interfaces*. To deal with GraphQL interfaces, we use an optional object gene for each type that implements the interface, together with an extra optional object gene (labeled with BASE) to specify the interface fields themself. Consider the following portion of the schema from *digitransit* (one of the SUTs used in our empirical study in Section 5).

```
type Query {
    node(id: ID!): Node
}
interface  Node{
    id: ID!
}
type Trip implements Node{
    id: ID!
    tripHeadsign: String
}
type TicketType implements Node{
    id: ID!
    price: Float
}
```

Here, the interface Node has two possible implementations, Trip and TicketType. A user can query different fields based on the concrete types of the returned objects. For example, assume querying the fields id, tripHeadsign and price, as described in the following.

```
{
  node(id: 3) {
    id
    ... on Trip {
      tripHeadsign
    }
    ... on TicketType {
      price
    }
  }
}
```

The tree in Figure 4 shows the genotype for this user query containing interfaces, based on the gene system used in our framework. Here, whether to query the different concrete types of the interfaces, and their fields, is optional. Indeed, a genotype representation needs to be able to express all these possible kinds of valid phenotypes.

After defining the possible type of genes supported by the proposed framework, we consider the solution space, where each solution is a set of test cases. A test case is composed of one or

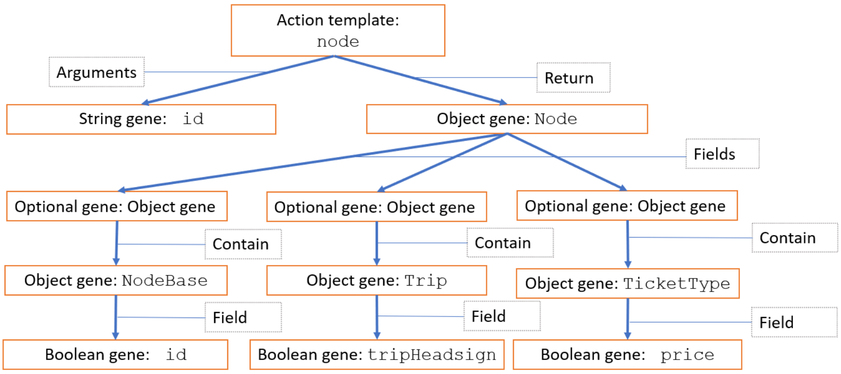


Fig. 4. A gene representation of a user query using GraphQL interfaces

more HTTP request. In order to represent an HTTP request, we typically need to deal with its components: HTTP verb, path and query parameters, body payloads (if any) and headers.

A GraphQL request can be sent via HTTP GET (used only for queries) or HTTP POST methods with a JSON body (used for queries and mutations). For simplicity, we only use the verb POST for both queries and mutations. A GraphQL server uses a single URL endpoint (typically `/graphql`), where the HTTP requests with the GraphQL queries/mutations will be sent. In the context of test generation for a GraphQL API, the main decisions to make are on how to create JSON body payloads to send. The genotype will contain genes (from the set defined above) to represent and evolve such JSON objects.

In Figure 5 there is an example of a test case generated automatically by EvoMaster for *petclinic* API, outputted in JUnit format. It is composed of two HTTP POST requests. The first call with a body payloads querying for the entry point `specialities` and the second requesting for the entry point `owner`. When a test case is generated and evaluated, we also provide assertions on the returned responses.

The test cases are generated in a random way, but they are still syntactically valid. For instance, if we consider the example illustrated in Figure 1a, the test cases are generated by exploring the fields of the pets node. For instance, if we consider the field ‘‘id’’ an integer represented in 32 bits. The possible test cases for the field ‘‘id’’ is 2^{32}. We also explore different combinations of two or more fields in each node. For instance, consider the same example illustrated in Figure 1a, the test cases might be generated from both fields ‘‘id’’, and ‘‘name’’ of the node pets. If we consider the length of the string is limited to 10, the possible tests cases for the field ‘‘name’’ is 2^{160} (assuming each character being 2-bytes). Therefore, the number of possible test cases by only exploring the fields ‘‘id’’, and ‘‘name’’ is $2^{32} \times 2^{160}$, which results an immense search space. Therefore, in our implementation, and in order to mitigate the combinatorial explosion, we use threshold to limit the number of generated test cases that can be evaluated (i.e., we limit the number of test cases we sample during the random search).

4.4 Search Operators

Once a chromosome representation is defined based on the GraphQL schema, test cases are evolved and evaluated in the same way as done for RESTful APIs in EvoMaster (recall Section 2.2), including testability transformations [25] and SQL database handling [24]. Internally, the search algorithms in EvoMaster are implemented in a generic way, independently of the addressed problem (e.g., REST

```java
@Test(timeout = 60000)
public void test_16() throws Exception {

    given().accept("application/json")
            .contentType("application/json")
            .body(" { " +
                " \"query\": \" {  specialties  {id,name}   }   \" " +
                " } ")
            .post(baseUrlOfSut + "/graphql")
            .then()
            .statusCode(200)
            .assertThat()
            .contentType("application/json")
            .body("'data'.'specialties'.size()", equalTo(3))
            .body("'data'.'specialties'[0].'name'", containsString("radiology"))
            .body("'data'.'specialties'[1].'name'", containsString("surgery"))
            .body("'data'.'specialties'[2].'name'", containsString("dentistry"));

    given().accept("application/json")
            .contentType("application/json")
            .body(" { " +
                " \"query\": \"  { owner  (id : 8)  {id,lastName,telephone,pets{birthDate,type{id},visits{totalCount}}}      } \" " +
                " } ")
            .post(baseUrlOfSut + "/graphql")
            .then()
            .statusCode(200)
            .assertThat()
            .contentType("application/json")
            .body("'data'.'owner'.'lastName'", containsString("Escobito"))
            .body("'data'.'owner'.'telephone'", containsString("6085557683"))
            .body("'data'.'owner'.'pets'.size()", equalTo(1))
            .body("'data'.'owner'.'pets'[0].'birthDate'", containsString("1997/02/24"))
            .body("'data'.'owner'.'pets'[0].'visits'.'totalCount'", numberMatches(0.0));
}
```

Fig. 5. Example of generated JUnit test for *petclinic*

and GraphQL APIs), and it is only a matter of defining an appropriate phenotype mapping function (e.g., how to create a valid HTTP request for a GraphQL API based on the evolved chromosome genotype).

As previously stated, an evolving individual will be a set of *actions* (i.e., calls to query and mutation endpoints) on the tested API. Each action is represented with a gene-tree template (e.g., recall examples in Figure 3 and Figure 4), which needs to be instantiated (i.e., set the values of the genes). As part of the search, there are 3 main search operators: (1) random sampling; (2) mutation on the structure of the tests; (3) mutation on the content of an action. Note: the term *"mutation"* in the context of GraphQL APIs (used to represent an endpoint in the API that can modify its state) has nothing to do with the term *"mutation"* used in the evolutionary computation literature (used to represent search operators that do small changes in the evolving individuals).

When sampling a new individual at random (e.g., needed for Random Search, as well as for evolutionary algorithms when they need to initialize their first population of individuals to evolve), first there is the need to choose how many actions K it contains. For example, it can be randomly chosen between 1 and N (where for example $N = 10$). Given A, the set of possible action templates (one for each query/mutation endpoint in the API), then each of these K actions in the sampled test will be chosen randomly from A. Then, the content of each gene in such trees is set at random (considering their types and constraints).

Mutation operators are used to do small changes to an evolving individual. The structure of the test can be modified by removing an action from the current K (if $K > 1$), or by adding a new random action from A (if $K < N$). This can be applied with a given probability P.

The content of an action a can be modified by selecting any from K, and then selecting randomly any gene from its tree. Given G_a the set of genes in the selected action a, each gene could be mutated with probability $1/|G_a|$. The type of mutation depends on the type of the genes. For example, a numeric gene could have its phenotype value been increased or decreased by a certain small delta. A boolean gene could be flipped from `true` to `false`, and vice-versa. A string gene could have some of its chars modified randomly. And so on (full details can be found in the source code of EvoMaster [32]).

Consider the last example in Figure 5, where the mutation operators are applied as the following.

```
@Test(timeout = 60000)
public void test_16() throws Exception {
 given().accept("application/json")
            .contentType("application/json")
            .body(" { " +
                " \"query\": \"  { owner  (id : 1)  {id,lastName,pets{birthDate,type{name}}}
      } \" " +
                " } ")
            .post(baseUrlOfSut + "/graphql")
            .then()
            .statusCode(200)
            .assertThat()
            .contentType("application/json")
            .body("'data'.'owner'.'lastName'", containsString("Franklin"))
            .body("'data'.'owner'.'pets'.size()", equalTo(1))
            .body("'data'.'owner'.'pets'[0].'birthDate'", containsString("2010/09/07"))
            .body("'data'.'owner'.'pets'[0].'type'.'name'", containsString("cat"))
}
```

Here, the test case is shrunk to only 1 action `owner` (from the original 2). The `owner`'s input argument `id` is mutated to 1 (from the original 8). The action is mutated by dropping the phenotype of the genes `telephone`, `id` and `visits` (i.e., their boolean genes were mutated from `true` to `false`), and it is extended by adding to the phenotype the field `name` (i.e., its boolean gene is mutated from `false` to `true`).

4.5 Fitness Function

The fitness function plays a critical role in an evolutionary algorithm, as it specifies which individuals will survive and reproduce. The main goal of our testing is to find faults in the tested API. A fault cannot manifest if the code in which it lies is not executed. Therefore, an indirect approach to try to find more faults is to maximize the code coverage achieved by the generated tests. However, generating high code coverage tests is a complex task, as the execution flow in the API might depend on complex constraints (e.g., complex predicates in `if` statements), which could be satisfied only with very specific inputs.

There is a large body of research literature on the topic of maximizing code coverage for software testing. In the case of search-based software testing [13], there are common techniques like *branch distance* [60] and *testability transformations* [52]. For the work in this paper, we do not define any new white-box heuristics. We rather rely on the state-of-the-art white-box heuristics for system test generation provided by EvoMaster. This includes advanced testability transformations [26], as well as different types of branch distance heuristics. All evolutionary algorithms compared in this paper use the same fitness function.

Besides testing targets based on the source code (e.g., line, statements and branches), there are other metrics of interest for practitioners. For example, for Web APIs using HTTP, covering different returned HTTP status codes can provide a better coverage of the API. For example, you can make a correct query, and receive a 200 status code in a HTTP response. For the same endpoint, you could send an invalid input (e.g., a number outside a specified range), which could lead the server to return a 400 status code (user error), although this depends on the server implementation (some GraphQL servers return 200 even in case of errors). A request with no authentication information could return a 401. A request with authentication but no authorization (i.e., no right permissions) could return 403. An input that leads to a crash (e.g., an exception thrown in the business logic of the API) could result in a 500 status code. And so on.

For each GraphQL endpoint, we create a different testing target for each returned HTTP status code. This enables EvoMaster to do not discard newly generated tests that cover endpoints returning different status codes (and so showing different behaviors of the API).

