Structure and Feedback in Cloud Service API Fuzzing

Evangelos Atlidakis

Submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy
under the Executive Committee
of the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

2021

ABSTRACT

Stucture and Feedback in Cloud Service API Fuzzing

Evangelos Atlidakis

Over the last decade, we have witnessed an explosion in cloud services for hosting software applications (Software-as-a-Service), for building distributed services (Platform-as-a-Service), and for providing general computing infrastructure (Infrastructure-as-a-Service). Today, most cloud services are programmatically accessed through Application Programming Interfaces (APIs) that follow the REpresentational State Transfer (REST) software architectural style and cloud service developers use interface-description languages to describe and document their services. My thesis is that we can leverage the structured usage of cloud services through REST APIs and feedback obtained during interaction with such services in order to build systems that test cloud services in an automatic, efficient, and learning-based way through their APIs.

In this dissertation, I introduce stateful REST API fuzzing and describe its implementation in RESTler: the first stateful REST API fuzzing system. Stateful means that RESTler attempts to explore latent service states that are reachable only with sequences of multiple interdependent API requests. I then describe how stateful REST API fuzzing can be extended with active property checkers that test for violations of desirable REST API security properties. Finally, I introduce Pythia, a new fuzzing system that augments stateful REST API fuzzing with coverage-guided feedback and learning-based mutations.

Contents

List of Figures

List of Tables

Acknowledgements

This thesis would have not been made possible had it not been for many amazing people that supported me throughout my PhD. I want to thank everyone who crossed paths with me. Your support—mental, psychological, and personal—made this happen. Your help has been invaluable. I am grateful to each and every one of you.

First and foremost, to my loving family: My parents, Pavlos and Chrysoula, and my brother, Konstantinos, helped me become what (I believe) an ethical human being is. This is above and beyond any diploma I will ever get. Thank you so very much for your endless love, support, and encouragement. My father, who is not with us today, would have particularly enjoyed this achievement. In loving memory of Pavlos: Your guiding hand on my shoulder will remain with me forever.

To my teachers and mentors: I stand on your shoulders. You helped me grow. I will always look up to you with admiration. I would like to thank my advisors Roxana Geambasu and Jason Nieh. They provided invaluable guidance and support during my time at Columbia. I am also grateful to all the faculty members of Columbia University whom I had the privilege of collaborating with, including Baishakhi Ray, Suman Jana, and Daniel Hsu as well as to Junfeng Yang for agreeing to serve on my committee. Furthermore, I will be eternally grateful to Patrice Godefroid, Marina Polishuck, and David Molnar for hosting me at Microsoft Research and for their constructive mentoring during my time there. Finally, I would like to thank Diomidis Spinellis and Panos Louridas, from Athens University of Economics and Business, for graciously hosting me in their lab during my time in Greece.

To my colleagues: I will remember all the days and nights that we spent in windowless labs. It has been an honor collaborating with you. No paper will ever count more than our good working environment. I would like to thank Jeremy Andrus, George Argyros, John Bell, Stefanos Chaliatsos, Konstantina Dritsa, Bryan Goodchild, Giannis Karamanolakis, Vasileios Kemerlis, Marios Kogias, George Koloventzos, Ios Kostogiannis, Mathias Lecuyer, Amit Levy, Jin Tack Lim, Charalambos Mitropoulos, Dimitris Mitropoulos, Andrei Papancea, Dimitris Paparas, John Paparrizos, Theofilos Petsios, Orestis Polychroniou, Marios Pomonis, Dennis Roellke, Suphannee Sivakorn, Thodoris Sotiropoulos, Riley Spahn, Ioannis Spiliopoulos, Florian Tramer, and Nicolas Viennot.

To my friends and partners: Thank you for tolerating me all these years. Despite the fact that I have, in times, been absent or not always in proximity, I have you all in dedicated special spots in my heart. I honestly apologize to those whom I might have hurt. I appreciate and value everything you did for me, reminiscing humbled on the good memories. In particular, I would like to thank my ex-domestic partner Iris Elizabeth Cardenas for her support and encouragement during the most difficult times of this journey. Thank you for believing in me and for standing by me when all the lights were off. No words will ever be enough to express my deep gratitude for everything you did for me. I would also like to thank my ex-partner Rosaria Orsitto for her support and inspiration which helped me put things in perspective when the rigors of research seemed too much to handle. Needless to say that I am also grateful to many more people not explicitly mentioned here; thank you all.

Finally, I acknowledge funding that made this work possible: DARPA contract FA8650-11-C-7190; NSF grants CNS-1351089, CNS-1162447, CNS-1422909, CNS-1514437, CCF-116202, CCF-1740833, and ONR N00014- 17-1-2010; a Microsoft Research Faculty Fellowship, two Sloan Fellowships, and two Google Faculty Fellowships.

My humble effort I dedicate to my loving family

My father & mother, and my brother.

Chapter 1

Introduction

Over the last decade, we have witnessed an explosion in cloud services for hosting software applications (Software-as-a-Service), for building distributed services (Platform-as-a-Service), and for providing general computing infrastructure (Infrastructure-as-a-Service). Thousands of new cloud services have been deployed by cloud platform providers, such as Amazon Web Services[10], Google Cloud[22], and Microsoft Azure[12], and support customers who are modernizing their processes by switching from the complexity of owning and maintaining their own, on-premise Information Technology (IT) infrastructure to instead simply access and pay on demand cutting edge technologies. According to a recent study, in 2020, the public cloud services market is expected to reach around 266 billion U.S. dollars in size and by 2022, its market revenue is forecast to exceed 350 billion U.S. dollars[18].

Today, most cloud services are programmatically accessed by third-party applications[38] and other services[143] through Application Programming Interfaces (APIs) that follow the REpresentational State Transfer (REST) software architectural style[84]. Cloud services that conform to the REST architectural style are called RESTful cloud services and their APIs are called REST APIs. REST APIs are implemented on top of the ubiquitous HTTP and HTTPS protocols, and provide requesting systems (clients) with a uniform and predefined set of stateless operations in order to create, monitor, manage, and delete cloud resources. By using a predefined set of stateless operations, RESTful cloud services aim for fast performance and the ability to grow by reusing components that can be managed and updated without affecting the system as a whole.

Lately, RESTful cloud service developers use interface-description languages to describe and document their services. Interface-description languages, such as the OpenAPI Specification (OAS)[29], provide an implementation-agnostic way to describe how a client can access a cloud service through its REST API, including what requests the service can handle through its REST API and the format of those requests, what responses may be received, and the respective response format[13;23;11]. Interface-description languages can be used by documentation generation tools to describe cloud service APIs and to automatically generate sample client code for testing purposes.

The fact that the vast majority of cloud services are accessed through REST APIs that are well-documented with interface-specification languages presents a unique opportunity to build systems that automatically test cloud services through their APIs. When is a RESTful cloud service reliable and secure to be publicly deployed? What kinds of software errors may be hiding behind REST APIs? And how critical these errors may be? Automatically answering such questions for production-scale, cloud services still remains an open research challenge and is of paramount importance in a multibillion-dollar market where competing cloud providers seek to avoid reliability and security incidents that will attract negative publicity.

Indeed, despite the rapidly evolving cloud ecosystem, systems for automatically testing cloud services through their REST APIs are still in their infancy. The most sophisticated testing systems currently available for REST APIs capture live API traffic, and then fuzz and replay the recorded traffic with the hope of finding software errors[8;34;6]. Since these fuzzing systems do not explicitly capture API request dependencies when generating new test cases, unfortunately, end up testing shallow service states reached by individual API requests and fail to uncover errors that require sequences of multiple API requests in order to be exposed.

Thesis. My thesis is that we can leverage the structured usage of cloud services through REST APIs and feedback obtained during interaction in order to build systems that test cloud services in an automatic, efficient, and learning-based way through their APIs.

My dissertation describes systems that are *automatic* in that they require minimal manual intervention to test target cloud services; *efficient* in that they find within a reasonable time-frame (e.g., in few hours) previously-unknown software errors that were beyond the reach of past systems; and *learning-based* in that they learn without predefined rules or heuristics how to test target cloud services. Test automation is achieved by leveraging the common structure present in the ecosystem of cloud services that are accessed through well-documented REST APIs. Efficiency is achieved by careful consideration of the semantics of the RESTful architectural design style, which allows to generate test sequences that consist of multiple API requests and test target services deeply, and by utilizing feedback obtained from the target services during testing in order to prune large search spaces. Finally, the systems described in my dissertation pursue the avenue of learning-based program analysis in order to learn from past tests common usage patterns of target cloud services and generate new tests.

Chapter 3 describes RESTler, the first stateful REST API fuzzing system. Stateful REST API fuzzing is the cornerstone of this dissertation. *Stateful* means that RESTler attempts to explore latent service states which are reachable only with sequences of multiple API requests. Unlike past REST API testing systems that issue individual API requests and test shallow service states, RESTler performs a lightweight static analysis on the target cloud service API specification and infers dependencies among API requests (e.g., inferring that a resource included in the response of a request A is necessary as input argument of another request B). RESTler then generates test cases that consist of multiple interdependent API requests and thoroughly test the corresponding cloud service. Experimental evaluation shows that RESTler is efficient in testing production-scale open-source and proprietary cloud services: RESTler has found tens of previously-unknown software errors (unhandled exceptions detected as “500 Internal Server Errors”) that have all been fixed.

Chapter 4 describes how stateful REST API fuzzing can be extended to capture errors beyond unhandled exceptions. It introduces four security rules that define desirable properties of cloud services and describes the implementation of active property checkers that generate API request sequences to specifically test for violations of these rules. By construction, active property checkers can find security rule violations beyond “500 Internal Server Errors” that can be detected by baseline stateful REST API fuzzing. Experimental evaluation shows that these checkers can report previously-unknown errors in production Azure and Office-365[28] cloud services.

Finally, Chapter 5 describes Pythia, a new fuzzing system that augments stateful REST API fuzzing with coverage-guided feedback and learning-based mutations. In baseline stateful REST API fuzzing, the automatically-generated fuzzing rules include few, predefined values for each primitive type in order to limit the combinatorial explosion of the possible fuzzing rules and values. These values remain static over time and lead to many redundant test cases (i.e., exercising identical functionality), which are also not prioritized in any way. Pythia augments stateful REST API fuzzing with learning-based mutations and coverage-guided feedback. Pythia uses a statistical model to learn common usage patterns of target cloud service APIs from seed test cases, and then generates new test cases with learning-based mutations. Additionally, coverage-guided feedback helps prioritize the test cases that are more likely to increase coverage coverage and find errors. Experimental evaluation shows that Pythia can report previously-unknown errors on production-scale open-source cloud services that were beyond the reach of baseline stateful REST API fuzzing.

Chapter 2

Related Work

The theme of this dissertation is testing of cloud services using fuzzing. Testing is a process used "to show the presence, not the absence, of software errors"[79] and. Therefore, software verification approaches using formal methods to prove the correctness of software with respect to a certain formal specification or property [126;155;72;74;71;49;96;180;181;139] are beyond the scope of this dissertation. Instead, the focus is on dynamic test input generation using fuzzing. Fuzzing[171] means automatic test input generation and execution with the goal of finding security vulnerabilities. Approaches based on static program analysis that automatically inspect source code and flag unexpected code patterns [82;85;83;60;114;113] are also orthogonal to the material presented in this dissertation, which relates to dynamic test input generation.

In the remainder of this chapter, we discuss the broader literature of test input generation approaches using fuzzing (Chapter 2.1), symbolic execution (Chapter 2.2), model-based testing (Chapter 2.3), and combinatorial test generation (Chapter 2.4).

2.1 Fuzzing

Traditionally, fuzzing has been used—with great success[4]—to test input-parsing programs written in low-level languages, such as C and C++. This dissertation is the first to introduce systems that use fuzzing to test complex distributed applications, such as cloud services, through their APIs in a stateful manner (i.e., by constructing tests with multiple interdependent API requests instead of individual API requests that test shallow service states.)

Conceptually, there are two main perspectives to categorize fuzzing approaches: based on how (and if) they use feedback from the target under test (e.g., in the form of code coverage or status codes returned) and based on how they generate new test inputs (e.g., by alternations of well-formed inputs or by using domain-specific rules). First, based on how feedback from the fuzzing target is being used, there is blackbox fuzzing, greybox fuzzing, and fully whitebox fuzzing. Second, based on how new inputs are being generated, there is random fuzzing, mutation-based fuzzing, and grammar-based fuzzing. Additionally, there are various combinations of the above approaches, some of which have also been augmented with learning-based capabilities.

Blackbox Fuzzing

Blackbox is the simplest appoach to fuzzing. A blackbox fuzzer is a client program which generates test inputs for a target program without any insight regarding the target's internal program structure. In that sense, the fuzzer treats the target under test as a "black box" which may only be monitored by a diagnostic tool[9;142;51;7] that could detect memory corruption errors (e.g., access violation exceptions and extreme memory consumption) when executing test inputs. The literature discussed in the remainder of this section includes random, mutation-based, grammar-based, and learning-based blackbox fuzzing.

Random Blackbox Fuzzing

The term "fuzzing" was originally introduced by Miller et al. in 1990 to refer to a client program that "generates a stream of random characters to be consumed by a target program."[136] One of the authors, while logged on to his workstation through a dial-up line, noticed that the rain had affected the phone lines leading to spurious characters which were causing basic operating systems utilities to crash ("core dump"). This observation motivated Miller et al.[136] to conduct a systematic testing on 90 utility programs, running on seven versions of the UNIX operating system[162], using what, today, is commonly referred to as random blackbox fuzzing.

The basic components involved in random blackbox fuzzing have remained largely unchanged until today, and include a module that generates random test inputs and a module that executes these test inputs on the target and identifies potential crashes. Miller et al.'s testing uncovered errors in 24% of UNIX utility programs, including errors in versions of UNIX that had underwent commercial product testing.

