ROCKFUZZ: fuzzing large-scale microservices in practice

Peng Chen
Ant Group
xxx@xxx.com

Wei Cao
Ant Group
xxx@xxx.com

Alex Liu
Ant Group
xxx@xxx.com

Peng Di
Ant Group
xxxx@xx.com

Abstract

Greybox fuzzing has made impressive progress in recent years, evolving from heuristics-based random mutation to approaches for solving individual path constraints.

Angora [14]

ACM Reference Format:
Peng Chen, Wei Cao, Alex Liu, and Peng Di. 2023. ROCKFUZZ: fuzzing large-scale microservices in practice. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Microservice architecture (MSA) is widely used in service-oriented software over the last few years due to the benefit of their agility, resilience, scalability and maintainability. The key idea behind MSA is to decomposed the monolithic application into a set of loosely coupled services. Services are self-contained and independently deployable with lightweight communication in a MSA, while monolithic applications are tightly coupled in one unit [16, 36, 38].

Microservices are riddled with vulnerabilities. From experience, smaller units should have less vulnerabilities since its simple logic. However, researchers observed that mircoservice applications have more vulnerabilities than monolithic applications [45]. A monolithic application has 39 vulnerabilities in average, while a microservice application has an average of 180 vulnerabilities. This indicates smaller is not always better from the view of total number of vulnerabilities. Though smaller make it easy to maintain and ease issues, but those reliability and security issues in service-oriented software still be inherited in microservices. With the complexity of communication between services, they exposes greater risk surface area. For example, any of the communication API may exits security issues such as unsafe deserialization and unsafe access control, or the dependencies with bugs are repeatedly used in many services, all of which makes the potential security threat expand significantly [17, 18, 48, 58].

Developers are suffered in testing microservices as whole [57]. Functionality of microservices goes beyond the functions performed by individual services, unit testing is not enough for ensuring software quality. Though mock technique can fake returns from function or responses from other services, it needs deep knowledge of the applications, and a lot of labor costs. In addition, these unit testing it hard to maintain due to rapid iteration of applications. End to end testing(E2E) is a technique that tests your entire application from start to finish, along with any of its dependencies, to ensure the application behaves as expected. However, an E2E testing, contains a series of requests in most time, are difficult to construct artificially.

Fuzzing becomes one of the most popular techniques to find vulnerabilities. In general, it give the software a large amount of test inputs, and see if unintended program behaviors happen. Fuzzing has been successfully applied to testing various applications, ranging from operating systems to database. Most state-of-art fuzzers have focused on adopting in system applications and finding memory safe bugs only [12, 14, 15, 40, 60]. Although there are some fuzzing tools focus on web services, none of them are suitable for microservices [2, 3, 7–9, 24, 27, 32, 42, 51]. EvoSuite produces unit tests automatically for Java programs that achieve high code coverage and provide assertions [20, 21]. EvoMaster is able to generate test cases automatically for integration testing of RESTful APIs using an evolutionary algorithm [2, 3]. RESTler generates requests for restful API automatically to reach deeper services state by grammar infer from Swagger specification [8, 24]. Its mutation are guided by feedback from service response with coarse granularity. Pythia fuzz stateful REST API guided by coverage feedback and a learning-based mutation strategy [7]. All of them view web-services as monolithic applications, aren't able to generate inputs and collect feedback across multiple services.

In this work, we designed and implemented a fuzzer for microservices, called RockFuzz. RockFuzz is able to generate multiple high-quality end to end test inputs and collect feedback across multiple services. It collects real world requests as as initial seeds, and generate request sequences by mutating seeds based on inferred grammar and coverage guidance. These request sequences are sent to multiple microservice services as end to end test inputs. instruments theses services to collect feedback and aggregates them by specific ID assigned for requests. In addition, we design test

oracles to verify vulnerabilities in microservices, e.g. error code detection, numerical verification.

To make microservices fuzzing available, RockFuzz solves several challenges.

- Test inputs for microservices, involves multiple difference units, has complicated structure and dependencies. RockFuzz collects lots of real worlds microservice traffic. They are used to determines grammar including structure of the input(Intermediate Representation) and which and what value should be mutated (Blacklist and Corpus). If the inputs have distinct coverage, they will be keep as seeds. Seeds will be selected and mutated grammar-aware with coverage guidance. RockFuzz designs multiple mutation operators for different types of values.
- Test oracles for microservices and coverage feedback across multiple services is absent. RockFuzz introduces specific IDs for each inputs, the IDs will propagate during communication between services. Thus, we can aggregate coverage from multiple services, and locate which inputs violate test oracles. Not only detecting exceptions and error codes of protocols, RockFuzz also check the numerical and error code of application logic.
- Microservice applications need substantial cost to deploy. We reused traditional test environments for microservices to reduce deploying cost. Since the environment may be used for other works, we isolates inputs generated by fuzzing by adding a specific tag. The tag will propagate during communication as the ID of input, too. RockFuzz won't collect any feedback from inputs without this tag.

Summary: TODO

2 Related work

RockFuzz is the first fuzzer for microservices that is able to generate multiple high-quality end to end test inputs and collect feedback across multiple services. The rest of this section introduces and compares the most relevant prior work with RockFuzz.

2.1 Feedback-Guided Fuzzing

Feedback-guided fuzzing mutates inputs via program's feedback in the hope to expore more branches or trigger some vulnerabilities. AFL [19, 61] and LibFuzzer [35] use code coverage to achieve the guidance by saving inputs that trigger new branches as seeds. AFLGo [12] and Hawkeye [13] generates inputs with the objective of reaching a given set of target program locations efficiently. Taint-based fuzzers [14, 15, 43] leverage dynamic taint analysis to identify interesting bytes of each input and then focuses on mutating them. REDQUEEN [6] and GREYONE [22] learn possible critical bytes by monitoring variable value changes. NEUZZ [46] uses neural network to smoothly approximate the program branching behavior and then generate inputs by gradient guidedance. SlowFuzz [41] uses the number of executed instructions as guidance to find algorithmic complexity vulnerabilities, while MEMLOCK [56] uses memory consumption information as the guidance to detect memory consumption bugs. ParmeSan [39] and Integrity [44] direct fuzzing towards triggering vulnerabilities accroding to the state of sanitizer checks. All of these feedback-guided fuzzing cannot be uesd in microservices since they are focused on a single application. RockFuzz collects coverage and error information from multiple application, and uses them as feedback.