When evaluating the fitness of an evolved test, besides considering testing targets related to code coverage and HTTP status coverage (for each different query/mutation operation), we also create new testing targets based on the returned responses. As discussed in Section 2.1, each response could contain either a `data` field or an `errors` field. For each query and mutation in the GraphQL schema, we consider two additional testing targets for those two possible outcomes. Note that a trivial way to get a response with `errors` is to send a syntactically invalid query. As such evolved test cases would be of little use, we explicitly avoid generating such kind of test cases (unless there are faults in EvoMaster).

As automated oracles to detect faults once a test is executed, we consider two properties: returned HTTP status code 500, and responses with `errors` fields. The former is a common oracle used in fuzzing HTTP-based APIs (e.g., [59, 62]). However, it is important to keep in mind that not all 500 responses are necessarily related to software faults. For example, an API could return a 500 when unable to communicate with its database because it is down, not reachable for some technical reasons. As `errors` fields might be due to *user errors* besides *server errors*, the users would still need to check those generated tests to see if actual faults are detected.

As a given query/mutation might fail for different reasons, we keep track of the last executed line in the business logic of the SUT. We further create a separated testing target for each combination of errored query/mutation and last executed line. Having explicit testing targets for those cases enables the search algorithms to keep those test cases, albeit the fitness function would have (currently) no gradient to lead to generate such kind of test cases in the first place.

4.6 Output Test Suite

Depending on how long the search is left running, millions of test cases could be evolved and evaluated. Huge test suites would be of little use for practitioners, as they are not manageable. For this reason, at end of the search, only a *minimized* test suite is given as output to the user. Each single test case contributes to the overall fitness of the output test suite (e.g., code and HTTP status coverage, detected faults). In particular, in EvoMaster for each search algorithm we employ an *archive*, that is updated each time a new test case covers a new testing targets. The output test suites is based on what stored in such archive during the search.

As final output, we generate executable test cases, in common formats such as JUnit (for Java/Kotlin) and Jest (for JavaScript), as shown for example in Figure 5. Each test case will have assertions on the obtained responses, i.e., they capture the current behavior of the API. If for example a JSON object is returned, assertions will be recursively created for each of its fields. These assertions do not directly detect faults, as without a formal specification we cannot know what are the expected outputs for the used inputs. However, these assertions can be used for regression testing (i.e., to check if the behavior of the API changes with new updates). Furthermore, as the output test suites are minimized, these assertions could be useful as well to point out possible faults if the users manually review the generated tests.

4.7 Black-Box Testing

We use the black-box testing when we do not have any knowledge about the source code of the GraphQL API, or it is not available for instrumentation (e.g., to calculate the search-based heuristics like the branch distance). It is not straightforward to get a high coverage value for such kind of tests [20], as little information from the SUT can be exploited. However, in some cases (e.g., when testing remote services), a black-box approach might be the only option available for automated testing. Also, as no code analysis is performed, black-box testing can be applied regardless of the programming language the API is written in, like for example Python and Ruby. On the other hand, currently for white-box testing with EvoMaster we are limited to languages running on the JVM (e.g., Java and Kotlin) and NodeJS (e.g., JavaScript and TypeScript).

We use the random search algorithm developed in EvoMaster. The main idea behind random search is performing a randomized process in generating the test cases, where no code-based fitness function is employed. The reason of not using search-based heuristics is due to the lack of the source code of the GraphQL APIs.

From a practical stand-point, our black-box testing is the same as random search, but without code-based heuristics. Like for white-box testing, we start from fetching the schema (Section 4.2), and create a problem representation (Section 4.3) from which new test cases are randomly sampled from (Section 4.4). No evolutionary mutation operator is applied here. We use the same fitness function to reward found faults (Section 4.5), but without any code metrics. At the end of the search, the final test suite is minimized, to contain only the test cases that contribute to the fitness (Section 4.6). In other words, for each query/mutation we retain test cases that lead to different HTTP status codes, and at least one with a correct `data` response and at least one with an `errors` response.

Both black-box and white-box testing share the same goal of detecting faults in the tested APIs. They use the same automated oracles to detect faults. Both testing approaches are important, as they have their own strengths and weaknesses. For example, black-box testing is easier to use (e.g., it requires no setup to specify how to start the application with automated instrumentation) and it is of wider applicability (e.g., it is not restricted to any specific programming language). On the other hand, white-box testing can achieve better results (i.e., code coverage and fault finding), as it can exploit information about the source code of the API. Furthermore, its generated tests can be used for regression testing (as the generated tests can start, stop and reset the API).

In this paper, we provide and empirically evaluate both approaches, as both of them are useful for practitioners in industry. As this is the first work in the literature addressing this problem, more can be done in future research. For example, our black-box approach is very basic, simply a random search on syntactically valid queries based on the schema.

4.8 Tool Support

All the novel techniques presented in this paper have been implemented as part of our existing tool EvoMaster. EvoMaster is open-source on GitHub, with each new release automatically uploaded to Zenodo for long term storage (e.g., [31]).

When a practitioner uses EvoMaster, they need to specify with command-line options whether they are testing a REST or GraphQL API. For example, black-box testing of an online API such as GitLab can be done on the command-line with:

```
evomaster.exe  --problemType GRAPHQL --bbTargetUrl https://gitlab.com/api/graphql --blackBox
    true --outputFormat JAVA_JUNIT_4 --maxTime 30s --ratePerMinute 60
```

Here, one needs to specify that we are fuzzing a GraphQL API (using `--problemType`), and not for example a RESTful one, where the API is located (`--bbTargetUrl`), the type of testing (`--blackBox`), the format of the output tests (`--outputFormat`), for how long to run the fuzzing session (`--maxTime`), and a rate-limiter (`--ratePerMinute`) to do not overload the tested API of requests (needed when testing APIs on the internet to avoid denial-of-service). For doing white-box testing, some manual effort is needed, as there is the need to implement a *driver* class to specify how to start and stop the API.

Extending an existing fuzzer for a new problem domain not only requires scientific research, but also significant engineering effort. What presented in this paper took 2 years of work. Considering the complexity of EvoMaster (which is currently more than 200 000 LOCs, not including tests), providing precise code metrics is not viable. Although modules specific for GraphQL can be identified (e.g., `org.evomaster.core.problem.graphql` with more than 4 000 LOCs of Kotlin code), changes were needed throughout the whole code base of EvoMaster to be able to support GraphQL. For example, the gene system of the evolutionary engine of EvoMaster needed to be extended with new genes like `TupleGene`. We can estimate around 10 000 to 15 000 LOCs of code needed to support GraphQL API testing.

To reduce the risk of publishing wrong results based of faulty software, this work has been carefully tested. For example, in unit tests (e.g., `GraphQLUtilsTest`) we parse 75 GraphQL schemas (having more than 860 000 lines), to make sure that our schema analysis algorithms do not crash and give the correct results (at least for those 75 schemas). Furthermore, EvoMaster has a sophisticated system of end-to-end (E2E) tests [30]. We created several artificial APIs, run EvoMaster on them, compile the generated tests, run them, and verify properties on those tests. This is all done automatically from JUnit tests (including the compilation and dynamic loading and execution of the new generated tests on-the-fly), and run in a Continuous Integration system (i.e., GitHub Actions) at each new Git commit (more details can be found in [30]). Due to all these E2E tests, current EvoMaster build takes more than 2 hours. For GraphQL, we currently have E2E tests for 39 artificial APIs (in the module `spring-graphql`), covering different aspects of GraphQL, for a total of more than 6 000 LOCs.

5 EMPIRICAL STUDY

5.1 Experimental Setup

In this section, several experiments have been carried out to validate the applicability of the proposed framework for GraphQL test generation. This can be achieved by answering the three following research questions:

RQ1: For white-box testing of GraphQL APIs, how effective are evolutionary algorithms at maximizing code coverage and fault detection compared to Random search?

RQ2: How does black-box testing fare on existing APIs on the Internet?

RQ3: What kinds of faults are found by our novel technique?

5.1.1 White-Box. GitHub [7], arguably the main repository for open-source projects, was used to find SUTs for experimentation. JVM and NodeJS projects were scanned and filtered, while excluding trivial projects. For example, we excluded APIs with less than 500 LOCs, and student projects. For this study, seven GraphQL web services are selected, which we could compile and run with no problems:

- The Spring *petclinic* [10] API (4 567 LOCs) is an animal clinic where a pet owner can register his pet for an examination. The examination is carried out by a veterinarian who has one or more specialist areas.
- *patio-api* [9] (12 552 LOCs) is a web application that attempts to estimate the happiness of a given team periodically by asking for a level of happiness.
- *graphql-ncs*(548 LOCs) and *graphql-scs* (577 LOCs) are based on artificial RESTfull APIs from an existing benchmark [6]. For this study, we adapted these two APIs into GraphQL APIs. *graphql-ncs* and *graphql-scs* are based on a code that was designed for studying unit testing approaches on solving numerical [21] and string [14] problems.
- *react-finland* [11] (16 206 LOCs) is an API for a week long developer conference focused on React.js and related technologies.
- *timbuctoo* [12] (85 365 LOCs) is an API that allows scientists to decide how data from different databases is shared.
- *e-commerce* [4] (1 791 LOCs) an e-commerce API built on Phoenix and Elixir that can be utilized in order to create interactive e-commerce web applications.

To the best of our knowledge, there is no other existing white-box fuzzer that can be used to test GraphQL APIs. Therefore, in this paper we cannot compare with any existing technique, as none is available. White-box fuzzing GraphQL APIs is a novel contribution of this paper. Still, it is important to verify whether a novel sophisticated technique is really warranted, and no simpler technique would be already as effective [13]. When nothing else is available, a common baseline in software testing research is *Random Testing* [28], in which an application is tested with random inputs. Still, sending random bytes on the TCP connection the SUT is listening on would be of little to no value, as the chances of generating a valid GraphQL query (or even simply a valid HTTP request) would be virtually non-existent. Therefore, for doing random testing, we still sample and send syntactically valid GraphQL queries based the schema of the SUT.

Once a software engineering problem is modelled as an optimization/search problem (e.g., by specifying the problem representation and the fitness function), different search algorithms can be applied and evaluated. But no search algorithm is best on all problem [77]. To improve performance on a specific problem, research is needed to customize the algorithms to exploit as much domain knowledge of this problem as possible. In the specific case of white-box test suite generation, the most used algorithms are WTS (Section 2.4), MOSA (Section 2.5), and MIO (Section 2.6). MOSA [65] replaced WTS [48] as the default search algorithm in EvoSuite [47] (which is the most famous search-based tool for *unit test* generation), based on large empirical studies comparing many different search algorithms [38, 66]. On the other hand, for the *system test* generation of RESTful APIs, MIO [18] provided the best results in search algorithm comparisons [18]. As the testing of GraphQL APIs with search algorithms is a novel contribution of this paper that has not been done before in the research literature, in this paper we applied and compare the 3 most common search algorithms for test suite generation, i.e., WTS, MOSA and MIO.

For the experiments, we set 1 hour as search budget for our white-box testing approach. To take into account the randomness of the algorithms, each experiment was repeated 30 times [23]. In total, these experiments took $7 \times 4 \times 30 = 840$ hours, i.e., 35 days. Experiments were run in parallel (15 at a time) on the same hardware: HP Z6 G4 Workstation with Intel(R) Xeon(R) Gold 6240R, 24 cores, CPU @2.40GHz 2.39GHz processor, 192G RAM, and 64-bit Windows 10.

