Integrating Fault Injection in Fuzzing: Design, Implementation and Preliminary Experiments

Gilles Coremans[1], Coen De Roover[1]

[1]*Vrije Universiteit Brussel*

Abstract

While microservice architectures offer many advantages to developers of web applications, it remains difficult to test the resilience of these applications to faults. Common approaches such as chaos engineering require dedicated personnel and carry the risk of the testing taking place in a production environment, while resilience testing tools require specifying execution scenarios and cannot test paths not exercised by those scenarios.

In this paper, we introduce Onweer, a resilience tester which integrates coverage-guided fuzzing and fault injection. Fuzzing is used to explore the execution space of the application and to collect potential fault injection points. Subsequent fuzzing iterations can then inject faults as part of the fuzzing process. That is, faults are added to test cases by mutators applied to the population, and if they increase coverage these test cases with faults are added to the population for further mutation.

We evaluate this approach by using three case studies of the *retry* resilience pattern and find that Onweer is able to both increase coverage by exploring otherwise inaccessible branches and reveal resilience defects.

Keywords

Fuzzing, Fault Injection, Resilience, Microservices, REST

1. Introduction

Microservice architectures are a popular choice for the development of web applications, as they offer many advantages in scaling and ease of development. However, they also introduce new complexities to the software development process. While in monolithic systems communication between modules happens via function calls which cannot themselves fail, in microservice systems this communication occurs over the network. Thus, messages may be dropped, potentially causing a transaction to occur twice, or individual services may crash, potentially bringing down the entire application. The application should be *resilient* to these *faults*, as they cannot be prevented.

Achieving resilience is quite difficult in practice, as faults mainly occur when deployed at scale and subjected to production loads, while testing environments or developer machines may not experience any faults at all. Thus, developers may not see the need for resilience or know how to implement it effectively and inadvertently introduce *resilience defects* into the application.

In order to test the resilience of their applications, many large organizations practice *chaos engineering* [1][2]. Chaos engineering involves the controlled injection of faults into a production system, which is then monitored for any adverse effects such as increased latency or decreased user engagement due to stability issues [3]. While this has proven successful at finding resilience defects, testing in production is inherently risky, does not prevent defects from making it to production in the first place, and requires specialized *site reliability engineers* (SREs) to carry out effectively.

Ideally, resilience would be tested automatically and in a staging environment. However, real-world resilience testing without the loads of a production system is difficult. Some defects may only occur on specific execution paths, or require a combination of multiple faults in a specific order. Defects requiring multiple faults are especially difficult to detect, as the search space quickly becomes intractable even with just a few faults. Several tools have been developed to solve this problem [4][5][2], but they have significant limitations. Generally, these tools require developers to specify the execution

BENEVOL24: The 23rd Belgium-Netherlands Software Evolution Workshop, November 21-22, Namur, Belgium

gilles.coremans@vub.be (G. Coremans); coen.de.roover@vub.be (C. De Roover)

0000-0003-2545-5223 (G. Coremans); 0000-0002-1710-1268 (C. De Roover)

```java
@PostMapping("/pong")
public ResponseEntity<Pong> pongPost(RestTemplate rest, @RequestBody Increment i) {
    long inc = i.increment();
    if(inc >= 0 && inc < 10000) {
        long cur = counter.addAndGet(inc);
        return ResponseEntity.ok().body(new Pong(cur));
    } else {
        return ResponseEntity.badRequest().build();
    }
}
```

Figure 1: The implementation of `pong` used for all examples and in the evaluation.

```java
@PostMapping("/ping")
public ResponseEntity<Ping> pingPost(RestTemplate rest, @RequestBody Increment i) {
    int retry = 0;
    long cur;
    while(retry < 3) {
        ResponseEntity<Pong> pong;
        try {
            pong = rest.postForEntity("http://pong/pong", i, Pong.class);
            cur = counter.addAndGet(i.increment());
            return ResponseEntity.ok(new Ping(cur, pong.getBody().id()));
        } catch(HttpClientErrorException e) { // 4xx status code
            logger.warn(e);
            return ResponseEntity.badRequest().build();
        } catch(RestClientException e) { // Timeout, connection reset, etc.
            logger.error(e);
            retry++;
            continue;
        }
    }
    return ResponseEntity.status(503).build();
}
```

Figure 2: Implementation #3 of `ping`, including a bounded retry mechanism.

scenario under which faults occur, and some tools also require developers to specify which faults are to be injected. This means that developers must spend time creating these specifications and that full coverage of the application is unlikely.

