

# Seeding and Mocking in White-Box Fuzzing Enterprise RPC APIs: An Industrial Case Study\*

Man Zhang

manzhang@buaa.edu.cn  
State Key Laboratory of Complex &  
Critical Software Environment  
(CCSE), Beihang University  
Beijing, China

Andrea Arcuri

andrea.arcuri@kristiania.no  
Kristiania University College and  
Oslo Metropolitan University  
Oslo, Norway

Piyun Teng

Kaiming Xue

Wenhao Wang

tengpiyun@meituan.com  
xuekaiming@meituan.com  
wangwenhao02@meituan.com  
Meituan  
Beijing, China

## ABSTRACT

Microservices is now becoming a promising architecture to build large-scale web services in industry. Due to the high complexity of enterprise microservices, industry has an urgent need to have a solution to enable automated testing of such systems. EvoMASTER is an open-source fuzzer, equipped with the state-of-the-art techniques for supporting automated system-level testing of Web APIs. It has been assessed as the most performant tool in two recent empirical studies in terms of line coverage and fault detection. In this paper, we carried out an empirical experiment to investigate how to better apply the state-of-the-art academic prototype (i.e., EvoMASTER) in industrial context. We extended the tool to handle seeding of existing industrial tests, and mocking of external services with their data handled as part of the input fuzzing. We studied two configurations of EvoMASTER, using two time budgets, on 40 enterprise RPC-based APIs (involving 5.6 million lines of code for their core business logic) at Meituan. Results show that, compared to existing practice of manual system-level testing and tests produced by record and replay of online traffic, EvoMASTER demonstrates clear additional benefits. EvoMASTER with the best configuration is capable of covering up to 32.4% line coverage, covering more than 10% line coverage on 36 out of 40 (90%) case studies, and identifying on average 3520 potential faults in these 40 APIs. In addition, we also identified and discussed important challenges in fuzzing enterprise microservices that must be addressed in the future.

## CCS CONCEPTS

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

\*Man Zhang is supported by State Key Laboratory of Complex & Critical Software Environment (CCSE, grant No. CCSE-2024ZX-01). Andrea Arcuri 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).

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

ASE '24, October 27-November 1, 2024, Sacramento, CA, USA

© 2024 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-1248-7/24/10

<https://doi.org/10.1145/3691620.3695265>

## KEYWORDS

Fuzzing, SBST, Microservices, Automated Test Case Generation

### ACM Reference Format:

Man Zhang, Andrea Arcuri, Piyun Teng, Kaiming Xue, and Wenhao Wang. 2024. Seeding and Mocking in White-Box Fuzzing Enterprise RPC APIs: An Industrial Case Study. In *39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24)*, October 27-November 1, 2024, Sacramento, CA, USA. ACM, New York, NY, USA, 11 pages. <https://doi.org/10.1145/3691620.3695265>

## 1 INTRODUCTION

Due to advantages of microservice architectures, such as rapid deployment and increased scalability, the use of microservices now has become a common practice to build large-scale enterprise services [36, 44], e.g., at Amazon [28], Netflix [28], Alibaba [32] and Meituan [49, 50]. Enterprise microservices can comprise hundreds of services [32, 49]. These services can communicate with each other through Application Programming Interface (API) techniques, e.g., REST, Remote Procedure Call (RPC), and GraphQL. However, such an architecture poses lots of challenges in its validation and verification [44]. Therefore, there is an increasing interest for tackling these challenges in academic research and in industry [32, 34, 49, 50, 53, 54].

To investigate how these challenges have been addressed, and learn what bottlenecks still exist that hamper the fuzzing of enterprise microservices, we chose the open-source fuzzer EvoMASTER [18] for this industrial study. In particular, we conducted an empirical study using EvoMASTER on 40 enterprise RPC APIs from a large-scale microservices at Meituan. Regarding the fuzzer selection, based on the results of two recent studies on REST APIs [30? ], white-box EvoMASTER is rated as the most performant tool in code coverage and fault detection. In addition, to the best of our knowledge, EvoMASTER is the only available open-source fuzzer that supports automated fuzzing of modern RPC APIs [49], in particular Thrift (used by those 40 APIs). To better adapt EvoMASTER in industrial settings, and improve performance, we also extended EvoMASTER to facilitate *seeding* existing enterprise tests and enabling *mocking* of external services with in-house developed techniques.

Experiments results on 40 RPC APIs at Meituan show that 1) compared to existing system-level testing practices at Meituan, EvoMASTER can bring clear additional benefits in covering more code, which existing manual tests do not cover, and identifying more potential faults; 2) as for fuzzing enterprise APIs, time budgets such

as 1 hour might not be enough, and more budgets (like 10 hours) can provide significantly better results; and 3) EvoMASTER with the best configuration is capable of achieving on average 17.0% (up to 32.4%) line coverage, on average 15.7% (up to 47.7%) line coverage for critical implementation parts, and detecting on average 3520 ( $88 \times 40$ ) potential faults in these 40 APIs. In addition, we identified important bottlenecks by investigating in details the APIs where EvoMASTER shows limited performance on. These issues include supporting Message Queues and Scheduled Tasks techniques. Finally, we summarize the lessons we learned in applying EvoMASTER in practice in a large enterprise such as Meituan.

## 2 BACKGROUND

### 2.1 Microservices and RPC

Microservices is an architecture to build modern large-scale enterprise web services that can comprise hundreds of services [32, 49]. Each service can interact with each other via APIs. Processing a request from customers can result in a series of invocations among multiple services [26, 40] and interactions with databases (Figure 1).

Remote Procedure Call (RPC) is a communication protocol which supports request–response interactions among services over network. For instance, as shown in Figure 1, using a client-stub of *service B* (e.g., *stubB*), *service A* can call methods of the *service B*, e.g., *stubB.method(42)*, then results of the call is utilized in handling a request to the *service A*. RPC has been widely applied in building distributed microservices [36], and nowadays, there exists various RPC frameworks, such as Apache Thrift [9], Apache Dubbo [2], gRPC [3] and SOFARPC [8]. However, all these frameworks share a concept of *interface* which comprises exposed functions to call [49], e.g., Apache Thrift [10] and gRPC [1] support to define the services using the Interface Definition Language (IDL). For instance, with Apache Thrift, a RPC service can be defined with IDL as:

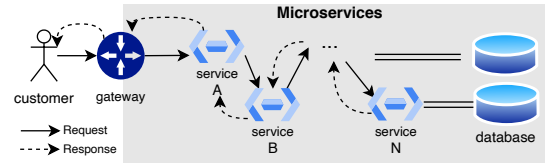
```
1 namespace java rpc
2 service SUT {
3     i32 method (1: i32 a, 2: i32 b, 3: i32 c)
4 }
5
6 Based on the IDL definition, the Thrift compiler is able to generate
7 source code of server and client library in different programming
8 languages, e.g., an example in Java:
9
10 package rpc;
11 @SuppressWarnings({"cast", "rawtypes", "serial", "unchecked", "unused"})
12 @javax.annotation.Generated(value = "Autogenerated by Thrift Compiler (0.15.0)", date = "2023-12-07")
13 public class SUT {
14     public interface Iface {
15         public int method(int a, int b, int c) throws org.apache.thrift.TException;
16     }
17     public static class Client extends org.apache.thrift.TServiceClient implements Iface {...}
18 }
19
```

The business logic can be developed by implementing the interface `rpc.SUT$Iface` at line 6, and the services can be accessed through the corresponding client at line 8.

### 2.2 EvoMaster

EvoMASTER [18] is an open-source fuzzer for enabling automated fuzzing for Web APIs, supporting both black-box and white-box testing [13]. For white-box mode, it supports APIs developed with Java/Kotlin [12], JavaScript/TypeScript [51] and C# [24].

EvoMASTER employs evolutionary algorithms to tackle test case generation problem for Web APIs. It integrates state-of-the-art



**Figure 1: An example of service invocations for handling a request from customers in Microservices**

search algorithms for test suite generation, such as the Many Independent Objective (MIO) algorithm [11] (default search algorithm in EvoMASTER), the Whole Test Suite (WTS) approach [22] and the Many-Objective Sorting Algorithm (MOSA) [37]. To better fuzz Web APIs with search, EvoMASTER has been enhanced with advanced techniques, such as *testability transformation* [16], *SQL handling* [15], and *adaptive hypermutation* [46].

With respects to various API techniques, EvoMASTER enables automated fuzzing of APIs built with REST [12, 47, 52], GraphQL [20] and RPC [49]. In terms of fuzzing REST APIs, EvoMASTER is the only fuzzer which supports white-box testing using an automated solution to extract runtime execution information [25], such as code coverage. In addition, two recent empirical studies demonstrate that white-box EvoMASTER is the most performant tool in code coverage and fault detection [30?]. Moreover, as EvoMASTER is the first approach for fuzzing RPC APIs and provide a generic solution to support diverse RPC frameworks (such as gRPC [3] and Apache Thrift [9]) [49], we chose EvoMASTER as the state-of-the-art to investigate its effectiveness and its application strategy in testing enterprise RPC APIs from large-scale microservices.

## 3 RELATED WORK

Most studies for Web API fuzzing are for REST APIs, e.g., [19, 27, 29–31, 35, 42, 45?], on open-source case studies. Regarding microservices testing, there exist approaches for regression testing [23, 32], handling the oracle problem with metamorphic testing [33], fault analysis and debugging [53, 54], and reliability testing [21].

In the context of fuzzing enterprise APIs that are parts of a large-scale microservices, there exist three most relevant works [43, 49, 50]. At the beginning of collaboration with Meituan, we conducted a study to assess applicability and effectiveness of EvoMASTER for industrial APIs from microservices with an aim at investigating potential integration into industrial pipelines [50]. The *user study* was conducted with EvoMASTER version 1.2.0 using REST API fuzzing on two enterprise APIs. This study was carried out mainly from the point of views of practitioners, e.g., how difficult to use EvoMASTER, how they rate quality of EvoMASTER, and what main challenges they face. With this study, one important challenge we identified is the lack of automated fuzzing solution for RPC APIs. Then, in [49], we proposed the first open-source solution to automate fuzzing RPC APIs, built on top of EvoMASTER. The approach for fuzzing RPC API with EvoMASTER was also assessed with case studies from Meituan. Compared to [49], in this paper we extended EvoMASTER for better adopting the tool in addressing enterprise APIs (e.g., with seeding and mocking), conducted a larger scale experiment with 40 RPC APIs (40 vs. 4) involving more types of APIs, carried out more analyses in terms of characteristics of the APIs and search budgets for applying EvoMASTER, and identified new bottlenecks and challenges. The third most relevant work is Zero-Config developed by Wang et al. [43] that facilitates automated test generations for C++

gRPC APIs based on gRPC schemas specified with Protobuf [6], using LibFuzzer [5]. It has been assessed by fuzzing C++ microservices at Google. Results show that Zero-Config can identify new bugs with a fix rate of 67.71%. Compared to the empirical study conducted for Zero-Config [43], our study provides detailed info on the case studies for understanding our experiment results, more analyses based on the results with respects to enterprise APIs, and recommendations of applying our fuzzer. Note that, as Zero-Config seems not open-source, it is impossible for us to assess it in our study (besides the fact that it targets gRPC APIs written in C++).

## 4 INDUSTRIAL ADOPTION

To better support integration into industrial processes, we have extended EvoMASTER to adapt it to mocking techniques developed in-house by enterprises (§4.1), and enabling the seeding of existing tests (§4.2), as discussed in more details next.

### 4.1 Mocking Adaption Support

With microservice architectures, a business logic implementation can spread over multiple services. Thus, how a service handles a request may depend on *what responses other dependent services reply or what data there exists in connected databases*. For instance, as an example shown in Figure 2, a service under test SUT interacts with a service Foo and connects to a database Bar. When a request to the abc of SUT (line 2), it first requires to get a response from the service Foo (line 3), and then query related data from the database Bar (line 4). In this case, the following execution flow not only relies on values of input parameters of the request (i.e., a, b, and c), but also it depends on the response from Foo and data in Bar. Therefore, to better test the services from microservices, it is necessary to handle such dependent external services (e.g., the external API Foo and the database Bar).