To evaluate and compare the effectiveness of the employed algorithms, we selected covered testing targets (#Targets), line coverage (%Lines), and the number of detected faults (#Errors)

Table 1. GraphQL APIs used for black-box experiments

Name	Description	Source Code
aniList	provides access to anime and manage entries, including character, staff, and live airing data.	Yes
deutsche-bahn	infrastructure Data, like realtime facility status and stations	Yes
barcelona-urban-mobility	combine information about the different urban mobility services of Barcelona city	Yes
buildkite	a platform for continuous integration and deployments	No
câmara-dos-deputados	an api to obtain the data from the brazilian deputies chamber	Yes
catalysis-hub	chemical surface reaction energies and structures	Yes
contentful	provides a content infrastructure for digital teams to power content in websites, apps, and devices	Yes
countries	information about countries, continents and languages	Yes
demotivational-quotes	get random demotivational quote	Yes
digitransit	journey planning solution combining several open source components into available route planning service	No
ehri	holocaust-related archival materials held in institutions across Europe and beyond	No
fauna	serverless GraphQL database	No
fake-graphQL-api	mock user and to do data	No
fruits	provides information of fruit trees of the world	Yes
ghibli	catalogs the people, places, and things found in the worlds of Ghibli	Yes
gitLab	host-your-own Git repository hosting service	No
google-directions	GraphQL wrapper over the google directions API	Yes
music-brainz	an open music encyclopedia that collects music metadata	Yes
pokémon	query for all the Pokémon data including their abilities, moves, items, learnsets, and type matchups	Yes
hivdb	a curated database to represent, store and analyze HIV drug resistance data	No
jobs	GraphQL jobs directory	No
melody	fast and reliable dependency manager for Go programming language	No
react-finland	GraphQL API for conferences and meetings	Yes
rickandmortyapi	based on the television show Rick and Morty. Provide the Rick and Morty information (characters, episodes, locations)	Yes
spacex	a non official platform for SpaceX's data	No
spotify	provides instant access to millions of songs, from old favorites to the latest hits	Yes
swapi	provides all the Star Wars data	Yes
swop	GraphQL foreign exchange rate API	No
travelgateX	a global marketplace for the travel trade	No
universe	check what your friends are doing and find unique events near you using our filter	No
weather	retrieve the current weather for any given city	Yes

as metrics for comparisons. The testing target (#Targets) is the default coverage criterion in EvoMaster. It comprises and aggregates different metrics, such as code coverage (including branch coverage), HTTP status code coverage and fault findings. The line coverage (%Lines) is collected as part of our code instrumentation. Furthermore, we also reported (#Errors) by identifying potential faults, i.e., 500 HTTP status codes and responses with `errors` entries (recall Section 4.5).

5.1.2 Black-Box. To evaluate the black-box testing, 31 online APIs with different domain applications, and different number of endpoints have been selected from *apis.guru* [2], a curated public listing of available web services on Internet. These APIs are written with different programming languages, such as JavaScript and Python. Some APIs provide their implementation (e.g., open-source), whereas others do not (e.g., commercial services). When an API required authentication, we created an account on these APIs, and added the right authentication information the HTTP headers of EvoMaster (e.g., authentication header can be set with the command-line argument `--header`). For our experiments, we considered all the GraphQL APIs listed on *apis.guru*, but excluded APIs that are no longer available (but still listed on *apis.guru*), or that required payment to create an account. Table 1 gives short description of the 31 APIs used in our experiments.

We ran our extension of EvoMaster on all those APIs with black-box mode. The stopping criterion was set to 1 000 HTTP calls per run. Each experiment was run only three times, since sending thousands and thousands of HTTP calls to live services could be interpreted as Denial-of-Service attack. For the same reason, we put a rate limiter of at most 10 HTTP requests per minute

Table 2. Results for 1-hour budget for white-box testing. Best results for each metric on each API are highlighted in bold.

SUT	Targets #				Line Coverage %				Errors #			
	RS	WTS	MOSA	MIO	RS	WTS	MOSA	MIO	RS	WTS	MOSA	MIO
ecommerce-server	**340.5**	340.4	330.4	331.6	**8.2**	8.1	7.9	7.9	**27.9**	27.7	26.7	26.4
graphql-ncs	456.3	635.4	635.7	**657.6**	59.5	88.2	87.4	**90.3**	6.0	6.0	6.7	**8.1**
graphql-scs	640.4	667.7	679.7	**775.3**	67.3	68.2	69.4	**78.6**	**11.0**	**11.0**	**11.0**	**11.0**
patio-api	3427.3	3452.1	3426.2	**3710.0**	32.7	32.7	32.7	**39.8**	58.8	63.1	42.9	**74.4**
petclinic-graphql	664.6	**666.6**	648.9	658.9	**60.6**	**60.6**	59.2	60.2	17.7	18.8	**19.0**	17.2
react-finland	486.9	0.0	501.2	**678.8**	1.7	0.0	1.7	**2.4**	28.0	0.0	28.9	**43.0**
timbuctoo	7441.9	7443.9	7459.9	**7501.3**	29.8	**29.8**	29.8	29.8	49.5	44.3	39.4	**89.7**
Average	1922.5	1886.6	1954.6	**2044.8**	37.1	41.1	41.2	**44.1**	28.4	24.4	24.9	**38.6**

Table 3. Detailed comparisons between MIO and RS. Statistically significant effect-sizes (at $\alpha \leq 0.05$ level) are marked in bold.

SUT	Targets #				Line Coverage %				Errors #			
	RS	MIO	$\hat{A}_{12}$	p-value	RS	MIO	$\hat{A}_{12}$	p-value	RS	MIO	$\hat{A}_{12}$	p-value
ecommerce-server	340.5	331.6	**0.04**	< 0.001	8.2	7.9	**0.01**	< 0.001	27.9	26.4	**0.03**	< 0.001
graphql-ncs	456.3	657.6	**1.00**	< 0.001	59.5	90.3	**1.00**	< 0.001	6.0	8.1	**1.00**	< 0.001
graphql-scs	640.4	775.3	**1.00**	< 0.001	67.3	78.6	**1.00**	< 0.001	11.0	11.0	0.50	1.000
patio-api	3427.3	3710.0	**1.00**	< 0.001	32.7	39.8	**1.00**	< 0.001	58.8	74.4	**0.99**	< 0.001
petclinic-graphql	664.6	658.9	**0.18**	< 0.001	60.6	60.2	**0.32**	< 0.001	17.7	17.2	**0.23**	< 0.001
react-finland	486.9	678.8	**1.00**	< 0.001	1.7	2.4	**1.00**	< 0.001	28.0	43.0	**1.00**	< 0.001
timbuctoo	7441.9	7501.3	**1.00**	< 0.001	29.8	29.8	0.37	0.066	49.5	89.7	**0.96**	< 0.001
Average	1922.5	2044.8	0.75		37.1	44.1	0.67		28.4	38.6	0.67	

(i.e., EvoMaster would make HTTP calls only every 6 seconds). In total, these experiments took $3 \times 31 \times (1000/10) = 9300$ minutes, i.e., 6.4 days. They were run in parallel in the same way as for the white-box experiments.

As we do not have any control on these remote APIs, repeating the experiments more times would not add much more information, as such runs would not be fully independent.

5.2 Experiment Results

5.2.1 Results for RQ1. To compare MIO, MOSA, WTS and Random Search (RS), Table 2 reports their average #Targets, %Lines, #Errors, for each employed search algorithm on each of the case studies. Overall, on average MIO provides the best results. However, there are 2 out of 7 APIs in which it provides worse results (i.e., *ecommerce-server* and *petclinic*).

Detailed analysis of MIO are provided with pairwise comparisons using Mann-Whitney-Wilcoxon U-tests (*p-value*) and Vargha-Delaney effect sizes ($\hat{A}_{12}$), when compared with RS (Table 3), WTS (Table 4) and MOSA (Table 5).

Note that there is no data collected for WTS on *react-finland*. On such API, WTS crashed due to out of memory in every single experiment.

Table 4. Detailed comparisons between MIO and WTS. Statistically significant effect-sizes (at $\alpha \leq 0.05$ level) are marked in bold.

SUT		Targets #				Line Coverage %				Errors #		
	WTS	MIO	$\hat{A}_{12}$	p-value	WTS	MIO	$\hat{A}_{12}$	p-value	WTS	MIO	$\hat{A}_{12}$	p-value
ecommerce-server	340.4	331.6	**0.13**	< 0.001	8.1	7.9	**0.14**	< 0.001	27.7	26.4	**0.11**	< 0.001
graphql-ncs	635.4	657.6	**1.00**	< 0.001	88.2	90.3	**1.00**	< 0.001	6.0	8.1	**1.00**	< 0.001
graphql-scs	667.7	775.3	**1.00**	< 0.001	68.2	78.6	**1.00**	< 0.001	11.0	11.0	0.50	1.000
patio-api	3452.1	3710.0	**1.00**	< 0.001	32.7	39.8	**1.00**	< 0.001	63.1	74.4	**0.92**	< 0.001
petclinic-graphql	666.6	658.9	**0.11**	< 0.001	60.6	60.2	**0.32**	< 0.001	18.8	17.2	**0.14**	< 0.001
react-finland	0.0	678.8	**1.00**	< 0.001	0.0	2.4	**1.00**	< 0.001	0.0	43.0	**1.00**	< 0.001
timbuctoo	7443.9	7501.3	**1.00**	< 0.001	29.8	29.8	**0.34**	0.041	44.3	89.7	**0.98**	< 0.001
Average	1886.6	2044.8	0.75		41.1	44.1	0.69		24.4	38.6	0.66	

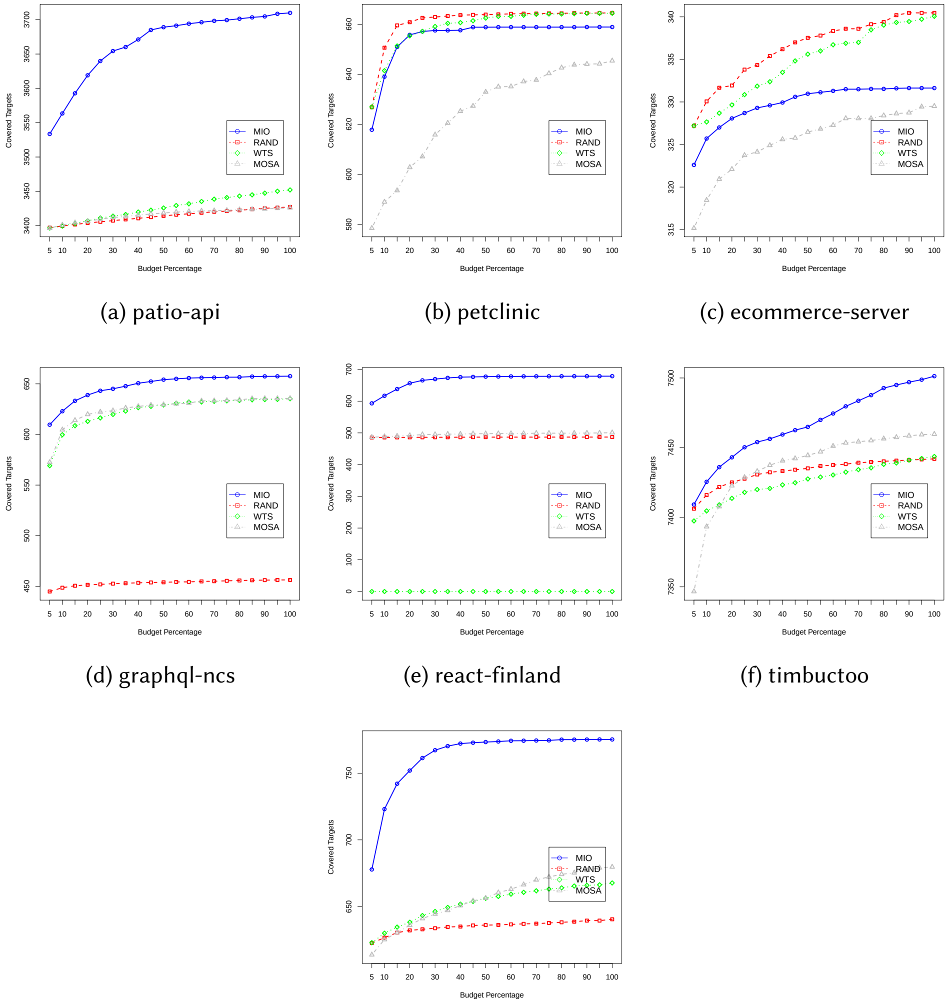
Table 5. Detailed comparisons between MIO and MOSA. Statistically significant effect-sizes (at $\alpha \leq 0.05$ level) are marked in bold.