In this paper we present a prototype of Onweer, a tool aimed at fully automating the resilience testing process. Our tool uses coverage-guided fuzzing to automatically explore the execution space of the system under test and generate a diverse set of test inputs. During this fuzzing process, potential fault injection points are collected, and faults are injected as part of the fuzzing process.

In this work, we answer the following research questions:

RQ1 Can fault injection as part of the fuzzing process be used to increase coverage?

RQ2 Can fault injection as part of the fuzzing process reveal resilience defects?

In order to answer these research questions, we use Onweer to test several different implementations of the *retry* resilience pattern, and show how fault injection reveals defects and explores otherwise inaccessible branches.

2. Motivating Example

As a motivating example, consider an application with two services, `ping` and `pong`. Both services have a counter, which they attempt to keep in sync. Sending a POST request to the `ping` service increments

both counters with the value given in the request body and returns the value of the counters.

As can be seen in Figure 1, `pong` is relatively simple: it checks whether the request is in the accepted range, increments its counter, and sends the value back. If the request is not in the valid range, `pong` responds with a generic `400 Bad Request` error.

`ping`, then, must simply pass along the amount to increment by, increment its own counter, and respond with both counter values. However, there are some extra complexities to consider. The response from `pong` must be checked to see whether the request was valid, but errors on the connection to `pong` must also be taken into account.

Figure 2 shows an implementation of `ping` which takes all of these factors into account, checking for all status codes and incorporating a bounded retry pattern. The implementation grows quite complex because of these additional checks, and hence features more conditional branches than an implementation that would ignore these checks would have. Manual testing of an application like this requires taking in account every failure condition to ensure that every branch is covered. If an exception is not handled, it is even possible that full code coverage is achieved in the presence of a defect, with no indication that some cases have not been tested.

Therefore, we would like to automatically test code like this, to achieve high coverage while minimizing manual effort, and to ensure that "invisible" defects are revealed. In order to find such resilience defects, we must explore both the input space of the application to determine which code is reached under which inputs, as well as the fault space to determine whether that code is resilient to faults.

Fuzzing allows us to efficiently explore the input space, especially if we incorporate coverage feedback. This still leaves the issue of fault injection, specifically knowing where faults can be injected and how to efficiently explore the fault space of the program. We propose that fuzzing can be leveraged to explore the fault space as well, by discovering potential fault injection points during fuzzing and by using coverage feedback to identify interesting faults similar to how we identify interesting inputs.

3. Onweer

3.1. Architecture

Figure 3 contains an overview of Onweer's architecture. The tool is divided into the *Onweer controller* and the *Onweer agents* attached to the system's microservices.

The agents are attached to each of the services in the system. They collect coverage information and fault injection points during execution. Agents are also responsible for injecting the faults associated with a test case. Currently, the only agent implementation is a Java agent[1], but communication occurs via a simple REST interface and thus other languages can be supported with a modest amount of effort.

The Onweer controller is a coverage-guided fuzzer written in Python. It uses a conventional fuzzing architecture: test cases are drawn from a population, mutated, ran, and the mutated test is added to the population if it increases coverage as reported by the agents. However, this architecture is complicated by several additions to support REST APIs and fault injection. For the purposes of mutation, test cases are split into REST request data and to fault injection data, and these two parts are mutated separately.