Mocking is a well-known technique to manipulate the dependent external services. However, each company might have their own mocking techniques specialized for their type of services. Those are needed to be able to write automated system and integration tests. Therefore, in order to test enterprise APIs in microservice architectures, a fuzzer needs to have a flexible solution to enable exploiting such existing mocking techniques developed in-house.

Designing a technique in EvoMASTER that can work only on the systems developed at Meituan would be of limited scientific value. It should rather be general, with minimal manual effort needed to adapt it to the different in-house mocking approaches. To satisfy this need, we extended EvoMASTER to make it more customizable by providing configurations for customizing mocking setups [7]. These configurations allow users to define how to use their own mocking techniques to set up mock objects evolved by EvoMASTER, i.e., responses of dependent external APIs, and results of queries of database. We further extended individual (which is the internal representation of evolving test cases in EvoMASTER) to evolve results of dependent external services as part of the search. Figure 3 shows an example chromosome of RPCIndividual, which is composed of a sequence of Genes associated with Actions. Note that EvoMASTER has a sophisticated search-based engine, needed to support different types of problems (e.g., REST, GraphQL and RPC). In the context of testing RPC APIs, besides RPCCallAction for making a RPC request, we introduced two new types of actions, i.e., RPCExternalServiceAction, representing an invocation of

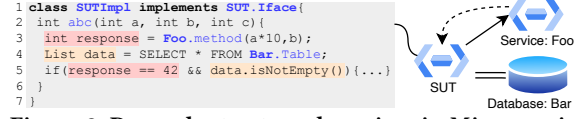


Figure 2: Dependent external services in Microservices

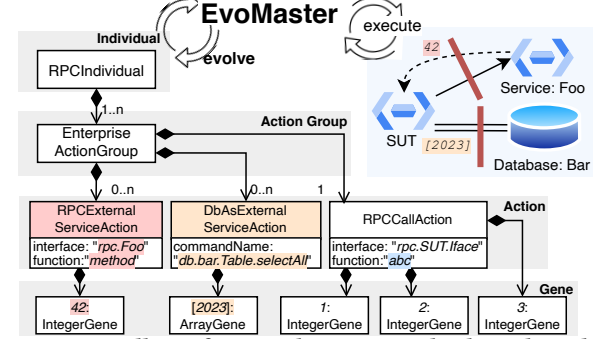


Figure 3: Handling of external services with adapted mocking techniques for covering line 5 in Figure 2

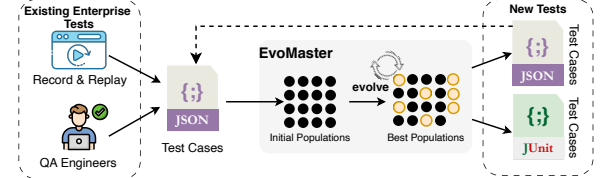


Figure 4: Seeding existing enterprise tests in fuzzing industrial APIs with EvoMASTER

external RPC APIs, and DbAsExternalServiceAction, representing a query of connected databases. Both types of actions contains a Gene for a result value being returned by the mocking. These genes will evolve throughout the search process (e.g., with different mutation operators based on their type, like strings, numbers and JSON objects). As shown in Figure 3, the method of the Foo API returns an integer represented as IntegerGene, and the query to Table returns a list of available rows represented as ArrayGene. With such reformulation, the returned results by the dependent external services can be further evolved for maximizing code coverage and fault findings as part of the search. In this example, line 5 in Figure 2 can be covered only if the method of the API Foo returns 42 (i.e., the value of IntegerGene is 42) and the query to the Table in database Bar return non-empty data (i.e., elements of ArrayGene is not empty) as shown in Figure 3.

### 4.2 Seeding

In industry, using test cases is a common practice to assure software quality. Typically, these tests can be implemented manually by engineers and testers, or can be generated automatically using some tools. For instance, at Meituan, there exists two types of system-level tests, as shown in Figure 4: one type is written manually by quality assurance engineers, and the other is generated automatically based on traffic record-and-replay. As such tests represent to some extent concerns of the services to test from practitioners' perspective and business logic, it may bring extra benefits to automated testing tools (such as EvoMASTER) by utilizing the tests.

Seeding is one possible method to improve efficiency of the search, instead of starting from a randomly initialized population [39]. In the context of Search-Based Software Engineering,



we have extended EvoMASTER to employ such existing tests to initialize populations instead of sampling the populations at random. Then, such tests can be further evolved during the search. To enable seeding of the existing enterprise tests, we extend EvoMASTER and define a domain specific language model (e.g., SeededRPCTestDto and SeededRPCActionDto [7]) that captures necessary information about tests for fuzzing RPC-based APIs and supports JSON serialization. For example:

```

1 {
2   "testName": "test_1",
3   "rpcFunctions": [{
4     "interfaceName": "rpc.SUT",
5     "functionName": "abc",
6     "inputParams": ["1", "2", "3"],
7     "inputParamTypes": ["int", "int", "int"],
8     "mockRPCExternalServiceDtos": [{
9       "interfaceFullName": "rpc.Foo",
10      "functionName": "method",
11      "responses": ["42"],
12      "responseTypes": ["int"]}],
13    "mockDatabaseDtos": [{
14      "databaseOrFrameworkType": "MyBatis",
15      "sqlCommand": "SELECT * FROM Bar.Table",
16      "commandName": "db.bar.Table.selectAll",
17      "response": "[\"2023\"]",
18      "responseFullType": "java.util.ArrayList",
19      "responseFullTypeWithGeneric": "int"]}]
20  }]
21 }
```

As the example shows, a test is composed of a sequence of RPC invocations, e.g., lines 3--20. For each RPC invocation, besides values of the input parameters, EvoMASTER can now set up responses of dependent external services, e.g., lines 8--12 for handling mocking of external RPC APIs and lines 13--19 for managing mocking of databases.

At Meituan, tests produced based on replay-and-record are specified in JSON. Therefore, it is straightforward to convert them into tests with SeededRPCTestDto. Regarding the existing manually written tests, to better handle complex test data in enterprises and maintain the tests according to updates of software, a typical approach is to keep test data separated from the test scripts. At Meituan, this test data is specified in JSON. With the additional info about the functions to test, these tests can be easily transformed into SeededRPCTestDto used by EvoMASTER. Such transformations have been automated at Meituan now with some scripts. This enabled to easily reuse for seeding in EvoMASTER thousands of already existing tests manually developed at Meituan.

## 5 EMPIRICAL STUDY

To assess EvoMASTER in an industrial context, we carried out an empirical study to answer the following research questions regarding fuzzing industrial microservices with EvoMASTER:

- RQ1:** How do characteristics of microservices impact on the effectiveness of EvoMASTER?
- RQ2:** Compared to existing tests in industry, what additional benefits can EvoMASTER bring?
- RQ3:** How does EvoMASTER perform by seeding existing industrial tests and mocking external services?
- RQ4:** How does the search budget impact on the effectiveness of EvoMASTER?

### 5.1 Experiment Setup

*Case studies.* We conducted our experiment with 40 Web APIs at Meituan. Table 1 presents descriptive information of these case studies. Such data includes the number of Java Files (#Files), lines of

**Table 1: Descriptive information for the used case studies**

SUT	#Files	LOC	LOC <sub>b</sub>	#Interfaces (#Functions)	#Dependent Services	#Tables (#Rows)
cs01	132	8,597	69,711	3(10)	100	2(476,872)
cs02	199	92,280	22,742	4(15)	102	0(0)
cs03	253	33,318	9,147	4(7)	143	21(0)
cs04	263	127,699	35,487	6(15)	142	0(0)
cs05	292	23,155	6,675	10(35)	100	30(6,658,077)
cs06	331	33,157	9,000	5(13)	97	1,480(45,507)
cs07	338	42,530	14,136	9(66)	79	79(110,573)
cs08	347	197,867	66,727	8(63)	96	8(938,719)
cs09	407	134,001	38,404	12(31)	97	119(83,576)
cs10	423	62,749	16,261	5(5)	70	45(1,117,918)
cs11	427	33,593	15,229	6(49)	55	13(2,313)
cs12	439	39,066	13,364	9(47)	129	568(23,350)
cs13	477	37,073	11,583	7(35)	111	42(2,596,605)
cs14	485	135,808	40,455	15(59)	107	0(0)
cs15	495	56,874	23,973	2(28)	83	17(50,310,974)
cs16	497	31,744	11,000	8(45)	113	24(134,201)
cs17	512	80,979	26,267	10(69)	157	159(7,145,998)
cs18	518	81,816	31,419	17(78)	104	24(563,336)
cs19	541	72,848	21,086	9(57)	129	1,480(45,507)
cs20	585	48,704	14,704	9(44)	136	78(11,573,934)
cs21	700	87,482	27,965	15(52)	97	27(9,426,345)
cs22	737	235,013	68,432	11(46)	74	72(1,141,760)
cs23	800	186,830	44,777	19(82)	103	47(12,266,746)
cs24	809	95,201	37,400	19(105)	78	82(373,707)
cs25	819	344,571	108,364	12(73)	102	108(24,226,760)
cs26	819	62,399	22,619	22(89)	119	70(38,548)
cs27	863	82,760	19,242	23(69)	103	421(635,842)
cs28	871	89,753	31,498	8(85)	153	8(48,458)
cs29	880	95,434	56,727	9(34)	132	203(686,358)
cs30	887	86,758	28,721	17(69)	116	22(114,636)
cs31	993	133,722	48,601	11(75)	142	97(1,858,471)
cs32	1,108	110,535	35,026	19(136)	141	16(23,693)
cs33	1,141	548,957	171,760	17(115)	90	2,165(2,398)
cs34	1,173	142,637	38,914	19(57)	165	115(6,801,794)
cs35	1,274	170,084	59,550	20(132)	81	106(24,324)
cs36	1,276	372,437	125,705	54(250)	76	175(3,162,469)
cs37	1,515	256,054	69,228	26(212)	147	325(3,784,103)
cs38	1,546	152,321	44,270	53(124)	105	133(11,964,007)
cs39	2,069	797,913	212,105	22(113)	91	158(536,338)
cs40	2,315	183,107	42,040	29(109)	100	421(635,842)
Sum	30,556	5,607,826	1,790,314	583(2798)	4,365	8,960(159,580,059)

code (LOC), total number of bytecode instructions (LOC<sub>b</sub>), the number of RPC interfaces (#Interfaces), the number of RPC functions that can be invoked (#Functions), the number of direct dependent API services (#Dependent Services), the number of tables defined in connected databases (#Tables), and the number of existing rows in these databases (#Rows). All of the services are JVM RPC APIs and parts of a large-scale distributed microservice architecture at Meituan. These 40 case studies cover a broad range of sizes of enterprise Web APIs, e.g., 30,556 Class Files (ranging from 132 to 2,315 per API), 5,607,826 of lines of code (ranging from 8,597 to 797,913 per API), and 583 RPC interfaces (ranging from 2 to 54 per API).

*Evaluation Metrics.* To assess the performance of EvoMASTER in industrial context, we used line coverage (Line%) and the number of detected faults (#Detected Faults), which are typical criteria in the context of software testing. In addition, in industrial settings, some existing code might be less critical (e.g., automatically generated code), and such code might be excluded in their testing. Thus, to represent the performance for covering *critical code*, we ran the final generated tests in the enterprise testing pipeline, and then reported line coverage using coverage report configurations at Meituan denoted as Critical Line%.

*Experiment configurations.* We defined two configurations:

- **Base:** default EvoMASTER which employed the default configuration of EvoMASTER, and

- **SM:** SM EVO MASTER which enabled the Seeding and Mock configurations in our extension and utilized existing enterprise tests to initialize populations in testing enterprise RPC APIs. In addition, the seeded tests are specified with mock objects for external service interactions, that can be further used and modified by EVO MASTER during search.