SUT		Targets #				Line Coverage %				Errors #		
	MOSA	MIO	$\hat{A}_{12}$	p-value	MOSA	MIO	$\hat{A}_{12}$	p-value	MOSA	MIO	$\hat{A}_{12}$	p-value
ecommerce-server	330.4	331.6	0.58	0.300	7.9	7.9	0.50	0.948	26.7	26.4	0.39	0.102
graphql-ncs	635.7	657.6	**1.00**	< 0.001	87.4	90.3	**0.94**	< 0.001	6.7	8.1	**0.99**	< 0.001
graphql-scs	679.7	775.3	**1.00**	< 0.001	69.4	78.6	**0.99**	< 0.001	11.0	11.0	0.50	1.000
patio-api	3426.2	3710.0	**1.00**	< 0.001	32.7	39.8	**1.00**	< 0.001	42.9	74.4	**1.00**	< 0.001
petclinic-graphql	648.9	658.9	**0.66**	0.042	59.2	60.2	**0.78**	< 0.001	19.0	17.2	**0.12**	< 0.001
react-finland	501.2	678.8	**1.00**	< 0.001	1.7	2.4	**1.00**	< 0.001	28.9	43.0	**1.00**	< 0.001
timbuctoo	7459.9	7501.3	**0.95**	< 0.001	29.8	29.8	0.38	0.113	39.4	89.7	**1.00**	< 0.001
Average	1954.6	2044.8	0.88		41.2	44.1	0.80		24.9	38.6	0.71	

When looking at the achieved target coverage, there is a clear improvement of white-box evolutionary search (e.g., MIO) compared to random search (Table 3). On 5 out of 7 APIs, the effect-size is maximum, i.e., $\hat{A}_{12} = 1$. This means that, in *each* of the 30 runs of MIO, the results were better than the best run of random search.

When looking at only line coverage, we can see that MIO enables covering 90.3% of lines in *graphql-ncs*. This is a large improvement compared to the 59.5% of Random (i.e., +30.8%). On average among the SUTs, the improvement is +7% (i.e., from 37.1% to 44.1%). When looking at the absolute values for line coverage, we need to point out an issue with collecting coverage for NodeJS applications (i.e., *ecommerce-server* and *react-finland*), as coverage computation is not considering what achieved during boot-time of the API.

When looking at line coverage along, there is statistically worse results for *petclinic* and *e-commerce*. However, the differences are minimal (i.e., at most -0.4%). For instance, *petclinic* is a simple API used for demonstration, where large parts of its code is not executed (e.g., it has 3 different implementations of its data-layer, where only one can be active at a time, and this has to be specified in a configuration file when the API is started). On simple problems, random search can be already very good, whereas evolutionary search can have some small side-effects (which would likely disappear when using a longer search budget). This is particularly the case if the fitness function does not provide gradient to the search, and the algorithm gets stuck on local optima. The



(a) patio-api (b) petclinic (c) ecommerce-server

(d) graphql-ncs (e) react-finland (f) timbuctoo

(g) graphql-scs

Fig. 6. Covered targets throughout the search

better the fitness function is (e.g., more advance white-box heuristics), the better the results can be achieved with evolutionary search.

In Table 2, we also report the number of potential faults identified by the different search algorithms. The table shows clear better results for MIO compared to Random, with an average high effect size of $\hat{A}_{12} = 0.67$ (Table 3). On *timbuctoo*, MIO achieves the most, finding 89.7 errors on average compared to 49.5 for Random. This result is achieved thanks to the evolutionary operators adopted in the proposed framework to handle GraphQL APIs.

Considering a 1-hour search-budget, an automatically achieved coverage of 39.8% for a complex API like *pation-api* could be considered a good, practical result, although more still need to be done (e.g., better search heuristics). For example, there was practically no difference in code coverage between random and evolutionary search on *timbuctoo*. An in-depth analysis of this API would be needed, to point out which branches were not covered, potentially pin-pointing which kinds of constraints were not be able to be solved with current search-based heuristics. This would be needed to design new heuristics to solve those kinds of constraints [81].

At any rate, in these experiments not only many potential faults are found, but the generated tests can also be used for regression testing (i.e., they can be added to the test suites of the SUTs, and run as part of Continuous Integration to check if any change is breaking any current functionality).

The choice of using 1-hour as stopping criterion is technically arbitrary. It could had been more, or less. Such choice was based on what practitioners could use in practice [84]. Nevertheless, the chosen search budget can impact the conclusions taken from the comparisons of search algorithms. For this reason, in Figure 6 we report plot-lines for demonstrating the performances of the compared techniques for the number of covered targets throughout the search, collected at each 5% intervals (i.e., at each 3 minute intervals).

According to the reported results, MIO outperforms Random for all cases except for *petclinic* and *ecommerce-server*, where we observed better results for Random. On the other APIs, the improvements of MIO are visible throughout the entire search. In a few cases, already with small budgets (e.g., 3 minutes) MIO gets better results than all the other algorithms running for 1 hour.

In these experiments, MOSA shows some interesting behavior. For example, it has a "slow start" on *timbuctoo* and *graphql-scs*, but then with the passing of time it gets better results than RS and WTS. Depending on the chosen search budget, different conclusions could be drawn from these empirical comparisons.

RQ1: *In terms of covered targets, MIO demonstrates a consistent and significant improvements (+7% line coverage and +10.2 more faults found on average) compared with random testing. In these experiments, MIO provides better results than other evolutionary algorithms such as WTS and MOSA. This shows the effectiveness of MIO adapted for GraphQL testing for maximizing code coverage and fault detection.*

5.2.2 *Results for RQ2.* Table 6 presents the results of the black-box testing on the 31 APIs described in Table 1. The results include the number of endpoints (#Endpoints) representing the number of queries and mutations present in the schema, the percentage of endpoints with generated tests without errors (%NoErrors), and the percentage of endpoints with generated tests with errors (%WithErrors). Tests with errors and others without errors could be generated for the same endpoint. However, the following formula would be satisfied:

$$\frac{\%NoErrors}{100} + \frac{\%WithErrors}{100} \leq 2 \tag{1}$$

with

$$\%NoErrors = \frac{\#NoErrors}{\#Endpoints} \tag{2}$$

and

$$\%WithErrors = \frac{\#WithErrors}{\#Endpoints} \tag{3}$$

In other words, for each endpoint, we want to see if we can generate at least one test case in which the endpoint returns data correctly (i.e., `data` field), and at least one test case in which there are issues (i.e., `errors` field). The former type of test could be useful for regression testing, and to manually check if the API behaves correctly and gives the expected outputs. The latter type of test could potentially detect faults, and so would be a first step for debugging the API. As an example, if an API has 10 endpoints (i.e., $\#Endpoints = 10$), then we want to see how many

Table 6. Results for black-box testing

SUT	#Endpoints	%NoErrors	%WithErrors
Anilist	56	1.8	98.2
Bahnql	7	100.0	100.0
barcelona-urban-mobility	10	50.0	100.0
Buildkite	70	7.6	99.0
Camara-deputados	33	23.2	100.0
Catalysis-hub	11	100.0	100.0
Contentful	23	11.6	100.0
Countries	8	87.5	12.5
Demotivation-quotes	2	100.0	0.0
Digitransit	33	26.3	93.9
Directions	6	33.3	83.3
Ehri	19	73.7	100.0
Fauna	13	10.3	100.0
Fruits	7	85.7	28.6
Ghibliql	10	100.0	30.0
Gitlab	270	1.9	95.2
Graphbrainz	6	0.0	100.0
Graphqlpokemon	13	53.8	76.9
Hivdb	9	44.4	77.8
Jobs	15	13.3	86.7
Melody	2	16.7	100.0
Mocki	4	100.0	0.0
React-finland	13	35.9	100.0
RickAndMortyapi	9	63.0	66.7
Spacex	43	93.8	68.2
Spotify	11	0.0	100.0
Swapi	13	46.2	100.0
Swop	6	16.7	100.0
Travelgatex	11	100.0	0.0
Universe	90	10.4	94.4
Weather	2	100.0	100.0
Total	825	48.6	77.8

tests with `data` ($0 \leq \#NoErrors \leq 10$) and with `errors` ($0 \leq \#WithErrors \leq 10$) responses are generated. Generating test with correct `data` responses is not necessarily trivial, as the inputs might have constraints that are not specified in the schema (e.g., a string input must match a specific regular expression). Likewise, if the API has no faults, or the testing tool is not able to create inputs that find any existing fault, or the API has no input validation, no test with response containing `errors` would be generated. Ideally, a fuzzer should aim at being able to generate valid inputs for all endpoints (i.e., $\%NoErrors = 100\%$), whereas whether it is possible to find any fault strongly depends on whether there is any fault in the tested API.

From Table 6, we remark that all endpoints were reached, and responses with either `data` or `errors` fields are effectively derived. From the table we can see that we can generate tests which lead to responses with `errors` fields for many endpoints (77.8%). However, there are also many queries/mutations for which we could not get back any valid data (i.e., responses with `data` field and no `errors` were less than 50%). This is likely due to input constraints which are unlikely to be satisfied with random data. Without code analysis (or constraints expressed directly on schema), likely there is not much a black-box tool can do here (besides having the user providing some sets of valid inputs to fuzz).

Similarly to the fuzzing of RESTful APIs, black-box testing can find faults in GraphQL APIs by just sending random (but syntactically valid) inputs, as often APIs are not particularly robust when dealing with such kind of random data [62]. However, without being able to analyze the source code, it can be hard to bypass their first layer of input validation and generate successful API requests [20].