2.2 Grammar-Based Fuzzing

Generation-based fuzzing leverages a grammar to generate syntactically correct inputs, which is very useful in programs with complex input format. LangFuzz [30], IFuzzer [50] and Montage [34] generate valid inputs for JavaScript interpreter based on the grammar of JavaScript language. SQUIRREL [62] and RATEL [55] performs type-based mutations on intermediate representations of SQL inputs. Superion [53] and NAUTILUS [4] combines the use of grammars with the use of code coverage feedback for input generation in fuzzing.

Some fuzzers try to learn the grammar automatically without any manual specification, and then leverages the learned grammar to generate seed inputs [11, 29, 52, 59]. GRIMOIRE [11] infers structural properties of the input language automatically by code coverage feedback. Profuzzer [59] probes the input fields and their semantics through per-byte mutations. Skyfire [52] learns a probabilistic context-sensitive grammar from inputs. Learn&Fuzz [29] learn grammar for fuzzing using neural-network-based statistical machine-learning techniques. Symbolic-assited fuzzers [25, 26, 28] get constaints of applications semantic insight from ynamic symbolic execution, and then use a a constraint solver to solve them. Hybrid fuzzers combines both fuzzing and concolic execution, e.g Driller [47], QSYM [60].

Requests in microservices has complex grammar that cannot be modeled by finited rules. RockFuzz infers encoding types of data in requests, and then use corresponding data encoding to decode the data to intermediate representations. These encodings includes comnication protocols(e.g. HTTP, RPC) and serialization format(e.g. JSON, ProtoBuf). If it meets some unknown encoding, RockFuzz try to tokenize it as a list of words. After that, RockFuzz applys type-based mutation on the nodes of IR.

2.3 Web Service Fuzzer

Java is the most popular language used in web services, especially in microservice. There are some fuzzers designed for Java application. Evosuit [20, 21] generates unit test cases with assertions for classes written in Java code automatically. JQF [40] and Kelinci [33] migrate AFL to support JAVA language. jFuzz [31] generates inputs by concolic execution for

Java built on top of JPF [1]. HotFuzz [10] fuzzes individual methodes in Java libraries to find inputs to these methods that demonstrate the presence of algorithmic complexity vulnerabilities.

Serverl fuzzers aim to generate inputs for REST APIs in web services. EvoMaster [2, 3] generate system level test cases for RESTful APIs using evolutionary algorithms. Restler [8] infers dependencies among request types declared in specification, and then generate inputs based on the responses observed during prior test executions in order to exercise the service more deeply. RESTTESTGEN [51] computes the next mutation operations to test inputs based on operation dependencies in a black-box way. [24] extends Restler's mutation techniques to more schema fuzzing rules and learning potential data values from specifications or responses. Pythia [7] augments grammar-based fuzzing with coverage-guided feedback and a learning-based mutation strategy for REST API fuzzing.

Fuzzing web service needs specific test oracles for evaluation. QuickREST [32] checks the responses of fuzzing tests for properties described in OpenAPI documents. [9] introduces four security checking rules used in fuzzing REST APIs. [27] compares the behavior of the same REST API in different versions on the same inputs against each other, and detect regressions in the observed differences.

All prior fuzzers can only generate inputs for single application or service. RockFuzz is the first fuzzer for microservices, which generates inputs for APIs from different appilications, and use coverage accross multiple appilications as guidance.

3 Design

Microservice applications has at least one small services that focus on one particular functionality. Current fuzzers for web services can only fuzz APIs from a single service. They can't not mutate and send requests, or collect feedback across multiple services. We now propose RockFuzz, a grammar-aware coverage-guided fuzzer designed for microservice applications. RockFuzz learns grammar of API and relationship between APIs from real world traffic. These APIs may from different services. According to the API's grammar and communication protocol, We designs and uses an intermediate representation (IR) to maintain API requests in a structural and informative manner. Then we perform type-based mutations on IR to generate syntactically correct requests. After that, attaches a specific tag and ID in the context of request from fuzzer and propagated them during communication between services. Thus it can find indirect vulnerabilities triggered at services receiving requests from other small services but not fuzzer. Through this ID, we can also aggregate coverage from each small services.

Figure 1 shows an high-level overview of our microservice fuzzing framework, RockFuzz. . It operates in five steps:

1. Record traffic from real world.
2. Grammar-aware mutation.
3. Requests sending.
4. Coverage collecting and analysis.
5. Test oracles checking.

3.1 Real world traffic record

Microservices has lots of different APIs, and any API may depend on another one, which forms many long and complicated series of API requests for different usages. Manually writing such requests for testing need huge labor costs. Therefore, such tests wrote by developers are insufficient for seeds in fuzzing test.

We collect seeds and learn how to send requests for mciroservices from real world traffic. We sample traffic in real world that send to the services for production (0.1% sample rate), and record the requests' detailed information for analysis. To sample and record requests, we inject code at the gateway application of microservices (e.g. HTTP gateway), or directly instrument code at Communication middleware (e.g. RPC), as shown in Figure 2. The inject code serializes the instance of requests from different machines at runtime and write them to a distributed log storage. The recording information enable RockFuzz to replay the request. For example, a request using HTTP protocol will include its URI, cookie, request header, body, and response. In addition, we also got the requests' context from the middleware of microservices, e.g. user ID invoking the service and the ID assigned for the request.