Considering the complexity of enterprise SUTs, the search budget likely has a major impact on the results. Therefore, we employed and evaluated two configurations of search budgets (i.e., 1-hour and 10-hours), which have been employed in existing work [48, 49]. To reduce the chance of results obtained by chance with search techniques, we repeated each configuration multiple times, i.e., 10 repetitions for the 1-hour time budget, and 2 repetitions for the 10-hours time budget. Thus, in total, our experiments took 100 days, i.e.,  $2 \times 40 \times (1 \text{ hour} \times 10 + 10 \text{ hours} \times 2) = 2400 \text{ hours} = 100 \text{ days}$ , if run in sequence. All experiments were conducted in the testing environment of Meituan.

## 5.2 Experiment Results

**5.2.1 RQ1 Results.** Table 2 presents the results, for each case study, achieved by EVO MASTER using the default configuration (i.e., *Base*) and 1-hour search budget. We show the line coverage (Line%), critical line coverage (Critical Line%) and the number of potential faults (#Detected Faults) with *Mean*, *Minimum*, *Median*, *Maximum* and *Standard Deviation* ( $\sigma$ ) statistics of 10 repetitions. In addition, we also analyzed the statistics over the whole 40 case studies. Based on these results, *Base* EVO MASTER is capable of covering on average 10.8% (up to 21.2%) line coverage and on average 9.3% (up to 46.1%) critical lines, and detecting on average 65.4 (up to 158.5) potential faults. However, the results vary significantly from case study to case study, e.g.,  $\sigma$  of Mean of Line% for 40 case studies is 4.5, and  $\sigma$  of Mean of Critical Line% for 40 case studies is 7.6 (see  $\sigma$  of Mean).

To investigate the variance in performance among the case studies, we first analyzed the Spearman Correlation Coefficient [41] between achieved line coverage (i.e., Line% and Critical Line%) and eight descriptive properties of the case studies, as shown in Table 1, e.g., #Files. These properties can reflect the complexity of the case studies from different perspectives. Results of  $\rho$  (summarizing the strength and direction, i.e., positive or negative, of monotonic relationship) and  $p$ -value (determining if the correlation is significant with a 95% level of confidence) are shown in Table 3. In addition, we rank the strength of correlation among properties based on Spearman's  $\rho$  (i.e.,  $abs(\rho)$ ). The property with Rank 1 represents the strongest correlation with performance of line coverage compared to other properties (see  $R_l$  for Line% and  $R_{cl}$  for Critical Line%). Based on the results in Table 3, negative correlations between line coverage performance and 6 out of 8 properties (except #Dependent Services and #Rows) are observed, indicating that performance of line coverage tends to decrease when the value of these 6 properties increases. However, the strength of the correlation shown by  $\rho$  is either *Weak* ( $\rho \in (0.1, 0.2]$ ) or *Moderate* ( $\rho \in (0.2, 0.4]$ ) [38]. With ranks using  $\rho$ , the most correlated properties are #Files for Line% and #Tables for Critical Line%. The overall most correlated property is #Files (see  $R_{mean}$ ), which may summarize to some extent the structure complexity of the Web API. Regarding #Rows (referring to the amount of data which SUT can access to) and #Dependent Services (referring to the amount of external APIs which SUT can invoke), no correlation is observed. For #Tables which somehow

**Table 2: Statistics of Line%, Critical Line%, and #Detected Faults achieved by tests generated by *Base* EVO MASTER configured with 1-hour time budget for 10 repetitions.**

SUT	Type	Line%		Critical Line%		#Detected Faults
cs01	CRUD	13.1	[10.6, 11.6, 17.3] (2.85)	11.3	[9.8, 11.6, 11.7] (0.8)	16.9 [16, 16, 20] (1.4)
cs02	QUERY	19.3	[19.0, 19.3, 19.5] (0.19)	46.1	[45.5, 46.2, 46.7] (0.4)	27.7 [23, 27, 33] (3.2)
cs03	FLOW	7.0	[6.7, 7.0, 7.0] (0.10)	2.3	[2.1, 2.3, 2.4] (0.1)	7.0 [7, 7, 7] (0.0)
cs04	QUERY	12.9	[12.4, 12.7, 14.2] (0.50)	10.4	[9.9, 10.1, 11.7] (0.5)	43.2 [36, 39, 62] (9.3)
cs05	CRUD	21.2	[19.2, 20.7, 26.8] (2.14)	16.9	[14.3, 16.9, 19.1] (1.5)	68.0 [58, 67.5, 83] (6.6)
cs06	RULE	8.0	[7.8, 7.9, 8.3] (0.14)	4.4	[4.1, 4.3, 4.7] (0.1)	13.0 [13, 13, 13] (0.0)
cs07	FLOW	12.1	[10.4, 11.8, 15.3] (1.38)	11.3	[9.8, 10.9, 14.1] (1.2)	74.9 [67, 76, 81] (4.9)
cs08	CRUD	14.2	[12.2, 14.3, 15.3] (0.78)	14.2	[11.9, 14.4, 15.2] (0.9)	61.3 [55, 62, 65] (3.7)
cs09	FLOW	14.8	[14.3, 14.8, 15.6] (0.51)	10.2	[8.8, 10.2, 11.5] (1.0)	33.9 [32, 34, 36] (1.2)
cs10	RULE	6.6	[6.2, 6.8, 7.3] (0.41)	2.5	[2.0, 2.7, 3.1] (0.4)	5.0 [5, 5, 5] (0.0)
cs11	RULE	9.1	[5.6, 9.5, 12.5] (2.45)	6.6	[4.3, 5.9, 9.0] (1.5)	45.1 [39, 46, 49] (3.4)
cs12	QUERY	9.9	[7.4, 10.1, 11.1] (0.81)	6.0	[2.9, 6.4, 7.6] (1.3)	52.5 [30, 54, 58] (5.9)
cs13	STATE	19.6	[19.4, 19.5, 19.9] (0.17)	19.4	[18.8, 19.4, 19.9] (0.3)	37.0 [37, 37, 37] (0.0)
cs14	QUERY	14.3	[13.0, 14.3, 15.3] (0.81)	22.8	[19.3, 22.6, 26.5] (2.5)	65.2 [60, 64, 71] (3.0)
cs15	STATE	7.3	[6.6, 7.1, 8.1] (0.44)	6.3	[5.6, 6.2, 6.9] (0.4)	33.4 [32, 34, 34] (0.9)
cs16	CRUD	14.3	[12.6, 14.3, 16.5] (1.03)	12.7	[10.9, 12.5, 15.0] (1.1)	44.7 [40, 45, 48] (2.1)
cs17	QUERY	14.7	[12.2, 14.5, 17.9] (1.63)	9.8	[7.7, 9.5, 12.4] (1.3)	65.0 [57, 66, 69] (3.8)
cs18	FLOW	5.8	[2.7, 6.0, 6.4] (0.85)	3.4	[1.5, 3.0, 6.0] (1.3)	60.2 [17, 64, 67] (11.5)
cs19	STATE	9.6	[7.9, 9.9, 10.6] (0.74)	5.9	[4.3, 6.0, 7.2] (0.7)	63.2 [57, 63.5, 70] (3.5)
cs20	CRUD	19.0	[14.7, 19.2, 22.8] (2.03)	16.6	[12.5, 16.6, 20.8] (2.0)	43.6 [41, 43, 46] (1.4)
cs21	CRUD	8.2	[6.8, 8.2, 9.6] (0.76)	6.3	[4.8, 6.3, 7.7] (0.9)	47.5 [44, 48, 50] (1.9)
cs22	CRUD	13.8	[12.0, 14.0, 14.5] (0.61)	4.1	[3.4, 4.1, 4.8] (0.4)	58.4 [57, 58.5, 60] (1.0)
cs23	RULE	4.4	[2.6, 4.5, 5.8] (0.85)	2.8	[1.6, 2.8, 4.2] (0.7)	133.1 [99, 137, 150] (15.7)
cs24	FLOW	9.4	[8.3, 9.3, 11.3] (0.93)	9.0	[7.8, 8.9, 10.9] (1.0)	90.4 [78, 90, 106] (8.2)
cs25	FLOW	9.8	[9.3, 9.7, 10.4] (0.37)	9.5	[8.3, 9.5, 10.3] (0.5)	62.8 [58, 63, 68] (2.6)
cs26	CRUD	9.0	[6.6, 8.9, 13.3] (1.49)	8.3	[4.8, 8.1, 13.8] (2.1)	100.3 [83, 100, 127] (10.7)
cs27	CRUD	18.8	[15.3, 18.8, 22.8] (2.23)	12.4	[10.2, 11.9, 15.2] (1.6)	62.6 [40, 63, 74] (9.4)
cs28	RULE	12.0	[11.1, 11.7, 14.3] (0.95)	9.9	[8.9, 9.7, 11.4] (0.8)	67.7 [64, 68, 72] (2.8)
cs29	FLOW	2.2	[2.1, 2.2, 2.4] (0.07)	4.2	[3.2, 4.1, 5.0] (0.5)	44.1 [42, 44.5, 46] (1.3)
cs30	RULE	5.2	[4.6, 5.1, 5.6] (0.28)	5.3	[4.2, 5.2, 6.5] (0.9)	66.1 [61, 66, 69] (2.4)
cs31	RULE	8.6	[6.7, 8.4, 10.1] (1.02)	7.5	[5.7, 7.5, 9.3] (1.0)	86.2 [76, 85.5, 94] (5.7)
cs32	FLOW	7.5	[5.9, 7.5, 9.3] (1.03)	5.6	[4.2, 5.6, 7.1] (0.9)	80.1 [63, 82, 96] (9.6)
cs33	FLOW	9.2	[7.1, 9.3, 11.4] (1.37)	5.3	[4.2, 5.2, 6.8] (0.9)	122.4 [97, 121.5, 150] (15.7)
cs34	RULE	8.5	[7.6, 8.4, 9.4] (0.50)	5.5	[4.8, 5.4, 6.2] (0.4)	76.5 [69, 77, 81] (3.5)
cs35	RULE	8.7	[6.8, 8.7, 11.5] (1.10)	5.4	[3.9, 5.4, 6.9] (0.9)	106.6 [78, 108, 132] (17.1)
cs36	FLOW	7.5	[6.0, 7.3, 10.6] (1.40)	9.3	[7.2, 8.6, 15.2] (2.3)	114.8 [95, 106.5, 175] (24.9)
cs37	CRUD	6.7	[5.3, 6.5, 9.0] (1.26)	4.5	[2.9, 4.4, 6.2] (1.1)	158.5 [112, 172, 223] (43.3)
cs38	RULE	14.4	[10.1, 14.6, 17.2] (2.02)	11.0	[7.1, 11.2, 13.0] (1.7)	99.4 [82, 97, 123] (13.2)
cs39	STATE	7.3	[6.3, 7.2, 8.4] (0.63)	3.1	[2.3, 3.2, 3.8] (0.5)	87.8 [73, 87, 99] (8.2)
cs40	STATE	9.4	[4.2, 9.3, 12.9] (2.08)	5.4	[1.5, 5.2, 7.8] (1.6)	87.9 [37, 89, 111] (19.1)
Mean of Mean		10.8		9.3		65.4
$\sigma$ of Mean		4.5		7.6		33.9

Note that, 1) for each metric, we report statistics with a format *Mean* [*Minimum*, *Median*, *Maximum*] (*Standard Deviation*); 2) **cell** for Mean of line coverage  $\in (0\%, 5\%)$ , **cell** for Mean of line coverage  $\in (5\%, 10\%)$ , **cell** for Mean of line coverage  $\in (10\%, 20\%)$  and **cell** for Mean of line coverage  $\in (20\%, 100\%)$ .