RQ2: *The Black-box testing implemented in our novel approach enables the automated test generation that can detect on average up to 641 endpoints with errors out of 825 endpoints (i.e., 77.8%).*

5.2.3 *Results for RQ3.*

As discussed in Section 2.1, currently GraphQL makes no distinction between user and server errors. So, without an in-depth manual analysis of the generated tests, it is hard to tell which responses with `errors` messages are due to actual software faults, and not a simple misuses of the API. Furthermore, without knowing the full details of the expected business logic of the specific API under analysis, it might be hard for researchers (which are not the developers of the API) to determine if a returned error is indeed due to a software fault. This problem is further exacerbated for the external APIs used for black-box testing experiments, where the source code is not available and cannot be used to validate if an error is indeed likely due to a fault. Still, when evaluating a novel fuzzing technique like we do in this paper, it is important to check if it can find any actual faults. For this reason, we did a manual analysis of hundreds of generated tests from our experiments. Here, we discuss some of the most interesting cases.

Let us start from the following generated test case for *petclinic*.

```java
@Test(timeout = 60000)
public void test_4() throws Exception {

   given().accept("application/json")
          .contentType("application/json")
          .body(" { " +
                    " \"query\": \"mutation{removeSpecialty(input:{specialtyId:643}){
     specialties{id}}}\" " +
                    " } ")
          .post(baseUrlOfSut + "/graphql")
          .then()
          .statusCode(200) // org/springframework/samples/petclinic/repository/springdatajpa/
     SpringDataSpecialtyRepositoryImpl_38_delete
          .assertThat()
          .contentType("application/json")
          .body("'data'", nullValue())
          .body("'errors'.size()", equalTo(1))
          .body("'errors'[0].'message'", containsString("Internal Server Error(s) while
     executing query"))
          .body("'errors'[0].'path'", nullValue())
          .body("'errors'[0].'extensions'", nullValue());
}
```

Here, the message *"Internal Server Error(s) while executing query"* is a clear example of a fault, even if the returned HTTP status code is 200. By debugging this test case, we found that the problem is due to a null pointer exception: trying to remove a specialty with id equals to 643 that does not exist. Ideally, the API should return an error message stating the requested resource does not exist. However, it looks like the implementation of such API is ignoring the cases when a user asks for something not in the database, which leads to an internal crash.

A similar case can be seen in the following test generated for *react-finland*.

```java
@Test(timeout = 60000)
public void test_0_with500() throws Exception {

   given().accept("application/json")
          .contentType("application/json")
          .body(" { " +
                " \"query\": \"{theme(conferenceId:\\\"coM_FEt0ANyU87\\\"){id,fonts{primary}}}\" " +
                " } ")
          .post(baseUrlOfSut)
          .then()
          .statusCode(500)
          .assertThat()
          .contentType("application/json")
          .body("'errors'.size()", equalTo(1))
          .body("'errors'[0].'message'", containsString("Conference id did not match series"))
          .body("'errors'[0].'locations'.size()", equalTo(1))
          .body("'errors'[0].'locations'[0].'line'", numberMatches(1.0))
          .body("'errors'[0].'locations'[0].'column'", numberMatches(5.0))
          .body("'errors'[0].'path'.size()", equalTo(1))
          .body("'errors'[0].'path'", hasItems("theme"))
          .body("'data'", nullValue());
}
```

Here, the API returns a meaningful error message stating *"Conference id did not match series"*. This means that the requested id *"coM_FEt0ANyU87"* is not matching any registered conference in the API. However, the API is returning the HTTP status code 500, which in HTTP represents a server error, and not a user error (where the most suited code for this case would likely be 404). Technically, this is a fault, although likely not a serious one. Not properly handling the requests for missing data seems common as well in RESTful APIs [62].

However, there are a few cases of more serious faults, like when the returned responses are not matching the constraints of the GraphQL schema of the API. For example, consider the following case of a HTTP call in a generated test for *Bahnql*.

```
given().accept("application/json")
       .contentType("application/json")
       .body(" { " +
             " \"query\": \"{parkingSpace(id : 842)  {name,label,responsibility,spaceType,location{latitude},url,operator,distance,facilityType,openingHoursEn,isSpecialProductDb,isOutOfService,occupancy{validData,timestamp,timeSegment},clearanceHeight,outOfService,isMonthSeason,tariffDiscount,tariffPaymentCustomerCards,tariffFreeParkingTimeEn,tariffPaymentOptionsEn,slogan}       } \" " +
             " } ")
       .post(baseUrlOfSut)
       .then()
       .statusCode(200)
       .assertThat()
       .contentType("application/json")
       .body("'errors'.size()", equalTo(2))
       .body("'errors'[0].'message'", containsString("Cannot return null for non-nullable field Location.latitude."))
       .body("'errors'[0].'locations'.size()", equalTo(1))
       .body("'errors'[0].'locations'[0].'line'", numberMatches(1.0))
       .body("'errors'[0].'locations'[0].'column'", numberMatches(77.0))
       .body("'errors'[0].'path'.size()", equalTo(3))
       .body("'errors'[0].'path'", hasItems("parkingSpace", "location", "latitude"))
       .body("'data'.'parkingSpace'", nullValue());
```

Here a 200 status code is returned, which would imply a success from the point of view of HTTP. However, the error message is *"Cannot return null for non-nullable field Location.latitude."*. This looks like a case of internal server error, where a test case is asking for a non-nullable field named *latitude*, but the server tried to return a *null* value. All types in GraphQL are nullable by default, and the *null* value is a valid response. However, when looking into its schema definition, the field *latitude* is defined as a *non-null* scalar. This is a clear example showing an actual fault in the SUT, where the API tries to return a response that violates the schema.

```
 "kind": "OBJECT",
 "name": "Location",
 "fields":[
            {
              "name": "latitude",
              "args": [],
              "type": {
                "kind": "NON_NULL",
                "name": null,
                "ofType": {
                  "kind": "SCALAR",
                  "name": "Float",
                  "ofType": null
                }
              },
              "isDeprecated": false,
              "deprecationReason": null
            },
```

Another interesting example of schema violation is the following test generated for the *Catalysis-hub* API.

```
@Test(timeout = 60000)
public void test_11() throws Exception {

   given().accept("application/json")
          .contentType("application/json")
          .body(" { " +
                " \"query\": \"{information(name:\\\"fyS8oO8Upt9ceuK\\\",value : \\\"
     CblJF7FCM_kT_\\\",distinct : false,op : \\\"4iyt\\\",search : \\\"Uk4WPkx7y\\\",jsonkey :
      \\\"dWws4\\\",order : \\\"pc\\\",before : \\\"\\\",after : \\\"CU1SdJQYVHfnce\\\",first
     : 298,last : 303)  {edges{node{value,id},cursor},totalCount}       } \" " +
                " } ")
          .post(baseUrlOfSut)
          .then()
          .statusCode(200)
          .assertThat()
          .contentType("application/json")
          .body("'errors'.size()", equalTo(1))
          .body("'errors'[0].'message'", containsString("Can't find property named \"cursor\"
     on mapped class Information->information in this Query."))
          .body("'errors'[0].'locations'.size()", equalTo(1))
          .body("'errors'[0].'locations'[0].'line'", numberMatches(1.0))
          .body("'errors'[0].'locations'[0].'column'", numberMatches(5.0))
          .body("'errors'[0].'path'.size()", equalTo(1))
          .body("'errors'[0].'path'", hasItems("information"))
          .body("'data'.'information'", nullValue());
```

Here a 200 status code is returned, which would imply a success from the point of view of HTTP. However in the body of the response the following error message appears: *"Can't find property named "cursor" on mapped class Information->information in this Query"*. It states that there is no field named *cursor* belonging to the root query *information*. We have extracted and analyzed the whole schema of the *Catalysis-hub* API by sending an introspective query to its endpoint. The schema reveals that the field *information* is of type *InformationCountableConnection*. The type *InformationCountableConnection* has the field named *edges* (that we have asked for) of type *InformationCountableEdge*. This latter, as shown below, has two fields, namely *node* and *cursor*, which shows a clear fault in the SUT. The internal implementation of the server does not respect the defined GraphQL schema.

```
          "kind": "OBJECT",
          "name": "InformationCountableEdge",
          "fields": [
            {
              "name": "node",
              "args": [],
              "type": {
                "kind": "OBJECT",
                "name": "Information",
                "ofType": null
              },
              "isDeprecated": false,
              "deprecationReason": null
            },
            {
              "name": "cursor",
              "args": [],
              "type": {
                "kind": "NON_NULL",
                "name": null,
                "ofType": {
                  "kind": "SCALAR",
                  "name": "String",
                  "ofType": null
                }
              },
              "isDeprecated": false,
              "deprecationReason": null
            }
          ],
          "inputFields": null,
          "interfaces": [],
          "enumValues": null,
          "possibleTypes": null
        },
```

Although schema violations might not be always easy to identify, there are other cases in which faults are very clear. For instance, consider the following test generated for the *Buildkite* API.

```
given().accept("application/json")
       .header("Authorization", "Bearer 992ae7ae4998a8a8faa7c762d74e3c20f2abe154")
       .contentType("application/json")
       .body(" { " +
             " \"query\": \"mutation{jobTypeBlockUnblock(input:{clientMutationId:\\\"hVHw\\\", id:\\\"R2qgRAwGnOo\\\", fields:\\\"6nj19G\\\"}){clientMutationId}}\" " +
             " } ")
       .post(baseUrlOfSut)
       .then()
       .statusCode(200)
       .assertThat()
       .contentType("application/json")
       .body("'type'", containsString("unknown_error"))
       .body("'errors'.size()", equalTo(1))
       .body("'errors'[0].'message'", containsString("An error occurred while executing your GraphQL query. Please contact hello@buildkite.com for help and provide this query in the email."));

```

Here, the server returned the status code 200. However, an internal server error is detected by showing this message in the body: *"An error occurred while executing your GraphQL query. Please contact hello@buildkite.com for help and provide this query in the email"*. Unfortunately, faults found by these black-box experiments from the requested APIs cannot be further analyzed, as we do not have access to the source code of this remote service. It seems like a case of crash due to requesting a resource that does not exist, but we cannot be sure.