After that, we analyse these recording traffic offline to learn their dependencies and grammar. The analysis step is invoked after a interval of time in a data processing platform (e.g. Spark) since it need enough requests for finding their dependencies. The data processing platform reads requests from log storage and analyses them at intervals. Some requests can't not send directly because they need specific context. Mocking can forge the context in replaying the same requests, but it is hard to restore unknown context after mutation. We try to finds requests' front and rear requests for construct their contexts and monitor their influence for later behaviors. We define scenario as a unique series of APIs for a specific function, and a request sequence is an instance of scenario which has detailed requests. As the example shown in Figure 3, a scenario may contains APIs for create a trade , pay the trade, and query the pay result finally. We assume that the requests sent from the same user in within a certain time are in the same scenario. Through grouping requests from the same user within time windows, we can find the related requests from the data stored in log storage, and merge them into a request sequence of a scenario. These sequences will been replaying in test environment, if they get successful response and trigger new branches, we save them as seeds for fuzzing. All seeds will be saved in database. In addition, we will parse all of the real world traffic as intermediate

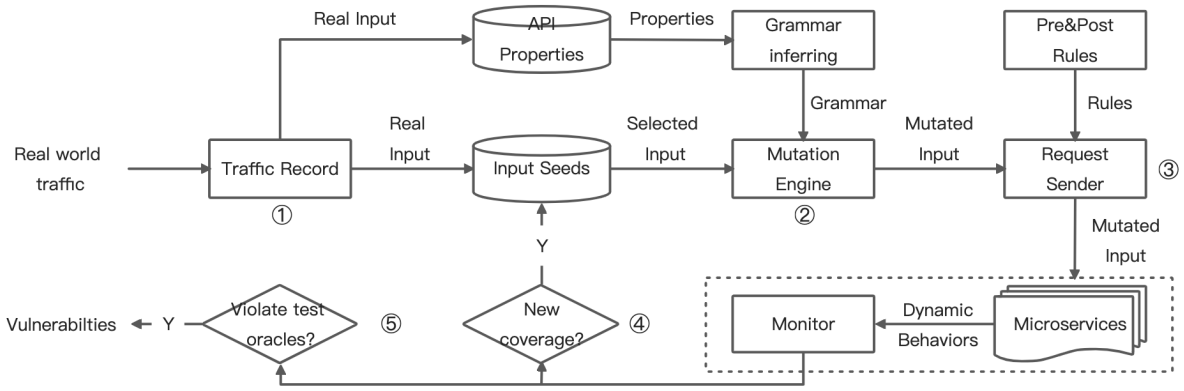

Figure 1. Overview of RockFuzz's architecture.

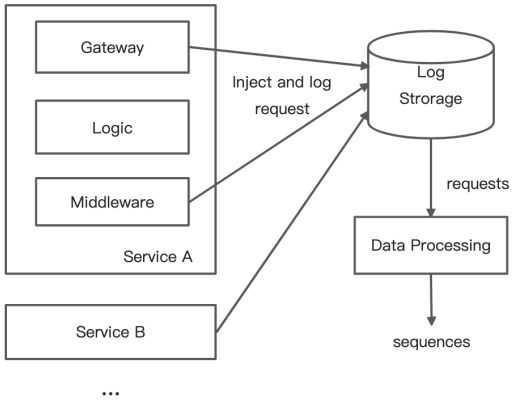

Figure 2. Interaction of recording real world traffic.

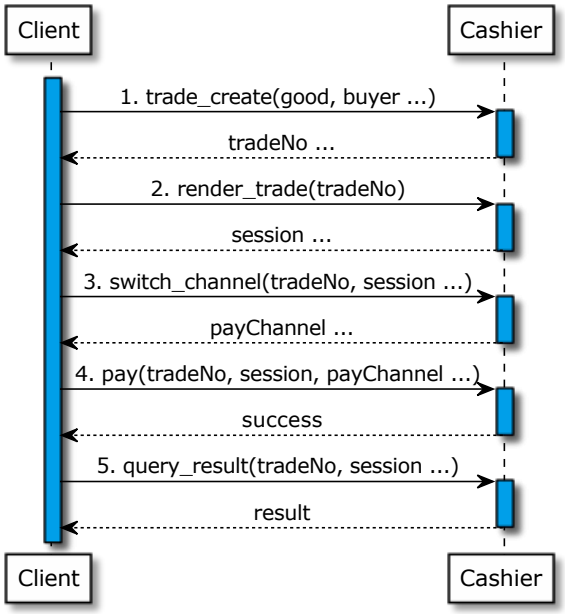

Figure 3. An example of sequence.

representation described in Section 3.2.1, and takes the value of leaf nodes or subtree with low height as corpus and saved them in database.

3.2 Grammar-aware mutation

3.2.1 Intermediate representation and Grammar inference. Communication protocols of microservices have many different types, e.g. HTTP, MQTT, gPRC, and Bolt. Data inside requests may have different encoding, e.g. JSON, ProtoBuf, and custom encoding. Even worse, the encoding may nest each other. Applying fuzzing techniques for these inputs needs do a lot of works for adaption. The mutation strategies should designed for different protocols and encoding since they has different structure, which is repeated. We design an intermediate representation (IR) to maintain API requests in a structural and informative manner. RockFuzz supports decoding and encoding of most common protocols and data encoding. It unpacks requests based on the specification of protocols, and then decode the internal data into intermediate representation with minimal granularity. After processing, they can be encode and pack back into requests that can be sent to services. The intermediate representation is a multi-branches tree, each node contains its type and value. If the value of a node can be break into smaller granularity, it will produces children storing decomposed value.