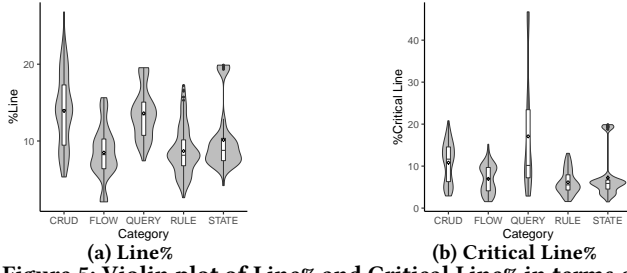
**Table 3: Results of Spearman correlation between line coverage and properties which represents size of case studies.**

Size Properties	Line%		Critical Line%		$R_{mean} = \frac{R_l + R_{cl}}{2}$
	$\rho$ ( $R_l$ )	$p$ -value	$\rho$ ( $R_{cl}$ )	$p$ -value	
#Files	-0.342(1)	$\leq 0.001$	-0.300(2)	$\leq 0.001$	1.5
LOC	-0.171(5)	$\leq 0.001$	-0.177(4)	$\leq 0.001$	4.5
LOC <sub>b</sub>	-0.246(3)	$\leq 0.001$	-0.212(3)	$\leq 0.001$	3.0
#Interfaces	-0.205(4)	$\leq 0.001$	-0.165(5)	$\leq 0.001$	4.5
#Functions	-0.255(2)	$\leq 0.001$	-0.134(6)	<b>0.002</b>	4.0
#Dependent Services	0.008(7)	0.847	0.055(7)	0.209	7.0
#Tables	-0.139(6)	<b>0.001</b>	-0.304(1)	$\leq 0.001$	3.5
#Rows	0.001(8)	0.979	0.029(8)	0.506	8.0

represents to some extent the complexity of the underlying data model, it correlates to the performance of line coverage achieved by EVO MASTER, but the rank of the strength to line coverage versus critical line coverage is distinctly different, i.e., 1 vs 6. This might give us a hint that the industry might concern more on the code relating to the data model and its handling.

In addition, we analyzed the performance of EVO MASTER with a consideration of business logic. To address a variety of business requirements, the services in enterprises might be categorized into various types. At Meituan, there exists five type definitions<sup>1</sup>:

<sup>1</sup>Note that the types might be applied for other e-commerce systems. However, these types may be domain-specific and depend on the business requirements and logic.



**Figure 5: Violin plot of Line% and Critical Line% in terms of each type of enterprise Web API at Meituan**

- CRUD stands for services which implement CRUD operations (i.e., Create, Read, Update, and Delete) that perform on data;
- FLOW stands for services which follow a particular sequence to process the business logic;
- QUERY stands for services which only perform query operations, i.e., do not modify any info in the microservices;
- RULE stands for services which operate or make decisions based on predefined rules;
- STATE stands for services which operate in a state-driven manner, executing actions based on the current state.

In these experiments, the 40 case studies comprises 10 CRUD, 10 FLOW, 5 QUERY, 10 RULE and 5 STATE. *Type* column in Table 2 shows the type for each case study. To investigate performance of EvoMASTER on the five types of services at Meituan, Figure 5 represents violin plots of Line% and Critical Line% for each type. Compared to other types, EvoMASTER achieved relatively better performance on CRUD and QUERY, i.e., the best two statistics for Line% and Critical Line% are either CRUD or QUERY (see Figure 5). CRUD and QUERY services implement basic actions that perform on the data and the logic is relatively less complex than other types. This might explain the better performance achieved by EvoMASTER. FLOW, RULE and STATE services are comparatively more complex, e.g., for FLOW services, whether the code can be reached might depend on whether some particular actions have been executed first. Then, with a 1-hour search budget, EvoMASTER has limited line coverage performance on such services.

**RQ1:** Among 8 size properties of RPC APIs, negative correlation to performance of EvoMASTER in terms of line coverage can be observed in 6 out of 8, i.e., #Files, LOC, LOC<sub>b</sub>, #Interfaces, #Functions and #Tables, except #Dependent Services and #Rows. #Files (i.e., a number of script files) has the strongest negative correlation with the performance. From the points of view of five types of services at Meituan relating to business logic, using 1-hour search budget, EvoMASTER performs best on CRUD and QUERY services, which implement basic actions for manipulating data or resources. On the other hand, EvoMASTER obtained limited results on FLOW, RULE and STATE services, which implement complex logic and likely need to coordinate with other functions and services.

**5.2.2 RQ2 Results.** Table 4 shows the number of existing enterprise tests (EETests) and Critical Line% covered by EETests. There exist in total 5067 tests (#Test) that have been used to perform system-level testing for the 40 case studies at Meituan, and 3372 out of 5067 (i.e., 66.5%) tests require mock objects (see #Test<sub>m</sub>). The existing tests achieve on average 6.2% Critical Line% among 40 case studies. Note that EETests do not encompass E2E tests or unit tests in enterprise. To analyze the potential benefits of EvoMASTER, we compared

**Table 4: Number of tests and Critical Line% achieved by Existing Enterprise Tests (EETests) and results of comparing tests generated by Base EvoMASTER and EETests.**

SUT	Type	EETests			Base vs. EETests		
		#Tests <sub>m</sub>	#Tests <sub>u</sub>	Critical Line%	Ratio of lines covered by EETests	Both	Base
cs01	CRUD	56 (8, 14.3%)		9.5	+18.5	27.4%	34.1%
cs02	QUERY	2 (0, 0.0%)		5.6	+719.0	0.0%	12.2%
cs03	FLOW	67 (21, 31.3%)		1.4	+62.0	0.3%	61.4%
cs04	QUERY	3 (0, 0.0%)		1.0	+895.6	0.3%	9.8%
cs05	CRUD	20 (6, 30.0%)		4.4	+286.5	7.0%	17.2%
cs06	RULE	10 (2, 20.0%)		1.9	+125.8	9.3%	31.0%
cs07	FLOW	21 (0, 0.0%)		7.0	+59.9	27.1%	18.7%
cs08	CRUD	345 (246, 71.3%)		7.7	+83.5	3.9%	48.6%
cs09	FLOW	13 (5, 38.5%)		4.9	+110.0	9.5%	33.9%
cs10	RULE	69 (0, 0.0%)		4.5	-43.2		47.4%
cs11	RULE	128 (38, 29.7%)		7.7	-14.9	37.9%	35.9%
cs12	QUERY	592 (399, 67.4%)		7.7	-22.4	43.5%	30.3%
cs13	STATE	194 (139, 71.6%)		16.2	+19.8	22.6%	42.1%
cs14	QUERY	2 (2, 100.0%)		2.4	+856.8	2.1%	8.3%
cs15	STATE	119 (26, 21.8%)		11.9	-47.3	55.0%	30.4%
cs16	CRUD	18 (6, 33.3%)		6.6	+91.0	16.3%	27.7%
cs17	QUERY	211 (129, 61.1%)		7.7	+26.7	24.6%	35.4%
cs18	FLOW	9 (1, 11.1%)		0.2	+1687.0	1.7%	4.6%
cs19	STATE	22 (18, 81.8%)		1.5	+307.6	11.9%	10.0%
cs20	CRUD	249 (247, 99.2%)		14.9	+11.1	31.8%	29.9%
cs21	CRUD	17 (16, 94.1%)		4.4	+43.4	27.3%	23.9%
cs22	CRUD	112 (109, 97.3%)		6.4	-36.2	55.6%	14.3%
cs23	RULE	350 (339, 96.9%)		6.2	-53.9	63.0%	18.1%
cs24	FLOW	108 (92, 85.2%)		3.2	+180.9	11.5%	20.2%
cs25	FLOW	51 (29, 56.9%)		3.2	+194.1	10.5%	20.0%
cs26	CRUD	81 (53, 65.4%)		6.2	+33.7	32.1%	20.4%
cs27	CRUD	12 (0, 0.0%)		3.3	+274.8	13.3%	10.1%
cs28	RULE	360 (323, 89.7%)		21.1	-53.2	60.4%	24.3%
cs29	FLOW	2 (2, 100.0%)		0.5	+721.4	4.4%	7.4%
cs30	RULE	130 (84, 64.6%)		14.6	-63.7	66.9%	24.6%
cs31	RULE	134 (80, 59.7%)		6.7	+11.8	17.2%	57.5%
cs32	FLOW	37 (37, 100.0%)		2.1	+162.9	17.2%	14.9%
cs33	FLOW	373 (168, 45.0%)		5.1	+5.2	30.1%	36.9%
cs34	RULE	14 (7, 50.0%)		1.6	+241.5	12.7%	13.0%
cs35	RULE	427 (300, 70.3%)		4.2	+28.8	22.4%	38.9%
cs36	FLOW	194 (53, 27.3%)		16.0	-41.7	55.2%	22.7%
cs37	CRUD	238 (171, 71.8%)		5.0	-9.2	40.0%	27.6%
cs38	RULE	58 (50, 86.2%)		8.2	+33.0	25.2%	31.7%
cs39	STATE	163 (123, 75.5%)		1.4	+126.6	13.1%	25.8%
cs40	STATE	56 (43, 76.8%)		3.6	+47.7	33.8%	13.4%
Sum		5067 (3372, 66.5%)					
Mean				6.2	+177.0	24.8%	25.9%
				Analysis with Ratio			
				Total	(0, 33.3%]	(33.3%, 50%]	(50, 100%]
				#Ratio <sub>EETests</sub> > Ratio <sub>Base</sub>	10	0	4
				#Ratio <sub>Base</sub> > Ratio <sub>EETests</sub>	30	2	10

Base EvoMASTER with EETests in terms of *Relative* improvement of Critical Line% and *Ratio* of lines covered by only EETests, both and only Base EvoMASTER (see Table 4). With these results, Base EvoMASTER is capable of improving Critical Line% on 30 out of 40 case studies (i.e., Relative % > 0), and relative improvements are up to 1687.0% and on average 177.0% of Critical Line% covered by EETests. In addition, by analyzing the ratio of lines covered by only EETests, Both and only Base EvoMASTER, we found that:

- EvoMASTER can cover on average 49.4% lines that EETests does not cover. As seen from #Ratio<sub>Base</sub> > Ratio<sub>EETests</sub>, on 30 out of 40 case studies (i.e., 75%), EvoMASTER can cover more unique lines than EETests. In terms of 30 case studies, compared to EETests, EvoMASTER can cover more than 50% new lines on 18 case studies, (33.3%, 50%] new lines on 10 case studies, and (0%, 33.3%] new lines on 2 case studies;
- There exist on average 25.9% lines that can be covered by both EETests and Base EvoMASTER;
- While there also exist 24.8% lines which EvoMASTER does not manage to cover. As seen from #Ratio<sub>EETests</sub> > Ratio<sub>Base</sub>, on 10 out of 40 case studies (i.e., 25%), EETests can cover more unique lines than EvoMASTER. In terms of 10 case studies, compared to EvoMASTER, EETests can cover more than 50% unique lines on 6 case studies and (33.3%, 50%] unique lines on 4 case studies.