Since then, fuzzing has become standard in the software development life-cycle and is required at every untrusted public interface of commercial products[111]. Despite its simplicity and its ease of adoption, random blackbox fuzzing has limited effectiveness. For instance, a trivial program with a 32-bit integer input, which is examined in a conditional, and when a specific value is supplied the taken branch triggers a memory exception error. Randomly generating the specific value that will uncover the error requires approximately 2^{32} trials. This trivial example highlights the main caveat of random blackbox fuzzing: random test input generation produces too many redundant test cases that exercise the same code paths without uncovering any errors. Mutation-based blackbox fuzzing has evolved as a promising alternative to harness this limitation.

Mutation-based Blackbox Fuzzing

In mutation-based blackbox fuzzing, an initial set of well-formed inputs, called seeds, is used for alternations, called mutations, which leads to new test inputs, called mutants.

HTTP-fuzzers like Burp[16], AppSpider[8], Qualys' WAS[34], and others[36;6], are representative examples of mutation-based blackbox fuzzers. These tools capture, and then randomly fuzz and replay HTTP traffic, hoping to find errors. Since all these tools use well-formed input seeds (live traffic) in order to produce new mutants, the new test inputs are relatively well-formed (instead of completely random). Hence, mutation-based blackbox fuzzing approaches have a higher probability to exercise a more varied set of code paths and find unexpected errors.

However, when testing programs that process structured, non-binary input formats, such as XML parsers[37], language compilers or interpreters[17;21;33], and cloud service APIs[13;23;11], the effectiveness of mutation-based fuzzing techniques (blackbox, greybox, or whitebox) is typically limited, and grammar-based fuzzing is a better alternative.

Grammar-based Blackbox Fuzzing

In grammar-based blackbox fuzzing, new test inputs are generated according to a grammar which describes the format of a target input domain. The use of input grammars for test case generation is not a new idea. In fact, it can be traced back to the 70s[107;133;154] and the use of Context-Free Grammars (CFG). Test generation from a grammar is usually either done using random traversals of the production rules of the grammar[107] or is exhaustive and covers all available production rules[121].

Today, the grammars used are not necessarily CFGs; yet, the basic idea remains the same since the 70s: grammar rules describe how to generate new test inputs that conform to a domain-specific format. Such test inputs satisfy complex structural constraints and exercise deep code paths because they are not rejected, early on, by syntactic lexers and semantic checkers. Typical examples of blackbox grammar-based fuzzers are HTTP testing tools, such as Sulley[35;15], Peach[2], and SPIKE[3]; network protocol testing tools such as Quivid[42], the PROTOS Test-Suite[32], and the Yagg test generator[77]; and URL parser testing tools[140]. In all these tools, sequences of messages are fed to the application under test and the user provides a grammar specifying the desired request format, what parts of each request are to be fuzzed, and with what values.

The main caveat of grammar-based blackbox fuzzing is that the compilation of a domain-specific fuzzing grammars is usually a non-trivial manual task. Recently, grammar-based fuzzing has been automated in the domain of cloud service API testing by RESTler[44;46] (described in Chapter 3). RESTler processes a REST API specification and automatically compiles a grammar describing how to test a target cloud service through its REST API. However, unlike REST APIs that are well-documented with interface-specification languages, in input domains where no specification is available, or where the specification is not machine-interpretable, automatically producing such fuzzing grammars is still infeasible. For instance, the Portable Document Format (PDF) input format is described in a $1,300-$pages document[30] and automatically deriving a fuzzing grammar is unrealistic. In such domains, learning-based approaches that learn how model a target input domain from corpora of existing test inputs have been shown more applicable.

Learning-based Blackbox Fuzzing

Given a testing target which processes inputs from a domain $\mathcal{D}$ (usually non-binary, structured input domain such as XML, PDF, or JavaScript), the main idea of learning-based blackbox fuzzing is two-fold: first, use a model $\mathcal{M}$ to approximate the structure of the target input domain $\mathcal{D}$; and then, sample the learnt model $\mathcal{M}$ to obtain new test inputs which are further alternated with mutations and then executed on the testing target, hoping to uncover unknown errors.

The models used are usually either formal grammars, such as CFGs, or statistical models, such as Recurrent Neural Networks[67;170]. In GLADE[50], the authors use an two-step algorithm involving a set of heuristic rules and counterexamples[41] to query a target program and incrementally built an oracle describing acceptable XML input formats. Ultimately, they synthesize a set of CFG rules. Godefroid et al.[102]

investigated the use generative RNNs[67;170] to model valid PDF formats. In Skyfire[177], the authors mine open-source repositories in order to compile Probabilistic Context Sensitive Grammars (PCSG) describing valid input formats for XML and JavaScript language interpreters.

All these blackbox learning-based approaches loosely relate to RESTler[44] (discussed in Chapter 3) since RESTler also follows an automatic process to learn from past tests how to prune invalid request sequences by analyzing service responses at specific states. However, the way RESTler utilizes service feedback more closely relates to feedback-directed test case generation[147;87;5] rather than to the learning-based approaches which model input domains.

Overall, the key benefit of blackbox fuzzing is its ease of adoption since there is no requirement for target source code availability. Moreover, grammar-based blackbox approaches have been shown effective in uncovering non-trivial errors in domains with complex input structure. However, the main questions that remain unanswered when using blackbox fuzzing are: (i) how much fuzzing is enough? and (ii) how can one steer the search towards test inputs that are more likely to trigger errors? Both limitations emanate from the fact that blackbox testing is completely agnostic to the internal program structure of the testing target. Greybox approaches discussed next attempt to address some of these limitations.

Greybox Fuzzing

In greybox fuzzing, the testing target is not perceived as a complete "black box.". Instead, there is some insight regarding the target's internal structure, usually in the form of code coverage feedback. The conventional wisdom is that if feedback is used to steer the search towards test cases that increase code coverage, this will increase the likelihood of uncovering unknown errors.

In principal, in greybox fuzzing, test case generation can be viewed as a search and optimization problem[134;73] (e.g., with respect to code coverage). Various heuristics and search techniques have been proposed and investigated for similar seach and optimization problems, including the “Hill Climbing“ search algorithm[163], the “Simulated Annealing” search algorithm[118], and evolutionary algorithms[137;48;103;150;119] which are the foundation of greybox mutation-based fuzzing.

Mutation-based Greybox Fuzzing

American Fuzzy Lop (AFL)[20] and LibFuzzer[25] are representative candidates of mutation-based greybox fuzzing. These tool require source code availability in order to perform a light-weight static analysis and instrument the fuzzing target to produce code coverage information. During fuzzing, test inputs that produce execution traces with unseen code paths are preserved and routed for further mutations; while inputs that do not trigger new execution traces are discarded. This is an evolutionary search trying to optimize (increase) code coverage without any computationally intensive comparisons.

Such tools are very successful and have found hundreds of CVEs[4] domains with relative simple input formats, such as audio, image, or video processing applications[24;26;27], ELF parsers[19], and binary utilities[14]. However, as explained earlier, when testing applications with complex inputs the effectiveness of mutation-based fuzzing techniques is limited and grammar-based fuzzing is preferable.

Grammar-based Greybox Fuzzing

In grammar-based greybox fuzzing, the new test inputs adhere to some domain-specific structure and, in addition, feedback is used to steer the search towards inputs that increase code coverage.

Greybox grammar-based fuzzers, like Superion[178] and AFLSmart[152], combine code coverage feedback and domain-aware heuristics that assert syntactic validity of new test inputs, or allow the user to define domain-aware fuzzing rules[149], or perform mutation of the Abstract Syntax Tree (AST) level of test cases[127]. Alternatively, Zent[148], uses code coverage feedback to guide grammar-based testing tools[70;167] for XML and JavaScript interpreters.

The main strength of grammar-based greybox tools is that, due to domain awareness, they can generate highly-structured inputs that go beyond syntax parsing and semantic checking and, at the same time, code coverage feedback helps prioritize test inputs that are more likely to uncover errors hidden in the implementation of core application logic. Like the above tools, Pythia (discussed in Chapter 5) also uses code coverage feedback to guide its search. However, unlike these tools that depend on user-provided domain rules, Pythia uses a statistical model to learn common usage patterns of target REST APIs from seed inputs.

Learning-based Greybox Fuzzing

Learning-based greybox fuzzing approaches [53;157;166;144;123;122] use machine learning to automatically model various program properties. The key benefit is that, such approaches, can automatically identify latent patterns[52] without domain-specific rules.

NEUZZ[166] uses feed-forward neural networks to model the branching behaviour of a target program given a corpus of seed inputs, and then leverages the gradients of the learn representation to mutate only a small subset of input locations that mostly affect target branches. A relevant approach, from Rajpal et al.[157], uses an RNN to predict mutations that are more likely to increase code coverage and select them while vetoing redundant ones. Similarly, AFLFast[53] and uses a Hidden Markov Model (HMM) to model the probability that mutating a certain seed will lead to new inputs that exercise a certain code path. With this information, AFLFast gravitates its search towards mutations that exercise less common paths and avoids generating redundant new inputs that repeatedly exercise the same “frequent” paths.

Nichols et el.[144] investigated the use of Generative Adversarial Networks (GAN)[104] to produce a diverse set of seeds which was, in turn, given to AFL for better code coverage. Similarly, Le et al.[122] used Markov Chain Monte Carlo methods[115] to promote diverse programs to more thoroughly exercise compilers. Finally, AFLNet[153] is a greybox fuzzer for network protocol implementations which learns from recorded server-client message exchanges variations that are effective at increasing the coverage.

All the above approaches relate to the material discussed in Chapter 5, where a statistical model is used to learn common usage patterns of target REST APIs from seed inputs. In general, learning-based approaches have at their core an inherent tension between learning and fuzzing which influences various design choices. On the one hand, the purpose of "learning" is to approximate as well as possible observed behaviours. While, on the other hand, the purpose of "fuzzing" is to confuse as much as possible, hoping to trigger unexpected, erroneous behaviours.

Whitebox Fuzzing

All the blackbox and greybox approaches discussed so far are easy to adopt and impose a relatively low performance overhead. However, when testing complex programs that process very long inputs and execute millions, or even billions, of instructions[55], simple code coverage feedback is not sufficient to guide the search towards test cases that will uncover errors. In such cases, more sophisticated program analysis techniques, such as dynamic taint analysis, symbolic execution, and constraint solving, are leveraged to improve test case generation precision.

The approaches discussed in this subsection are called whitebox because they utilize insights obtained from the testing target and convert a black box (or a grey box) into a white box. Furthermore, they borrow ideas from fuzzing (e.g., operate on well-formed initial inputs). Thus, we put them under the umbrella of whitebox fuzzing.

Mutation-based Whitebox Fuzzing

Mutation-based whitebox fuzzing systems, such as SAGE[100], collect symbolic taints at the x86 instruction level of a testing target which executes initially well formed inputs and then systematically solve symbolic path constraints in order to derive new concrete test inputs that will force new execution paths. TaintScope[179], BuzzFuzz[88], Angora[65] employ symbolic taint tracing to identify which input bytes of well-formed inputs are used in branch conditionals and then focus on modifying specifically those bytes. Others systems, such as T-Fuzz[151], Steelix[125], and VUzzer[159], use static and dynamic taint analysis to detect input checks that the fuzzer-generated inputs fail, and then transform the testing target by removing these checks to increase code coverage.

Grammar-based Whitebox Fuzzing

The use of mutation-based whitebox fuzzing approaches have been very successful in testing programs that process inputs with relatively simple structure[55]. However, mutation-based whitebox fuzzing, like its blackbox and greybox counterparts, is limited when testing programs that process with highly-structured inputs. Although tracing is more precise due to the use of symbolic taints, the granularity of the taints is too fine-grained (e.g., at the byte-level). This leads to an enormous number of control flow paths in an early processing stage and many redundant mutations that produce mutants which cannot reach deep parts of the target program, beyond lexers and syntactic parsers.

To tackle this problem, grammar-based whitebox fuzzing approaches[99;130] combine symbolic execution and grammar-based specifications of valid inputs formats to greatly reduce the set of new test inputs generated. These approaches either use grammars to pre-generate strings and then, starting from those pre-generated strings, leverage symbolic execution (treating grammar primitives as symbolic) to generate new test inputs[130], or involve custom grammar-based constraint solvers that exploit the grammar for solving constraints and generate new test inputs which are, by construction, grammatically valid[99].

Overall, it is still largely unclear how to adopt whitebox fuzzing approaches—which have at their core symbolic execution, and more specifically, a synergy between symbolic and concrete execution—in the domain of cloud service testing which is the theme of this dissertation. However, symbolic execution has been extensively used to test systems written in low-level languages and the relevant literature is described next.

2.2 Symbolic Execution

Symbolic execution of programs was first introduced in the 70s by EFFIGY[116] and SELECT[56]. Instead of supplying normal, concrete inputs to a program (e.g., integer numbers), one supplies symbols representing arbitrary values and the execution proceeds arriving at expressions in terms of those symbols and constraints encoding the possible outcomes of each conditional branch. In essence, symbolic execution summarizes classes of inputs of a path into a boolean formula. Eventually, symbolic execution partitions the space of program inputs into classes of inputs that follow the same path.

Initially, symbolic execution was tout as a program verification technique but till today this goal remains elusive for large, complex software. Instead, symbolic execution and symbolic execution engines are primarily used for dynamic, high-coverage test case generation[62;63;64;173;39;124].

Unfortunately, dynamic symbolic execution has some fundamental limitations when testing large, complex software: first, the path explosion problem: the number of feasible paths can be exponential in the program size, or even infinite if the program has a single loop whose number of iterations may depend on some unbounded input. Second, the environment modeling problem: the symbolic execution engine must mediate between the program and its environment (e.g., external libraries and various OS components). Existing approaches usually incomplete because they either concretize calls to the environment or use abstract abstract models of the environment. Third, unsupported path constrains: pure symbolic execution engines cannot reason on the feasibility of paths that depend on constraints outside the theories supported by the solver (e.g., non-linear) or that depend on unknown or non-invertible operations (e.g., hash functions). Furthermore, there is significant engineering effort involved in building symbolic execution engines and the performance overhead incurred (due to tracing and solving complex path constraints) is also non-negligible.