3.2. REST API Fuzzing

REST requests are generated with the help of an OpenAPI [6] schema. OpenAPI is a standard for specifying the endpoints, arguments and responses of a REST API. The schema for the `ping` service is shown in Figure 4. It first defines a schema for two JSON objects, an `IncrementRequest` object containing a single integer and a `PingResponse` object containing two integers. It then defines the paths and methods available on `ping`, in this case only the `POST` method on the `/ping` path. It requires the body of requests to this endpoint to be `IncrementRequest` and the response to be either a `200` status code with a `PingResponse` body or a `400` status code with no schema specified for its body. We

[1]Using JaCoCo for code coverage and ByteBuddy for instrumentation

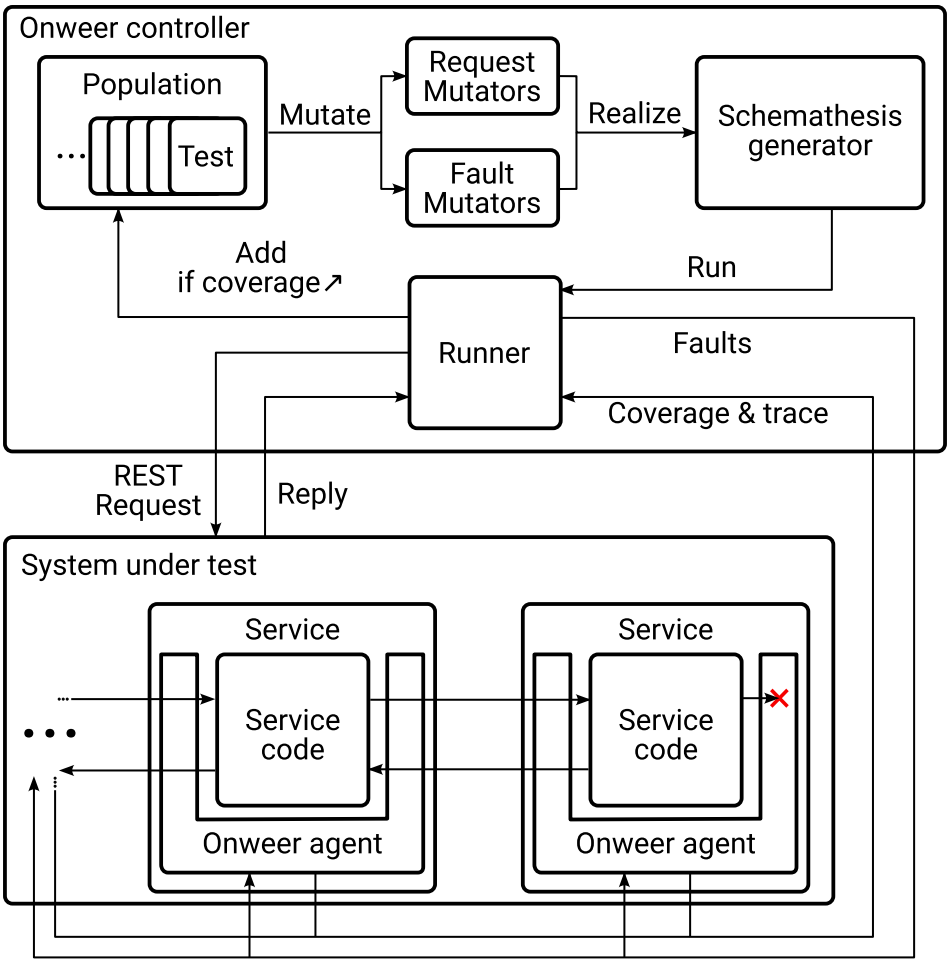

Figure 3: An overview of Onweer's architecture and operation, showing an injected fault represented by a red cross.

base our fuzzer on such a schema as it is otherwise infeasible to automatically discover all the endpoints in an API, which arguments these endpoints take, etc.

Test cases are generated using Hypothesis [7] data generation strategies. Hypothesis is a property-based testing library for Python, and as such includes advanced data generation strategies. Specifically, we use the Schemathesis [8] library to create a Hypothesis data generation strategy for every operation in the API. The Schemathesis data generation strategies produce executable test cases which conform to the schema.