**Table 5: Mean of Line%, Critical Line%, and #Detected Faults achieved by tests generated by SM EvoMASTER with 1-hour time budget for 10 repetitions, and results of comparing with Base 1-hour using Relative% and Vargha-Delaney  $\hat{A}_{12}$**

SUT	Budgets by seeds	Line%		Critical Line%		#Detected Faults	
		Mean	Relative%( $\hat{A}_{12}$ )	Mean	Relative%( $\hat{A}_{12}$ )	Mean	Relative%( $\hat{A}_{12}$ )
cs01	6.6%	16.6	+26.7 (0.78)	15.4	+36.6 (1.00)	17.2	+2.1 (0.42)
cs02	0.2%	19.3	+0.0 (0.53)	45.4	-1.5 (0.42)	26.4	-4.8 (0.38)
cs03	3.8%	7.1	+1.4 (0.40)	2.4	+1.6 (0.57)	7.0	+0.0 (0.50)
cs04	0.3%	13.0	+1.0 (0.57)	10.5	+1.7 (0.59)	43.8	+1.3 (0.64)
cs05	1.8%	22.6	+6.5 (0.77)	18.1	+7.1 (0.71)	63.2	-7.1 (0.27)
cs06	0.5%	8.5	+6.1 (0.97)	4.8	+9.6 (0.96)	13.0	+0.0 (0.50)
cs07	1.5%	17.6	+44.9 (0.99)	16.3	+44.7 (0.99)	79.7	+6.5 (0.72)
cs08	59.2%	10.1	-28.8 (0.05)	10.8	-23.5 (0.06)	38.8	-36.8 (0.04)
cs09	1.4%	15.3	+2.8 (0.65)	11.2	+9.4 (0.73)	35.1	+3.4 (0.75)
cs10	7.0%	9.2	+39.1 (1.00)	4.6	+83.6 (1.00)	5.0	+0.0 (0.50)
cs11	9.7%	16.0	+76.1 (1.00)	13.2	+101.4 (1.00)	46.9	+4.0 (0.66)
cs12	31.3%	12.4	+24.6 (1.00)	9.0	+50.8 (1.00)	56.1	+6.9 (0.77)
cs13	12.9%	24.6	+25.8 (1.00)	24.0	+23.8 (1.00)	36.6	-1.1 (0.33)
cs14	0.2%	14.6	+1.8 (0.52)	24.1	+5.9 (0.62)	67.6	+3.8 (0.59)
cs15	11.6%	15.9	+118.1 (1.00)	15.9	+153.5 (1.00)	33.7	+1.0 (0.55)
cs16	1.6%	16.7	+17.1 (0.94)	15.1	+19.4 (0.92)	45.4	+1.6 (0.59)
cs17	22.0%	15.8	+7.4 (0.71)	11.3	+15.8 (0.82)	60.9	-6.3 (0.23)
cs18	0.6%	6.9	+18.5 (0.88)	6.4	+89.3 (0.97)	70.6	+17.2 (0.83)
cs19	2.1%	10.8	+11.6 (0.84)	6.9	+16.3 (0.83)	63.9	+1.1 (0.54)
cs20	28.8%	21.8	+14.8 (0.85)	19.0	+14.9 (0.82)	35.8	-17.8 (0.30)
cs21	1.7%	10.3	+25.7 (0.99)	8.8	+40.3 (0.99)	47.8	+0.5 (0.50)
cs22	9.9%	16.3	+18.1 (0.97)	8.4	+103.5 (1.00)	56.8	-2.7 (0.39)
cs23	36.6%	10.2	+134.3 (1.00)	6.7	+135.6 (0.93)	110.7	-16.8 (0.31)
cs24	8.8%	10.4	+11.1 (0.77)	11.2	+24.9 (0.94)	93.2	+3.1 (0.60)
cs25	6.3%	10.0	+2.1 (0.69)	9.9	+4.3 (0.69)	61.2	-2.6 (0.31)
cs26	6.1%	12.8	+42.7 (0.96)	12.9	+56.0 (0.92)	106.8	+6.5 (0.60)
cs27	1.0%	22.1	+17.3 (0.84)	14.3	+15.8 (0.83)	69.0	+10.2 (0.76)
cs28	83.8%	23.1	+92.7 (1.00)	21.6	+119.0 (1.00)	48.6	-28.2 (0.07)
cs29	0.2%	2.4	+9.5 (0.59)	3.9	-8.0 (0.38)	42.6	-3.4 (0.24)
cs30	6.5%	8.7	+66.5 (1.00)	12.5	+136.7 (1.00)	69.0	+4.4 (0.88)
cs31	18.1%	9.6	+11.5 (0.77)	8.2	+10.3 (0.72)	72.4	-16.0 (0.12)
cs32	4.0%	9.4	+24.9 (0.91)	7.4	+32.2 (0.94)	84.9	+6.0 (0.59)
cs33	47.8%	9.7	+4.7 (0.61)	6.2	+16.9 (0.77)	105.1	-14.2 (0.34)
cs34	0.9%	9.8	+14.9 (0.99)	6.4	+17.4 (0.98)	77.4	+1.1 (0.57)
cs35	31.5%	10.1	+16.5 (0.85)	7.0	+28.6 (0.93)	111.3	+4.4 (0.59)
cs36	29.2%	11.8	+58.1 (0.95)	18.5	+98.1 (1.00)	129.1	+12.4 (0.80)
cs37	34.6%	9.1	+35.6 (0.93)	6.7	+47.0 (0.96)	104.7	-33.9 (0.11)
cs38	5.1%	18.7	+30.1 (0.99)	14.9	+36.0 (1.00)	101.0	+1.7 (0.58)
cs39	16.5%	8.0	+9.0 (0.62)	3.6	+13.2 (0.68)	96.4	+9.7 (0.61)
cs40	4.3%	12.6	+34.4 (0.91)	7.9	+47.1 (0.91)	100.0	+13.8 (0.68)
Mean		13.2	+26.9 (0.8)	12.0	+40.9 (0.8)	63.4	-1.7 (0.5)
#Relative > 0			39		37		23
#SM > Base			30		31		7
#Base > SM			1		1		8

Regarding fault detection, the EETests do not identify any existing faults (if they did, these faults would have already been fixed by the engineers at Meituan). These thousands of existing tests are used for regression testing. On the other end, all potential faults detected by EvoMASTER ( $65.4 \times 40 = 2616$ ) in these experiments can be considered as an extra benefit of fuzzers such as EvoMASTER (besides using the generated tests for regression testing).

**RQ2:** Compared to existing API testing practice at Meituan, it is clear that EvoMASTER (with 1-hour search budget) is able to bring additional benefits, i.e., covered more unique lines on 30 out of 40 case studies, achieved on average 49.4% new lines that the existing tests do not cover, and identify 2616 new faults. However, EvoMASTER cannot replace yet the current practice in enterprise as there exist on average 24.8% lines that EvoMASTER does not cover.

**5.2.3 RQ3 Results.** Table 5 presents Mean of 10 repetitions of Line%, Critical Line% and potential faults achieved by SM EvoMASTER. Results show that SM EvoMASTER is capable of covering on average 13.2% Lines (up to 24.6%), 12.0% Critical Lines (up to 45.4%), and 63.4 potential faults (up to 129.1) among 40 case studies. In Table 5, we performed further analyses for each case study using Relative improvement (i.e.,  $(a - b)/b$  for a vs. b), Vargha-Delaney  $\hat{A}_{12}$ , and Mann-Whitney U test. Note that a value in **bold** indicates

that SM is significantly better than Base in statistic (i.e.,  $p$ -value  $\leq 0.05$  with a 95% level of confidence and  $\hat{A}_{12} > 0.5$ ). A value with the normal format indicates that there is no significant difference between SM and Base (i.e.,  $p$ -value  $> 0.05$ ), while a value in **red** indicates that Base is significantly better than SM (i.e.,  $p$ -value  $\leq 0.05$  and  $\hat{A}_{12} < 0.5$ ). Compared to Base, we found that SM EvoMASTER significantly improve Line% on 39 out of 40 case studies, Critical Line% on 37 out of 40 case studies, and #Detected Faults on 23 out of 40 case studies. The significant improvements can be observed in 30 case studies for Line%, 31 case studies for Critical Line%, 7 case studies for #Detected Faults. But, on cs08, SM significantly underperform over Base in terms of Line% and Critical Line%. By looking at budget used by seeds, we found that the seeded tests (i.e., EETests) on cs08 took 59.2% of the search budget, and achieved modest results, i.e., 7.7% critical line coverage (Table 4). Thus, there is limited remaining search budget for EvoMASTER, and this might explain the observed underperformance of EvoMASTER on this case study. Moreover, we analyze line coverage results by considering seeded tests. By checking the 10 case studies where EETests obtains better results (Table 4), SM EvoMASTER obtains significant improvements on all case studies. The results are expected as the seeded tests already have better performance. In terms of 30 case studies where Base EvoMASTER obtains better results, SM EvoMASTER can obtain further significant improvements by seeding EETests.

Regarding #Detected Faults, there is no clear difference between Base and SM. More specifically, Mean of  $\hat{A}_{12}$  with 0.5 indicates that the two configurations have the same chance (i.e., 50% probability) to obtain better result than each other. Negative relative improvement with -1.7% shows that SM is slightly worse than Base, but #Relative  $> 0$  demonstrates that SM achieved better results on more case studies than Base (23 vs. 17). As EETests cover lines where there do not exist faults, when employing SM configuration, it might affect search starting from the region where there are less faults. This might explain limited improvement obtained by SM.

**RQ3:** With 1-hour search budget, SM EvoMASTER clearly outperformed Base on 39 out of 40 case studies in terms of line coverage related metrics (i.e., Line% and Critical Line%), and relative improvements are on average 26.9% (up to 134.3%) for Line% and 40.9% (up to 153.5%) for Critical Line%. Regarding capability of fault detection, there is no difference between Base and SM, indicating that seeding EETests likely could not contribute to identify more faults for a search-based fuzzer such as EvoMASTER.

**5.2.4 RQ4 Results.** Table 6 reports results of SM and Base EvoMASTER configured with 10-hours search budget, pair comparison results of performance achieved by using 10-hours and 1-hour search budgets for SM and Base respectively (i.e., SM 10h vs. SM 1h, and Base 10h vs. Base 1h), and pair comparison results of performance achieved by SM 10h and Base 10h (i.e., SM 10h vs. Base 10h). As there are only two repetitions for 10-hours configuration, we analyze the results only with Mean and  $\hat{A}_{12}$ .

Regarding performance of SM EvoMASTER with 10-hours search budget, results demonstrate that it is capable of covering on average more than 15% lines for 40 case studies (i.e., 17.0% (up to 32.4%) for Line% and 15.7% (up to 47.7%) for Critical Line%), and detecting on average 88.0 (up to 287) potential faults. In addition, on 36 (i.e., 25 + 11) out of 40 case studies (except cs03, cs06, cs18, and cs29), SM EvoMASTER achieved more than 10% Line%, and it is noteworthy

**Table 6: Mean of 2 repetitions of Line%, Critical Line% and #Detected Faults achieved by SM EvoMASTER with 10-hours search budget, and results (i.e., Relative and  $\hat{A}_{12}$ ) of pair comparison of SM EvoMASTER using 10 hours and 1 hour (i.e., 10h vs. 1h).**