Over the last decades, especially after significant progress on constraint solvers[80], many attempts have arisen trying to harness some of these limitations. For example, various works have investigated heuristics to alleviate the path explosion[54;158;59] and the environment problems[64;66]. Others have investigated parallelization of symbolic execution on clusters of commodity hardware[58] or have exploited the compositionality of test cases[97;112], state merging[47;40], targeted program transformations[61], and loop handling optimizations[164]. Indeed, all these ideas have largely improved the applicability of symbolic execution to test real software. However, the ability for full and complete symbolic execution remains elusive on complex software. In practice, today many systems built upon a synergy between concrete and symbolic execution.

When dealing with complex software, symbolic reasoning may be impossible for some inputs (e.g., due path contains outside the reasoning scope of the solver). In such cases, dynamic symbolic execution can leverage runtime information and use concrete values of inputs can be used instead. For example, image a conditional checking the values of two inputs, one of which is fed to a hash-function. Since the constraint solver cannot solve the respective path constraint, there is no way to symbolically reason about both outcomes of the branch in question. Nevertheless, concrete values can be used to simplify the constrain and allow dynamic test case generation to proceed.

The key design choice is to sacrificing completeness (i.e., concretize symbolic inputs) in favor of precision of test case generation. This idea was first presented in DART[98] and was later extended with symbolic memory pointers in CUTE[165]. The synergy between concrete and symbolic execution, also known as concolic execution, is at the core of mutation-based whitebox fuzzing systems, such as SAGE[100]. Concolic execution has also motivated various hybrid approaches that combine symbolic execution with random testing[129] or fuzzing[169;183;146], and active property checking[101;138].

Despite the undeniable success of whitebox approaches using symbolic execution to test software (especially written in low-level languages), the material presented in this dissertation regards testing of cloud services. It remains currently unclear how to employ symbolic execution to test complex software conglomerates, such as cloud services. Nonetheless, it is an interesting avenue to further investigate.

2.3 Model-based Testing

In model-based testing, an abstraction of a target system, called model, and its environment are used to produce test cases. Model-based testing is not to be confused with traditional model checking which is a formal formal method to test if the computations of a system are "models" with respect to temporal logic formulas. Model-based testing approaches to test Finite State Machines (FSM) [110;174;105;156;182] appeared in the 90s.

Compared to material discussed in this dissertation, model-based testing is closer to blackbox grammar-based fuzzing. The difference between model-based testing and grammar-based fuzzing is that the former generates test inputs to test an abstraction of a system, whereas the latter uses grammar rules describing how inputs conforming to a specific domain can be generated. Recently, model-based testing has been used to test file systems[106], operating system device drivers[161;68], and systems on chip[81;141].

2.4 Combinatorial Test Generation

Given a program and a set of input parameters, combinatorial test generation aims at efficiently generating test inputs which cover as many combinations of input parameters as possible [75;160;120;78;128;76]. The input parameter combinations exercised can be pairwise[120] or n-way[78;128;76]. In practice, these techniques are today used to test system configuration parameters, where the number of distinct input parameters is sufficiently small. The material discussed in the next chapters shares some common ideas with combinatorial test generation. Specifically, in grammar-based fuzzing, the number of distinct values used to fuzz each primitive type needs to be small in order to avoid combinatorial explosions.

Chapter 3

Stateful REST API Fuzzing

In this chapter, we describe RESTler, the first *stateful REST API fuzzing system.* Stateful REST API fuzzing is the cornerstone of this dissertation. It was first introduced by RESTler[44] and was then extended with active checkers capturing desirable REST API security properties[46] (see Chapter 4) and with learning-based mutations and coverage-guided feedback[45] (see Chapter 5).

3.1 Background and Motivation

Today, most cloud services are programmatically accessed by third-party applications[38] and other services[143] through Application Programming Interfaces (APIs) that follow the REpresentational State Transfer (REST) software architectural style[84]. Meanwhile, cloud service developers increasingly use interface-description languages, such as the OpenAPI Specification (OAS)[29], to describe and document their REST APIs[13;23;11] or to automatically generate sample client code in various programming languages.

Despite the rapid evolution of cloud services, systems to automatically test cloud services through their REST APIs are still in their infancy. The most sophisticated testing systems currently available, capture live traffic, and then fuzz and replay the recorded traffic with the hope of finding unknown errors[8;34;6]. Such systems do not explicitly capture dependencies among multiple API requests and, unfortunately, test only shallow service states reached by individual requests.

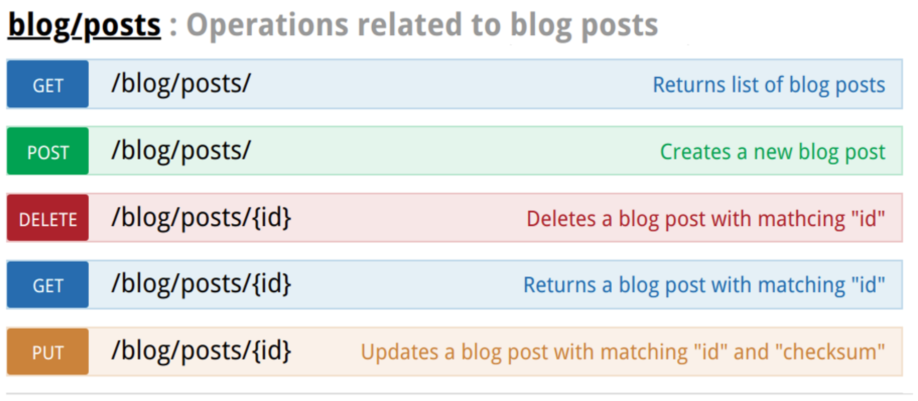
In contrast, RESTler performs stateful REST API fuzzing. *Stateful* means that RESTler generates test cases which consist of multiple interdependent API requests that explore subtle states and thoroughly test the target cloud service. To accomplish this, RESTler first carries out a lightweight static analysis of an entire OpenAPI specification, and then generates (and executes) test cases that exercise the target cloud service in a stateful manner. The static analysis performed by RESTler on API specifications of target cloud services is described next.

3.2 Processing API Specifications

An interface-description language, such as OpenAPI[29], describes what requests a cloud service can handle through its REST API and what responses may be expected. Given such a specification (e.g., in JSON or YAML format), open-source tools automatically generate web User Interfaces (UI) that allow users to view the documentation and interact with the API a service via a web browser.

A sample OpenAPI specification, in its web-UI form, is shown in Figure 3.1. This specification describes the API of a simple blog posts hosting service, which consists of five *request types* to query, create, delete, access, and update, and delete blog posts respectively. Each request type specifies the endpoint (i.e., HTTP path), the method (i.e., HTTP verb), and required parameters (i.e., HTTP body).

Clicking on any of these five request types in a web browser expands the description of the request type. For instance, selecting the second (POST) request in Figure 3.1 reveals text similar to the left of Figure 3.2. This text describes (in YAML format) the expected syntax for the specific request and its response: the `definition` part of the specification indicates that an object named `body` of type `string` is required and that an object named `id` of type `integer` is optional (since it is not required); the `path` part


blog/posts : Operations related to blog posts

GET	/blog/posts/	Returns list of blog posts
POST	/blog/posts/	Creates a new blog post
DELETE	/blog/posts/{id}	Deletes a blog post with mathcing "id"
GET	/blog/posts/{id}	Returns a blog post with matching "id"
PUT	/blog/posts/{id}	Updates a blog post with matching "id" and "checksum"

Figure 3.1: **Sample OpenAPI specification of blog posts service.** Shows the OpenAPI specification of a sample blog posts service.

of the specification describes the HTTP-syntax for this POST request and the format of the expected response.

From such a specification, RESTler automatically constructs the test-generation grammar shown on the right of Figure 3.2. RESTler's test generation (fuzzing) grammar is encoded in executable `python` code. It consists of code to generate an HTTP request (of type POST in this case) and of code to process the expected response of this request (which produces a `post id` in this case).

Each function `restler_static` simply appends the string it takes as argument without modifying it. In contrast, the function `restler_fuzzable` takes as argument a value type (like `string` in this example) and replaces it by one value of that type taken from a small *dictionary* of values for that type. How dictionaries are defined and how values are selected is discussed in the next subsection of this chapter. The specific response is expected to return a new *dynamic object* (a dynamically created resource id) named `id` of type `integer`. Using the schema shown on the left, RESTler automatically generates the function `parse_posts` shown on the right.

Furthermore, by leveraging the semantics of the RESTful architectural design style (i.e., the naming conventions and the placement of resource hierarchies in paths), RESTler automatically infers all request dependencies globally available in an API spec-

```
basePath: ’/api’
swagger: ’2.0’
definitions:
 ”Blog Post”:
  properties:
   body:
    type: string
   id:
    type: integer
  required:
  −body
  type: object

paths:
 ”/blog/posts/”
  post:
   parameters:
   −in: body
    name: payload
    required: true
   schema:
    ref: ”/definitions/Blog Post”
```

```
from restler import requests
from restler import dependencies

def parse_posts(data):
 post_id = data[”id”]
 dependencies.set_var(post_id)

request = requests.Request(
 restler_static(”POST”),
 restler_static(”/api/blog/posts/”),
 restler_static(”HTTP/1.1”),
 restler_static(”{”),
 restler_static(”body:”),
 restler_fuzzable(”string”),
 restler_static(”}”),
 ’post_send’: {
   ’parser’: parse_posts,
   ’dependencies’: [
      post_id.writer(),
   ]
 }
)
```

Figure 3.2: **OpenAPI specification and automatically-generaterd RESTler grammar.** Shows a snippet of OpenAPI specification in YAML format (left) and the corresponding grammar automatically generated by RESTler (right).

ification. For instance, in the particular example, RESTler will infer that the `ids` “produces” in the response of the second request are necessary to generate well-formed paths for the last three requests whose path “consume” such `ids`. Such *producer-consumer dependencies* extracted by RESTler, are used by the core test case generation algorithm to create sequences of interdependent request (i.e., requests with producer-consumer dependencies) which thoroughly test target cloud services in a stateful manner.

3.3 Test Case Generation for Stateful REST API Fuzzing

The main algorithm used by RESTler for test generation is shown in Figure 3.3 in `python`-like notation. It starts (line 3) by processing an OpenAPI specification as discussed in the previous section. The result of this processing is a set of request types, denoted `reqSet`, and of their dependencies.

The algorithm operates on a set of request sequences, denoted `seqSet`, which initially only contains empty sequence ϵ (line 5). A request sequence is *valid* if every response in the sequence has a valid return code (defined as HTTP status codes in the `200` range). At each iteration of the main loop (line 8), starting with $n = 1$, the algorithm generates and executes all valid request sequences of length n before moving on to length $n + 1$, and so on, until a user-specified `maxLength` is reached. Generating valid request sequences and adding them in `seqSet` is done in two steps.

First, the set of valid request sequences of length $n-1$ is *extended* (line 9) to create a set of new sequences of length n by appending each request with satisfied dependencies at the end of each sequence, as described in the EXTEND function (line 13). The function DEPENDENCIES (line 36) checks if all dependencies of the specified request are *satisfied*. This is true when every dynamic object that is a required parameter of the request, denoted by CONSUMES(req), is produced by some response to the request sequence preceding it, denoted by PRODUCES(seq). If all the dependencies are satisfied, the new sequence of length n is retained (line 18); otherwise it is discarded.

Second, each newly-extended request sequence whose dependencies are satisfied is *rendered* (line 10) one by one as described in the RENDER function (line 21). For every newly-appended request (line 24), the list of all fuzzable primitive types in the request is computed (line 25) (those are identified by `restler_fuzzable` in the code

```
Inputs: openapi_spec, maxLength
# Set of requests parsed from the OpenAPI spec
reqSet = PROCESS(openapi_spec)
# Set of sequences (initially an empty sequence)
seqSet = {$\epsilon$}
# Main loop: iterate up to a given maximum sequence length
n = 1
while (n <= maxLength):
  seqSet = EXTEND(seqSet, reqSet)
  seqSet = RENDER(seqSet)
  n += 1
# Extend all sequences in seqSet by appending new requests
def EXTEND(seqSet, reqSet):
  newSeqSet = {}
  for seq in seqSet:
    for req in reqSet:
      if DEPENDENCIES(seq, req):
        newSeqSet = newSeqSet + concat(seq, req)
  return newSeqSet
# Concretize all newly appended requests using dictionary values
def RENDER(seqSet):
  newSeqSet = {}
  for seq in seqSet:
    req = last_request_in(seq)
    $\vec{V}$ = tuple_of_fuzzable_types_in(req)
    for $\vec{v}$ in $\vec{V}$:
      newReq = concretize(req, $\vec{v}$)
      newSeq = concat(seq, newReq)
      response = EXECUTE(newSeq)
      if response has a valid code:
        newSeqSet = newSeqSet + newSeq
      else:
        log error
  return newSeqSet
# Check that all objects used in req are produced by seq
def DEPENDENCIES(seq, req):
  if CONSUMES(req) $\subseteq$ PRODUCES(seq):
    return True
  return False
# Objects required in a request
def CONSUMES(req):
  return object_types_required_in(req)
# Objects produced in the responses of a sequence of requests
def PRODUCES(seq):
  dynamicObjects = {}
  for req in seq:
    newObjs = objects_produced_in_response_of(req)
    dynamicObjects = dynamicObjects + newObjs
  return dynamicObjects
```

Figure 3.3: **Main test case generation algorithm used by RESTler.** Shows the implementation of the test case generation algorithm used by RESTler.

shown on the right of Figure 3.2). Then, each fuzzable primitive type in the request is *concretized*, which substitutes one concrete value of that type taken out of a finite, user-configurable *dictionary* of values.