Rather than drawing from a seeded random generator, Hypothesis strategies instead draw from an input bitstream [9], using the values of that bitstream to generate values or decide which branch to take. When a strategy is used to generate a structured input, such as a REST request, each part of the input bitstream corresponds to a specific structural element in the generated output. Furthermore, by construction every input bitstream results in either a schema-conforming REST request or an exception indicating that generation failed for this bitstream.

As described in [10], by mutating the underlying bitstream we can thus safely mutate structured test cases without having to implement complex structural and schema-preserving mutators. For example, if the schema requires that a request body contains one of two fields, then the first byte of the bitstream might encode which field is used. If then those two fields have different types, the second byte could be interpreted as either an integer or a floating point number depending on which field is chosen. If a mutator changes the first byte of the bitstream, the request produced will use a different field, and the second byte will be transparently reinterpreted as a different type by the generator, requiring no extra code in the mutator. On the other hand, if the mutator changes the second byte of the bitstream, this

```yaml
components:
  schemas:
    IncrementRequest:
      type: object
      properties:
        increment: {format: int32, type: integer}
      required: [increment]
      additionalProperties: false
    PingResponse:
      type: object
      properties:
        ping: {format: int32, type: integer}
        pong: {format: int32, type: integer}
      required: [ping, pong]
      additionalProperties: false
paths:
  /ping:
    post:
      requestBody:
        required: true
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/IncrementRequest'
      responses:
        '200':
          content:
            application/json:
              schema:
                  $ref: '#/components/schemas/PingResponse'
        '400':
```

Figure 4: An excerpt of the OpenAPI schema describing the interface offered by `ping`.

will change the value in that field to another valid value of its type, without the mutator needing to know which type that is or even how that byte is interpreted at all.

3.3. Tracing & Fault Injection

Fault injection is fully automated as part of the fuzzing process. In order to inject faults, Onweer must also automatically discover fault injection points.

When the system under test starts, all potential fault injection points (such as REST call sites) are instrumented so that whenever a point is executed, it is added to a *trace* of fault injection points. This instrumentation also adds code to enable fault injection at these points. After the execution of a test case, Onweer fetches the traces from every agent and merges them, storing it with the test case. This merged trace then contains every fault injection point that is executed by that test case.

This allows us to add faults to a test case by defining a mutator, in the same way we define mutators to generate new REST requests from the population. When this mutator is run, it picks a fault from the test's trace and adds it to the list of faults to be injected. Then, before the test is executed, the controller sends this list of faults to the appropriate agents, which will inject it when the execution reaches the specified point.

Currently, the only agent implementation is a Java instrumentation agent which intercepts Spring Boot API calls, and thus fault injection points correspond to the source code location of Spring REST requests. There are currently two kinds of faults that can be injected at these points: a `RestClientException` thrown before or thrown after sending the corresponding request. An exception thrown before a REST request simulates a connection error causing the initial request to be lost, whereas an exception thrown after simulates the response to that request being lost. These two fault types cause

```java
@PostMapping("/ping")
public ResponseEntity<Ping> pingPost(RestTemplate rest, @RequestBody Increment i) {
    ResponseEntity<Pong> ponge = rest.postForEntity("http://pong/pong", i, Pong.class);
    long cur = counter.addAndGet(i.increment());
    Pong pong = ponge.getBody();
    return ResponseEntity.ok(new Ping(cur, pong.id()));
}
```

Figure 5: Implementation #1 of `ping`, which has no guards against errors on the connection and does not handle the `400` response from `pong`.

```java
@PostMapping("/ping")
public ResponseEntity<Ping> pingPost(RestTemplate rest, @RequestBody Increment i) {
    long cur;
    ResponseEntity<Pong> ponge;
    try {
        ponge = rest.postForEntity("http://pong/pong", i, Pong.class);
        cur = counter.addAndGet(i.increment());
    } catch(HttpClientErrorException e) {
        logger.warn(e);
        return ResponseEntity.badRequest().build();
    } catch(RestClientException e) {
        // Retry
        logger.error(e);
        ponge = rest.postForEntity("http://pong/pong", i, Pong.class);
        cur = counter.addAndGet(i.increment());
    }
    Pong pong = ponge.getBody();
    return ResponseEntity.ok(new Ping(cur, pong.id()));
}
```

Figure 6: Implementation #2 of `ping`, a naive implementation which only retries the request once.

different behavior which must be appropriately handled by the application for correct operation. Other types of faults are possible and occur in production systems, but the faults we have implemented are common in production systems, and even these two simple fault types already enable us to explore interesting behaviors of the application under test. Furthermore, these faults also clearly demonstrate the principle of Onweer, as it is clear that any application should be resilient to them, and thus it is reasonable to consider any failure to handle these faults to be a bug.

