2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE) | 979-8-3503-2996-4/23/$31.00 ©2023 IEEE | DOI: 10.1109/ASE56229.2023.00036

Zero-Config Fuzzing for Microservices

Wei Wang
Google Inc.
wwweiwang@google.com

Andrei Benea
Google Inc.
abenea@google.com

Franjo Ivančić
Google Inc.
ivancic@google.com

***Abstract*—The microservice paradigm is a popular software development pattern that breaks down a large application into smaller, independent services. While this approach offers several advantages, such as scalability, agility, and flexibility, it also introduces new security challenges. This paper presents a novel approach to securing microservice architectures using fuzz testing. Fuzz testing is known to find programming errors in software by feeding it with unexpected or random inputs. In this paper, we propose a zero-config fuzz test generation technique for microservices that can maximize coverage of internal states by mutating both the incoming requests and the backend responses from dependent services. We successfully deployed our technique to over 95% of C++ services built on Google's internal microservice platform. It reported and got fixed thousands of errors in real-world microservice applications.**

***Index Terms*—fuzz testing, automation, microservice**

I. Introduction

The microservice paradigm is a software design approach that structures an application as a suite of small, independent services. Each service is self-contained and performs a single business function. Services communicate with each other using well-defined APIs. Microservice architectures have several advantages over traditional monolithic architectures, including flexibility, scalability and agility [1]. However, they also have some security and reliability risks. One of the security risks is that they have more potential entry points (e.g. microservice APIs) for attackers than monolithic architectures [2]. If an attacker can find a vulnerability in one API, they can use it to gain access to the microservice behind the API. Once they have access to one microservice, they may access other microservices and even take the entire application down.

Fuzz testing is a valuable addition to existing testing techniques. It works by generating a large number of randomly generated test inputs and feeding them to the software under test. This can find programming errors such as buffer overflows and memory leaks. Developers rarely write tests to catch these types of errors, which are exactly the kinds of issues that lead to security vulnerabilities and reliability problems. Fuzz testing has been widely adopted in real-world applications, such as general libraries [3], Internet of Things (IoT) devices [4], firmware [5], kernels [6], and smart contracts [7]. However, it has not been widely used at the service level. This is likely due to the complexity of monolithic services, which can make it difficult to generate effective fuzz test inputs. Furthermore, developers need to worry about potential side-effects of fuzzing some services under test on other production backends that are not safe to be fuzzed.

Microservices, on the other hand, are designed to be more modular and easier to test. This makes them more amenable to fuzz testing. We have successfully integrated fuzz testing into Google's microservice development framework. This new integration automatically generates fuzz tests for all C++ microservices, without any user intervention. Fuzz tests, also known as fuzz drivers, are pieces of code that are used to exercise target software by providing it with inputs. In most cases (if not all), writing fuzz tests is still a primarily manual process, which is a major hindrance to the widespread adoption of fuzzing. There are a few efforts to automate the process, such as FUDGE [8], FuzzGen [9], and other tools [10]–[12]. These tools try to learn the patterns of existing code bases and use them to generate fuzz tests automatically. However, it still requires humans to review and validate the generated fuzz tests before they can be used.

Our microservice fuzz test generation does not require human intervention. It uses the well-defined APIs of microservices to generate ready-to-run tests. First, the fuzz test randomly selects an exposed API. Next, it uses structure-aware fuzzing [13] to feed the API with coverage-guided randomly generated inputs that match the corresponding argument types. Then, it mocks out all the backend microservice components and injects coverage-guided mutated responses or errors. We call this *backend fuzzing*: It provides a hermetic test environment, and it also tests the resilience of the target microservice against faults or malicious behavior in dependent components. As a result, thousands of services are fuzzed continuously, and thousands of bugs have been reported to developers.

II. gRPC-based Microservice Architecture

A gRPC-based microservice architecture is an architectural style that uses gRPC [14] to connect microservices. gRPC is an open-source RPC framework that is built on top of HTTP/2. It supports load balancing, tracing, health checking and authentication that make it well-suited for microservices architectures.