Another clear example of a major problem can be seen here in the following test generated for the *Catalysis-hub* API. The test requests the resource called *hasPreviousPage*, but the actual call (done with the library RestAssured) throws an exception.

```
try{
    given().accept("application/json")
           .contentType("application/json")
           .body(" { " +
                 " \"query\": \"{information(name:\\\"8qpQyCBnwDOgP2\\\",value:\\\"fU5gDkH6
     \\\",distinct: false,op : \\\"J0ZA3ahBYJTLgMp\\\",search : \\\"59W\\\",jsonkey : \\\"
     TMJOz\\\",order : \\\"aHdTeItZ\\\",before : \\\"g7_U2sanJ_l0\\\",after : \\\"bEUm\\\",
     first : 141,last : 847){pageInfo{hasPreviousPage}}} \" " +
                 " } ")
           .post(baseUrlOfSut);
} catch(Exception e){
}
```

After a manual investigation, we found that the SUT is returning a failure in an HTML page instead of a JSON object, meaning a significant crush in the SUT.

```
"<!DOCTYPE html>\n\t<html>\n\t  <head>\n\t\t<meta name=\"viewport\" content=\"width=device-
     width, initial-scale=1\">\n\t\t<meta charset=\"utf-8\">\n\t\t<title>Application Error</
     title>\n\t\t<style media=\"screen\">\n\t\t  html,body,iframe {\n\t\t\tmargin: 0;\n\t\t\
     tpadding: 0;\n\t\t  }\n\t\t  html,body {\n\t\t\theight: 100%;\n\t\t\toverflow: hidden;\n\
     t\t  }\n\t\t  iframe {\n\t\t\twidth: 100%;\n\t\t\theight: 100%;\n\t\t\tborder: 0;\n\t\t
     }\n\t\t</style>\n\t  </head>\n\t  <body>\n\t\t<iframe src=\"//www.herokucdn.com/error-
     pages/application-error.html\"></iframe>\n\t  </body>\n\t</html>"
```

However, this might had been due to hardware issues, and not a software fault in the business logic of the API. For example, it could had well been that, although we did successfully generate an introspective query on this API, after a few hundreds HTTP calls the cloud provider Heroku (mentioned in that error page) temporarily disabled the API due to bandwidth usage constraints.

The last error we are going to discuss is for the following test case for the *ecommerce-server* NodeJS API. Note that, in contrast to the previous examples, the generated test here is in Jest (used for JavaScript/TypeScript APIs) format instead of JUnit.

```
test("test_6", async () => {

    let token_foo = "Bearer ";
    await superagent
            .post(baseUrlOfSut + "/graphql")
            .set('Content-Type','application/json')
            .send(" { " +
                " \"query\": \"mutation{login(data:{email:\\\"foo@foo.com\\\",password:\\\"
     bar123\\\"}){token}}\" " +
                " } ")
            .then(res => {token_foo += res.body.data.login.token;},
                error => {console.log(error.response.body); throw Error("Auth failed.")});;


    const res_0 = await superagent
            .post(baseUrlOfSut + "/graphql").set('Accept', "application/json")
            .set("Authorization", token_foo) // foo-auth
            .set('Content-Type','application/json')
            .send(" { " +
                " \"query\": \"  { findStoreById  (id : \\\"Z\\\")  {id,products{categories{
     name},brand,images},employees{createdAt,username,email,bio,image},createdAt,bio,rate,city
     ,state,number,sales}       } \" " +
                " } ")
            .ok(res => res.status);

    expect(res_0.status).toBe(200); // build/src/store/store.service.js_39_40
    expect(res_0.header["content-type"].startsWith("application/json")).toBe(true);
    expect(res_0.body.errors.length).toBe(1);
    expect(res_0.body.errors[0].message).toBe("invalid input syntax for integer: \"Z\"");
    expect(res_0.body.errors[0].locations.length).toBe(1);
    expect(res_0.body.errors[0].locations[0].line).toBe(1.0);
    expect(res_0.body.errors[0].locations[0].column).toBe(5.0);
    expect(res_0.body.errors[0].path.length).toBe(1);
    expect(res_0.body.errors[0].path[0]).toBe("findStoreById");
    expect(res_0.body.errors[0].extensions.code).toBe("INTERNAL_SERVER_ERROR");
    expect(res_0.body.errors[0].extensions.exception.query).toBe("SELECT \"Store\".\"id\" AS \"Store_id\", \"Store\".\"created_at\" AS \"Store_created_at\", \"Store\".\"updated_at\" AS \"Store_updated_at\", \"Store\".\"name\" AS \"Store_name\", \"Store\".\"bio\" AS \"Store_bio\", \"Store\".\"rate\" AS \"Store_rate\", \"Store\".\"slug\" AS \"Store_slug\", \"Store\".\"street\" AS \"Store_street\", \"Store\".\"city\" AS \"Store_city\", \"Store\".\"state\" AS \"Store_state\", \"Store\".\"country\" AS \"Store_country\", \"Store\".\"neighborhood\" AS \"Store_neighborhood\", \"Store\".\"number\" AS \"Store_number\", \"Store\".\"zipCode\" AS \"Store_zipCode\", \"Store\".\"sales\" AS \"Store_sales\", \"Store__employees\".\"id\" AS \"Store__employees_id\", \"Store__employees\".\"created_at\" AS \"Store__employees_created_at\", \"Store__employees\".\"updated_at\" AS \"Store__employees_updated_at\", \"Store__employees\".\"name\" AS \"Store__employees_name\", \"Store__employees\".\"username\" AS \"Store__employees_username\", \"Store__employees\".\"email\" AS \"Store__employees_email\", \"Store__employees\".\"bio\" AS \"Store__employees_bio\", \"Store__employees\".\"image\" AS \"Store__employees_image\", \"Store__employees\".\"role\" AS \"Store__employees_role\", \"Store__employees\".\"status\" AS \"Store__employees_status\", \"Store__employees\".\"password\" AS \"Store__employees_password\", \"Store__products\".\"id\" AS \"Store__products_id\", \"Store__products\".\"created_at\" AS \"Store__products_created_at\", \"Store__products\".\"updated_at\" AS \"Store__products_updated_at\", \"Store__products\".\"title\" AS \"Store__products_title\", \"Store__products\".\"description\" AS \"Store__products_description\", \"Store__products\".\"brand\" AS \"Store__products_brand\", \"Store__products\".\"sku\" AS \"Store__products_sku\", \"Store__products\".\"price\" AS \"Store__products_price\", \"Store__products\".\"thumbnail\" AS \"Store__products_thumbnail\", \"Store__products\".\"images\" AS \"Store__products_images\", \"Store__products\".\"reviews\" AS \"Store__products_reviews\", \"Store__products\".\"quantity\" AS \"Store__products_quantity\", \"Store__products\".\"dimension\" AS \"Store__products_dimension\", \"Store__products\".\"storeId\" AS \"Store__products_storeId\" FROM \"store\" \"Store\" LEFT JOIN \"store_employees_users\" \"Store_Store__employees\" ON \"Store_Store__employees\".\"storeId\"=\"Store\".\"id\" LEFT JOIN \"users\" \"Store__employees\" ON \"Store__employees\".\"id\"=\"Store_Store__employees\".\"usersId\" LEFT JOIN \"product\" \"Store__products\" ON \"Store__products\".\"storeId\"=\"Store\".\"id\" WHERE \"Store\".\"id\" IN ($1)");
    expect(res_0.body.errors[0].extensions.exception.parameters.length).toBe(1);
    expect(res_0.body.errors[0].extensions.exception.parameters[0]).toBe("Z");
    expect(res_0.body.errors[0].extensions.exception.driverError.length).toBe(90.0);
    expect(res_0.body.errors[0].extensions.exception.driverError.name).toBe("error");
    expect(res_0.body.errors[0].extensions.exception.driverError.severity).toBe("ERROR");
    expect(res_0.body.errors[0].extensions.exception.driverError.code).toBe("22P02");
    expect(res_0.body.errors[0].extensions.exception.driverError.file).toBe("numutils.c");
    expect(res_0.body.errors[0].extensions.exception.driverError.line).toBe("62");
    expect(res_0.body.errors[0].extensions.exception.driverError.routine).toBe("pg_atoi");
    expect(res_0.body.errors[0].extensions.exception.length).toBe(90.0);
    expect(res_0.body.errors[0].extensions.exception.severity).toBe("ERROR");
    expect(res_0.body.errors[0].extensions.exception.code).toBe("22P02");
    expect(res_0.body.errors[0].extensions.exception.file).toBe("numutils.c");
    expect(res_0.body.errors[0].extensions.exception.line).toBe("62");
    expect(res_0.body.errors[0].extensions.exception.routine).toBe("pg_atoi");
    expect(res_0.body.errors[0].extensions.exception.stacktrace.length).toBe(9);
    expect(res_0.body.errors[0].extensions.exception.stacktrace[0]).toBe("QueryFailedError: invalid input syntax for integer: \"Z\"");
    expect(res_0.body.errors[0].extensions.exception.stacktrace[1]).toBe("    at QueryFailedError.TypeORMError [as constructor] (D:\\WORK\\EXPERIMENTS\\graphql-journal\\y100k_0_9\\ecommerce-server\\node_modules\\typeorm\\error\\TypeORMError.js:9:28)");
    expect(res_0.body.errors[0].extensions.exception.stacktrace[2]).toBe("    at new QueryFailedError (D:\\WORK\\EXPERIMENTS\\graphql-journal\\y100k_0_9\\ecommerce-server\\node_modules\\typeorm\\error\\QueryFailedError.js:13:28)");
    // Skipping assertions on the remaining 6 elements. This limit of 3 elements can be increased in the configurations
    expect(res_0.body.data.findStoreById).toBe(null);
});
```

The API here returns the error message *invalid input syntax for integer: "Z"*. In the schema, the `id` input for the query `findStoreById` is of type string (and so EvoMaster did send an input string like *"Z"*), but the API is expecting an integer. Technically, this could be considered as a schema-related fault. However, there might be good reasons for sending a numeric value as a string in JSON, as JSON numbers are considered as 64-bit double-float values. This is an issue if one rather needs to deal with 64-bit integers (e.g., for numeric ids in SQL databases). The major issue here, though, is "where" the check is done. The GraphQL specification allows to provide extra information in the `errors` objects, under an optional field called `extensions`. What can be present in this field depends on the different GraphQL framework implementations. In this particular case, the full stack-trace of an internal thrown exception is added to the response.

First, this could be technically a security issue if the API was in production and not just run locally for testing. Full stack-trace details are useful for debugging, but they expose internal details of the API that could be exploited by external attackers. Second, the exception seems to happen in a SQL SELECT query, which is malformed. The point here is that the id in the SQL database is of type numeric, and so a value like *"Z"* is invalid. However, the API does not check for such integer constraints as a first layer of input validation when a GraphQL query is executed, and rather fail afterwards. This can be a serious problem if there are modifications to the internal state of API before the thrown exception, as the API might be left in a inconsistent state.

> ***RQ3****: Different kinds of faults were automatically detected with our novel techniques, including wrong handling of requests for missing data, and generated responses that do not match the API schemas.*

6 DISCUSSION AND FUTURE DIRECTIONS

This section discusses the main findings of the paper, followed by possible future work. The main findings of using the EvoMaster for automated GraphQL APIs testing can be summarized as follows:

(1) The first finding of this study consists on the difficulty of automatically identifying test cases of GraphQL APIs compared to the RESTful APIs. Indeed, the graph representation of the actions is more complex than the traditional representation of the RESTful APIs. This representation is rich and might be used in different domain applications, however, this needs a careful care of the automated test generation process. Furthermore, whereas a RESTful API can have clear relations between resources based on hierarchical URIs, and that information can be successfully exploited by test generation tools [86], this does not seem the case for GraphQL APIs (e.g., no easy heuristics to determine which resources on the graph each mutation operation might manipulate).

(2) The second finding of this study is that the EvoMaster tool proved its applicability in handling other kinds of web service APIs, represented by GraphQL APIs. EvoMaster was implemented and architectured from the start to be able to be extended and adapted to other system test generation domains besides REST APIs [16]. The results obtained in this paper shows that EvoMaster is a generic enough framework for evolutionary-based system test

generation, at least for applications where the entry point is a TCP connection. Being released as open-source [27], EvoMaster can be further extended and used in other domains as well.