Along with the point and type, the controller also determines the number of times executing the point results in a fault being injected. For example, if a fault is injected once, an exception will be thrown only the first time the fault injection point is executed. If it is injected five times, an exception will be thrown the first five times the point is executed. This allows Onweer to explore retry mechanisms, which might require multiple consecutive faults to reach some branches.

4. Evaluation

4.1. Design

To evaluate Onweer we use three implementations of the *retry* pattern as case studies. The retry pattern is a simple resilience pattern where a failed request is retried a bounded number of tries to attempt to successfully complete the request [11]. However, it is still not trivial to implement correctly, as the case where all retries fail must be handled, idempotency of the retried request must be taken into account, etc. Each of these case studies implements `ping` in a different way while using the same `pong` discussed in Section 2 and shown in Figure 1.

Case study #1, shown in Figure 5, is a naive implementation without any resilience pattern, so any fault injected here will result in a bug being found. This case study also does not check for the `400 Bad Request` error condition and thus contains a bug that can be found without fault injection.

Case study #2, shown in Figure 6, implements a "naive" retry pattern, as it assumes that the fault is transient and will not occur a second time. Hence, injecting one fault will not reveal the defect, but injecting two faults will.

Case study #3, as shown in Figure 2 and discussed in Section 2, features a correct implementation of the retry pattern. As the implementation is correct, Onweer will not be able to find a defect, but it does include several branches which are only reachable if a fault occurs.

We compare the coverage achieved and defects found between Onweer using only fuzzing but without the fault injection mutator enabled, and Onweer with fault injection enabled to determine the advantages of using fault injection. In both cases, both the `ping` and `pong` services are instrumented by Onweer agents in order to gather coverage and trace data and inject faults.

We perform 10 fuzzing runs for every case study with and without fault injection. Every fuzzing run starts with a seed population of 1 test case and runs for 2 minutes or until a bug is found. Bugs are identified by a `500` status code returned by the server.

4.2. Results & Discussion

Table 1 contains the results of the experiment, listing how often an error was found, the percentage of lines covered, population size (how many coverage-increasing test cases were found), the number of fuzzing iterations ran and the time taken.

Starting with case study #1, we can see that both with and without fault injection Onweer quickly finds an error and terminates. However, the error found differs. Without fault injection, the error found is that the increment is out of range for `pong`, which case study #1 does not check for. With fault injection, Onweer can find both errors, finding the resilience defect 4 out of 10 times in our tests. Notably, this also results in a significantly higher coverage in the runs where the resilience defect was found, resulting in a higher average coverage.

In case study #2, running without fault injection results in no error being found, as the bug cannot occur without a fault. Thus, Onweer executes around 4800 requests over the 2 minutes of allowed runtime. The final population is 2.7 and the final coverage is 87.8% on average. There is some variance in these results because the fuzzing process depends on the initial population and which mutations are

Table 1
Results from the experiments. Each cell contains the average of 10 runs.
† Onweer found the resilience defect in 4 out of 10 runs and found the unhandled `400` status code in the other 6 runs.

	Error found	Coverage	Population	Iterations	Time (*s*)
	Case study #1 (naive)				
No fault injection	100%	33.3%	1	1	1.8
Fault injection	100%†	60%	1	1.6	1.8
	Case study #2 (simple retry)				
No fault injection	0%	87.8%	2.7	4804	120.1
Fault injection	100%	65.6%	1.8	4.7	1.9
	Case study #3 (bounded retry)				
No fault injection	N/A	87.5%	2.5	4770	120.1
Fault injection	N/A	91.4%	4.6	4488	120.1

chosen, and we use a short runtime and small initial population for this evaluation. Proper tuning of the fuzzing parameters and a longer runtime is likely to make results more consistent.