Let us say you are building a video streaming application. The application would have a number of features, such as video streaming, user authentication and payment processing. In a gRPC based microservice architecture, each feature would be implemented in a separate gRPC service and can also talk to each other.

When a user wants to watch a video, the client first sends a request to the video streaming service. The video streaming service then authenticates the user by sending a request to the user authentication service. The user authentication service validates the user's credentials and returns a token to the video streaming service. The video streaming service then sends the token to the payment processing service to process the payment. The payment processing service authorizes the payment and returns a confirmation message to the video streaming service. The video streaming service then looks up the video in the database and returns a stream of video data to the user.

The following is a code snippet that defines three services with Protocol Buffers (a.k.a. Protobuf) [15]. Protocol Buffers are a language-neutral, platform-neutral, extensible mechanism for serializing structured data. They are often used to define the interface of microservices. Each service is defined as a collection of service methods that can be called remotely with their parameters and return types. It has a client side and a server side, as shown in Fig. 1. The client side is responsible for making requests to the server and receiving responses, acting as an interface. The server side is responsible for handling requests from the client and returning responses, by implementing the provided service methods.

```
service PaymentProcessing {
 rpc Process (ProcessPaymentRequest) returns (
      ProcessPaymentResponse);
}
service UserAuthentication {
 rpc Authenticate (AuthenticateUserRequest) returns
        (AuthenticateUserResponse);
}
service VideoStreaming {
 rpc GetVideo (GetVideoRequest) returns (
      GetVideoResponse);
 rpc ListVideos (ListVideosRequest) returns (
      ListVideosResponse);
}
```

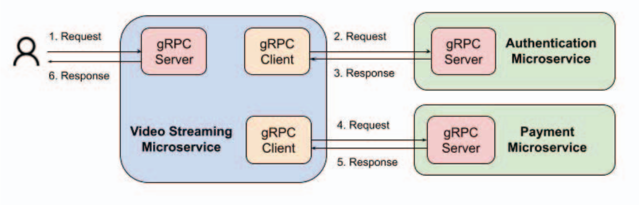

Fig. 1. Video Streaming Application.

III. Automatic Fuzz Test Generation for Microservices

The fuzz test generation framework for a microservice consists of two parts:

1) Generating the test harness. The test harness is a piece of code that starts up the service under test and invokes the service methods with corresponding arguments.
2) Generating the specification of the test input. It specifies the service method and its arguments.

In the described solution, both parts are fully automatic, requiring no manual intervention.

A. Input Data Specification

Given a Protobuf definition of a microservice, we can dynamically generate an input data specification also in a Protobuf message at runtime. The dynamically generated message represents both the service methods to invoke and their arguments.

For example, consider the Protobuf definition of service `VideoStreaming`. It has two methods, both of which could be used to attack the service. We could fuzz each method separately but this would not be efficient. And this approach could miss bugs that are triggered by a sequence of mixed method calls. Instead, we generate a Protobuf specification that allows us to perform a sequence of calls with their corresponding arguments. Here is our generated specification:

```
message RpcCall {
 oneof rpc_call {
   GetVideoRequest get_video = 1;
   ListVideosRequest list_videos = 2;
 }
}
message FuzzSession {
 repeated RpcCall rpc_calls = 1;
}
```

The `FuzzSession` message specifies the input data for fuzzing. It contains a repeated field `rpc_calls` with type `RpcCall`. In the definition of `RpcCall`, we use the keyword `oneof` to specify that only one of the `RpcCall` fields can be used at a time. The name of each `RpcCall` field is encoded from the name of a method defined in the service, and the field type is the method's argument type.

Based on the structure of the `FuzzSession` message, we can send a sequence of calls to the service under test using libprotobuf-mutator [16]. For each call, we can specify the service method that we want to invoke and the input that we want to pass to the method. This approach can be used to find bugs that would not be found by fuzzing each method separately. For example, if a service has a bug that is only triggered when two methods are called in a specific order, then fuzzing can find it by generating an input message with the two methods in that order.