For instance, for fuzzable type `integer`, RESTler might use a small dictionary with the values 0, 1, and -10, while for fuzzable type `string`, a dictionary could be defined with the values “sampleString”, the empty string, and a very long fixed string. The function RENDER generates all possible such combinations (line 26). Each combination thus corresponds to a fully-defined request `newReq` (line 27) which is HTTP-syntactically correct. The function RENDER then *executes* this new request sequence (line 29), and checks its response: if the response has a valid status code, the new request sequence is valid and retained (line 31); otherwise, it is discarded and the received error code is logged for analysis and debugging.

More precisely, the function EXECUTE executes each request in a sequence one by one, each time checking that the response is valid, extracting and memoizing dynamic objects (if any), and providing those in subsequent requests in the sequence if needed, as determined by the dependency analysis; the response returned by function EXECUTE in line 29 refers to the response received for the last, newly-appended request in the sequence. Note that if a request sequence produces more than one dynamic object of a given type, the function EXECUTE will memoize all of those objects, but will provide them later when needed by subsequent requests in the exact order in which they are produced; in other words, the function EXECUTE will not try different ordering of such objects. If a dynamic object is passed as argument to a subsequent request and is “destroyed” after that request, i.e., it becomes unusable later on, RESTler will detect this by receiving an invalid status code, outside the `200` range (e.g., a `400` or a `500` status code) when attempting to reuse that unusable object, and will then discard that request sequence.

By default, the function RENDER of Figure 3.3 generates *all* possible combinations of dictionary values for every request with several fuzzable types (see line 26). For large dictionaries, this may result in astronomical numbers of combinations. In that case, a more scalable option is to randomly sample each dictionary for one (or a few) values, or to use *combinatorial-testing* algorithms[75] for covering, say, every dictionary value, or every pair of values, but not every k-tuple. In the experiments reported later, we used small dictionaries and the default RENDER function shown in Figure 3.3.

Optimizations

The function EXTEND of Figure 3.3 generates *all* request sequences of length $n + 1$ whose dependencies are satisfied. Since n is incremented at each iteration of the main loop of line 8, the overall algorithm performs a complete *breadth-first search* (BFS) in the search space defined by all possible request sequences. To harness the combinatorial explosion of a complete BFS we investigate two optimization.

BFS-Fast. In function EXTEND, instead of appending every request to every sequence, every request is appended to at most one sequence. This results in in a smaller set `newSeqSet` which covers (i.e., includes at least once) every request but does not generate all valid request sequences. Like BFS, BFS-Fast still exercises every executable request type at each iteration of the main loop in line 8: it still provides full grammar coverage with respect to all possible rendering of each individual request but does not explore all possible request sequence combinations for a given sequence length. This allows BFS-Fast to scale better than BFS and to generate longer request sequences.

RandomWalk. In function EXTEND, the two loops of line 15 and line 16 are eliminated; instead, the function now returns a single new sequence of requests whose dependencies are satisfied and which is generated by *randomly* appending one `req` from `reqSet` to the current (one) `seq` in `seqSet`. (The function always chooses a `req` that satisfies all dependencies of `seq`.) Once a `seq` of length n has been constructed, the main loop of line 8 will proceed to construct a sequence of length $n + 1$ while maintaining a `seqSet` with exactly one sequence (of the previous iteration). Furthermore, in function RENDER, when the condition of line 30 is satisfied (i.e., a valid rendering of the last request has been found), the loop of line 26 will stop.

In essence, RandomWalk sacrifices full grammar coverage both with respect to all possible renderings of individual requests and with respect to all possible request combinations for a given sequence length. This design choice allows RandomWalk to explore the search space of possible request sequences deeper more quickly than BFS or BFS-Fast in a random, non-systematic manner. When RandomWalk can no longer extend the current request sequence, it restarts from scratch from an empty `seqSet` and repeats the same process, trying to construct a new sequence. Since RandomWalk does not memoize past request sequences between restarts, it might regenerate the same request sequence multiple times.

In Chapter 3.6, we report experiments performed on three production-scale open-source services comparing the scalability of BFS, BFS-Fast, and RandomWalk in term of code coverage achieved and new bugs found. Next, we discuss some implementation details of RESTler.

3.4 Implementation of Stateful REST API Fuzzing in RESTler

We have implemented RESTler in $3,151$ lines of modular python code split into: the parser and compiler module, the core fuzzing runtime module, and the garbage collector (GC) module.

The parser and compiler module is used to parse an OpenAPI specification and to generate the RESTler grammar describing how to fuzz a target service. (In the absence of an OpenAPI specification, the user could directly provide the RESTler grammar.) The core fuzzing runtime module implements the algorithm of Chapter 3.3 and its variants. It renders API requests, processes service-side responses to retrieve values of the dynamic objects created, and analyzes service-side feedback to decide which requests should be reused in future generations while composing new request sequences. Finally, the GC runs as a separate thread that tracks the creation of the dynamic objects over time and periodically deletes aging objects that exceed some user-defined limit.

RESTler is currently a command-line tool that takes as input an OpenAPI specification, service access parameters (e.g. IP, port, authentication), the mutations dictionary, and the search strategy to use during fuzzing. After compiling the OpenAPI specification, RESTler displays the number of endpoints discovered and the list of resolved and unresolved dependencies, if any. In case of unresolved dependencies, the user may provide additional annotations or resource-specific mutations (see Section 3.5) and re-run this step to resolve them. Alternatively, the user may choose to start fuzzing right away and RESTler will treat unresolved dependencies in consumer parameters as `restler_fuzzable` string primitives.

During fuzzing, RESTler treats each `500` status code (`500` "Internal Server Error") received after executing a request sequence as a *bug* and uses a bucketization scheme to cluster similar `500` "Internal Server Error" incidents.

Bug Bucketization

When fuzzing, different instances of a same bug are often found repeatedly. Since all the bugs found have to be inspected by the user, it is therefore important in practice to aid this analysis by identifying likely-redundant instances of a same unique bug.

Since we define a *bug* to be a `500` HTTP status code received after executing a request sequence, each bug found is associated with the request sequence that was executed to uncover it. Given this property, we use the following bucketization procedure for bugs found by RESTler:

> Whenever a new bug is found, we compute all non-empty suffixes of its non-rendered request sequence[1] (starting with the smallest one) and check whether some suffix is a previously-recorded sequence leading to a bug found earlier. If there is a match, the new bug is added to the bucket of that previous bug. Otherwise, a new bucket is created with the new bug and its request sequence.

In the above procedure, requests in sequences are identified by their type, not by how they are rendered: fuzzable primitive types are not taken into account and different renderings of the same request are equivalent. When using BFS or BFS-Fast, This bucketization scheme will identify bugs by the shortest sequence necessary to find it.

Baseline stateful REST API fuzzing described so far, can only find bugs manifested as `500` "Internal Server Errors." However, in Chapter 4, we describe how stateful REST Fuzzing can be extended with active checkers that capture desirable REST API security properties and detect errors beyond `500` "Internal Server Errors."

3.5 Experiences with RESTler in Proprietary Cloud Services

In this section, we describe our experiences running RESTler to test proprietary Azure[12] and Microsoft Office365[28] cloud services performing resource management and real-time data aggregation. RESTler found several bugs (that have now been fixed) in all four services tested. However, we also faced a number of challenges unique to proprietary services, including resource quota limitations, short-lived access tokens, and

[1] A request sequence of length n has n suffixes of length $1, 2, \ldots, n$.

complex API dependencies beyond the canonical REST API structure with application-specific resource values and naming schemes. We describe the extensions made to address these challenges.

Resource Quotas. Proprietary services that run in public clouds are deployed with default resource quotas. Once the specific resource quotas are reached, RESTler's test generation algorithm will repeatedly attempt to execute request sequences containing requests that can no longer succeed due to exceeded quotas (since these requests were valid in prior tests and generated lots of new resources). This impedes test case generation progress. This challenge is unique to proprietary cloud deployments, contrary to self-contained deployments where one can easily control and reconfigure resource quotas. To address this problem, RESTler includes a garbage collector (GC). The GC runs as a separate thread that monitors the creation of dynamic resources over time and periodically deletes dynamic objects that are no longer used in order to avoid exceeding resource quotas. This allows RESTler to continuously test new sequences for hours or days without hitting resource-quota-related errors.

Short-lived Access Tokens. Unlike in self-contained deployments where an admin can pre-populate static or long-lived authentication tokens, public cloud services use short-lived, refreshable authentication tokens. Usually, a public endpoint, accessible with some type of static credentials (e.g., a username-password pair or a master token) and service-specific logic, generates fresh, short-lived access tokens. The latter are added in the header of HTTPS requests. Since different services may require custom logic to access their public authentication endpoints, RESTler provides an authentication hook which periodically executes a user-provided piece of code (e.g., a script) and propagates fresh values in the pool of refreshable authentication tokens.

Application-specific Naming Schemes. As discussed in Chapter 3.2, RESTler performs a light-weight static analysis of a OpenAPI specification to infer dependencies among requests of the target REST API. However, part of a target API may not be fully REST compliant, or the specification may be incomplete, and consequently the inferred dependencies will also be incomplete. To address this challenge, RESTler supports annotations, which can be added directly to the specification (as OpenAPI extensions), in order to explicitly declare dependencies, as well as resource-specific mutations, which can be used for the creation of resources that require some custom format (e.g., an IP address). These two features have proven useful in practice because, unlike the typical design of REST APIs, Azure services use PUT requests to create resources whose user-provided names are passed as URL parameters and, after successful creation, are also returned in the response. For this scenario, one can use resource-name-specific mutations to indicate that a PUT request should create a resource named in a custom format, and then use that name to identify the corresponding dynamic object in subsequent requests.

3.6 Evaluation

In this section, we present experimental results obtained with RESTler that answer the following questions:

Q1: Are both inferring dependencies among request types and analyzing dynamic feedback necessary for effective automated REST API fuzzing?

Q2: Are tests generated by RESTler exercising deeper service-side logic as sequence length increases?

Q3: How do the three search strategies implemented in RESTler compare across various APIs in terms of code coverage?

Q4: How do the three search strategies implemented in RESTler compare across various APIs in terms of new bugs found?

Experimental Setup

We answer the first question (Q1) using a simple blog posts service with a REST API. We answer (Q2), (Q3), and (Q4) using three open-source, production-scale web services with REST APIs.

Blog posts service. To answer (Q1) we use a simple blog posts service, written in 189 lines of python code using the Flask web framework[86]. Its functionality is exposed over a REST API with a OpenAPI specification shown in Figure 3.1. The API contains five request types: (i) GET on `/posts`: returns all blog posts currently registered; (ii) POST on `/posts`: creates a new blog post (body: the text of the blog post); (iii) DELETE `/posts/id`: deletes a blog post; (iv) GET `posts/id`: returns the body and the checksum of an individual blog post; and (v) PUT `/posts/id`: updates the contents of a blog post (body: the new text of the blog post and the checksum of the older version of the blog post's text).

In order to model an imaginary subtle bug, at every update of a blog post (PUT request with body text and checksum) the service checks if the checksum provided in the request matches the recorded checksum for the current blog post, and if it does, an uncaught exception is raised. Thus, this bug will be triggered only if dependencies on dynamic objects shared across requests are taken into account during test generation.

Production-scale cloud services. To answer (Q2), (Q3), and (Q4) we use three open-source, production-scale cloud services; namely GitLab, Mastodon, and Spree.

First, we use GitLab enterprise version 11.11 and test API families related to common version control operations. GitLab is an open-source web service for self-hosted Git, its back-end is written in over 376K lines of ruby code using ruby-on-rails, and its functionality is exposed through REST APIs. It is used by more than $100,000$ organizations, has millions of users, and has currently a 2/3 market share of the self-hosted Git market[95]. We configure GitLab to use Nginx HTTP web server and 20 Unicorn

Target Service	**API Family**	**Total Requests**
GitLab	Commits	15 (*11)
	Brances	8 (*2)
	Issues & Notes	25 (*20)
	User Groups	53 (*2)
	Projects	54 (*5)
	Repos & Files	12 (*22)
Mastodon	Accounts & Lists	26 (*3)
	Statuses	18 (*19)
Spree	Storefront Cart	8 (*11)

Table 3.1: **Target Cloud Service APIs.** Shows the number of distinct request types in each API family and the average number of primitive value combinations that are available for each request type (*).

workers with 4GB of physical memory. We use postgreSQL with 20 workers and the default GitLab configuration for sidekiq queues and redis workers. According to GitLab's recommendations, such configuration scales up to $4,000$ concurrent users[94].

Second, we use Mastodon version 3.1.1, an open-source, self-hosted social networking service with more than 4.4M users[132]. We follow the same configuration, as in GitLab, regarding Unicorn workers and persistent storage.

Third, we use Spree, an open-source e-commerce platform for Rails 6 with over 1M downloads[168]. We test the default sandbox configuration of version 4.0. All the experiments discussed in this evaluations were run on Ubuntu 18.04 Google Cloud VMs[22] with 8 logical CPU cores and 52GB of physical memory. Each fuzzing client is used to test a target service deployment running on the same machine

Table 3.1 shows characteristics of the target service APIs, such as the total number of requests in each API family and the average number of different values available to fuzz each request. The total number of requests in each API family and the average number of values available per request indicate the size of the respective fuzzing space.

Fuzzing dictionaries. To maintain the size of the search space small and, at the same time, perform some meaningful fuzzing we use a small, predifined dictionary of values available for each primitive type. *string* fuzzable types have possible values “sampleString” and “” (empty string); *integer* fuzzable types have possible values “0” and “1”; *boolean* fuzzable types have possible values “$true$” and “$false$”.

Techniques for effective REST API fuzzing (Q1)

In this section, we report results with our blog posts service to determine whether both (1) inferring dependencies among request types and (2) analyzing dynamic feedback are necessary for effective automated REST API fuzzing (Q1). We choose a simple service in order to clearly measure and interpret the testing capabilities of the two core techniques being evaluated. Those capabilities are evaluated by measuring service code coverage and client-visible HTTP status codes.