With fault injection, however, the resilience defect in case study #2 is found quickly, even though it requires two consecutive faults to be injected rather than only one. However, a few more iterations are necessary than for case study #1 and the population size is slightly larger on average. This is because injecting one fault increases the coverage and is thus saved to the population, which can then be mutated to inject the second fault and find the defect. This shows that the combination of fuzzing and fault injection is effective in finding resilience defects. Note that while the coverage is lower with fault injection, this is expected as the fuzzing process stops after finding an error and thus does not fully explore the application.

In case study #3, Onweer runs for 2 minutes both with and without fault injection since there is no bug to be found. It again executes about 4800 iterations without fault injection and slightly less with fault injection. However, both the population and the coverage are notably higher with fault injection, as Onweer is able to use fault injection to explore several branches that are otherwise unreachable and find more interesting inputs. In fact, Onweer with fault injection is able to explore all branches in case study #3 and achieved this on several runs. The metric in Table 1 is somewhat lower as the short runtime and small initial population mean this was not achieved on every run due to randomness.

While our evaluation is limited by the small number of case studies, we can answer our research questions affirmatively:

RQ1 In case studies #1 and #3, we see that adding fault injection to the fuzzing process can increase the coverage of a fuzzer. The reduced coverage of case study #2 does not invalidate this, as it is easily explained by early stopping.

RQ2 Case studies #1 and #2 show that adding fault injection to the fuzzing process enables our fuzzer to find resilience defects that cannot be found by a regular fuzzer.

5. Future Work & Limitations

5.1. Stateful REST API Fuzzing

Currently, Onweer only supports sending single REST requests. Most REST APIs, however, are stateful; intended for use with sequences of requests where an object is created and then manipulated on the server. In order to be able to effectively test real-world applications, we thus plan on adding stateful fuzzing to Onweer, whereby it should be able to automatically create interesting sequences of requests rather than only single requests.

Several tools already exist on which our approach can be modeled, such as RESTler [12]. However, most approaches currently described in the literature are black-box methods. Ideally we would extend the methods used in these tools to use coverage feedback. Of course, we will have to integrate fault injection into these frameworks as well.

5.2. Flaky Test Cases

As most microservice applications are stateful, it is likely that some test cases will either not always take the same path through the application or be flaky. For example, when using certain resilience patterns it is possible that a request influences the path taken or result of a subsequent request, even if those requests operate on different server-side objects. Because startup times are typically quite long for these systems, restarting after every test case is not a viable solution. To ensure our tool is robust and efficient, we must take these cases into account and attempt to mitigate them. This could take the form of periodically "resetting" the system if its API exposes such a functionality, using distributed tracing to accurately track all paths associated with a test case, or a combination of these methods.

5.3. Additional Fault Types

As mentioned in Section 3.3, currently the only fault that is injected is a generic exception thrown before or after a Spring REST request. This is a restricted view of fault injection, as many other kinds of faults occur in real-world systems [13]. Thus, we potentially miss errors that could be induced by, for example, crashing a service and letting it come back online. Expanding the range of faults that can be injected would allow Onweer to detect more kinds of bugs in real systems.

We have designed Onweer so that the framework can easily accommodate new kinds of faults. The controller only receives traces from the agents and sends back which faults are to be injected, without making many assumptions about what a fault is. All other logic, such as which fault injection points are recorded and how a fault is injected, is handled by the agents. Thus, a new fault can be implemented in an agent simply by reporting it as a fault injection point in the trace and implementing the fault injection logic, without modifying the controller at all.

Furthermore, the controller and agents communicate through a simple REST API, making it straightforward to write new agents. This way, Onweer can be expanded to include fault injection not only in the business logic of a microservice application, but also in the infrastructure surrounding it such as load balancers or API gateways.