B. Test Harness

Typically, the microservice platform standardizes the process of starting up a service. This process usually involves a few key steps: loading the configuration files, initializing dependencies, and launching the main loop. Once the service is up and running, we load the corresponding method for each `RpcCall` field of the input `FuzzSession` message, then we invoke the method with the argument specified by that `RpcCall` field. Google's internal microservice platform allows us to dynamically generate all the related code for any service.

IV. Backend Fuzzing

In the previous section, we presented a method for fuzzing a service by invoking its methods in a randomized order with mutated arguments. This approach, in practice, has many difficulties and risks.

a) In-process Fuzzing of Microservices: In-process fuzzing and out-of-process fuzzing are two types of fuzzing. In-process fuzzing executes the fuzz engine in the same process as the target program, while out-of-process fuzzing executes the fuzz engine in a separate process. In general, in-process fuzzing is more efficient than out-of-process fuzzing because it can collect coverage feedback from the target program more quickly. This feedback can then be used to guide the fuzz engine in generating new inputs that are more likely to find bugs. However, in-process fuzzing does not fit the microservice platform very well where different services are running in different processes. Once running with one service under test, the fuzz engine would not be able to collect coverage feedback from its backend services.

b) Fuzzing Performance of Microservices: Execution speed is the number of times a fuzz engine can execute a program per second [17]. It is a crucial factor in measuring fuzzing performance. The faster the execution speed, the faster a fuzz test can generate new test inputs and cover new code paths. The complexity of the program being fuzzed can affect the execution speed. In general, services are considered more complex than typical library-level fuzz targets. This is because services are typically larger and have more functionality. Microservices, on the other hand, are designed to be smaller and more focused. This makes them less complex and can improve the execution speed of fuzzing. However, the communication between microservices can add latency and overhead overall. Additionally, communication can fail for a variety of reasons, such as service outages or network outages. All of these factors can slow down fuzzing.

c) Non-deterministic Behavior of Backend Services: The interaction between two or more microservices and their backends, such as storage layers for example, can introduce non-deterministic behaviors. This means that the same request can sometimes produce different responses, depending on the state of the service that processes the request. For example, consider microservice A that sends a request to another microservice B. If B is overloaded or unavailable at the time, A may receive B's response after a much longer period of time or may not receive anything at all. This type of non-deterministic behavior can make it difficult to reproduce bugs.

d) Potential Risk to Backend Services: When fuzzing a microservice, it is important to ensure that it is talking to a *test instance* of backend services, not the *production instance*. This is because we do not want the mutated test inputs to cause any outages in any production system not under test. However, whether a microservice talks to the test instance or the production instance depends on how the developers configure the connections. If the connections are not configured correctly, it is possible that fuzzing a service may cause potential risk to its backend services.

To address these risks and challenges, we propose a new approach called *backend fuzzing*. It uses an intercepting channel to intercept an RPC and inject a mutated response or an error status from the backend service. This allows fuzzing to run in a hermetic mode, which means that it is isolated from the production environment. Backend fuzzing does not make any real connection to backend services, so it can largely reduce the overhead of network communication and the non-deterministic backend behaviors. In this sense, backend fuzzing is more reliable and efficient. With the coverage feedback of the service under test only, backend fuzzing can generate and inject mutated responses and error statuses from the backend services. This allows backend fuzzing to fully exercise the code handling backend responses and thus is more likely to explore new code paths and find more bugs that can be triggered by the interaction with backend services.

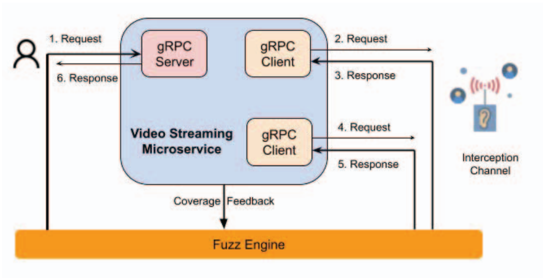

Fig. 2. Video Streaming Application in Backend Fuzzing Mode.