(3) To obtain better code coverage, white-box heuristics based on search-based techniques can help significantly. However, existing APIs can have many faults that can be easily detected by simply sending random inputs. This makes even simple approaches like black-box testing potentially useful for practitioners.

In order to improve the effectiveness of the automated GraphQL API testing, several directions may be investigated in the future:

(1) **Test oracle problem.** Given a test case, whether the result of its execution is correct or not can be determined with an automated *oracle* [35]. Without an automated oracle, the developer has to determine manually whether the observed test results are as expected or not. But having to manually check hundreds/thousands of generated test cases might not viable. As discussed in the paper, query responses with `errors` fields might not be representing actual faults in the SUT, but rather just the user sending wrong data. In order to mitigate the test oracle problem, an intelligent automated strategy is needed to differentiate between the actual faults from the user errors for a given GraphQL API response. One approach is to use machine learning, in particular supervised classification, to automatically label whether a response with `errors` field should be treated as a potential fault that the developer should investigate. When test suites are generated at the end of the search, the test cases could be ordered based on their probability of representing actual faults.

(2) **Evolutionary Computation.** Evolutionary computation is an intelligent mechanism of exploring large and big solution spaces, inspired by the evolutionary process from nature. In this research work, we only used the evolutionary algorithms MIO, MOSA and WTS, but others might be more fitting for the case of GraphQL API testing. In order to further improve the code coverage of the automated testing in this domain, further investigation should be carried out in this area. For example, other techniques such as Particle Swarm Optimization [55] and Ant Colony Optimization [78] could be considered. Combining other testing techniques (e.g., Symbolic Execution [34]) with evolutionary computation can also be considered a good direction to further address this problem [49].

(3) **Knowledge Discovery.** Data mining and knowledge discovery is the process of extracting hidden patterns from a large data collection. Decomposition is a widely used technique in solving complex problems [43, 44]. The aim is to create highly correlated clusters, where each cluster contains similar data. In our context, the idea is to apply the decomposition method to the GraphQL schema in order to derive sub-graphs of schema. Each sub-graph might contain highly connected actions. Good decomposition methods allow to find independent sub-graphs as much as possible, in order to enable the same test case generation while dealing with the sub-graphs as when dealing with the entire GraphQL schema.

(4) **Industrial Settings.** Further investigations with more case studies will be essential to generalize the effectiveness of our novel technique. Of particular importance it will be to apply our technique in industrial settings, to see and evaluate how engineers would use tools like EvoMaster in practice on their APIs.

7 THREATS TO VALIDITY

Threats to internal validity come from the fact that our experiments are derived from a software tool. Errors in such a tool could negatively affect the validity of our empirical results. Although our EvoMaster extension was carefully tested, we cannot provide any guarantee of not having software faults. However, as it is open-source, anyone can review its source code. Another potential issue is that the implemented solution in this research work is based on random algorithms. This happens in particular for population initialization of the evolutionary algorithm, where different test cases may be generated. To deal with this issue, each experiment for white-box testing was repeated 30 times [23], with different random seeds, and the appropriate statistical tests were used to analyze the results. All the APIs used for the white-box experiments are collected in a GitHub repository called EMB [6], which is stored on Zenodo as well [33]. Furthermore, all of our scripts used to carry out our experiments are stored as part of the repository of EvoMaster. This is done to enable third-parties to replicate and validate our experiments. However, experiments for black-box testing cannot be reliably replicated, as they rely on live services on which we do not have any control on (e.g., they can be modified at any time by their owners).

Threats to external validity are due to the fact that only 7 GraphQL APIs for white-box testing, and 31 GraphQL APIs for black-box testing, were used in our empirical analysis. The generalization of such results to other APIs might not be possible at this stage. More APIs should be investigated in the future. However, as this is the first work on white-box testing of GraphQL APIs, already achieving good coverage and finding real faults on a complex GraphQL API provide a promising first step.

8 CONCLUSIONS

This paper introduced a new approach for automated testing for GraphQL APIs. It is a full complete solution, starting from the schema extraction and ending by automatically generating test cases outputted in JUnit and Jest format. Two testing modes are implemented and evaluated: white-box and black box testing.

In order to intelligently explore the test case space, evolutionary computation techniques are used in the white-box testing. Two mutation operators (internal and structure mutation) are defined, where the goal is to maximize code coverage and fault-finding. In addition, random testing is used for the black-box mode.

To validate the applicability of the proposed framework, it is integrated into the EvoMaster open-source tool. Our empirical analysis was carried out on 7 GraphQL APIs for white-box testing, empirically comparing 3 different evolutionary algorithms, and 31 GraphQL APIs for black-box testing. The results show the clear improvement of using evolutionary computation compared with the random search baseline for white-box testing. Regarding black-box testing, several real faults were found by random testing in the analyzed APIs.

The work presented in this paper is currently integrated in our EvoMaster tool. It is open-source on GitHub [5], with each new release automatically uploaded and stored on Zenodo (e.g., [29]). To learn more about EvoMaster, visit `www.evomaster.org`.

ACKNOWLEDGMENTS

This work is funded by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (EAST project, grant agreement No. 864972).

REFERENCES

[1] [n.d.]. AFL. https://github.com/google/AFL.
[2] [n.d.]. apis.guru. https://apis.guru/graphql-apis/.
[3] [n.d.]. Apollo GraphQL. https://github.com/apollographql.
[4] [n.d.]. e-commerce. https://github.com/react-shop/react-ecommerce.
[5] [n.d.]. EvoMaster. https://github.com/EMResearch/EvoMaster.
[6] [n.d.]. EvoMaster Benchmark (EMB). https://github.com/EMResearch/EMB. Online, Accessed May 20, 2022.
[7] [n.d.]. Github. https://github.com.
[8] [n.d.]. GraphQL Foundation. https://graphql.org/foundation/.

[9] [n.d.]. Patio-api. https://github.com/patio-team/patio-api.

[10] [n.d.]. petclinic. https://github.com/spring-petclinic/spring-petclinic-graphql.

[11] [n.d.]. react-finland. https://github.com/ReactFinland/graphql-api.

[12] [n.d.]. timbuctoo. https://github.com/HuygensING/timbuctoo.

[13] Shaukat Ali, Lionel C Briand, Hadi Hemmati, and Rajwinder Kaur Panesar-Walawege. 2009. A systematic review of the application and empirical investigation of search-based test case generation. *IEEE Transactions on Software Engineering* 36, 6 (2009), 742--762.

[14] Mohammad Alshraideh and Leonardo Bottaci. 2006. Search-based software test data generation for string data using program-specific search operators. *Software Testing, Verification, and Reliability* 16, 3 (2006), 175--203. https://doi.org/10.1002/stvr.v16:3

[15] Andrea Arcuri. 2017. Many Independent Objective (MIO) Algorithm for Test Suite Generation. In *International Symposium on Search Based Software Engineering (SSBSE)*. 3--17.

[16] Andrea Arcuri. 2018. EvoMaster: Evolutionary Multi-context Automated System Test Generation. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE.

[17] Andrea Arcuri. 2018. An experience report on applying software testing academic results in industry: we need usable automated test generation. *Empirical Software Engineering* 23, 4 (2018), 1959--1981.

[18] Andrea Arcuri. 2018. Test suite generation with the Many Independent Objective (MIO) algorithm. *Information and Software Technology* 104 (2018), 195--206.

[19] Andrea Arcuri. 2019. RESTful API Automated Test Case Generation with EvoMaster. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 28, 1 (2019), 3.

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

[21] A. Arcuri and L. Briand. 2011. Adaptive Random Testing: An Illusion of Effectiveness?. In *ACM Int. Symposium on Software Testing and Analysis (ISSTA)*. 265--275.

[22] A. Arcuri and L. Briand. 2011. Formal Analysis of the Probability of Interaction Fault Detection Using Random Testing. *IEEE Transactions on Software Engineering (TSE)* (2011). to appear.

[23] A. Arcuri and L. Briand. 2014. A Hitchhiker's Guide to Statistical Tests for Assessing Randomized Algorithms in Software Engineering. *Software Testing, Verification and Reliability (STVR)* 24, 3 (2014), 219--250.

[24] Andrea Arcuri and Juan P Galeotti. 2020. Handling SQL databases in automated system test generation. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 29, 4 (2020), 1--31.

[25] Andrea Arcuri and Juan P Galeotti. 2020. Testability transformations for existing APIs. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 153--163.

[26] Andrea Arcuri and Juan P Galeotti. 2021. Enhancing Search-based Testing with Testability Transformations for Existing APIs. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 1 (2021), 1--34.

[27] Andrea Arcuri, Juan Pablo Galeotti, Bogdan Marculescu, and Man Zhang. 2021. EvoMaster: A Search-Based System Test Generation Tool. *Journal of Open Source Software* 6, 57 (2021), 2153.

[28] A. Arcuri, M. Z. Iqbal, and L. Briand. 2012. Random Testing: Theoretical Results and Practical Implications. *IEEE Transactions on Software Engineering (TSE)* 38, 2 (2012), 258--277.

[29] Andrea Arcuri, Man Zhang, Asma Belhadi, Bogdan, Juan Pablo Galeotti, Seran, Amid Gol, Alberto Martín López, Agustina Aldasoro, Annibale Panichella, and Kyle Niemeyer. 2023. *EMResearch/EvoMaster: 1.6.1*. https://doi.org/10.5281/zenodo.7821550

[30] Andrea Arcuri, Man Zhang, Asma Belhadi, Bogdan Marculescu, Amid Golmohammadi, Juan Pablo Galeotti, and Susruthan Seran. 2023. Building an open-source system test generation tool: lessons learned and empirical analyses with EvoMaster. *Software Quality Journal* (2023), 1--44.

[31] Andrea Arcuri, ZhangMan, asmab89, Bogdan, Amid Golmohammadi, Juan Pablo Galeotti, Seran, Alberto Martín López, Agustina Aldasoro, Annibale Panichella, and Kyle Niemeyer. 2022. *EMResearch/EvoMaster:*. https://doi.org/10.5281/zenodo.6651631

[32] Andrea Arcuri, ZhangMan, asmab89, Bogdan, Amid Golmohammadi, Juan Pablo Galeotti, Seran, Alberto Martín López, Agustina Aldasoro, Annibale Panichella, and Kyle Niemeyer. 2022. *EMResearch/EvoMaster:*. https://doi.org/10.5281/zenodo.6651631

[33] Andrea Arcuri, ZhangMan, Amid Golmohammadi, and asmab89. 2022. *EMResearch/EMB:*. https://doi.org/10.5281/zenodo.6106830

[34] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2018. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)* 51, 3 (2018), 1--39.

[35] Earl T Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering (TSE)* 41, 5 (2015), 507--525.

[36] Asma Belhadi, Man Zhang, and Andrea Arcuri. 2022. Evolutionary-based Automated Testing for GraphQL APIs. In *Genetic and Evolutionary Computation Conference (GECCO)*.

[37] Edwin Cabrera, Paola Cárdenas, Priscila Cedillo, and Paola Pesántez-Cabrera. 2020. Towards a Methodology for creating Internet of Things (IoT) Applications based on Microservices. In *2020 IEEE International Conference on Services Computing (SCC)*. IEEE, 472--474.