Specifically, we compare results obtained when exhaustively generating all possible request sequences of length up to three, with three different test-generation algorithms:

1. RESTler ignores dependencies among request types and treats dynamic objects – such as post `id` and `checksum` – as fuzzable primitive type `string` objects, while still analyzing dynamic feedback.
2. RESTler ignores service-side dynamic feedback and does not eliminate invalid sequences during the search, but still infers dependencies among request types and generates request sequences satisfying those.
3. RESTler uses the algorithm of Figure 3.3 using both dependencies among request types and dynamic feedback.

Figure 3.4 shows the number of tests, i.e., request sequences, up to maximum length 3, generated by each of these three algorithms, from left to right. The top plots show the cumulative code coverage measured in lines of code over time, as well as when the

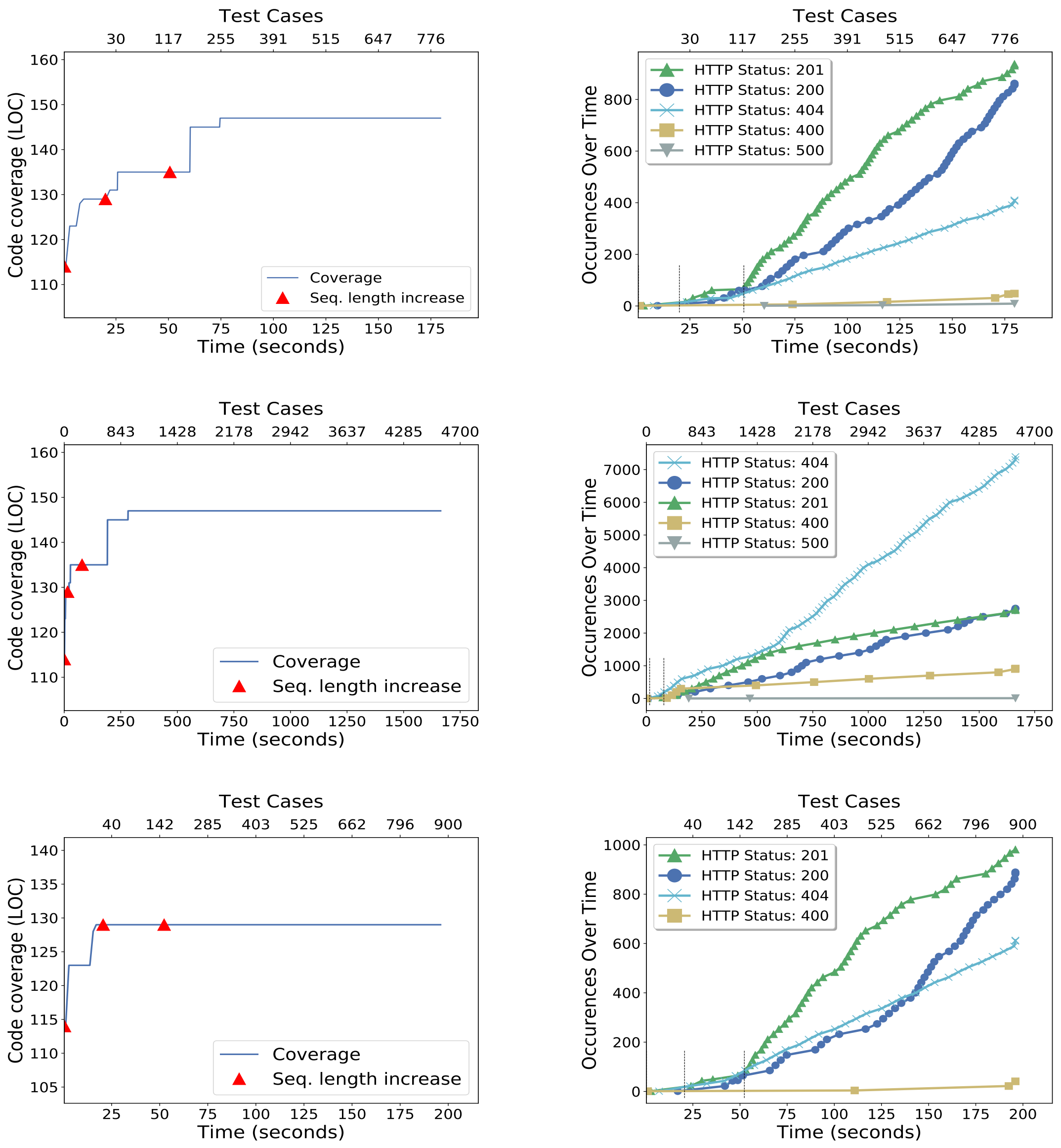


Figure 3.4: **Blog posts service code coverage and HTTP status codes over time.** Shows the increase in code coverage over time (left) and the respective cumulative number of HTTP status codes received over time (right). *Top:* RESTler utilizes both dependencies among request types and dynamic feedback. *Middle:* RESTler ignores dynamic feedback. *Bottom:* RESTler ignores dependencies among request types. When leveraging both techniques, RESTler achieves the best code coverage and finds the planted 500 "Internal Server Error" bug with the least number of tests.

sequence length increases. The bottom plots show the cumulative number of HTTP status codes received.

Code Coverage. First, we observe that without considering dependencies among request types (Figure 3.4, bottom left), code coverage is limited to up to 130 lines and there is no increase over time, despite increasing the length of request sequences. This illustrates the limitations of using a naive approach to test a service where values of dynamic objects like `id` and `checksum` cannot be randomly guessed or picked among values in a small predefined dictionary. In contrast, by inferring dependencies among requests and by processing service responses RESTler achieves an increase in code coverage up to 150 lines of code (Figure 3.4, middle left and top left).

Second, we see that without considering dynamic feedback to prune invalid request sequences in the search space (Figure 3.4, middle) the number of tests generated grows quickly, even for a simple API. Specifically, without considering dynamic feedback (Figure 3.4, middle right), RESTler produces more than $4,600$ tests that take $1,750$ seconds and cover about 150 lines of code. In contrast, by considering dynamic feedback (Figure 3.4, top), the state space is significantly reduced and RESTler achieves the same code coverage with less than 800 test cases and only 179 seconds.

HTTP Status Codes. We make two observations. First, focusing on `40X` status codes, we notice a high number of `40X` responses when ignoring dynamic feedback (Figure 3.4, middle right). This indicates that without considering service-side dynamic feedback, the number of possible invalid request sequences grows quickly. In contrast, considering dynamic feedback dramatically decreases the percentage of `40X` status codes from 60% to 26% without using dependencies among request types (Figure 3.4, bottom right) and to 20% with using dependencies among request types (Figure 3.4, top right). Moreover, when using dependencies among request types (Figure 3.4, bottom right), we observe the highest percentage of `20X` status codes (approximately 80%), indicating that RESTler then exercises a larger part of the service logic – also confirmed by coverage data (Figure 3.4, top left).

Second, when ignoring dependencies among request types, we see that no `500` status

API	**Total Requests**	**Sequence Length**	**Coverage Increase**	**Tests**	**seqSet Size**	**Dynamic Objects**
Commits	15 (*11)	1	598	1	1	1
		2	1108	7	5	10
		3	1196	250	46	521
		4	1760	2220	1341	6577
		5	1760	3667	20679	12518
Branches	8 (*2)	1	598	1	1	1
		2	1089	8	6	11
		3	1172	58	44	107
		4	1182	576	387	1279
		5	1185	3644	5528	9336
Issues	25 (*20)	1	816	37	37	37
		2	1163	2444	1839	4245
		3	1163	4156	15658	8870
User Groups	53 (*2)	1	887	39	39	38
		2	1177	3508	3360	5204
		3	1177	4817	79518	8946
Projects	54 (*5)	1	934	42	41	38
		2	1192	1870	1781	3343
		3	1203	3226	18173	7374
Repos & Files	12 (*22)	1	598	1	1	1
		2	1117	97	65	206
		3	1181	5153	2194	15472
Accounts & Lists	26 (*3)	1	4	30	8	8
		2	215	470	424	544
		3	656	30229	24300	42199
Statuses	18 (*19)	1	4	60	8	8
		2	333	5908	416	6272
		3	631	28926	5192	55376
Storefront Cart	8 (*11)	1	10	1	1	1
		2	208	2	6	2
		3	1473	47	98	62
		4	1943	6380	6201	21309

Table 3.2: **Testing common GitLab, Mastodon, and Spree APIs with RESTler.** Shows the increase in sequence length, code coverage, tests executed, `seqSet` size, and the number of dynamic objects being created using BFS for 5 hours. Longer request sequences gradually increase service-side code coverage.

codes are detected (Figure 3.4, bottom right), while RESTler finds a handful of `500` status codes when using dependencies among request types (see Figure 3.4, top right and middle right). These `500` responses are triggered by the unhandled exception we planted in our blog posts service after a PUT blog update request with a checksum matching the previous blog post's body (see Chapter 3.6). When ignoring dependencies among request types, RESTler misses this bug (Figure 3.4, bottom right). In contrast, when analyzing dependencies across request types and using the checksum returned by a previous GET `/posts/id` request in a subsequent PUT `/posts/id` update request with the same `id`, RESTler does trigger the bug. Furthermore, when additionally using dynamic feedback, the search space is pruned while preserving this bug, which is then found with the least number of tests (Figure 3.4, bottom right).

Overall, these experiments illustrate the complementarity between utilizing dependencies among request types and using dynamic feedback, and show that both are needed for effective REST API fuzzing.

Deeper service exploration (Q2)

In this section, we present experiments to determine whether the tests generated by RESTler exercise deeper service-side logic as sequence length increases (Q2). We perform individual experiments on the nine API families using 5h fuzzing sessions with the default test-generation algorithm of RESTler (i.e., BFS). For each experiment, we limit the number of fuzzable primitive-type combinations to maximum $1,000$ combinations per request. Between experiments, we force the entire GitLab, Mastodon, or Spree service to restart from the same initial state.

Table 3.2 shows the increase (going down) in the sequence length, the increase in code coverage (new lines after boot), the total number of tests executed, the `seqSet` size (see Figure 3.3), and the number of dynamic objects created until the 5-hours timeout.

API	**BFS**			**BFS-Fast**			**RandomWalk**		
	SeqSet	**Len**	**Req**	**SeqSet**	**Len**	**Req**	**SeqSet**	**Len**	**Req**
Commits	64579	5	15/15	20	27	15/15	1	9 (2850)	10/15
Branches	44805	6	8/8	8	100	8/8	1	14 (3392)	8/8
Issues & Notes	58332	3	6/25	99	17	25/25	1	17 (376)	24/25
User Groups	31621	3	28/53	93	33	50/53	1	40 (1341)	51/53
Projects	50541	3	36/54	16	48	54/54	1	13 (2047)	52/54
Repos & Files	77666	4	11/12	121	19	11/12	1	8 (3352)	12/12
Accounts & Lists	142124	4	24/26	24	75	24/26	1	3 (17652)	24/26
Statuses	288448	4	18/18	40	39	18/18	1	2 (13672)	15/18
Storefront Cart	98571	5	8/8	45	100	8/8	1	24 (1481)	7/8

Table 3.3: **Comparison of *BFS*, *BFS-Fast* and *RandomWalk* after** 48 **hours**. Shows the `seqSet` size for each search strategy, the maximum sequence length (RandomWalk restarts in parenthesis), and the request coverage (i.e., requests that have been executed at least once) after 48 hours in GitLab, Mastodon, and Spree. *BFS-Fast* and *RandomWalk* maintain a much smaller `seqSet` compared to *BFS*, and usually construct longer request sequences and achieve better request coverage.

Code Coverage. The fourth column of Table 3.2 shows the cumulative code coverage achieved after executing all the request sequences generated by RESTler for each sequence length, or until the 5-hours timeout expires. The results are incremental on top of the initial lines of code executed, by default, during booting each service (i.e., 16,836 lines for GitLab, 6,434 for Mastodon, and 3,359 for Spree). Later on, we complement these results, in the next subsection, with graphs showing how coverage evolves over time 48h in Figure 3.5. However, here, our purpose is not to compare RESTler's code coverage against developers' test cases code coverage or how it evolves over time. Instead, the focus is to demonstrate that RESTler test-cases exercise deeper states of server-side logic as sequence length increases.

From Table 3.2, we can clearly see that longer sequence lengths consistently increase service-side code coverage across all nine APIs and all three services. This is the desired and expected behaviour which demonstrates the key strength of stateful REST API fuzzing: the test sequences generated by stateful REST API fuzzing successfully test functionality that can be exercised only by sequences of interdependent requests.

As an example, consider the GitLab functionality of "selecting a commit". According to GitLab's specification, selecting a commit requires two dynamic objects, a *project-id* and a *commit-id*, and the following dependency of requests is implicit: (1) a user needs to create a project, (2) use the respective *project-id* to post a new commit, and then (3) select the commit using its *commit-id* and the respective *project-id*. Clearly, this operation can only be performed by sequences of three requests or more. For the Commit APIs, note the gradual increase in coverage from 598 to $1,108$ to $1,196$ lines of code for sequence lengths of one, two, and three, respectively. Most notably, for the Branches API, service-side code coverage keeps gradually increasing for sequences of length up to five, and reaches $1,185$ lines when the 5-hours limit expires. Similarly, for the Storefront Cart APIs, service-side code coverage keeps gradually increasing for sequences of length up to four, and reaches $1,943$ lines when the 5-hours limit expires.

Tests, Sequence Sets, and Dynamic Objects. In addition to code coverage, Table 3.2 also shows the increase in the number of tests executed, the size of `seqSet` after the RENDER function returns in line 10 of Figure 3.3, and the number of dynamic objects created by RESTler. One *test* here means, one stateful sequence of requests of length depending on the value of n in line 7 of Figure 3.3.

We observe, in the last three columns of Table 3.2 that the number of tests, the size of the `seqSet`, and the number of dynamic objects created rapidly increases with sequence length across all APIs due to the exhaustive nature of the BFS search strategy. Nevertheless, we emphasize that without the two key techniques evaluated in Chapter 3.6 this growth would be much worse.