Figure 4 shows such an example of HTTP request. We parse the request arguments, header and body based on HTTP protocol specification, then we infer their encoding and types, and parse them based on the encoding recursively. According the parsing algorithm shown in Algorithm 2, the example of HTTP request will be converted to the intermediate representation shown in Figure 5. For HTTP arguments, they are URI encoding at most time, and we can view them as a map. The body of requests is a byte string, but they can be encoded differently. We infer their encoding by the features

Algorithm 1 RockFuzz's fuzzing loop.

```
1: function FUZZ
2:     seeds ← initialize from real world traffic.
3:     reports ← ∅
4:     repeat
5:         seed ← selectSeed(seeds)
6:         seq_irs ← map(seed.requests, parseAndDecodeReque
   ▷ Algorithm 2
7:         strategy ← fetchStrategy(seed)
8:         for k ← 1 to N do
9:             if strategy.mutateSequence then
10:                sequenceBasedMutate(seq_irs)          ▷
   Algorithm 3
11:            end if
12:            if strategy.mutateRequest then
13:                typeBasedMutate(seq_irs)              ▷
   Algorithm 4
14:            end if
15:            status                                    ←
   sendSequence(seq_irs, strategy.pre_rules, strategy.post_rules)
16:            if has new coverage then
17:                input                                 ←
   enchodeAndPackSequence(seq_irs)
18:                seeds ← seeds ∪ input
19:            end if
20:            if violate test oracles then
21:                report ← generate report based on dy-
   namic information
22:                reports ← reports ∪ report
23:            end if
24:        end for
25:    until interrupt loop
26: end function
```

```
POST /trade/create?token=test HTTP/1.1
Content-Type: application/json
{
  "good": { name: "milk", "price": 12.0 },
  "buyer":"Bob",
  "ext_params": "channel^main#seller^shopA
#categories^2221|2223"
}
```

Figure 4. An example of HTTP request.

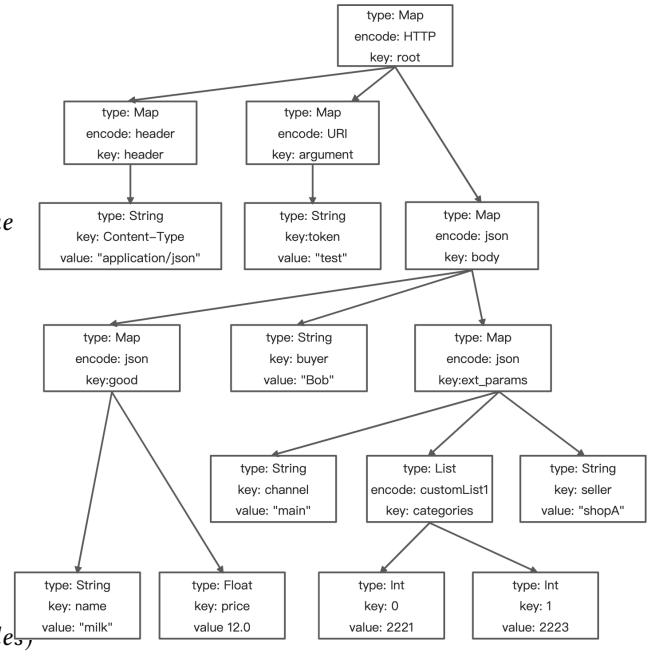

Figure 5. Intermediate representation of Figure 4.

Algorithm 2 Parse and decode requests as IR.

```
1: function PARSEANDDECODEREQUEST(request)   ▷ request: a request to microservice API. return: request's IR.
2:     ir ← parseByProtocol(request)
3:     stack ← ir.getLeafNodes()
4:     while stack is not empty do
5:         leafNode ← stack.pop()
6:         if leafNode is string type then
7:             encoding ← inferEncoding(leafNode)
8:             if encoding is not null then
9:                 subTree                          ←
   parseByEncoding(leafNode, encoding)
10:                ir.replace(leafNode, subTree)
11:                stack.addAll(subtree.getLeadNodes())
12:            end if
13:        end if
14:    end while
15:    return ir
16: end function
```

of encoding using regular expression. Since the encoding is inferred as JSON, we use JSON decoder to disassemble the string, and put them as a subtree in the IR. Once we find any leaf node of the tree is String type, we try to infer its encoding and disassemble them repeatedly. Finally, every leaf nodes in IR will be a primitive type such as String Int, and intermediate nodes are types of collections such as Map, List.

ProtoBuf is a specific encoding that we need its protocol file to decode and encode its data. It is widely used in communication of mircroservices. We extract files of protocol buffer specification from executable file, and associate them with APIs provided by the executable programs. The file can be loaded in RockFuzz and try to use them to parse data sent or received by the related APIs.

Algorithm 3 Sequence based mutation.

```
1: function SEQUENCEBASEDMUTATE(seq_irs)  ▷ seq_irs: IR array for request sequence . return: mutated IR of requests.
2:     mutated_irs ← seq_irs.clone()
3:     i ← random select between 0 to seq_length
4:     if flip a coin then
5:         mutated_irs.remove(i)
6:     else
7:         mutated_irs.insert(i, seq_irs[i])
8:     end if
9:     return mutated_irs
10: end function
```

3.2.2 Sequence based mutation. Though a scenario has fixed order and kinds of requests, mutate the sequence in request-level may trigger new behaviors. Take Figure 3 as example, if we pay twice after trade creation, the services may enter a new state which is unexpected. We design two mutation operators for request-level mutation, as described in Algorithm 3. RockFuzz randomly select a request in the sequence, and skip this request during sending or send this request more than one time up to a coin flip.

3.2.3 Type based mutation. To generate syntactically correct requests, we perform type-based mutations on IR, including node insertion, deletion and replacement. Algorithm 4 shows the ideas. First, it puts all nodes in IRs of a sequence in a set, then random select nodes with weight based on their types. Instead of selecting nodes after requests selection, we directly select nodes in all the requests, which makes sure every node have fair opportunity to be chosen. Function *selectNodes* select N nodes configured by user. Only leaf nodes and intermediate nodes whose height (view it as subtree) is lower than a configured value will be selected. After that, we apply following mutation operators on these selected nodes based on their types.