[38] José Campos, Yan Ge, Nasser Albunian, Gordon Fraser, Marcelo Eler, and Andrea Arcuri. 2018. An empirical evaluation of evolutionary algorithms for unit test suite generation. *Information and Software Technology (IST)* 104 (2018), 207--235.

[39] Alan Cha, Erik Wittern, Guillaume Baudart, James C Davis, Louis Mandel, and Jim A Laredo. 2020. A principled approach to GraphQL query cost analysis. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 257--268.

[40] Flavio Cirillo, David Gómez, Luis Diez, Ignacio Elicegui Maestro, Thomas Barrie Juel Gilbert, and Reza Akhavan. 2020. Smart city IoT services creation through large-scale collaboration. *IEEE Internet of Things Journal* 7, 6 (2020), 5267--5275.

[41] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation (TEVC)* 6, 2 (2002), 182--197.

[42] Tomás Díaz, Federico Olmedo, and Éric Tanter. 2020. A mechanized formalization of GraphQL. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*. 201--214.

[43] Youcef Djenouri, Asma Belhadi, Philippe Fournier-Viger, and Jerry Chun-Wei Lin. 2018. Fast and effective cluster-based information retrieval using frequent closed itemsets. *Information Sciences* 453 (2018), 154--167.

[44] Youcef Djenouri, Jerry Chun-Wei Lin, Kjetil Nørvåg, Heri Ramampiaro, and Philip S Yu. 2021. Exploring Decomposition for Solving Pattern Mining Problems. *ACM Transactions on Management Information Systems (TMIS)* 12, 2 (2021), 1--36.

[45] S. Droste, T. Jansen, and I. Wegener. 2002. On the analysis of the (1+1) evolutionary algorithm. *Theoretical Computer Science* 276 (2002), 51--81.

[46] Carles Farré, Jovan Varga, and Robert Almar. 2019. GraphQL schema generation for data-intensive web APIs. In *Model and Data Engineering: 9th International Conference, MEDI 2019, Toulouse, France, October 28--31, 2019, Proceedings 9*. Springer, 184--194.

[47] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: automatic test suite generation for object-oriented software. In *ACM Symposium on the Foundations of Software Engineering (FSE)*. 416--419.

[48] Gordon Fraser and Andrea Arcuri. 2013. Whole Test Suite Generation. *IEEE Transactions on Software Engineering* 39, 2 (2013), 276--291.

[49] Juan Pablo Galeotti, Gordon Fraser, and Andrea Arcuri. 2014. Extending a search-based test generator with adaptive dynamic symbolic execution. In *ACM Int. Symposium on Software Testing and Analysis (ISSTA)*. ACM, 421--424.

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

[51] Amid Golmohammadi, Man Zhang, and Andrea Arcuri. 2022. Testing RESTful APIs: A Survey. *arXiv preprint arXiv:2212.14604* (2022).

[52] Mark Harman, Lin Hu, Rob Hierons, Joachim Wegener, Harmen Sthamer, André Baresel, and Marc Roper. 2004. Testability transformation. *IEEE Transactions on Software Engineering (TSE)* 30, 1 (2004), 3--16.

[53] Mark Harman, S Afshin Mansouri, and Yuanyuan Zhang. 2012. Search-based software engineering: Trends, techniques and applications. *ACM Computing Surveys (CSUR)* 45, 1 (2012), 11.

[54] Olaf Hartig and Jorge Pérez. 2018. Semantics and complexity of GraphQL. In *Proceedings of the 2018 World Wide Web Conference*. 1155--1164.

[55] M Shamim Hossain, Mohd Moniruzzaman, Ghulam Muhammad, Ahmed Ghoneim, and Atif Alamri. 2016. Big data-driven service composition using parallel clustered particle swarm optimization in mobile environment. *IEEE Transactions on Services Computing* 9, 5 (2016), 806--817.

[56] Stefan Karlsson, Adnan Čaušević, and Daniel Sundmark. 2020. Automatic Property-based Testing of GraphQL APIs. *arXiv preprint arXiv:2012.07380* (2020).

[57] Raees Khan and Adnan Noor Mian. 2020. Sustainable IoT sensing applications development through graphQL-based abstraction layer. *Electronics* 9, 4 (2020), 564.

[58] Myeongsoo Kim, Qi Xin, Saurabh Sinha, and Alessandro Orso. 2022. Automated Test Generation for REST APIs: No Time to Rest Yet. https://doi.org/10.48550/ARXIV.2204.08348

[59] 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 (ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 289–301. https://doi.org/10.1145/3533767.3534401

[60] Bogdan Korel. 1990. Automated software test data generation. *IEEE Transactions on software engineering* 16, 8 (1990), 870--879.

[61] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: Multi-objective automated testing for android applications. In *ACM Int. Symposium on Software Testing and Analysis (ISSTA)*. ACM, 94--105.

[62] Bogdan Marculescu, Man Zhang, and Andrea Arcuri. 2022. On the Faults Found in REST APIs by Automated Test Generation. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 3 (2022), 1--43.

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

[64] Sam Newman. 2015. *Building Microservices.* " O'Reilly Media, Inc.".

[65] Annibale Panichella, Fitsum Kifetew, and Paolo Tonella. 2018. Automated Test Case Generation as a Many-Objective Optimisation Problem with Dynamic Selection of the Targets. *IEEE Transactions on Software Engineering (TSE)* 44, 2 (2018), 122--158.

[66] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. 2018. A large scale empirical comparison of state-of-the-art search-based test case generators. *Information and Software Technology* 104 (2018), 236--256.

[67] Antonio Quiña-Mera, Pablo Fernandez, José María García, and Antonio Ruiz-Cortés. 2023. GraphQL: A Systematic Mapping Study. *Comput. Surveys* 55, 10 (2023), 1--35.

[68] Roberto Rodriguez-Echeverria, Javier Luis Cánovas Izquierdo, and Jordi Cabot. 2018. Towards a UML and IFML Mapping to GraphQL. In *Current Trends in Web Engineering: ICWE 2017 International Workshops, Liquid Multi-Device Software and EnWoT, practi-O-web, NLPIT, SoWeMine, Rome, Italy, June 5-8, 2017, Revised Selected Papers 17*. Springer, 149--155.

[69] José Miguel Rojas, José Campos, Mattia Vivanti, Gordon Fraser, and Andrea Arcuri. 2015. Combining multiple coverage criteria in search-based unit test generation. In *International Symposium on Search Based Software Engineering*. Springer, 93--108.

[70] José Miguel Rojas, Mattia Vivanti, Andrea Arcuri, and Gordon Fraser. 2017. A detailed investigation of the effectiveness of whole test suite generation. *Empirical Software Engineering (EMSE)* 22, 2 (2017), 852--893.

[71] Pavel Seda, Pavel Masek, Jindriska Sedova, Milos Seda, Jan Krejci, and Jiri Hosek. 2018. Efficient architecture design for software as a service in cloud environments. In *2018 10th International Congress on Ultra Modern Telecommunications and Control Systems and Workshops (ICUMT)*. IEEE, 1--6.

[72] Ruben Taelman, Miel Vander Sande, and Ruben Verborgh. 2018. GraphQL-LD: linked data querying with GraphQL. In *ISWC2018, the 17th International Semantic Web Conference*. 1--4.

[73] Daniela Meneses Vargas, Alison Fernandez Blanco, Andreina Cota Vidaurre, Juan Pablo Sandoval Alcocer, Milton Mamani Torres, Alexandre Bergel, and Stéphane Ducasse. 2018. Deviation testing: A test case generation technique for GraphQL APIs. In *11th International Workshop on Smalltalk Technologies (IWST)*. 1--9.

[74] Milena Vesić and Nenad Kojić. 2020. N. Comparative Analysis of Web Application Performance in Case of Using REST versus GraphQL. In *Proceedings of the Fourth International Scientific Conference on Recent Advances in Information Technology, Tourism, Economics, Management and Agriculture (ITEMA), Online-Virtual*. 17--24.

[75] Emanuele Viglianisi, Michael Dallago, and Mariano Ceccato. 2020. RESTTESTGEN: Automated Black-Box Testing of RESTful APIs. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE.

[76] Maximilian Vogel, Sebastian Weber, and Christian Zirpins. 2018. Experiences on migrating RESTful web services to GraphQL. In *Service-Oriented Computing--ICSOC 2017 Workshops: ASOCA, ISyCC, WESOACS, and Satellite Events, Málaga, Spain, November 13--16, 2017, Revised Selected Papers*. Springer, 283--295.

[77] D. H. Wolpert and W. G. Macready. 1997. No Free Lunch Theorems for Optimization. *IEEE Transactions on Evolutionary Computation* 1, 1 (1997), 67--82.

[78] Quanwang Wu and Qingsheng Zhu. 2013. Transactional and QoS-aware dynamic service composition based on ant colony optimization. *Future Generation Computer Systems* 29, 5 (2013), 1112--1119.

[79] Louise Zetterlund, Deepika Tiwari, Martin Monperrus, and Benoit Baudry. 2022. Harvesting Production GraphQL Queries to Detect Schema Faults. In *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 365--376.

[80] Man Zhang and Andrea Arcuri. 2021. Adaptive Hypermutation for Search-Based System Test Generation: A Study on REST APIs with EvoMaster. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 1 (2021).

[81] Man Zhang and Andrea Arcuri. 2022. Open Problems in Fuzzing RESTful APIs: A Comparison of Tools. *arXiv preprint arXiv:2205.05325* (2022).

[82] Man Zhang and Andrea Arcuri. 2023. Open Problems in Fuzzing RESTful APIs: A Comparison of Tools. *ACM Transactions on Software Engineering and Methodology (TOSEM)* (may 2023). https://doi.org/10.1145/3597205 Just Accepted.

[83] Man Zhang, Andrea Arcuri, Yonggang Li, Yang Liu, and Kaiming Xue. 2023. White-Box Fuzzing RPC-Based APIs with EvoMaster: An Industrial Case Study. *ACM Transactions on Software Engineering and Methodology (TOSEM)* (feb 2023).

https://doi.org/10.1145/3585009 Just Accepted.

[84] Man Zhang, Andrea Arcuri, Yonggang Li, Kaiming Xue, Zhao Wang, Jian Huo, and Weiwei Huang. 2022. Fuzzing Microservices In Industry: Experience of Applying EvoMaster at Meituan. https://doi.org/10.48550/ARXIV.2208.03988

[85] Man Zhang, Asma Belhadi, and Andrea Arcuri. 2022. JavaScript Instrumentation for Search-Based Software Testing: A Study with RESTful APIs. In *IEEE International Conference on Software Testing, Verification and Validation (ICST).* IEEE.

[86] Man Zhang, Bogdan Marculescu, and Andrea Arcuri. 2019. Resource-based test case generation for RESTful web services. In *Proceedings of the Genetic and Evolutionary Computation Conference.* 1426--1434.

[87] Man Zhang, Bogdan Marculescu, and Andrea Arcuri. 2021. Resource and dependency based test case generation for RESTful Web services. *Empirical Software Engineering* 26, 4 (2021), 1--61.