SUT	Type	Mean of SM 10h; SM 10h vs. SM 1h						Mean of Base 10h; Base 10h vs. Base 1h						SM 10h vs. Base 10h			
		Line%		Critical Line%		#Detected Faults		Line%		Critical Line%		#Detected Faults		Line%		Critical Line%	
		Mean	Relative%( $\hat{A}_{12}$ )	Mean	Relative%( $\hat{A}_{12}$ )	Mean	Relative%( $\hat{A}_{12}$ )	Mean	Relative%( $\hat{A}_{12}$ )	Mean	Relative%( $\hat{A}_{12}$ )	Mean	Relative%( $\hat{A}_{12}$ )	Mean	Relative%( $\hat{A}_{12}$ )	Mean	Relative%( $\hat{A}_{12}$ )
cs01	CRUD	20.8	+25.3 (0.88)	18.4	+19.5 (0.96)	20.7	+20.1 (0.71)	15.9	+21.6 (0.80)	12.3	+8.9 (0.88)	18.8	+10.9 (0.47)	+30.6 (0.84)	+49.8 (1.00)	+10.5 (0.61)	
cs02	QUERY	19.6	+1.7 (0.93)	47.7	+4.9 (0.98)	34.5	+30.7 (0.99)	19.6	+1.2 (0.95)	47.2	+2.4 (1.00)	31.5	+13.6 (0.84)	+0.4 (0.75)	+0.9 (0.75)	+9.5 (0.75)	
cs03	FLOW	7.0	-1.1 (0.61)	2.4	-0.2 (0.64)	7.0	+0.0 (0.50)	7.4	+5.8 (0.47)	2.6	+10.4 (0.65)	7.0	+0.0 (0.50)	-5.1 (0.57)	-8.2 (0.54)	+0.0 (0.50)	
cs04	QUERY	15.7	+20.3 (0.97)	12.7	+20.9 (0.97)	47.0	+7.4 (0.75)	17.5	+36.0 (1.00)	14.2	+37.5 (1.00)	40.0	-7.5 (0.69)	-10.7 (0.00)	-10.6 (0.00)	+17.5 (1.00)	
cs05	CRUD	28.3	+25.6 (0.95)	23.7	+30.9 (1.00)	102.5	+62.2 (1.00)	29.5	+39.1 (0.96)	21.5	+27.4 (1.00)	125.0	+83.8 (1.00)	-3.8 (0.38)	+10.1 (0.88)	-18.0 (0.12)	
cs06	RULE	8.9	+5.4 (0.88)	5.2	+8.5 (0.98)	13.0	+0.0 (0.50)	8.9	+11.8 (1.00)	5.0	+15.5 (0.98)	13.0	+0.0 (0.50)	-0.0 (0.55)	+3.0 (0.55)	+0.0 (0.50)	
cs07	FLOW	22.4	+27.6 (1.00)	20.2	+24.2 (0.90)	86.9	+9.0 (0.90)	20.7	+70.3 (1.00)	19.6	+73.9 (1.00)	86.5	+15.5 (1.00)	+8.5 (1.00)	+3.4 (0.86)	+0.4 (0.61)	
cs08	CRUD	16.1	+59.7 (1.00)	16.8	+55.0 (1.00)	66.5	+71.5 (0.99)	16.4	+16.0 (1.00)	17.3	+21.8 (1.00)	66.5	+8.4 (1.00)	-2.0 (0.25)	-2.7 (0.25)	+0.0 (0.50)	
cs09	FLOW	15.8	+3.9 (0.85)	12.1	+8.0 (0.79)	35.4	+1.1 (0.83)	16.0	+7.9 (1.00)	11.9	+16.0 (1.00)	35.0	+3.2 (0.77)	-1.1 (0.82)	+1.9 (0.86)	+1.2 (0.86)	
cs10	RULE	10.3	+11.0 (0.93)	5.0	+7.6 (0.96)	5.0	+0.0 (0.50)	11.1	+67.2 (1.00)	4.0	+57.2 (1.00)	5.0	+0.0 (0.50)	-7.7 (0.33)	+25.7 (1.00)	+0.0 (0.50)	
cs11	RULE	17.4	+8.5 (0.77)	14.6	+10.6 (0.99)	49.0	+4.5 (0.85)	13.5	+48.4 (0.95)	10.1	+53.5 (1.00)	49.0	+8.6 (0.90)	+28.8 (1.00)	+45.2 (1.00)	+0.0 (0.50)	
cs12	QUERY	14.2	+15.1 (0.99)	12.1	+35.1 (1.00)	60.0	+7.0 (1.00)	13.3	+34.0 (1.00)	9.9	+66.6 (1.00)	60.0	+14.3 (1.00)	+7.0 (0.88)	+22.3 (0.96)	+0.0 (0.50)	
cs13	STATE	27.0	+9.8 (0.96)	26.8	+12.0 (0.92)	37.1	+1.5 (0.72)	21.4	+9.2 (1.00)	27.4	+41.4 (1.00)	37.0	+0.0 (0.50)	+26.5 (1.00)	-2.0 (0.29)	+0.4 (0.57)	
cs14	QUERY	17.4	+19.8 (1.00)	31.1	+29.0 (1.00)	77.5	+14.6 (0.91)	17.2	+20.4 (1.00)	30.8	+35.0 (1.00)	86.5	+32.7 (1.00)	+1.3 (1.00)	+1.2 (0.75)	-10.4 (0.19)	
cs15	STATE	17.8	+12.1 (0.82)	18.5	+16.7 (0.82)	35.2	+4.5 (0.78)	9.6	+32.6 (0.95)	6.9	+10.3 (0.92)	34.0	+1.8 (0.69)	+84.4 (1.00)	+168.1 (1.00)	+3.7 (0.75)	
cs16	CRUD	23.1	+38.1 (1.00)	21.7	+43.6 (1.00)	48.2	+6.1 (0.96)	20.6	+44.1 (1.00)	20.4	+61.0 (1.00)	48.5	+8.5 (0.99)	+12.1 (1.00)	+6.5 (0.79)	-0.7 (0.33)	
cs17	QUERY	22.1	+40.2 (1.00)	16.3	+44.2 (1.00)	70.5	+15.7 (1.00)	20.3	+38.0 (1.00)	14.6	+49.3 (1.00)	70.0	+7.7 (1.00)	+9.1 (1.00)	+11.9 (1.00)	+0.7 (0.75)	
cs18	FLOW	8.2	+18.7 (1.00)	9.6	+49.7 (1.00)	78.0	+10.5 (0.83)	8.2	+40.7 (1.00)	8.8	+158.4 (1.00)	78.0	+29.5 (1.00)	-0.1 (0.48)	+9.7 (0.96)	+0.0 (0.50)	
cs19	STATE	14.3	+33.0 (1.00)	9.4	+36.0 (1.00)	72.0	+12.7 (0.95)	14.0	+45.2 (1.00)	9.2	+55.1 (1.00)	72.0	+13.9 (1.00)	+2.3 (0.78)	+2.0 (0.67)	+0.0 (0.50)	
cs20	CRUD	32.4	+48.4 (1.00)	30.1	+58.0 (1.00)	47.7	+33.1 (0.92)	31.3	+64.5 (1.00)	29.1	+75.4 (1.00)	49.5	+13.6 (1.00)	+3.6 (0.83)	+3.5 (0.67)	-3.7 (0.00)	
cs21	CRUD	12.9	+24.5 (1.00)	11.1	+26.3 (1.00)	52.0	+8.8 (0.90)	11.5	+39.5 (1.00)	9.7	+55.1 (1.00)	52.0	+9.4 (1.00)	+12.2 (1.00)	+14.2 (1.00)	+0.0 (0.50)	
cs22	CRUD	18.0	+10.4 (1.00)	9.2	+9.9 (0.89)	59.0	+3.8 (0.71)	15.1	+9.6 (1.00)	4.1	+0.9 (0.54)	59.2	+1.3 (0.74)	+18.9 (1.00)	+121.5 (1.00)	-0.3 (0.40)	
cs23	RULE	13.4	+30.7 (1.00)	9.8	+46.5 (1.00)	221.0	+99.6 (1.00)	8.1	+86.7 (1.00)	4.0	+41.3 (0.68)	191.5	+43.9 (1.00)	+64.1 (1.00)	+144.3 (1.00)	+15.4 (0.88)	
cs24	FLOW	16.8	+60.9 (1.00)	15.9	+41.6 (1.00)	121.6	+30.5 (1.00)	16.0	+70.8 (1.00)	15.9	+76.8 (1.00)	123.7	+36.9 (1.00)	+4.6 (0.47)	+0.0 (0.47)	-1.7 (0.03)	
cs25	FLOW	13.1	+31.2 (1.00)	14.9	+50.4 (1.00)	73.0	+19.3 (1.00)	13.2	+34.9 (1.00)	15.0	+58.2 (1.00)	73.0	+16.2 (1.00)	-0.6 (0.67)	-0.8 (0.56)	+0.0 (0.50)	
cs26	CRUD	17.4	+35.7 (1.00)	16.8	+30.8 (0.94)	132.1	+23.7 (1.00)	15.1	+68.6 (1.00)	14.8	+79.9 (1.00)	141.7	+41.2 (1.00)	+14.9 (0.90)	+13.4 (1.00)	-6.7 (0.10)	
cs27	CRUD	28.7	+30.0 (1.00)	20.1	+40.4 (1.00)	74.2	+7.5 (0.87)	28.0	+48.6 (1.00)	19.7	+59.1 (1.00)	74.5	+18.9 (0.95)	+2.6 (0.80)	+2.2 (0.60)	-0.4 (0.35)	
cs28	RULE	29.3	+27.1 (1.00)	27.4	+26.5 (1.00)	107.0	+120.0 (1.00)	18.9	+57.8 (1.00)	15.7	+58.4 (1.00)	85.6	+26.4 (1.00)	+55.2 (1.00)	+75.0 (1.00)	+25.0 (1.00)	
cs29	FLOW	2.6	+9.9 (0.93)	5.5	+43.0 (0.96)	48.5	+13.9 (1.00)	2.9	+30.0 (1.00)	5.7	+35.0 (1.00)	50.2	+14.0 (1.00)	-7.4 (0.00)	-2.5 (0.31)	-3.5 (0.06)	
cs30	RULE	12.6	+45.3 (0.72)	12.8	+2.4 (0.69)	70.0	+1.4 (0.71)	9.2	+76.9 (1.00)	7.7	+45.4 (1.00)	70.0	+5.9 (1.00)	+36.7 (0.56)	+66.7 (1.00)	+0.0 (0.50)	
cs31	RULE	12.3	+28.3 (1.00)	10.3	+25.5 (1.00)	94.0	+29.8 (1.00)	11.7	+35.4 (1.00)	9.7	+29.9 (1.00)	94.7	+9.8 (0.97)	+5.6 (0.90)	+6.6 (1.00)	-0.7 (0.33)	
cs32	FLOW	17.1	+82.0 (1.00)	13.4	+80.9 (1.00)	136.5	+60.8 (1.00)	16.7	+122.5 (1.00)	13.4	+139.6 (1.00)	135.8	+69.5 (1.00)	+2.2 (0.70)	-0.2 (0.50)	+0.5 (0.80)	
cs33	FLOW	12.8	+32.2 (1.00)	9.3	+48.5 (1.00)	152.7	+45.3 (1.00)	12.4	+34.7 (1.00)	7.9	+47.9 (1.00)	153.0	+25.0 (0.99)	+2.7 (0.92)	+17.3 (1.00)	-0.2 (0.54)	
cs34	RULE	11.4	+16.7 (1.00)	7.5	+17.0 (1.00)	84.6	+9.3 (1.00)	10.2	+19.6 (1.00)	6.7	+22.8 (1.00)	83.3	+8.9 (1.00)	+12.1 (1.00)	+11.9 (1.00)	+1.5 (0.77)	
cs35	RULE	12.9	+27.8 (1.00)	9.2	+31.1 (1.00)	141.0	+26.7 (1.00)	13.7	+57.8 (1.00)	9.1	+66.9 (1.00)	170.3	+59.7 (1.00)	-5.6 (0.33)	+1.1 (0.67)	-17.2 (0.00)	
cs36	FLOW	20.6	+74.5 (1.00)	31.8	+72.0 (1.00)	268.0	+107.6 (1.00)	16.5	+121.0 (1.00)	22.9	+145.5 (1.00)	256.5	+123.4 (1.00)	+24.8 (1.00)	+38.8 (1.00)	+4.5 (1.00)	
cs37	CRUD	14.5	+59.1 (1.00)	10.5	+56.6 (1.00)	287.0	+174.2 (1.00)	12.6	+87.5 (1.00)	8.8	+93.5 (1.00)	300.0	+89.3 (1.00)	+15.0 (1.00)	+19.0 (1.00)	-4.3 (0.25)	
cs38	RULE	23.1	+23.5 (0.97)	19.6	+31.3 (1.00)	149.0	+47.5 (1.00)	22.9	+59.2 (1.00)	18.1	+65.2 (1.00)	178.0	+79.2 (1.00)	+0.9 (0.50)	+8.0 (1.00)	-16.3 (0.25)	
cs39	STATE	12.1	+51.3 (1.00)	6.2	+73.4 (1.00)	127.5	+32.3 (1.00)	11.8	+61.2 (1.00)	6.0	+89.8 (1.00)	129.0	+46.8 (1.00)	+2.3 (1.00)	+3.4 (1.00)	-1.2 (0.38)	
cs40	STATE	19.3	+52.9 (1.00)	13.3	+68.6 (1.00)	129.2	+29.2 (1.00)	19.3	+105.2 (1.00)	13.4	+150.5 (1.00)	131.3	+49.4 (1.00)	+0.1 (0.58)	-1.0 (0.42)	-1.6 (0.25)	
Mean		17.0	+29.4 (1.0)	15.7	+33.4 (1.0)	88.0	+30.1 (0.9)	15.5	+47.0 (1.0)	13.8	+56.0 (1.0)	89.2	+25.8 (0.9)	+11.1 (0.7)	+22.0 (0.8)	+0.1 (0.5)	
#(10%, 20%]		25		18				25		15							
#(20%, 100%]		11		10				8		7							

that Line% is more than 20% on 11 case studies. In term of Critical Line%, SM EvoMASTER achieved more than 10% line coverage on 28 (i.e., 18 + 10) out of 40 case studies, and Critical Line% is more than 20% on 10 case studies. For Base, EvoMASTER with 10-hours search budget, it can cover on average 15.5% (up to 31.3%) Line% and 13.8% (up to 47.2%) Critical Line%, and identify on average 89.2 (up to 300.0) potential faults on 40 case studies.