- **Integer/Float/Double/Char**. If node's type can be calculated with arithmetic, we will use arithmetic operations to mutate it, such as add, subtraction. Also we will try to set some interesting values for them, e.g. -1 for integer.
- **Boolean/Enum**. The potential values of boolean and enum are in a finite set, thus we randomly choose a value in the set, then assign it as the node's value. The enum type inference and value sets got from lightweight static code analysis, which finds definition of enum type in programs and associate them with IR nodes.
- **String**. String is the most complicated types, since many types of content will be encoded as string type. As Section 3.2.1 described, we will infer their encoding methods, and decode them recursively. If they were decoded as an subtree, the types become collections or other composite structure, which will uses other operators or select their descendant nodes. For those values that can't be decoded into smaller granularity, we try to tokenize them, then mutate these tokens. Operators will randomly delete these tokens, or insert other tokens between them, or swap the order of tokens. Because the tokens in determined by they are words, symbols or digits, we can mutate them based on token's type. Digital tokens will be processed like integer type, but we need to encode them as string finally; symbol tokens is simply replace as other symbols; words tokens will be randomly insert, delete, or replace characters.
- **Map/List/Array/Structure**. Collection and structure types are composited data type, which can be viewed as subtrees. The operators will randomly insert nodes as their children, delete their children, or swap the order of their children (only for list and array).
- **All**. We apply two mutation operators for any types. First, we query corpus by node's key in database which collected from Section 3.1, and randomly select one of the corpus to replace node's value. Second, we not only mutate the values in nodes, but also their types. The operator updates the nodes by constructing new nodes whose type differ from the original nodes.

Algorithm 4 Type based mutation.

```
1: function TYPEBASEDMUTATE(seq_irs)  ▷ seq_irs: IR array for request sequence . return: mutated IR of requests.
2:     nodes ← ∅
3:     for each ir ∈ req_irs do
4:         nodes ← nodes ∪ ir.nodes
5:     end for
6:     select_nodes ← selectNodes(nodes)
7:     mutated_irs ← seq_irs.clone()  ▷ node-level clone
8:     for each node ∈ select_nodes do
9:         operator ← findOperators(node)  ▷ Random select operator for node based on its type
10:        mutated_node ← operator.mutate(node)
11:        mutated_irs.update(mutated_node)  ▷ If mutated_node has children, update their chilren recursively.
12:    end for
13:    return mutated_irs
14: end function
```

3.2.4 Pilot stage. Similar to most fuzzers, RockFuzz has a pilot stage for new added seeds, which is invoked only once for each seed. The stages will traverse every request and nodes, and adopt deterministic sequence based mutation and type-based mutation on them respectively. However,

the potential type-based mutation behaviors is too huge to traverse. Therefore, we limit the number of node to 1 during mutation, and provide a small set of deterministic mutation behaviors for type-based operators. For example, the operand used in arithmetic operators for integer will be configured to a small value. After pilot stage, RockFuzz enters havoc stages, which random select nodes, operators and operands.

3.2.5 Seed maintenance. Sequence inputs will be saved as seeds if they trigger new behaviors. The inputs include real world traffic and mutated sequences. However, the code and dynamic configurations (stored in database or memory) may changes. Thus, some seeds may not available after a period of time. We design a mechanism for keeping seeds available. This process runs every once in a while. First, we backup the seeds, and reset the database of seeds and coverage. Then, we send lots of fresh real world traffic and the backup seeds. If any of them find new coverage, we save it to the seed database.

3.3 Requests sending to microservices

Executing Inputs in microservice programs is totally different from traditional system software which most fuzzers focus on. In microservices, inputs is sent via communication protocol such as HTTP. Its fuzzer should be a client of services, too. The requests is not dependent, they should send as a sequence by a specific order. In additions, the APIs of requests may in different services and have different communication protocols. The client should associate the sequential requests, using a ID to group these requests and replacing arguments based prior requests. Algorithm 5 shows the algorithm. It initializes an empty context, and generates a unique ID for a sequence. For each request in the sequence, we use some defined rules to check and replace the arguments based on the context. We also generate a unique ID and assign fuzzing tag for a request, then we pack and encode the IR to an request that can be sent. RockFuzz will select corresponding clients of communication according to the protocol types. The response of requests will be converted to an IR, too. We check the arguments and get values that context need. The defined rules are some scripts that can determine if the sequence is success or not, and store values to context or put context's value to IR. Most of the rules are wrote by developers, and we also try to infer some of them by analyzing real world traffic. If IRs of request and response exist nodes have the same key and dynamic related, we generate a rule for storing and putting the node's value in context.

3.3.1 Reused test environment. Most of traditional fuzzing works set up a standalone environment for running inputs generated by fuzzers since these inputs may trigger crashes or pollute database. Fuzzing microservices also needs such an environment. However, deploying many services and middlewares needs a lot of cost. We reuses test environment which used in the development process. The test environment has its own middlewares and databases, so it won't pollute functions and data in production environment. Though reused test environment reduces the effort of delopying, but the environments are shared by other developers or testers. The behaviors triggered by other developers are viewed as noise, and we should eliminate them. As all requests sent by our fuzzers have a "FUZZ" tag, we can distinguish effects caused by fuzzer or others. Based on this, we can focus on analysing behaviors caused by fuzzers. Requests also bring some side effects. For example, other requests increases the connections and make the services unavailable since they are busy. We exclude errors caused by side effects in Section 3.5.