Let us take another look at the video streaming service in backend fuzzing mode, as shown in Fig. 2. In this mode, the interaction between the video streaming service and its backend services (e.g. the user authentication and payment processing services) is cut off by the intercepting channel. The fuzz engine can generate the test input (e.g. `FuzzSession` message) that includes both incoming requests from users and backend responses, by using the coverage feedback solely from the video streaming service.

```
message ClientRpcCall {
  repeated ProcessPaymentResponse
       paymentprocessing_process_response = 1;
  repeated AuthenticateUserResponse
       userauthentication_authenticate_response = 2;
}
message FuzzSession {
  repeated RpcCall rpc_calls = 1;
  optional ClientRpcCall client_rpc_call = 2;
}
```

The `FuzzSession` message is updated as above to include a new field called `ClientRpcCall` to specify backend responses. This field enumerates all the methods provided by backend services and encodes them as different subfields. The type of each subfield is the method's return type, and the name is encoded from the name of the method and the corresponding backend service. Each subfield is tagged as a `repeated` field in order to match multiple invocation of the same method. During fuzzing, once a backend service method is invoked, the intercepting channel will lookup `ClientRpcCall` for the corresponding subfield and return the next *unused* response. If there is no unused one left, it simply returns an empty response. We also inject an error status as another response type for each subfield. However, we will not discuss this here due to space constraints.

Using Google's internal microservice platform, we can dynamically locate all the backend services for a given microservice and generate the updated `FuzzSession` message and the related test harness code. This allows us to focus on the microservice's code without having to worry about the behavior of its backend services.

Backend fuzzing is an effective way to find bugs in services triggered by mutated backend responses. However, it can also generate results that users may perceive as *false positives*. This is because the service under test may have expectations on the backend responses that are not explicitly validated.

For example, a video streaming service may expect the user authentication service to always return a non-empty token if the user's credentials are valid. However, with backend fuzzing, it may receive an empty token. If the service tried to read or write this token string using a fixed offset, then a stack buffer overflow would occur. Or, if the service has a check in the code that the token is non-empty, then a check failure would occur.

In this case, we would report the stack buffer overflow to developers, not the assertion failure. This is because stack buffer overflows can be exploited by attackers to gain control of the program, while assertion failures typically only cause the program to terminate immediately. In addition, assertion failures are common in the code to check for unexpected conditions, and they often occur during fuzzing, so we choose to filter them out. We also do not report bugs for timeouts since they often happen during the service initialization and, in most cases, timeouts are transient issues and often are not indicative of a serious problem.

V. Industry Experience

We successfully deployed our technology to over 95% of C++ services built on Google's internal microservice platform without any developer intervention. Any newly added C++ service will automatically receive a fuzz test. In this section, we share some evaluation statistics and several lessons learned from our experience deploying continuous fuzz testing for our service platforms.

A. Fuzzing Statistics

Fuzzing engines like libFuzzer [18] explore a large search space by randomly mutating inputs. To be effective, they need to run as fast as possible to cover as many code paths as possible [19].

For most services, we observed that the generated fuzz tests can handle hundreds of inputs per second. This is generally slower when compared to traditional unit-level fuzz tests, but still fast enough for effective coverage-guided fuzzing at the service-level. However, to further improve the efficacy, we took a few actions:

- Statically initialize a global instance of the service under test. This will allow the fuzz test to exercise all inputs without having to perform setup and teardown on each input.
- With backend fuzzing, the speed can be further improved by not connecting to real backends.
- Limit the instrumentation scope to service related code rather than some frequently used base libraries that are often already covered by unit-level fuzz tests.

B. Bug Statistics

As of today, the automatically generated fuzz tests have reported *thousands of bugs* among more than 1K microservices, with a fix rate of 67.71%. We list the top ten bug types that were most triggered by fuzzing in Table I and their fix rates.