For instance, for the Commit API, the `SeqSet` size is 20, 679 and there are 12, 518 dynamic objects created by RESTler for sequences of length up to five. By comparison, since the Commits API has 15 request types with an average of 11 rendering combinations, the number of all possible rendered request sequences of up to length four is already more than 741 millions and a naive brute-force enumeration of those would clearly be intractable. Similarly, for the Storefront Cart API, the `SeqSet` size is 6, 201 and there are 21, 309 dynamic objects created by RESTler for sequences of length up to four. By comparison, since the API has 8 request types with an average of 11 rendering combinations, the number of all possible rendered request sequences of up to length four is already more than 59 millions. Still, even with the two core techniques used in RESTler, the search space explodes quickly. Next, we evaluate two BFS optimizations.

Search strategies: Code coverage (Q3)

We now present results of experiments comparing the BFS, BFS-Fast, and RandomWalk search strategies defined in Chapter 3.3 (Q3). For each search strategy, Table 4.1 shows the maximum sequence length achieved after 48 hours, the respective request coverage (i.e., the number of requests that have been used in at least one test case), and the size of the `seqSet` when the 48-hours timeout is reached. For the RandomWalk search strategy the total number of restarts is also shown in parenthesis. Additionally, Figure 3.5 shows the cumulative service-side code coverage increase (on top of the boot coverage) for the three search strategies over 48 hours. We compare the search strategies.

First, we examine BFS. From Table 4.1 we observe that across all APIs BFS maintains the largest `seqSet` size, compared to BFS-Fast and RandomWalk. This is inevitable since BFS provides full grammar coverage both with respect to all possible value combinations of each individual request and with respect to all possible request type combinations given a sequence length (see Chapter 3.3) and, therefore, explores a

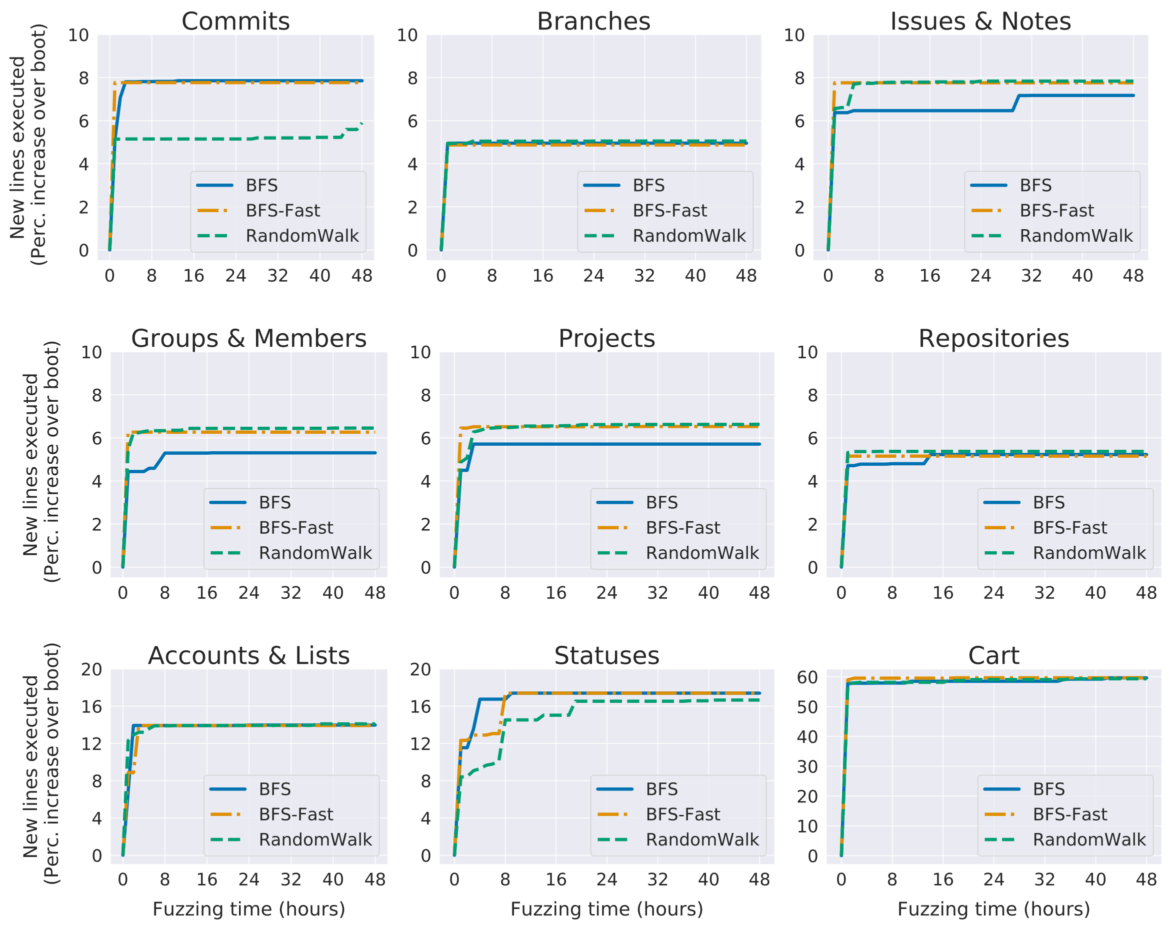


Figure 3.5: **Comparison of *BFS*, *BFS-Fast* and *RandomWalk* code coverage over 48 hours in GitLab, Mastodon, and Spree.** Shows the percentage code coverage increase (on top of the initial boot coverage) for *BFS*, *BFS-Fast* and *RandomWalk* over 48 hours in GitLab, Mastodon, and Spree. *BFS-Fast* and *RandomWalk* perform evidently better than BFS in API families with many requests or many possible value combinations per request.

considerably larger search space. Consequently, BFS does not scale well: it constructs shorter sequences than BFS-Fast and RandomWalk, especially in APIs with relatively many possible value combinations per request (e.g., Issues: 3 versus 17 and 17; Repos & Files: 4 versus 19 and 8) and in APIs with relatively many request types (e.g., User Groups: 3 versus 33 and 40; Projects: 3 versus 16 and 13). Notable exceptions are the APIs of Mastodon (Accounts & Lists and Statuses), in which RandomWalk performs worst in terms of maximum sequence length achieved.

To understand the reason behind this differentiation we explain how these two API families differ from all the rest. All APIs have a base parent request. That is, a producer of dynamic resources which are a necessary dependencies for all other requests. For example, in GitLab, the base parent is a request type which produces a project-id correlated with some user that can perform various actions. Similarly, in Spree, the base parent is a request type which constructs a fresh user-id correlated with some user that can then perform various actions. These base parent request types have exactly one possible value combination both in GitLab and in Spree. However, the respective base parent request type in Mastodon (which creates a fresh user account) has thirty possible value combinations, out of which, only one is valid. To make matters worse, RandomWalk does not memoize request renderings between restarts (see Chapter 3.3). Therefore, RandomWalk has a relatively low probability of randomly selecting the one valid value combination compared to all other APIs of GitLab and Mastodon (i.e., 1 out of 30 versus 1 out of 1). This forces RandomWalk to perform many restarts. Specifically, there are $17,652$ restarts for Accounts & Lists and $13,672$ restarts for Statuses, which is an order of magnitude more than all other APIs.

Finally, from Table 4.1, we see that BFS provides the worst request coverage and in various API families (such as Issues & Notes, User Groups, and Projects) there are many requests than have never been executed after 48h. For example, in Issues & Notes, out of the 25 total requests, only 6 have been covered after 48h. To understand the exact size of the state space explored by BFS in Issues after 48h, consider that for sequence length three the most complex test case creates a project (one possible value combination), then creates an issue (108 value combinations), and then edits an issue (324 value combinations). Fully exploring the search space of this single one test case, using BFS, leads to approximately 35K value combinations. (Let alone exploring the search space defined by all sequences in `seqSet` after multiple hours.) The conclusions drawn from Table 4.1, with respect to request coverage, are also in line with what can be observed in Figure 3.5, regarding service-side code coverage. In particular, Figure 3.5 shows that BFS performs worst in Issues & Notes, User Groups, and Projects, which is exactly what was discussed above.

Second, we examine BFS-Fast. We observe, in Table 4.1, that BFS-fast maintains a considerably smaller `seqSet` than BFS (in fact many orders of magnitude) across all APIs. This is expected because, by design, BFS-Fast sacrifices full grammar coverage with respect to all possible request combinations for a given sequence length, and instead, given a sequence length, it appends each request exactly once per generation. Specifically, the largest `seqSet` for BFS-Fast is 121 (for Repos & Files), while the largest `seqSet` for BFS is $288,448$ (for Statuses). In comparison with BFS-Fast, only RandomWalk (discussed next) maintains a smaller `seqSet` which always consists of exactly one sequence. Additionally, BFS-Fast always constructs longer sequences than BFS after 48h. In Branches and in Storefront Chart BFS-Fast it even reaches length 100, which is the maximum, user-defined allowed sequence length. Furthermore, from Table 4.1, we observe that BFS-Fast achieves considerably better request coverage than BFS and similar to RandomWalk. Indeed, after 48h, BFS-Fast covers all requests in most API families, except User Groups, Repos & Files, and Accounts & Lists where it covers 50 out of 53, 11 out of 12, and 24 out of 26 requests respectively.

The better coverage in API requests is also reflected in service-side code coverage. Figure 3.5 shows that BFS-Fast performs better (or at least as good) compared to BFS and RandomWalk across all API families and always reaches its plateau faster than all other search strategies reache theirs – usually in just few hours. As explained earlier, the difference between BFS-Fast and BFS is more apparent in API families with many requests or many possible value combinations per request because BFS-Fast sacrifices full grammar coverage to scale better.

Third, we examine RandomWalk. RandomWalk always maintains the smallest `seqSet` of exactly one sequence. By construction, RandomWalk sacrifices full grammar coverage both with respect to all possible request combinations for a given sequence length and with respect to all posible value renderings of individual requests, and just tries to incrementally extend the current sequence in `seqSet` with one random request from `reqSet` until no other request can be used because there are no valid dependencies (see Chapter 3.3). After 48 hours, RandomWalk usually explores deeper request sequences compared to BFS and less deep compared to BFS-Fast. Indeed, in APIs that contain requests with relatively more value combinations (such as Commits, Issues & Notes, and Repos & Files) randomly selecting valid values and constructing longer sequences than BFS-Fast is unlikely, especially because RandomWalk does not memoize valid/invalid value combinations after restarts and may produce many duplicate test cases. By contrast, in User Groups there are 53 requests with only 2 possible value combinations when, in comparison, in Projects there are 54 requests but with 5 possible renderings. Consequently, the probability of randomly generating valid and long sequences is higher in Projects (despite the high number of API requests) and, indeed, after $1,341$ restarts RandomWalk produces longer sequences than BFS-Fast (sequence length 40 versus 33).

Regarding RandomWalk request coverage, we observe that it behaves similarly with BFS-Fast (and better than BFS) in most APIs, except, most notably, for Commits API. In the later, RandomWalk covers only 10 out of 15 API requests, whereas both BFS and BFS-Fast cover 15 out of 15 API requests. To understand why RandomWalk underperforms in the specific API family, we need to analyze the dependencies across the requests of Commits APIs, model the probability RandomWalk has to cover all request types in the specific API family, and compare this probability with other APIs.

We clarify that the five request types which have never been covered by RandomWalk in Commits are all requests that consume three dynamic resource types: A project-id, a branch-id, and a commit-id, and we will model the probability RandomWalk has to generate at least one sequence that satisfies the dependencies of these five requests: (1) The producer for project-id is a base parent and has a probability one to be selected when the sequence length is one; (2) the producer for a branch-id consumes a project-id and can be selected only after sequence length is two, with probability 0.25 because three other requests (i.e., create project, delete project, query project) have valid dependencies when the sequence length is two; (3) the producer for commit-id consumes a branch-id and can be selected only after sequence length is three, with probability 0.16 because six other requests (i.e., create project, delete project, query project, create branch, delete branch, query branch) have valid dependencies when the sequence length is three; (4) finally, each one of the five requests, that have never been rendered, may be selected when sequence length is four with probability 0.06 since all fifteen requests now have valid dependencies.

From (1), (2), (3), and (4), the probability to cover render one of the five uncovered requests using RandomWalk is 0.25*0.16*0.06 or approximately 0.2%. However, the actual probability in the given time frame of 48 hours is much less than 0.2% because the above analysis did not consider the time spend—since RandomWalk does not memoize between restarts—until a valid value combination for each request is found. The same rationale can be applied to other APIs with uncovered API requests, such as Issues & Notes or Mastodon Statuses. For example, in Issues & Note (one out of 25 requests uncovered), with a similar analysis we get that the uncovered request for posting a note on an issue of a project may be selected with probability 0.05% (without considering the time spent until valid value combinations are found). Finally, the regarding service-side code coverage shown in Figure 3.5), we observe that RandomWalk behaves like BFS-Fast across most APIs. The only expected exception is in Commits (top-left corner) where clearly the service-side code coverage achieved with RandomWalk is less than BFS and BFS-Fast.

Overall, both controlling the size of `seqSet` when facing broad search spaces due to large APIs with many requests or when reaching greater depths as well as controlling the size of `seqSet` when facing broad search spaces due to complex APIs with request that have many possible value combinations are both key to delivering good code coverage quickly. However, as shown in Figure 3.5, when running long fuzzing sessions (e.g., as long as 48 hours) all search strategies plateau in relatively similar incremental code coverage. In order to go beyond such plateaus when performing stateful REST API fuzzing, one needs to investigate other types of mutations which are discussed extensively in Chapter 5. Nevertheless, the ultimate goal of testing is to find bugs as quickly as possible, and maximizing code coverage should always be perceived as heuristic towards reaching that goal. Therefore, next, we investigate the performance of the three search strategies explicitly with respect to the number of bugs found in a limited time frame.