Algorithm 5 The process of sending requests to microservices, including pre and post process. Microservices are test environment used by developers which we reused it for fuzzing.

```
1:  function SENDSEQUENCE(seq_irs, pre_rules, post_rules)
    ▷ seq_irs: IR array for request sequence . rules: rules for pre and post processing. return: result of sending requests.
2:      context ← empty map
3:      seq_id ← generateUniqueId()
4:      for each req_ir ∈ seq_irs do
5:          pre_pule ← pre_rules.find(req_ir)
6:          status ← pre_rule.check(req_ir, context)
7:          if status is not SUCCESS then
8:              return status
9:          end if
10:         req_ir ← pre_rule.replace(req_ir, context)   ▷ Replace arguments inside request based on values in context.
11:         request ← encodeAndPackIR(req_ir)
12:         addFuzzTag(request)
13:         request_id ← generateUniqueId()
14:         assignTraceId(request, seq_Id, request_id)
15:         response ← sendToService(requst)
16:         resp_ir ← parseResponse(response)
17:         post_pule ← post_rules.find(req_ir)
18:         status ← post_pule.check(resp_ir)
19:         if status is not SUCCESS then
20:             return status
21:         end if
22:         resp_context ← post_pule.parseCtx(resp_ir)
23:         context ← context ∪ resp_context
24:     end for
25:     return SUCCESS
26: end function
```

3.4 Coverage collecting and analysis

A more precise and fast coverage metric can meke fuzzing more efficient [5, 23, 37, 54]. There are serveral different approaches to collect coverage information. Fuzzers instrument

Figure 6. The workflow of coverage collection and analysis.

code during compile time or runtime to collect coverage of each input. These instrumentation can only collect coverage in standalone application, and can not used in shared envioronment. Jacoco [49] is used to measure code coverage in microservice by merging collected information from different machine, but it can't distinguish which inputs trigger these code coverage, which is not fit to fuzzing.

We design a new mechanism of coverage collection and analysis for fuzzing, as Figure 6 shows. Both the collection and analysis steps are asynchronous.

3.4.1 Coverage Collection. The coverage collection step are run in services under test. We instrument code at the beginning of each basic blocks in applications. Once a request is invoked in a service, the instrumentation records the request's ID(req_id) and which blocks it vitisted(bb). We construct a cache map in applications's runtime, whose key and value are req_id and $unit_covov$ respectively. $unit_covov$ is a set of bb. Visited blocks with the same req_id will be merged by putting the (req_id, bb) to the cache map. Because the handling of requests may create new threads and run asynchronous, determining whenever it finish is hard. We assume the process will finish in a fixed time such as 1 second, and set the cache with timed eviction. The expired entries $(req\_id, unit\_cov)$ will be sent to a distributed computing system.

3.4.2 Coverage Analysis. We analsyse coverage at distributed computing system, which deployed independent with fuzzing engine. These temperal coverage data are stored in a time series database. Since they may be generated from different services, we need to merge them by their req_id again. Similar to aggregation in application's service, we use a cache map to merge $unit\_cov$, and get a req_cov finnally, which includes all visited blocks accross multiple services. We also maintain a global coverage map in a persistent database. Request's coverage will be compared to global coverage map. If a request triggers any new basic block, we send a guidance to fuzzing engine and update global coverage map.

3.4.3 Coverage Guidance. The guidance includes req_id. It can be used to find out which sequence discover the new coverage since we have held all sent sequences in a time series database (**??**). The sequence with the detailed arguments of requests will be saved in a persistent database of seed inputs. Accroding to genetic algorithm, these new saved sequences wiil be preferentially seleceted as seeds for mutation, guiding the mutating to more likely to trigger vulnerabilities or new branches.

3.5 Test Oracles Checking

Fuzzers for system software use crashes or sanitizer tools as test oracle for checking vulnerabilities in applications. Existed web service fuzzers validate the response content. For example, RESTler [8] checks 500 error code in HTTP responses. Errors that triggered inside may be ignored and processs silently, and return successful responses finally. Thus only cosidering response in microservices is not enough. Besides checking responses, we also checking logging contents and verify numerical values in calculation.

3.5.1 Response Validation. There are multiple requests in microservice communication even we only send one request. We instrument code in every service in microservices to collect all responses, then we check them by following rules.

- Protocol errors rule. For different protocols, they have their own specification for errors. HTTP and MQTT will return error codes, e.g 500 error code in HTTP. RPC protocol may throw exceptions if they meet errors, so we should capture them during invoking. RockFuzz classifies these error pattern systematically from specifications, and use them to detect errors.
- Custom error code rule. Some deverlopers perfer to encode errors in request's body and define their own custom error codes, e.g. an "error" field in response body encoded by JSON. We implement common rules to analyse the error message in response's content, and allow developers to add their own definition of error code in our platform.
- Timeout rule. We use the processing time of requests to detect availability vulnerabilities. If the processing time of a request is slow than a threshold value after mutation, it may trigger an availability vulnerabilitiy potentially.

3.5.2 Error Logging Capture. Web services, especially microservices, have vast amout of log data. Developers use these log for debugging or error analysis. Every logging message has a log level that indicates how important a given log message is. Among them, "ERROR" and "FATAL" level designate error events happen in the application. We instrument code to collect these high severity logging at runtime and associate them with the ID of request. Finnally, we can find whether a reqeust prints error logs or not.

3.5.3 Numerical Consistency Verification. Microservices is one way to do distributed computing, which may lead to inconsistent data caused by misuse of lock, incorrect use of parameters, or uncontrollable factors, and so on. These wrong data will be used to computing and get a wrong result, and pollute the database finally. Take **??** as an example. Both "Pay" and "Risk" services read balance from "Account", there may exist inconsistent data if they "Pay" update the balance before "Risk" query the data. Moreover, if the *queryDiscount* is failed since network problem, "Cashier" may suppose the *discount* is 0 and continue to deal with the request, which bring losses to users.

However, detecting such logic bugs automatically is difficult since it need significant amount of effort and expertise for writing rules that checking consistency. We alleviates the work for writing these rules by providing unified methods for data collection and verification.