TABLE I
Bug Statistics of Auto-generated Service Fuzz Tests

Bug Type	Proportion	Fix Rate
Integer-overflow	27.1%	54.62%
Heap-use-after-free	11.2%	82.73%
Float-cast-overflow	10.0%	48.87%
Stack-overflow	8.9%	49.49%
Segfault	8.6%	84.97%
Null-dereference	6.8%	67.76%
Stack-use-after-return	6.6%	93.19%
Heap-buffer-overflow	5.5%	85.24%
Stack-use-after-scope	2.1%	74.46%
Stack-buffer-overflow	1.8%	77.50%

Initially, we were wondering how unsuspecting software developers would react to bugs filed from zero-configuration fuzzing setups that they may not have even been aware of. Our prior experience with continuous fuzzing was only with respect to fuzz tests either written or at least reviewed by a software engineering team member. Since they wrote or accepted the fuzz test, there is an explicit request for fuzzing and thus an understanding that the teams will resolve reported findings on such tests. However, we are observing that software engineering teams are resolving reported issues for these fully automated fuzzing setups at an approximately equivalent velocity and rate compared to the manually contributed fuzz tests. This may partially be due to the fact that service-level bugs have high fidelity and acceptance as realistic issues, since they are often easy to replicate via simple RPC calls.

It is also important to note that there is no indication that users tend to ignore the bugs triggered by backend fuzzing. In fact, as long as the bugs are reproducible, developers are convinced that these are real bugs. Of the 32.29% of bugs that have not yet been fixed, we observed the following reasons:

- Memory bugs caused by race conditions can be challenging to reproduce.
- Some bugs may be considered low priority or not worth fixing, depending on the severity of the bugs and the type of the services (e.g. whether they handle untrusted external requests or not).
- Developers sometimes argue that the inputs generated by fuzzing are not realistic and will not occur in production. However, we argue that if there are any expected invariants of the inputs, it is best practice to implement

a validation process to filter out invalid inputs before processing them.

Overall, service-owners are entrusted to decide which bugs to prioritize fixing, given that they understand their use cases. For example, some services have to process external requests, while others only receive internal requests. These could be further classified by whether these requests could originate from internal humans or just from automated production accounts. Some backend services may only process requests from another frontend service owned by the same team, and may thus have a very high trust expectation. The fuzzing infrastructure described here does not yet distinguish these different usage models and trust boundaries.

C. *Improve the Readability of Test Inputs*

When receiving a bug reported by fuzzing, developers often struggle to identify the root cause by just looking at the test input or the generated crashing stack trace. This is because test inputs generated by randomized mutations can be extremely difficult to read.

To address the issue of large test inputs, we simplified the generated fuzz tests. We did this by reducing the size of the test input, setting a size limit to the sequence of RPC calls that can be sent to the service under test. This is because we found that in most cases, the reported bug is triggered by the last one or two RPC calls. The prior RPC calls are redundant. This change may prevent fuzz tests from identifying bugs related to state changes among a large number of RPC calls. However, we believe that the improved readability of the test inputs is worth the trade-off. We are also working on techniques to analyze the data flow dependency among RPC calls and generate more sensible sequences.

It is important to note that a large amount of repeated calls to the service under test during fuzzing can accumulate state changes that are able to cause crashes with stack traces to inspect. However, in such scenarios, the reproducibility of the issue is somewhat hampered.

D. *Concurrency and Heap-use-after-free*

Heap-use-after-free bugs are the most difficult to reproduce. This kind of bugs occurs when a program tries to access memory that has already been freed. It can lead to undefined behavior. To reproduce this type of bug, we need to find the exact sequence of memory access that led to the error. This is particularly difficult in services where multiple threads are running in a concurrent way. Each time the program is run, the threads may access memory in different ways, making it difficult to reproduce. We have found that running the fuzz test with the same input N times can help to increase the variety of thread schedules, but it does not always reproduce the problem.