6. Related Work

6.1. Resilience Testing

The concept of using fault injection to test the resilience of distributed applications is not new, and many tools exist in the literature. However, as discussed in Section 1, most tools require the developer to specify part of the execution or fault scenario.

FATE and DESTINI [4] are some of the earliest examples of a resilience tester, though it focuses on traditional distributed systems rather than microservice applications. These tools gather IO points such as network accesses and disk reads during execution of the system. FATE will try automatically inject failures on these IO points using heuristics, whereas DESTINI integrates FATE in a declarative specification framework where faults to be injected and expected responses are encoded.

Gremlin [5] is one of the first resilience testing tools to focus on microservice applications specifically. It works purely on the network layer between services to avoid having to write language- or framework-specific instrumentation, so as to more easily support polyglot microservice systems. It provides a declarative fault specification language with a standard library of patterns that can be used to check whether services implement resilience patterns correctly or to introduce faults into a system. These faults are injected during an execution of the system which is not managed by Gremlin.

Chaokka [14] is focused on Akka actor systems. It traces the execution of a user-defined test case, and injects faults using delta-debugging to try and find a minimal set of faults which can change the outcome of the test case.

Filibuster [2] operates under the assumption that microservices fully encapsulate their dependencies to efficiently explore multiple-fault scenarios. It injects faults as HTTP status codes during the execution of end-to-end tests. It also allows developers to enhance tests with information about which faults are injected and the expected behavior of the system under those faults.

6.2. API Fuzzing

Even though Onweer is focused on resilience testing, it uses REST API fuzzing to explore the execution space of the program and enable fault injection. Thus, it is important to consider the state of the art in this field. Most of the current approaches are black-box, and not all of them easily permit integration with fault injection.

Schemathesis [8] is used as a library by Onweer, but it can also be used a standalone property-based testing tool for REST API testing. It focuses on efficient generation of diverse inputs that conform to an OpenAPI schema and checks the responses to verify whether they match the responses defined in the

schema. However, Schemathesis features no automated generation of request sequences and requires dependencies between operations to be specified in the schema instead. A Schemathesis run produces a test suite which may be kept for continued integration testing of the application.

RestTestGen [15] is an extensible REST API testing framework, which features a variety of built-in data generation strategies. Like Schemathesis, it relies on OpenAPI schemas to generate request values and to determine dependencies between operations. However, it supports custom ordering algorithms for request sequences.

Evomaster [16] is a search-based test generation tool for REST APIs. It works only for Java applications, as it uses code coverage and branch distance metrics to speed up testing, but also has a black-box mode usable on any REST API. It uses predefined "templates" to create sequences of requests, which may be mutated somewhat. Similarly to Schemathesis, Evomaster produces a test suite which aims to cover as much of the application as possible in as few tests as possible.

RESTler [12] is a REST API fuzzer focused on building sequences of requests, using objects returned in previous requests as arguments in future requests. It iteratively builds request sequences by appending API operations to sequences. An operation may only be appended when its dependencies are fulfilled, that is, when operations earlier in the sequence have created or fetched the server-side objects it uses. RESTler uses several heuristics to make this process more efficient. It also uses feedback from the server, in the form of response codes, to infer which sequences result in an error and should thus not be further extended.

7. Conclusion

In this paper we presented Onweer, a prototype resilience testing tool which combines fault injection and fuzzing. Onweer leverages fuzzing to generate interesting paths for fault injection and to collect fault injection points. It integrates fault injection into the fuzzing process by making faults part of the test case and mutating them in the same way as other inputs. This approach allows resilience testing to be fully automated with minimal developer intervention. Through three case studies, we show that Onweer is able to find resilience defects in microservice systems and test branches which are otherwise unreachable to regular testing.

References

[1] H. Tucker, L. Hochstein, N. Jones, A. Basiri, C. Rosenthal, The Business Case for Chaos Engineering, IEEE Cloud Computing 5 (2018) 45–54. URL: https://ieeexplore.ieee.org/document/8383672. doi:10.1109/MCC.2018.032591616.