1. Collect data from microservices. We instrument code to collect data across multiple services, includes the content of communication between services(e.g RPC and message) and middlewares (e.g. database). We model all these content as IR defiened in Section 3.2.1 to make the expert users identify specific fields easier. In addition, we adding the reqeust's ID during collection.
2. Associate data that have consistent relationship. Experts define rules to find which data are related. For example, requests are related if they have same *buyerId* within a certain time window after pay request was invoked. These rules will be used in a distributed computing system for processing data.
3. Check related data by dicision rules. Decision rules are also defined by experts, which use to determine if those related data is in a consistent state or not. For example, the rule can check if the values of balance in "Pay" and "Risk" are the same or not. Also, we can define rules to check if data satisfy certain properties in specific conditions. For example, if the risk checking is failed, the buyer's balance should always the same and can't be reduced.

Besides collecting steps, all other steps are ran aside microservices under test like **??**. The checkings will return "TRUE" or "FALSE" for every rules. If any rule return "FALSE", we generate a report with the cause of error and the request's ID that triggered it.

4 Implementation

4.1 Instrumentation

Instrumentation: java agent

4.2 FaaS

stack: JAVA, microservice support protocol: rpc, http(Restful), mosn

5 Evaluation

1. Experiment Setup 2. Metrics

5.1 Real world traffic analysis result

5.2 Success Rate

5.3 Type and Operation distribution

5.4 Coverage Improvement

5.5 Vulnerabilities

5.5.1 Exception.

5.5.2 Error Code.

5.5.3 Timeout.

5.5.4 Numerical verification.

5.6 Case Study

5.7 Speed

5.8 Noise and false negatives

6 Conclusion

References

[1] Saswat Anand, Corina S Păsăreanu, and Willem Visser. 2007. JPF–SE: A symbolic execution extension to java pathfinder. In *International conference on tools and algorithms for the construction and analysis of systems*. Springer, 134–138.

[2] Andrea Arcuri. 2017. RESTful API automated test case generation. In *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 9–20.

[3] Andrea Arcuri. 2019. RESTful API automated test case generation with EvoMaster. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 28, 1 (2019), 1–37.

[4] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. 2019. NAUTILUS: Fishing for Deep Bugs with Grammars.. In *NDSS*.

[5] Cornelius Aschermann, Sergej Schumilo, Ali Abbasi, and Thorsten Holz. 2020. Ijon: Exploring deep state spaces via fuzzing. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1597–1612.

[6] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. 2019. REDQUEEN: Fuzzing with Input-to-State Correspondence. (2019).

[7] Vaggelis Atlidakis, Roxana Geambasu, Patrice Godefroid, Marina Polishchuk, and Baishakhi Ray. 2020. Pythia: grammar-based fuzzing of REST APIs with coverage-guided feedback and learning-based mutations. *arXiv preprint arXiv:2005.11498* (2020).

[8] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. 2019. Restler: Stateful rest api fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 748–758.

[9] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. 2020. Checking Security Properties of Cloud Service REST APIs. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 387–397.

[10] William Blair, Andrea Mambretti, Sajjad Arshad, Michael Weissbacher, William Robertson, Engin Kirda, and Manuel Egele. 2020. HotFuzz: Discovering Algorithmic Denial-of-Service Vulnerabilities Through Guided Micro-Fuzzing. *arXiv preprint arXiv:2002.03416* (2020).

[11] Tim Blazytko, Matt Bishop, Cornelius Aschermann, Justin Cappos, Moritz Schlögel, Nadia Korshun, Ali Abbasi, Marco Schweighauser, Sebastian Schinzel, Sergej Schumilo, et al. 2019. {GRIMOIRE}: Synthesizing structure while fuzzing. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*. 1985–2002.

[12] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2329–2344.

[13] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. 2018. Hawkeye: towards a desired directed grey-box fuzzer. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2095–2108.

[14] Peng Chen and Hao Chen. 2018. Angora: efficient fuzzing by principled search. In *IEEE Symposium on Security and Privacy (SP)* (2018-05-21/2018-05-23). San Francisco, CA.

[15] Peng Chen, Jianzhong Liu, and Hao Chen. [n.d.]. Matryoshka: fuzzing deeply nested branches. In *ACM Conference on Computer and Communications Security (CCS)* (2019-11-11/2019-11-15). London, UK.

[16] Paolo Di Francesco, Ivano Malavolta, and Patricia Lago. 2017. Research on architecting microservices: trends, focus, and potential for industrial adoption. In *2017 IEEE International Conference on Software Architecture (ICSA)*. IEEE, 21–30.

[17] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. 2017. Microservices: yesterday, today, and tomorrow. *Present and ulterior software engineering* (2017), 195–216.

[18] Christian Esposito, Aniello Castiglione, and Kim-Kwang Raymond Choo. 2016. Challenges in delivering software in the cloud as microservices. *IEEE Cloud Computing* 3, 5 (2016), 10–14.

[19] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++: Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association.

[20] Gordon Fraser and Andrea Arcuri. 2011. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 416–419.

[21] Gordon Fraser and Andrea Arcuri. 2014. A large-scale evaluation of automated unit test generation using evosuite. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 24, 2 (2014), 1–42.

[22] Shuitao Gan, Chao Zhang, Peng Chen, Bodong Zhao, Xiaojun Qin, Dong Wu, and Zuoning Chen. 2020. GREYONE: Data Flow Sensitive Fuzzing. In *29th USENIX Security Symposium (USENIX Security 20)*. Boston, MA.

[23] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. 2018. Collafl: Path sensitive fuzzing. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 679–696.

[24] Patrice Godefroid, Bo-Yuan Huang, and Marina Polishchuk. 2020. Intelligent REST API data fuzzing. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 725–736.

[25] Patrice Godefroid, Adam Kiezun, and Michael Y Levin. 2008. Grammar-based whitebox fuzzing. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 206–215.

[26] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: directed automated random testing. In *ACM SIGPLAN Notices*, Vol. 40. 213–223.

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

[28] Patrice Godefroid, Michael Y Levin, and David A Molnar. 2008. Automated whitebox fuzz testing.. In *NDSS*, Vol. 8. 151–166.