E. *Human Written Fuzz Tests vs. Auto-Generated Fuzz Tests*

We still recommend that developers write their own fuzz tests. To make this task easier, Google has open-sourced FuzzTest [20], a framework that simplifies the process by allowing developers to write fuzzer-executed property-based tests within a standard unit testing framework [21]. Half of the tens of thousands of fuzz tests that we have are human-written, and the other half are auto-generated for the microservice platform. Since its launch, auto-generated fuzz tests have contributed to the discovery of 33% of bugs. This is a significant contribution. While human-written fuzz tests are typically more targeted at specific libraries, auto-generated fuzz tests cover a wider range of code. This combination of approaches allows us to find a wider range of bugs and to improve the security in general.

VI. Related Work

Our work focuses on the automatic generation of fuzz tests for microservice platforms. The idea of automatic tests generation is not new, and it has been previously explored in the areas of unit test generation and library-level fuzz testing. However, to the best of our knowledge, our work is the first to demonstrate the applicability of such technology to a large microservice platform without any human intervention.

A. *Automated Test Generation for RESTful APIs*

RESTful APIs are web APIs that follow the REST architectural style [22]. REST stands for Representational State Transfer. RESTful APIs are built on top of the HTTP protocol and often use JSON or XML as the message format. gRPC is another type of interservice communication protocol. It is built on top of HTTP/2 and uses Protocol Buffer as the message format. Techniques have been proposed to automate the generation of test cases for REST APIs [23]. However, most of these techniques use purely randomized testing, which can be ineffective without code coverage feedback of the service under test. EvoMaster [24] is a tool that generates high-coverage test cases for REST APIs with code coverage feedback of the target service. However, it has the testing harness and the SUT running in different processes. This can add overhead to the runtime cost. Our approach is to generate fuzz tests that run in the same process as the SUT. This avoids the communication overhead to an external process. While EvoMaster's focus is on system/integration level testing, our focus is a single service. We can achieve high code coverage of the target service by mocking out the backend RPC services and mutating their responses.

B. *Autonomous Testing of Services*

To improve the quality of software, an autonomous testing extension has been developed that automatically generates random test inputs to exercise the service under test. The extension claims to be able to generate tests for a wider range of code paths than manual testing and find bugs that would otherwise go undetected [25].

This technology focuses on integration testing, while our focus is unit testing. It requires developers to define the test environment and create the test harness. It also aims for a zero-config setup, but this is difficult to achieve for all services without mature microservice platforms. It uses unguided fuzzing, which randomly chooses appropriate values for each data type. Unlike our coverage-guided fuzzing, this approach does not provide feedback of the executed code.

C. Automatic Fuzz Tests Generation for Libraries

Automatic fuzz tests generation is still a relatively new research area and there are a couple of recent efforts on automated fuzz tests generation for complex libraries. FUDGE [8] is a tool that generates fuzz tests by scanning the entire codebase for client code of target libraries. It then automatically synthesizes test candidates based on a single client code and ranks the candidates with their runtime statistics. However, FUDGE still requires a human to choose a suitable fuzz test from the candidates. FuzzGen [9] is another tool to generate fuzz tests based on existing codebase. It merges multiple client code of the target libraries and synthesizes fuzz tests that are more complex than those generated by FUDGE. However, FuzzGen can produce over-approximate results and thus false positives. It still requires manual verification of the generated fuzz tests by double checking all the API sequences. In this sense, both approaches require a human in the loop.

VII. Conclusion

In this paper, we presented a zero-config fuzz test generation approach for microservices. This technology can achieve automated fuzzing by feeding the service under test with randomized incoming requests and backend responses. It has been successfully deployed to 95% of C++ services built on Google's internal microservice architecture, and has reported thousands of bugs to service developers. We believe that it has the potential to be widely used in the industry to further improve the security and reliability of service applications. We also envision expanding this capability to microservices developed in other programming languages.

References

[1] "Microservice architectures: more than the sum of their parts?" March 2020. [Online]. Available: https://www.ionos.com/digitalguide/websites/web-development/microservice-architecture

[2] J. Kanjilal, "Security challenges and solutions for microservices architecture," October 2021. [Online]. Available: https://www.developer.com/security/security-solutions-microservices

[3] K. Serebryany, "OSS-Fuzz - Google's continuous fuzzing service for open source software." Vancouver, BC: USENIX Association, August 2017.