[2] C. S. Meiklejohn, A. Estrada, Y. Song, H. Miller, R. Padhye, Service-Level Fault Injection Testing, in: C. Curino, G. Koutrika, R. Netravali (Eds.), SoCC '21: ACM Symposium on Cloud Computing, Seattle, WA, USA, November 1 - 4, 2021, ACM, 2021, pp. 388–402. doi:10.1145/3472883.3487005.

[3] M. A. Chang, B. Tschaen, T. Benson, L. Vanbever, Chaos Monkey: Increasing SDN Reliability through Systematic Network Destruction, in: S. Uhlig, O. Maennel, B. Karp, J. Padhye (Eds.), Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication, SIGCOMM 2015, London, United Kingdom, August 17-21, 2015, ACM, 2015, pp. 371–372. doi:10.1145/2785956.2790038.

[4] H. S. Gunawi, T. Do, P. Joshi, P. Alvaro, J. M. Hellerstein, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, K. Sen, D. Borthakur, {FATE} and {DESTINI}: A Framework for Cloud Recovery Testing, in: 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11), 2011. URL: https://www.usenix.org/conference/nsdi11/fate-and-destini-framework-cloud-recovery-testing.

[5] V. Heorhiadi, S. Rajagopalan, H. Jamjoom, M. K. Reiter, V. Sekar, Gremlin: Systematic Resilience Testing of Microservices, in: 2016 IEEE 36th International Conference on Distributed Com-

puting Systems (ICDCS), 2016, pp. 57–66. URL: https://ieeexplore.ieee.org/document/7536505. doi:10.1109/ICDCS.2016.11.

[6] D. Miller, J. Whitlock, M. Gardiner, M. Ralphson, R. Ratovsky, U. Sarid, OpenAPI Specification, 2021. URL: https://spec.openapis.org/oas/v3.1.0.html.

[7] Hypothesis, 2024. URL: https://hypothesis.works/.

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

[9] D. MacIver, How Hypothesis Works, 2016. URL: https://hypothesis.works/articles/how-hypothesis-works/.

[10] R. Padhye, C. Lemieux, K. Sen, M. Papadakis, Y. Le Traon, Semantic fuzzing with zest, in: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Association for Computing Machinery, New York, NY, USA, 2019, pp. 329–340. URL: https://dl.acm.org/doi/10.1145/3293882.3330576. doi:10.1145/3293882.3330576.

[11] R. Kuhn, B. Hanafee, J. Allen, Reactive Design Patterns, Manning, 2017.

[12] V. Atlidakis, P. Godefroid, M. Polishchuk, RESTler: Stateful REST API Fuzzing, in: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), 2019, pp. 748–758. doi:10.1109/ICSE.2019.00083.

[13] F. Silva, V. Lelli, I. Santos, R. Andrade, Towards a Fault Taxonomy for Microservices-Based Applications, in: Proceedings of the XXXVI Brazilian Symposium on Software Engineering, SBES '22, Association for Computing Machinery, New York, NY, USA, 2022, pp. 247–256. doi:10.1145/3555228.3555245.

[14] J. De Bleser, D. Di Nucci, C. De Roover, A Delta-Debugging Approach to Assessing the Resilience of Actor Programs through Run-time Test Perturbations, in: Proceedings of the IEEE/ACM 1st International Conference on Automation of Software Test, AST '20, Association for Computing Machinery, New York, NY, USA, 2020, pp. 21–30. URL: https://dl.acm.org/doi/10.1145/3387903.3389303. doi:10.1145/3387903.3389303.

[15] D. Corradini, A. Zampieri, M. Pasqua, M. Ceccato, RestTestGen: An Extensible Framework for Automated Black-box Testing of RESTful APIs, in: 2022 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2022, pp. 504–508. URL: https://ieeexplore.ieee.org/document/9978261. doi:10.1109/ICSME55016.2022.00068.

[16] A. Arcuri, RESTful API Automated Test Case Generation with EvoMaster, ACM Transactions on Software Engineering and Methodology 28 (2019) 3:1–3:37. URL: https://dl.acm.org/doi/10.1145/3293455. doi:10.1145/3293455.