Search strategies: Bugs found (Q4)

Lastly, we discuss the number of uniques bugs found by each search strategy across each API family of three cloud services. We cosider as bug each `500` "Internal Server Error" incident triggered after executing a sequence and we cluster similar instances according to the bucketization scheme described in Chapter 3.4. Table 3.4 shows the sets of bug buckets found by each search strategy after 5 hours and it also shows the union and intersection of the respective sets to obtain a perspective of complementarity (and of overlap) regarding the bug-finding capability of each search strategy.

Although previously in this section we reported experiments with longer fuzzing sessions (up to 48h), here we report experiments with shorter sessions in order to

Target Service	**API Family**	**BFS**	**BFS-Fast**	**Random-Walk**	**Intersection**	**Union**
GitLab	Commits	5	1	5	1	5
	Brances	7	7	7	5	8
	Issues & Notes	0	1	1	0	1
	User Groups	0	0	2	0	2
	Projects	2	1	3	1	3
	Repos & Files	2	3	3	2	3
Mastodon	Accounts & Lists	0	0	0	0	0
	Statuses	1	1	0	0	1
Spree	Storefront Cart	1	1	1	1	1
Total		18	15	22	10	24

Table 3.4: **Bug buckets found by BFS, BFS-Fast, and RandomWalk.** Shows the the number of bug buckets found by each search strategies after 5h.

demonstrate the effectiveness of each search strategy in a limited time-frame. After running each search strategy for 5 hours on each API family, RESTler found 24 new *unique* bugs accross all services and APIs.

RandomWalk stands out in Table 3.4 by finding the most bugs: 22 compared to 18 and 15 for BFS and BFS-Fast respectively. However, it is particularly intriguing that service-side code coverage, as shown in Figure 3.5 for each search strategy in the first five hours, is not always on par with the respective total nummber of bug buckets found by each search strategy. There are two cases.

First, there are APIs where service-side code coverage and number of bug buckets found are largely on par. In specific, in Branches all search strategies have almost identical code coverage and find the same total number of bug buckets; in Issues & Notes BFS-Fast and RandomWalk find one bug, while BFS finds zero, which is consistent with the service-side code coverage of the three strategies; in Repos & Files both BFS-Fast and RandomWalk find three bugs versus two bugs of BFS, which is inline with service-side code coverage because both BFS-Fast and RandomWalk outperform BFS code coverage in the first five hours; in Accounts & Lists all search strategies also have identical code coverage and find the same number (zero) of bug buckets; in Statuses BFS and BFS-Fast find one bug bucket, while RandomWalk finds zero. This is also consistent with service-side code coverage because in the first five hours RandomWalk underperforms and catchs up with BFS and BFS-Fast after approximately 20 hours; finally, in Storefront Cart all search strategies find the same number of bug buckets and their service-side coverage identical as well.

On the other hand, there are APIs where service-side code coverage and number of bug buckets found are not on par and we explain why. In Commits, it is particularly intriguing that RandomWalk and BFS find the highest number of bugs compared to BFS-Fast (five versus one) which is inconsistent with service-side code coverage (see Figure 3.5, top-left, for the first five hours) where RandomWalk never achieves the same code-coverage increment with BFS or with BFS-Fast. In addition, we can see from Table 3.4 that RandomWalk and BFS find exactly the same bug buckets because the number of the union of bug buckets (five) equals the size of the individual sets for RandomWalk and BFS. Although this inversion is counter-intuitive at first sight, it is informative regarding the impact of the trade-off followed by BFS-Fast (which systematically sacrifices full grammar coverage with respect to all possible requests combinations for a given sequence length in order to construct longer sequences).

The fact that BFS-Fast covers more API requests and also produces longer sequences than RandomWalk (see first row of Table 4.1) but still RandomWalk finds more bugs, indicates that particularly in Commit APIs sacrificing full grammar coverage with respect to all possible request combinations has a negative impact on the bug-finding capability of BFS-Fast. To understand why, recall the scenario discussed earlier, where a user edits (e.g., cherry-pick) a commit on a branch of a project (required sequence lenght is four). In this scenario, because BFS-Fast attempts to prune the search space by appending each request to at most one sequence in `SetSeq` for a given sequence length, it ends up discarding all test cases whose third request can create a commit and therefore it fundamentally cannot exercise functionality related to editing commits (nor discover any related bugs). One such Commits bug, found by BFS and RandomWalk but not by BFS-Fast, is discussed in Example 1 of Chapter 3.7. Similar conclusions regarding the BFS-Fast trade-off can be drawn in User Groups and Projects where RandomWalk achieves almost identical service-side code coverage with BFS-Fast (see second row of Figure 3.5) but finds more bugs buckets (two versus zeros for User Groups and three versus one for Projects).

Overall, within the 5-hours time-frame of our experiments, RandomWalk finds more bugs than BFS or BFS-Fast despite the fact that it does not always deliver the best coverage. In the same spirit, although BFS delivers worse coverage than BFS-Fast Projects, the former still finds more bugs (two versus one). Such inversions stress out the fact that code coverage increase should not always dictate the selection of a search strategy: code coverage is a indication of the progress of a search strategy; yet, different search strategies may have complementary value, especially within large search spaces. Additionally, it becomes apparent that pruning the search space early on, by sacrificing grammar coverage with respect to possible request combinations for a given sequence length, may fundamentally limit bug-finding capability, although it benefits scalability and allows for greater sequence length.

Indeed, when testing APIs that have never been fuzzed before, exploring a relatively small depth of the search space (i.e., generating tests cases that consist of a only handful of API requests) while, at the same time, systematically exploring the breadth of the search space defined by the possible API request combinations for the given depth (i.e., not pruning the breadth of the search space early on) is sufficient to achieve non-trivial code coverage, exercise rare service-side code paths, and eventually uncover previously-unknown errors. Armed with this realization, in Chapter 4.4 we present a new search strategy (called BFS-Cheap) which combines the best of both worlds and provides a middle-ground between BFS-Fast and RandomWalk.

3.7 New Bugs Found

During all our fuzzing experiments with RESTler on our local GitLab, Mastodon, and Spree deployments we found 30 previously-unknown, unique bugs that have now been fixed. Furthermore, RESTler found handful of bugs in each of the four proprietary Azure and Office365 cloud services discussed in Chapter 3.5. These bugs range from mis-handled invalid inputs (e.g., using a wrong ID or enum value), executing operations in invalid states (e.g., updating a resource that no longer exists), and inconsistent parameter validations (e.g., using a valid request body with incorrect metadata) and have all been fixed. To give the reader a flavor of the nature of those bugs, here, we describe two representative examples. (See[90;91;92;1] for other examples of bugs found.)

Example 1: Bug in Commits API of GitLab. One of the bugs found by RESTler in the Commits API is triggered when a user tries to cherry-pick a commit to a branch with an empty name. Due to incomplete input validation, an invalid branch name can be passed between two different layers of abstraction as follows: The ruby code that checks if a target branch exists, invokes a native C function whose return value is expected to be either NULL or an existing entry. However, if an unmatched entry type (e.g., an empty string) is passed to the native function, an exception is raised. This exception is unhandled by the higher-level ruby code, and therefore it causes a `500` "Internal Server Error". The bug can be reproduced by (1) creating a project, (2) creating a new branch (in addition to master branch which is created by default), (3) posting a valid commit with action "create" in the branch created in (2), and (4) cherry-picking the commit to a branch whose name is set to the empty string.

Example 2: Bug in Branches API of GitLab. Another bug, found by RESTler in the Branches API, is triggered when a user tries to edit a branch of a recently deleted project. The bug is due to invalid serialization of operations which results in an database entry update using an invalid foreign key of a deleted project. Since the project-id (foreign key) is not present in the respective "projects" table, a *PG::ForeignKeyViolation* exception causes a `500` "Internal Server Error". The bug can be reproduced by (1) creating a project, (2) creating a branch, (3) deleting the project created in (1), and (4) quickly editing the branch of the deleted project.

From the above bug descriptions, we see a two-fold pattern. First, RESTler produces a sequence that exercises the target service deep enough so that it reaches a particular valid "state". Second, while the service is in such a state, RESTler produces an additional request with an unexpected fuzzed value (e.g., an empty string) or an unexpected action (e.g., edit a branch of a recently deleted project). Most bugs found by RESTler require a combination of these two features in order to be found.

3.8 Summary

RESTler is the first automatic tool for stateful fuzzing of cloud services through their REST APIs. RESTler analyzes a OpenAPI specification of a REST API, and generates tests by inferring dependencies among request types and by learning invalid request combinations from the service's responses. We presented empirical evidence showing that these techniques are necessary to thoroughly exercise a service while pruning its large search space of possible request sequences. We also evaluated three different search strategies on three open-source, production-scale cloud services and found tens of bugs and several bugs in each of the four proprietary Azure and Office365 cloud services tested.

Chapter 4

Checking Security Properties of Cloud Service REST APIs

In this chapter, we describe how stateful REST API fuzzing can be extended to capture errors beyond the generic class of unhandled exceptions. We introduce four security rules that define desirable properties of cloud services and describe how RESTler can be extended with active checkers that generate API request sequences to specifically test for violations of these rules.

4.1 Background and Motivation

As explained earlier, the target of this dissertation is cloud services accessible through REST APIs. A REST API is defined as a finite set of requests and each request r is a tuple of the form $\langle a, t, p, b \rangle$ where

- a is an *authentication token*,
- t is the *request type*,
- p is a *resource path*, and
- b is the *request body*.

A request type t is any of the following five REST-allowed values: PUT (create or update), POST (create or update), GET (read, list or query), DELETE (delete), PATCH (update). The resource path p is a string identifying a cloud resource and its parent hierarchy. Typically, p is a (non-empty) sequence matching the regular expression

$$(/\langle\texttt{resourceType}\rangle/\langle\texttt{resourceName}\rangle/)+$$

where `resourceType` denotes the type of a cloud resource and `resourceName` is the specific name of the resource of that type. The last resource named in the path is typically the specific resource that the request tries to create, access, or delete. The request body b may include additional parameters and their values that may be required or optional for the request to be executed successfully.

For instance, here is a request to get the properties of a specific Azure DNS zone[135] (shown on multiple lines):

```
⟨ User-auth-token ⟩ GET
https://management.azure.com/
subscriptions/{subscriptionId}/
resourceGroups/{resourceGroupName}/
providers/Microsoft.Network/
dnsZones/{zoneName}
?api-version=2018-03-01 { }
```

This request is of type GET, its path requires three resource names, namely a `subscriptionID`, a `resourceGroupName`, and a `zoneName`, and its body denoted by `{ }` is empty.

REST API requests of type PUT or POST typically create new resources, while DELETE requests destroy existing resources. A request whose execution creates a new resource of type T is called a *producer* for the resource type T. A newly created resource is represented by its *identifier*, or *id* for short. Because resources are dynamically created, we will sometimes call them *dynamic objects*. A request which requires a resource name of type T in its path or in its body is called a *consumer* for the resource type T. We will sometimes refer to the resource name of type T as the *dynamic object type*. In the Azure DNS zone example above, the GET request shown consumes three resources of type `subscriptions`, `resourceGroups`, and `dnsZones` respectively, but does not produce any new resource.

Inside resource paths or request bodies of individual requests, the user is allowed to specify that some specific values, called *fuzzable values*, are to be chosen randomly among a (small finite) set of specific values. For instance, a user might specify that a given integer value in the body of a request may be, say, either `0`, `10`, `1000000`, or `-10`. Such a set of values is called a *fuzzing dictionary*. Given a request with fuzzable values, a *rendering* of that request denotes a mapping of each fuzzable value to a single concrete value selected in its fuzzing dictionary. Thus, a request with n fuzzable values which can each take k possible values results in n^k possible renderings. A rendering is called *valid* if the execution of the corresponding request returns a valid response (defined in the next paragraph). Users are responsible for identifying values they want to fuzz and their associated fuzzing dictionaries.

We define the *state space* of a service as a directed graph where nodes represent service states and edges are transitions between these. Given a state s of the service, executing a single request r leads to a successor state s': this execution is denoted by $s \xrightarrow{r} s'$. The execution of a request r in a state s is either *valid* if it triggers a `2xx` response, *invalid* if it triggers a `3xx` or `4xx` response, or a `bug` if it triggers a `5xx` response. Given an initial state where no resources exist, the state space of the service reachable from that initial state can be explored by executing *sequences of requests*. Such an exploration is *stateful* when it attempts to explore service states that are reachable only using sequences of multiple requests: earlier requests in a sequence may produce resources that are consumed in subsequent requests in that sequence in order to exercise more requests and reach deeper service states. State-space exploration can be performed using various search strategies, e.g., a systematic Breadth First Search (BFS) or a random search, as explained earlier in Chapter 3. We wil call the default

BFS state-space exploration algorithm the *main driver* of *stateful REST API fuzzing*.

In addition to the generic `5xx`-related bugs that can be detected by baseline stateful REST API fuzzing, in this chapter we also introduce four security rules that capture desirable properties of REST APIs and services. We treat violations of these rules as new classes of bugs. Briefly, these rules are:

- **Use-after-free rule.** A resource that has been deleted must no longer be accessible.
- **Resource-leak rule.** A resource that was not created successfully must not be accessible and must not "leak" any side-effect in the backend service state.
- **Resource-hierarchy rule.** A child resource of a parent resource must not be accessible from another parent resource.
- **User-namespace rule.** A resource created in a user namespace must not be accessible from another user namespace.

Violations of such rules might allow an attacker to hijack cloud resources or bypass quotas (*Elevation-of-Privilege* attack), or to steal information from other users (*Information-Disclosure* attack), or to corrupt the backend service state so that it no longer operates properly (*Denial-of-Service* attack). These rules and the ramifications of such violations are discussed in detail in Chapter 4.2. Furthermore, in Chapter 4.3 we show how RESTler can be extended with active property checkers that est and detect violations of such rules.

4.2 REST API Security Properties

We introduce four security rules that capture desirable properties of REST APIs and services. We illustrate each rule with an example and discuss its security implications. All four rules are inspired by past real bugs in deployed cloud services, which were found either by manual penetration testing or by root cause analysis of customer-visible incidents. Examples of new, previously-unknown bugs we found as rule violations in deployed production Azure and Office-365 services are presented later in Chapter 4.6.