[4] J. Chen, W. Diao, Q. Zhao, C. Zuo, Z. Lin, X. Wang, W. Lau, M. Sun, R. Yang, and K. Zhang, "IoTFuzzer: Discovering Memory Corruptions in IoT Through App-based Fuzzing," in *Network and Distributed System Security Symposium*, 2018.

[5] Y. Zheng, A. Davanian, H. Yin, C. Song, H. Zhu, and L. Sun, "FIRM-AFL: High-Throughput greybox fuzzing of IoT firmware via augmented process emulation," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, August 2019, pp. 1099–1114.

[6] M. Xu, S. Kashyap, H. Zhao, and T. Kim, "Krace: Data race fuzzing for kernel file systems," in *IEEE Symposium on Security and Privacy (S&P)*, 2020, pp. 1643–1660.

[7] B. Jiang, Y. Liu, and W. K. Chan, "Contractfuzzer: Fuzzing smart contracts for vulnerability detection," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '18. New York, NY, USA: Association for Computing Machinery, 2018, pp. 259–269.

[8] D. Babic, S. Bucur, Y. Chen, F. Ivancic, T. King, M. Kusano, C. Lemieux, L. Szekeres, and W. Wang, "Fudge: Fuzz driver generation at scale," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019.

[9] K. Ispoglou, D. Austin, V. Mohan, and M. Payer, "FuzzGen: Automatic fuzzer generation," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, August 2020, pp. 2271–2287.

[10] M. Kelly, C. Treude, and A. Murray, "A case study on automated fuzz target generation for large codebases," in *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2019, pp. 1–6.

[11] M. Zhang, J. Liu, F. Ma, H. Zhang, and Y. Jiang, "Intelligen: Automatic driver synthesis for fuzz testing," in *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2021, pp. 318–327.

[12] B. Jeong, J. Jang, H. Yi, J. Moon, J. Kim, I. Jeon, T. Kim, W. Shim, and Y. Hwang, "Utopia: Automatic generation of fuzz driver using unit tests," in *2023 2023 IEEE Symposium on Security and Privacy (SP) (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2023, pp. 2676–2692. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SP46215.2023.00043

[13] "Structure-Aware Fuzzing with libFuzzer." [Online]. Available: https://github.com/google/fuzzing/blob/master/docs/structure-aware-fuzzing.md

[14] "gRPC - A High-Performance, Open-Source Universal RPC Framework," 2018. [Online]. Available: http://www.grpc.io

[15] "Protocol buffers documentation." [Online]. Available: https://protobuf.dev

[16] "libprotobuf-mutator." [Online]. Available: https://github.com/google/libprotobuf-mutator

[17] "Analyzing fuzzer performance." [Online]. Available: https://google.github.io/clusterfuzz/using-clusterfuzz/workflows/analyzing-fuzzing-performance

[18] K. Serebryany, "Simple guided fuzzing for libraries using LLVM's new libFuzzer," 04 2015. [Online]. Available: http://blog.llvm.org/2015/04/fuzz-all-clangs.html

[19] Efficient fuzzing guide. [Online]. Available: https://chromium.googlesource.com/chromium/src/+/main/testing/libfuzzer/efficient_fuzzing.md

[20] "FuzzTest." [Online]. Available: https://github.com/google/fuzztest

[21] "GoogleTest." [Online]. Available: https://github.com/google/googletest

[22] R. T. Fielding and R. N. Taylor, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, 2000, aAI9980887.

[23] M. Kim, Q. Xin, S. Sinha, and A. Orso, "Automated test generation for REST APIs: no time to rest yet," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, jul 2022. [Online]. Available: https://doi.org/10.1145%2F3533767.3534401

[24] A. Arcuri, "RESTful API Automated Test Case Generation," in *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, 2017, pp. 9–20.

[25] P. Marinescu, "Autonomous testing of services at scale," October 2021. [Online]. Available: https://engineering.fb.com/2021/10/20/developer-tools/autonomous-testing