By comparing results obtained by 10-hours and 1-hour search budgets, for SM, the improvement achieved by 10-hours is significant with more 29% positive relative improvement (i.e., +29.4% for Line%, +33.4% for Critical Line% and +30.1% for #Detected Faults) and high  $\hat{A}_{12}$  (i.e., 1.0 for Line% and Critical Line%, and 0.9 for #Detected Faults) for all three metrics. For Base, compared to 1-hour, the improvements using 10-hours are significant (i.e., relative improvements are +47.0% for Line%, +56.0% for Critical Line% and +25.8% for #Detected Faults; and  $\hat{A}_{12}$  are 1.0 for Line% and Critical Line%, and 0.9 for #Detected Faults). Besides, we compared SM with 10-hours and Base with 10-hours. SM EvoMASTER achieved overall better performance for line coverage related metrics compared to Base with 10-hours budget configuration, i.e., relative improvements for Line% and Critical Line% are +11.1% and +22.0% respectively, and  $\hat{A}_{12}$  for Line% and Critical Line% are 0.7 and 0.8 respectively. However, in terms of detected faults, we still do not observe improvements obtained by SM EvoMASTER.

Comparing results of Table 5 (i.e., SM 1h vs. Base 1h) and Table 6 (SM 10h vs. Base 10h), we found that relative improvements for

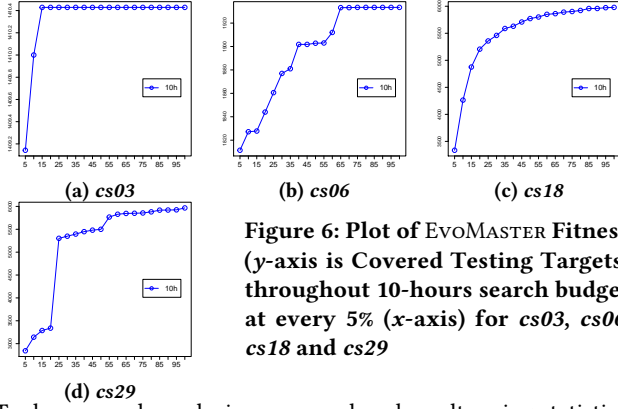
line coverage achieved by SM vs. Base are reduced (i.e., +26.9% vs. +11.1% for Line%, 40.9% vs. 22.0% for Critical Line%), while  $\hat{A}_{12}$  for line coverage remains the similar (i.e., 0.8 vs. 0.7 for Line%, 0.8 vs. 0.8 for Critical Line%). As more budget is allocated to EvoMASTER, the extent of relative improvements attributed to EETests might depend on how effectively the lines covered by EETests can be covered by Base EvoMASTER. However, EvoMASTER with SM continues to maintain a higher probability of outperforming Base as indicated by the results of  $\hat{A}_{12}$ .

**RQ4:** On 40 enterprise case studies, EvoMASTER with 10-hours search budget obtained consistently and significantly better results compared to 1-hour. Compared to Base 10h, EvoMASTER with SM configuration still obtain better overall performance. EvoMASTER with SM and 10-hours search budget is capable of generating system-level tests that can cover on average 17.0% (up to 32.4%) line coverage, 15.7% (up to 47.4%) line coverage for critical code, and 88.0 (up to 287.0) potential faults. In addition, SMEvoMASTER is effective at covering more than 10% line coverage on 90% (and more than 20% on 27.5%) case studies and cover more than 10% critical line on 70% (and more than 20% on 25%) case studies. This demonstrates the effectiveness of EvoMASTER on enterprise case studies.

### 5.3 Threats To Validity

**Conclusion validity:** As EvoMASTER is a SBSE fuzzer, we followed the guidelines in [14] about repeating each configuration of our experiments multiple times (e.g., at least 10 times for SM 1h and Base 1h, and at least 2 times for SM 10h and Base 10h), in order to reduce the negative impact of randomness on the results.





**Figure 6: Plot of EvoMASTER Fitness (y-axis is Covered Testing Targets) throughout 10-hours search budget at every 5% (x-axis) for cs03, cs06, cs18 and cs29**

To draw sound conclusions, we analyzed results using statistical methods, such as Spearman correlation for correlation analysis, Mann–Whitney U test and Vargha-Delaney effect size for comparison analysis; **Construct validity**: All experiments were deployed on the same enterprise testing pipeline at Meituan; **Internal validity**: EvoMASTER is an open-source fuzzer that can be used by anyone who are interested in. As proprietary enterprise APIs, it is not possible to provide these case studies as part of a replication package. However, we provide raw data with removed confidential info for verifying our analyses, i.e., re-generating tables and figures reported in the paper with analysis scripts [7]; **External validity**: We conducted our experiment on 40 enterprise RPC APIs at Meituan (consisting of 5.6 million lines of code for their business logic) covering a broad range of size of enterprise Web APIs, e.g., ranging from 132 to 2,315 for Class files existing in an API, and ranging from 8,597 to 797,913 lines of code for an API.

## 5.4 Lessons Learned

**5.4.1 Identified Bottleneck.** SM EvoMASTER with 10h had limited performance on four case studies, i.e., cs03, cs06, cs18, and cs29 (Table 6). On these four case studies, we further plot the performance of SM EvoMASTER based on *Testing Targets* [11, 12] (an aggregated score of all fitness metrics EvoMASTER optimizes for) For cs03, we found that there is no improvement in EvoMASTER fitness starting from 15% used budget (Figure 6a), and this result also explains the equivalent performance achieved by 10-hours of SM and 1-hour of SM (Table 6). Such results on cs03 revealed that EvoMASTER equipped with current techniques (e.g., *testability transformation* [17] and *adaptive mutation* [46]) can at maximum solve around 7% lines of code of cs03, and more search budgets (e.g., more than 1-hour) likely would not lead to better results. We also observed convergence where it occurs at 65% of used budgets on cs06 (Figure 6b). For these two case studies, in order to cover further lines of code, new techniques (e.g., more advanced white-box heuristics) are needed to improve the fuzzing.

To investigate what the bottlenecks are, we analyzed the two case studies with our industrial partner. For cs03, we found that most of the business logic of this RPC API is mainly triggered by consuming *Message Queue* (MQ), implemented with an in-house modified version of Kafka [4]. An example is shown as follows:

```
1 @MKafkaListener(topics = "topic", groupId = "group")
2 public void listen(String message) {
3     //consume MQ
4 }
```

All the exposed RPC functions (i.e., 7 as shown in Table 1) are used to query info related only to a small amount of code.

On cs06, the core business logic are enabled by scheduled tasks using in-house developed framework, e.g., an example as:

```
1 @ScheduledTasks(task.id)
2 public void taskExecute(Long A ,Long B){
3     //perform the scheduled task
4 }
```

For cs03 and cs06, the current EvoMASTER by modifying RPC call and mock objects for dependent services cannot access the core implementation, which explains the worse results. After these findings, our industrial partner systematically checked all these 40 RPC APIs and found that there exist in total 273 functions developed as MQ consumers and 421 functions defined as scheduled tasks. Thus, supporting these two features would probably further improve the effectiveness of EvoMASTER on enterprise APIs.

**Summary:** *Message Queue and Scheduled Task commonly exist in large distributed enterprise microservices. To better test these services, there is the need for new solutions for facilitating automatically identifying if such techniques have been employed in the service under test, and further enable testing for such scenarios. In addition, in large enterprises they might use in-house develop libraries/frameworks. Thus, the proposed solutions should be customizable and extensible.*

For cs18 and cs29, the performance tends to improve with the passage of time, but still it is interesting to investigate if the performance can be more efficient. Therefore, we took a look into cs18 and cs29 with our industrial partner in more details. For cs18, we found that the logic of the RPC API is complex, performing various calculations by relying on multiple data sources, e.g., an example:

```
1 public Dto assess(Request request){
2     Response r1 = ExternalService.get(request);
3     Result r2 = queryComputingRules(request); // 1 Table
4     Result r3 = queryInfoForFoo(request) //Redis, 3 Tables
5     Result r4 = queryInfoForBar(request) // 2 Tables
6     ComputedResult result = compute(r1,r2,r3,r4);
7     // process bussiness logic based on the computed result
8 }
```

To carry out the computation at line 6, it requires to perform an external API call (line 2), load rules<sup>2</sup> for the computation (line 3) and query multiple related data from Redis and SQL databases (lines 4 and 5). If any of such data is not valid, line 6 cannot be reached. Different data and rules would also result in different execution flows, e.g., loaded computing rules r2 directly impacts the computed result at line 6.

**Summary:** *Enterprise APIs heavily rely on data which can be from various sources, e.g., external services, Redis and databases. EvoMASTER can manipulate external services and SQL databases if they are specified in seeded tests, but it is unsure whether the seeded tests define all direct connected external services and databases. It first requires an automated solution to identify what kind of sources exist, and then enable preparation of the data. However, as the enterprise services can be complex, how to prepare required and varied data in a cost-effective manner is also an important challenge to address.*

<sup>2</sup>an example of a rule, e.g., {"id":1,"name":"ruleA","pass":30.0,"target":10.0,"weight":10.0,"orderType":1,"valueType":2}. The rule defines: if the given value is a type of 2 and the request is a type of 1 and the value is between 10.0 and 10.0+30.0, it scores  $1 \times 10/100$ . How to use the rule depends on compute at line 6.

For *cs29*, this RPC API is responsible for dispatching tasks. Before starting to dispatch a task, it needs first to query data from various sources, and then carry out a restrict pre-validation to ensure if the dispatching can be performed. Then, this case faces a similar problem as *cs18*. Besides, to satisfy diverse business requirements, there might exist different strategies to handle tasks (e.g., a new version might be released to a small subset of users at the beginning, referred as *Gray release*), and which strategy to use can be managed by remote configurations, e.g., an example:

```
1 Map configValue = configClient.get(API_ID);
2 if(!isConfigInGrayState(configValue))
3     return null;
```

Whether the task can be further processed totally depends on the loaded configurations (see line 1). Without editing the configuration, it is not feasible to access other strategies. To support such cases, EVO MASTER needs to have a solution to identify such remote configurations and modify them. In addition, as such configuration manages the business logic and might globally impact the microservices, to maintain consistency of business logic within a test, it is better to manipulate the configurations before executing each test.

**Summary:** *In a large-scale enterprise microservices, business logic of a Web API can be managed remotely by configurations. In an industrial context, a fuzzer would need the capability of identifying such configurations and facilitating manipulating them properly during the testing. Besides, it might need to design test criteria specific to coverage of such configurations, for better covering business logic.*

**5.4.2 Application of EVO MASTER.** Regarding search budget configuration in an industrial setting, based on pair comparison results achieved by 10 hours and 1 hour, we found that that results with 10-hours are significantly better than 1 hour, consistently on both configurations of *Base* and *SM*. In Table 6, we also found that there exist 7 case studies which *SM* 10h achieved more than 50% relative improvements of Line% compared to *SM* 1h, i.e., +82.0% for *cs32* (Flow), +74.5% for *cs36* (Flow), +60.9% for *cs24* (Flow), +59.7% for *cs08* (CRUD), +59.1% for *cs37* (CRUD), +52.9% for *cs04* (STATE), and +51.3% for *cs39* (STATE). 5 out of 7 case studies are relatively complex types, i.e., either FLOW or STATE type, except *cs08* and *cs37* (CRUD). For *cs08*, because *SM* 1h underperformed as seeded EETests take most of search budget while covering only little code, the improvement made by 10-hours might be further enlarged. For *cs37*, it might relate to its size, i.e., the fourth most #Files (see Table 1). This can also be observed in the results of comparing *Base* 10h and 1h, i.e., EVO MASTER likely obtained more improvements on relatively complex case studies.