[29] Patrice Godefroid, Hila Peleg, and Rishabh Singh. 2017. Learn&fuzz: Machine learning for input fuzzing. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 50–59.

[30] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with code fragments. In *21st {USENIX} Security Symposium ({USENIX} Security 12)*. 445–458.

[31] Karthick Jayaraman, David Harvison, Vijay Ganesh, and Adam Kiezun. 2009. jFuzz: A Concolic Whitebox Fuzzer for Java.. In *NASA Formal Methods*. 121–125.

[32] Stefan Karlsson, Adnan Čaušević, and Daniel Sundmark. 2020. QuickREST: Property-based Test Generation of OpenAPI-Described RESTful APIs. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 131–141.

[33] Rody Kersten, Kasper Luckow, and Corina S Păsăreanu. 2017. POSTER: AFL-based Fuzzing for Java with Kelinci. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2511–2513.

[34] Suyoung Lee, HyungSeok Han, Sang Kil Cha, and Sooel Son. 2020. Montage: A Neural Network Language Model-Guided JavaScript Engine Fuzzer. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*. 2613–2630.

[35] LLVM. 2021. libFuzzer – a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html

[36] Irakli Nadareishvili, Ronnie Mitra, Matt McLarty, and Mike Amundsen. 2016. *Microservice architecture: aligning principles, practices, and culture*. " O'Reilly Media, Inc.".

[37] Stefan Nagy and Matthew Hicks. 2019. Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 787–802.

[38] Sam Newman. 2019. *Monolith to microservices: evolutionary patterns to transform your monolith*. O'Reilly Media.

[39] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2020. Parmesan: Sanitizer-guided greybox fuzzing. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*. 2289–2306.

[40] Rohan Padhye, Caroline Lemieux, and Koushik Sen. 2019. JQF: coverage-guided property-based testing in Java. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 398–401.

[41] Theofilos Petsios, Jason Zhao, Angelos D Keromytis, and Suman Jana. 2017. Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2155–2168.

[42] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. 2020. AFLNet: a greybox fuzzer for network protocols. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 460–465.

[43] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing.. In *NDSS*, Vol. 17. 1–14.

[44] Yuyang Rong, Peng Chen, and Hao Chen. [n.d.]. Integrity: Finding Integer Errors by Targeted Fuzzing. In *International Conference on Security and Privacy in Communication Networks (SecureComm)* (2020-10-21/2020-10-23).

[45] WhiteHat Security. 2018. 2018 Application Security Statistics Report. https://info.whitehatsec.com/Content-2018StatsReport_LP.html?utm_source=website&utm_medium=Website-Content-Ongoing-2018StatsReport

[46] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. 2018. NEUZZ: Efficient Fuzzing with Neural Program Learning. *arXiv preprint arXiv:1807.05620* (2018).

[47] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: augmenting fuzzing through selective symbolic execution. In *Proceedings of the Network and Distributed System Security Symposium*.

[48] Yuqiong Sun, Susanta Nanda, and Trent Jaeger. 2015. Security-as-a-service for microservices-based cloud applications. In *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 50–57.

[49] EclEmma team. 2021. JaCoCo Java Code Coverage Library. https://www.jacoco.org/jacoco/

[50] Spandan Veggalam, Sanjay Rawat, Istvan Haller, and Herbert Bos. 2016. IFuzzer: An evolutionary interpreter fuzzer using genetic programming. In *European Symposium on Research in Computer Security*. Springer, 581–601.

[51] Emanuele Viglianisi, Michael Dallago, and Mariano Ceccato. 2020. RestTestGen: automated black-box testing of RESTful APIs. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 142–152.

[52] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2017. Skyfire: Data-driven seed generation for fuzzing. In *Security and Privacy (SP), 2017 IEEE Symposium on*. IEEE, 579–594.

[53] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2019. Superion: Grammar-aware greybox fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 724–735.

[54] Jinghan Wang, Yue Duan, Wei Song, Heng Yin, and Chengyu Song. 2019. Be sensitive and collaborative: Analyzing impact of coverage metrics in greybox fuzzing. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses ({RAID} 2019)*. 1–15.

[55] Mingzhe Wang, Zhiyong Wu, Xinyi Xu, Jie Liang, Chijin Zhou, Huafeng Zhang, and Yu Jiang. 2021. Industry Practice of Coverage-Guided Enterprise-Level DBMS Fuzzing. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 328–337.

[56] Cheng Wen, Haijun Wang, Yuekang Li, Shengchao Qin, Yang Liu, Zhiwu Xu, Hongxu Chen, Xiaofei Xie, Geguang Pu, and Ting Liu. 2020. Memlock: Memory usage guided fuzzing. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 765–777.

[57] Li Wu, Johan Tordsson, Erik Elmroth, and Odej Kao. 2020. Microrca: Root cause localization of performance issues in microservices. In *NOMS 2020-2020 IEEE/IFIP Network Operations and Management Symposium*. IEEE, 1–9.

[58] Tetiana Yarygina and Anya Helene Bagge. 2018. Overcoming security challenges in microservice architectures. In *2018 IEEE Symposium on Service-Oriented System Engineering (SOSE)*. IEEE, 11–20.

[59] Wei You, Xueqiang Wang, Shiqing Ma, Jianjun Huang, Xiangyu Zhang, XiaoFeng Wang, and Bin Liang. 2019. ProFuzzer: On-the-fly Input Type Probing for Better Zero-Day Vulnerability Discovery. In *IEEE Symposium on Security and Privacy (SP)*.

[60] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. {QSYM}: A practical concolic execution engine tailored for hybrid fuzzing. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 745–761.

[61] Michal Zalewski. 2021. American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/

[62] Rui Zhong, Yongheng Chen, Hong Hu, Hangfan Zhang, Wenke Lee, and Dinghao Wu. 2020. SQUIRREL: Testing Database Management Systems with Language Validity and Coverage Feedback. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 955–970.