**Summary:** *Considering the complexity of the enterprise services, we recommend fuzzing enterprise APIs with EVO MASTER for longer time, such as 10 hours. Larger budgets likely result in better results.*

For *SM* EVO MASTER, the quality of seeded tests (i.e., EETests) may have a strong impact on results achieved by EVO MASTER. As discussed with the industrial partner, we found that there is no optimization phase in their current practice. For instance, the problem revealed in *cs08* is due to lots of redundant tests generated by *Record & Replay*. With EVO MASTER, it can optimize such tests by *archive-based populations management* [11], i.e., removing tests that cannot contribute to covering new testing targets. In addition, industrial partner would be more interested in tests that can identify

code that EETests do not currently cover. Thus, it would be useful to have a separate file for those tests. But, currently EVO MASTER generates too many tests, and it is difficult for the test engineers to identify the most useful tests for them. To better use the generated tests, it would be useful to design some strategies to output a small set of tests, e.g., the tests that are most likely *important* to users based on EETests or critical code. Hence, such tests can be moved to EETests, and users can also create new ones by modifying them.

**Summary:** *EVO MASTER can help to optimize and maintain existing enterprise tests. In addition, tests generated by EVO MASTER also need to be optimized from industrial perspectives for industrial practitioners in order to better use them.*

It is a clear advantage that EVO MASTER can identify new faults. Comparing *SM* and *Base*, there is no much different between these two configurations. However, our industrial partner stated that many faults are due to a lack of input validations in terms of null/-format check. This type of faults are less important, because most of APIs at back-end are not used directly by customers (i.e., external users), and it is rare that other APIs inside the microservice architecture give inputs which have format errors or NULL. Thus, such faults might have low priority to be fixed, but identifying the faults are time-consuming. For instance, *SM* EVO MASTER with 10-hours identified  $88.0 \times 40 = 3520$  faults on average. It is impractical for them to check all those faults in a reasonable amount of time. Then, it is possible that critical faults might be revealed in the tests, but be ignored. To better use the tests generated by EVO MASTER, it needs to have strategy to distinguish faults. For example, as faults existing in the implementation for *input validation* might be less important, then we can categorize them into a separate test suite file, to emphasize other faults which might be more critical.

**Summary:** *EVO MASTER on both configurations (i.e., Base and SM) demonstrates its advantage of identifying potential faults. However, with current fault identification strategy, EVO MASTER outputs many not so important faults, which makes it hard for industrial practitioners to find potential critical ones among all those found faults. There is an urgent need to have a prioritization strategy to help checking the identified potential faults.*

## 6 CONCLUSIONS

Microservices pose lots of testing challenges. In this paper, we carried out a large-scale empirical study for investigating how the challenges have been tackled by EVO MASTER (a state-of-the-art academic prototype) on 40 enterprise RPC APIs that are parts of a large e-commerce microservices (i.e., more than 1,000 services in the back-end) at Meituan. Results show that seeding existing enterprise tests and using mocking can help a search-based fuzzer such as EVO MASTER to improve performance significantly in terms of line coverage, but not fault detection. EVO MASTER with the best configuration is effective at covering more than 10% line coverage on 90% of the case studies and more than 20% line coverage on 27.5% of the case studies. It achieved on average 17% (up to 32.4%) line coverage, identifying 3520 potential faults. To further improve the performance and better fuzz enterprise APIs, a set of problems commonly existing in enterprise APIs need to be addressed by the academic community, e.g., support the handling of *Message Queue* and *Scheduled Tasks*.

## REFERENCES

- [1] [n. d.]. Core concepts, architecture and lifecycle. <https://grpc.io/docs/what-is-grpc/core-concepts/>.
- [2] [n. d.]. Dubbo. <https://dubbo.apache.org/en/>.
- [3] [n. d.]. gRPC. <https://grpc.io/>.
- [4] [n. d.]. Kafka. <https://kafka.apache.org/>. Online, Accessed August 6, 2024.
- [5] [n. d.]. LibFuzzer. <https://llvm.org/docs/LibFuzzer.html>.
- [6] [n. d.]. Protocol Buffers. <https://protobuf.dev/>.
- [7] [n. d.]. Replication Package for Seeding and Mocking in White-box Fuzzing Enterprise RPC APIs: An Industrial Case Study. <https://github.com/man-zhang/seeding-mocking-packages>.
- [8] [n. d.]. SOFARPC. <https://www.sofastack.tech/en/>.
- [9] [n. d.]. thrift. <https://thrift.apache.org/>.
- [10] [n. d.]. Thrift interface description language. <https://thrift.apache.org/docs/idl.html>.
- [11] Andrea Arcuri. 2018. Test suite generation with the Many Independent Objective (MIO) algorithm. *Information and Software Technology* 104 (2018), 195–206.
- [12] Andrea Arcuri. 2019. RESTful API Automated Test Case Generation with EvoMaster. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 28, 1 (2019), 3.
- [13] Andrea Arcuri. 2020. Automated Black-and White-Box Testing of RESTful APIs With EvoMaster. *IEEE Software* 38, 3 (2020), 72–78.
- [14] 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.
- [15] 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.
- [16] 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.
- [17] 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.
- [18] 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.
- [19] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. 2019. RESTler: Stateful REST API Fuzzing. In *ACM/IEEE International Conference on Software Engineering (ICSE)*. 748–758.
- [20] Asma Belhadi, Man Zhang, and Andrea Arcuri. 2023. Random Testing and Evolutionary Testing for Fuzzing GraphQL APIs. *ACM Transactions on the Web* (2023).
- [21] Matteo Camilli, Antonio Guerriero, Andrea Janes, Barbara Russo, and Stefano Russo. 2022. Microservices integrated performance and reliability testing. In *Proceedings of the 3rd ACM/IEEE International Conference on Automation of Software Test*. 29–39.
- [22] Gordon Fraser and Andrea Arcuri. 2013. Whole Test Suite Generation. *IEEE Transactions on Software Engineering* 39, 2 (2013), 276–291.
- [23] Luca Gazzola, Maayan Goldstein, Leonardo Mariani, Marco Mobilio, Itai Segall, Alessandro Tundo, and Luca Ussi. 2022. ExVivoMicroTest: ExVivo Testing of Microservices. *Journal of Software: Evolution and Process* (2022), e2452.
- [24] Amid Golmohammadi, Man Zhang, and Andrea Arcuri. 2023. .NET/C# instrumentation for search-based software testing. *Software Quality Journal* (2023), 1–27.
- [25] Amid Golmohammadi, Man Zhang, and Andrea Arcuri. 2023. Testing RESTful APIs: A Survey. *ACM Transactions on Software Engineering and Methodology* (aug 2023). <https://doi.org/10.1145/3617175>
- [26] Xiaofeng Guo, Xin Peng, Hanzhang Wang, Wanxue Li, Huai Jiang, Dan Ding, Tao Xie, and Liangfei Su. 2020. Graph-based trace analysis for microservice architecture understanding and problem diagnosis. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1387–1397.
- [27] Zac Hatfield-Dodds and Dmitry Dygalo. 2022. Deriving Semantics-Aware Fuzzers from Web API Schemas. In *2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 345–346.
- [28] Pooyan Jamshidi, Claus Pahl, Nabor C Mendonça, James Lewis, and Stefan Tilkov. 2018. Microservices: The journey so far and challenges ahead. *IEEE Software* 35, 3 (2018), 24–35.
- [29] Myeongsoo Kim, Davide Corradini, Saurabh Sinha, Alessandro Orso, Michele Pasqua, Rachel Tzoref-Brill, and Mariano Ceccato. 2023. Enhancing REST API Testing with NLP Techniques. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1232–1243.
- [30] Myeongsoo Kim, Qi Xin, Saurabh Sinha, and Alessandro Orso. 2022. Automated Test Generation for REST APIs: No Time to Rest Yet. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (Virtual, South Korea) (ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 289–301. <https://doi.org/10.1145/3533767.3534401>
- [31] Nuno Laranjeiro, João Agnelo, and Jorge Bernardino. 2021. A black box tool for robustness testing of REST services. *IEEE Access* 9 (2021), 24738–24754.
- [32] Jiangchao Liu, Jierui Liu, Peng Di, Alex X Liu, and Zexin Zhong. 2022. Record and replay of online traffic for microservices with automatic mocking point identification. In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*. 221–230.
- [33] Gang Luo, Xi Zheng, Huai Liu, Rongbin Xu, Dinesh Nagumothu, Ranjith Jana-pareddi, Er Zhuang, and Xiao Liu. 2019. Verification of microservices using metamorphic testing. In *International Conference on Algorithms and Architectures for Parallel Processing*. Springer, 138–152.
- [34] Meng Ma, Weilan Lin, Disheng Pan, and Ping Wang. 2021. ServiceRank: Root Cause Identification of Anomaly in Large-Scale Microservice Architectures. *IEEE Transactions on Dependable and Secure Computing* 19, 5 (2021), 3087–3100.
- [35] Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. 2020. RESTTest: Black-Box Constraint-Based Testing of RESTful Web APIs. In *International Conference on Service-Oriented Computing*.
- [36] Sam Newman. 2021. *Building microservices*. "O'Reilly Media, Inc."
- [37] 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.
- [38] Louis M Rea and Richard A Parker. 2014. *Designing and conducting survey research: A comprehensive guide*. John Wiley & Sons.
- [39] José Miguel Rojas, Gordon Fraser, and Andrea Arcuri. 2016. Seeding strategies in search-based unit test generation. *Software Testing, Verification and Reliability* 26, 5 (2016), 366–401.
- [40] Benjamin H Sigelman, Luiz Andre Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspán, and Chandan Shanbhag. 2010. Dapper, a large-scale distributed systems tracing infrastructure. (2010).
- [41] Charles Spearman. 1961. The proof and measurement of association between two things. (1961).
- [42] Emanuele Viglianisi, Michael Dallago, and Mariano Ceccato. 2020. RESTTEST-GEN: Automated Black-Box Testing of RESTful APIs. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE.
- [43] Wei Wang, Andrei Benea, and Franjo Ivancic. 2023. Zero-Config Fuzzing for Microservices. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1840–1845.
- [44] Muhammad Waseem, Peng Liang, Mojtaba Shahin, Amleto Di Salle, and Gastón Márquez. 2021. Design, monitoring, and testing of microservices systems: The practitioners' perspective. *Journal of Systems and Software* 182 (2021), 111061.
- [45] Huayao Wu, Lixin Xu, Xintao Niu, and Changhai Nie. 2022. Combinatorial Testing of RESTful APIs. In *ACM/IEEE International Conference on Software Engineering (ICSE)*.
- [46] 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).
- [47] Man Zhang and Andrea Arcuri. 2021. Enhancing Resource-Based Test Case Generation for RESTful APIs with SQL Handling. In *International Symposium on Search Based Software Engineering*. Springer, 103–117.
- [48] Man Zhang and Andrea Arcuri. 2023. Open Problems in Fuzzing RESTful APIs: A Comparison of Tools. (2023). <https://doi.org/10.1145/3597205>
- [49] 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* 32, 5 (2023), 1–38.
- [50] 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>
- [51] Man Zhang, Asma Belhadi, and Andrea Arcuri. 2023. JavaScript SBST Heuristics To Enable Effective Fuzzing of NodeJS Web APIs. *ACM Transactions on Software Engineering and Methodology* (2023).
- [52] 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.
- [53] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Wenhai Li, and Dan Ding. 2018. Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study. *IEEE Transactions on Software Engineering (TSE)* (2018).
- [54] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Dewei Liu, Qilin Xiang, and Chuan He. 2019. Latent error prediction and fault localization for microservice applications by learning from system trace logs. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 683–694.