Use-after-free rule. A resource that has been deleted must not be accessible. In other words, after a successful DELETE operation on any resource, any subsequent operation – like read, update, or delete – on that resource must fail.

For example, after issuing a `DELETE` request to URI `/users/user-id1` in order to delete the account with identifier `user-id1`, all subsequent attempts to use `user-id1` must fail and thus return a "`404` Not Found" HTTP status code in their response.

A use-after-free violation occurs when a resource that has been deleted still remains accessible through the API. This must never happen. It is a clear bug that may lead to bypassing resource quotas and corrupting the service backend state.

Resource-leak rule. A resource that was not successfully created must not be accessible, and must not "leak" any associated resources in the backend service state. In other words, if the execution of a `PUT` or `POST` request to create a new resource fails (for any reason), any subsequent operation on that resource must also fail with a `4xx` response. Furthermore, no side-effects associated with successful creation of that resource type must occur in the backend service state and be visible to the user. For instance, a failed-to-be-created resource must not be counted in the user's resource counter towards service quotas, and the name of the failed-to-be-created resource must be reusable by the user.

As an example, after issuing a malformed `PUT` request to create URI `/users/user-id1`, a `4xx` response must be received. Any subsequent request to access (read, update, or delete) this URI must also fail.

A resource-leak violation occurs when a resource that was not successfully created nevertheless "leaks" some side-effect in the backend service state. For instance, the resource may be listed by a subsequent `GET` request, yet it cannot be deleted with a `DELETE` request, or subsequent attempts to re-create this resource return "`409` Conflict" responses. Such violations must never happen, as they may have unintended consequences on the capacity for that resource type (e.g., if resource quota limits are reached and no new resources can be created) and on the performance of the service (e.g., due to unnecessarily large database tables).

Resource-hierarchy rule. A child resource of a parent resource must not be accessible from another parent resource. In other words, if a resource `child` is successfully created from a resource `parent` and identified as such in service resource paths of the form $\langle$`parentType`$\rangle$`/parent/`$\langle$`childType`$\rangle$`/child/`, the `child` resource must not be accessible (i.e., must not be successfully read, updated or deleted) when substituting the `parent` resource by any other parent resource.

For example, after issuing `POST` requests to URIs `/users/user-id1`, `/users/user-id2`, and `/users/user-id1/reports/report-id1` to create users `user-id1`, `user-id2`, and then add report `report-id1` to user `user-id1`, subsequent requests to URI `/users/user-id2/reports/report-id1` must fail since, according to the resource-hierarchy rule, report `report-id1` belongs to user `user-id1` but not to user `user-id2`.

A resource-hierarchy violation occurs when a sub-resource originally created from a parent resource is accessible from a different parent resource with no parent-child relationship. When such violations are possible, an attacker might be able to provide an unauthorized parent object identifier (e.g., `user-id3`), and then steal (read) or hijack (write) an unauthorized child object (e.g., `report-id1`). Resource-hierarchy violations are clear bugs, are potentially dangerous, and must never happen.

User-namespace rule. A resource created in a user namespace must not be accessible from another user namespace. In the context of REST APIs, we consider user namespaces defined by the user token used to interact with the API (e.g., OAUTH token-based authentication[145]).

For example, after issuing a `POST` request to create URI `/users/user-id1` using token `token-of-user-id1`, resource `user-id1` must not be accessible using another token `token-of-user-id2` of another user.

A user namespace violation occurs when a resource created within the namespace of one user is accessible from within the namespace of another user. If such a violation ever occurs, an attacker might be able to execute REST API requests using an unauthorized authentication token, and perform unauthorized operations on resources belonging to another (victim) user.

4.3 Active Checkers for REST API Security Properties

Ideally, the desired property for any REST API is that none of the violations defined in Chapter 4.2 occur. In practice, however, there may be violations and we implement active checkers to monitor for those violations. An *active checker* monitors the state space exploration performed by the main driver of stateful REST API fuzzing and suggests new tests to assert that specific rules are not violated. Thus, an active checker augments the search space by executing new tests targeted at violating specific rules. In contrast, a *passive checker* monitors the search performed by the main driver without executing new tests. Hence, a passive checker does not augment the state space explored by stateful REST API fuzzing.

```
Inputs: seq, global_cache, reqCollection
# Retrieve the object types consumed by the last request and
# locally store the most recent object id of the last object type.
n = seq.length
req_obj_types = CONSUMES(seq[n])
# Only the id of the last object is kept, since this is the
# object actually deleted.
target_obj_type = req_obj_types[−1]
target_obj_id = global_cache[target_obj_type]
# Use the latest value of the deleted object and execute
# any request that type−checks.
for req in reqCollection:
  # Only consider requests that typecheck.
  if target_obj_type not in CONSUMES(req)
    continue
  # Restore id of deleted object.
  global_cache[target_obj_type] = target_obj_id
  # Execute request on deleted object.
  EXECUTE(req)
  assert ''HTTP status code is 4xx''
  if mode != 'exhaustive':
      break
```

Figure 4.1: **Use-after-free checker.** Shows the implementation of the use-after-free checker.

We design active checkers following a modular style, based on two principles:

1. Checkers are independent from the main driver of stateful REST API fuzzing and do not affect its state space exploration.
2. Checkers are independent from each other and generate tests by analyzing the requests executed by the main driver, excluding those executed by other checkers.

We enforce the first principle by running all the checkers whenever the main driver has finished executing a new test case. We enforce the second principle by prioritizing the order of applying checkers based on their semantics, so that they operate on different test cases and do not interfere with each other (more on this later in this section). In what follows, we present implementation details of each checker as well as optimizations to limit state-space explosion.

```
Inputs: seq, global_cache, reqCollection
# Retrieve the object types produced by the whole sequence and by
# the last request separately to perform type checking later on.
seq_obj_types = PRODUCES(seq)
target_obj_types = PRODUCES(seq[−1])
for target_obj_type in target_obj_types:
  for guessed_value in GUESS(target_obj_type):
    global_cache[target_obj_type] = guessed_value
    for req in reqCollection:
      # Skip consumers that don't consume the target type.
      if CONSUMES(req) != target_obj_type:
        continue
      # Skip requests that don't typecheck.
      if CONSUMES(req) − seq_obj_types:
        continue
      # Execute the request accessing the ''guessed'' object id.
      EXECUTE(req)
      assert ''HTTP status code in 4xx class''
      if mode != 'exhaustive':
        break
```

Figure 4.2: **Resource-leak checker.** Shows the implementation of the resource-leakage checker.

Use-after-free checker. The implementation of the use-after-free rule checker is described in Figure 4.1 in python-like notation. The algorithm is called after the main driver executes a DELETE request (see Figure 3.3) and takes three inputs: a sequence `seq` of requests, which is the latest test case executed by the main driver; the global cache of dynamic objects, denoted `global_cache`, which contains the most recent object types and ids for the dynamic objects created so far; and the request collection, denoted `reqCollection`, which is the set of all available API requests.

First, the types of the dynamic objects consumed by the last request are retrieved (line 5) and the id of the last object type, denoted `target_obj_type`, is stored in a temporary variable, denoted `target_obj_id`. Although the last request may be consuming more than one object type, we consider the last type in `req_object_types` as the actual type of the deleted object. (For example, a DELETE request on the URI `/users/userId1/reports/reportId1` consumes two object types (users and reports) but only deletes report objects.) After this initial setup, the for-loop (line 12) iterates over all requests available in `reqCollection` and skips those that do not consume the target object type (line 14). Once a request, `req`, that consumes the target object type is found, the target object id is restored in the global cache of dynamic objects (line 17) and is therefore used by the function EXECUTE (line 19) which executes request `req`. Note that the target object id is repeatedly restored in the global cache because the function EXECUTE uses object ids available in `global_cache` when executing a request. If any of these requests succeeds, line 20 will trigger a use-after-free violation (see Chapter 4.2). or assert that no such violation occurs for the given request sequence .

Finally, in order to limit the number of additional tests generated for each request sequence, the inner loop (optionally) terminates when one request for each target object type is found (line 21). This option is used if the variable `mode` is not set to value `exhaustive`. We present detailed experimental results regarding the impact of this optimization in Chapter 4.5.

Resource-leak checker. The resource-leak rule checker is described in Figure 4.2. The algorithm takes the same three inputs as the use-after-free checker. This checker operates on request sequences executed by the main driver whose last request led to an invalid HTTP status code in the response (see Figure 3.3).

The intuition behind this design decision is that when an invalid status code is returned and the last request was attempting to create one or several new resources (i.e., the last request is a resource producer), the requested dynamic objects must not be created in the backend state; otherwise, a leak occurs: (some of these) dynamic objects may have been created in the backend state yet the user may not have access to these through the API.

Initially, the algorithm identifies the dynamic object types produced by the whole sequence, denoted `seq_obj_types`, and produced by the last request, denoted

```
1  Inputs: seq, global_cache
2  # Record the object types consumed by the last request
3  # as well as those of all predecessor requests.
4  n = seq.length
5  last_request = seq[n]
6  target_obj_types = CONSUMES(seq[n])
7  predecessor_obj_types = CONSUMES(seq[:n])
8  # Retrieve the most recent id of each child object consumed
9  # only by the last request. These are the objects whose
10 # hierarchy we will try to violate.
11 local_cache = {}
12 for obj_type in target_obj_types − predecessor_obj_types:
13   local_cache[obj_type] = global_cache[obj_type]
14 # Render sequence up to before the last request
15 EXECUTE(seq, n−1)
16 # Restore old children object ids that do NOT belong to
17 # the current parent ids and must NOT be accessible from those.
18 for obj_type in local_cache:
19     global_cache[obj_type] = local_cache[obj_type]
20 EXECUTE(last_request)
21 assert ''HTTP status code is 4xx''
```

Figure 4.3: **Resource-hierarchy checker.** Shows the implementation of the resource-hierarchy checker.

`target_obj_types` (lines 4 and 5). The main logic of the algorithm is implemented in three nested for loops. The first loop (line 6) iterates over all object types produced by the last request. The second loop (line 7) iterates over object ids "guessed" for the current object type for which an invalid HTTP status code was received. The function GUESS takes as argument an object type and returns a set of possible object ids matching this type and which were not created successfully. For instance, if the creation of a dynamic object with object type "x" and object id "objx1" fails through the API (according to the response received), the checker will attempt to execute any request that consumes the object type "x" and assert it fails when using the object id "objx1". Note that the total number of guessed values per object id is limited to a user-provided parameter value in order to avoid an explosion in the number of additional tests.

In line 8, a guessed object-id value is temporarily added to the global cache of properly-created dynamic objects. Then the inner loop (line 9) iterates over all requests in `reqCollection` to find requests that are executable (given the object types produced by the current sequence) and that consume the given target object type. These requests are executed (line 17) using the "guessed" object ids previously registered in the global cache. This way, the algorithm tries to trigger a resource-leak violation (see Chapter 4.2) or asserts that no such violation occurs for the given sequence (line 18).

Finally, in order to limit the number of additional tests generated for each input sequence, the inner loop (optionally) terminates when one request for each guessed object is found (line 19). We evaluate this optimization in Chapter 4.5.

Resource-hierarchy checker. The implementation of the resource-hierarchy rule checker is described in Figure 4.3. The algorithm takes two inputs: a sequence of requests, denoted `seq`, which is the latest test case executed by the main driver and the current global cache of dynamic objects, denoted `global_cache`.

First, the algorithm records the object types consumed by the last request of the current sequence, denoted `target_obj_types` (line 6), and the object types consumed by all other requests of the sequence before the last request, denoted `predecessor_obj_types` (line 7). Afterwards, the ids of the objects consumed only by the last request are stored locally (lines 12 and 13). These are the child objects whose hierarchy the checker will try to violate by executing requests that try to access them using invalid parent objects. To this end, in line 15, the current sequence is executed up to (and not including) the last request. Finally, the old child object ids are restored (lines 18 and 19) and the last request is executed using the old child object ids on top of new parent object ids (line 20). These parent object ids are not proper parent objects of the restored child object ids. This way, the algorithm tries to trigger a resource-hierarchy violation (see Chapter 4.2) or asserts that no such violation occurs for the given request sequence (line 21).

User-namespace checker. The implementation of the user namespace rule checker

```
 1 Inputs: seq, global_cache
 2 # Because this checker is applied last, we need to re−render the
 3 # current sequence in order to propagate proper object ids.
 4 EXECUTE(seq)
 5 # Retrieve the object types consumed by the whole sequence and
 6 # locally store the most recent object ids of those objects.
 7 target_obj_types = CONSUMES(seq)
 8 local_cache = {}
 9 for obj_type in target_obj_types:
10   local_cache[obj_type] = global_cache[obj_type]
11 for i, req in enumerate(seq):
12   # If not in exhaustive mode, render only the last request.
13   if mode != 'exhaustive' and i != seq.length:
14     continue
15   # Skip requests that are not consumers.
16   if not CONSUMES(req):
17     continue
18   # Reset global cache of object ids and use an alternate (attacker)
19   # token to execute the sequence up to before the last request.
20   global_cache.reset()
21   EXECUTE(seq − req, use_attacker_token)
22   # Restore the object ids belonging to the benign user and try to hijack
23   # them by executing the last request using an attacker token which is
24   # not authorized for those object ids.
25   for obj_type in local_cache:
26     global_cache[obj_type] = local_cache[obj_type]
27   EXECUTE(req, use_attacker_token)
28   assert "HTTP status code is 4xx"
```

Figure 4.4: **User-namespace checker.** Shows the implementation of the user-namespace checker.

is described in Figure 4.4, in python-like notation. The algorithm takes three inputs: a sequence of requests, denoted `seq`, which is the latest test case executed by the main driver; the global cache of dynamic objects, denoted `global_cache`, which contains the most recent object types and ids for the dynamic objects created so far; and an attacker token (an alternate token representing the attacker's identity, which must not have access to the same resources as the token used by the main driver), denoted `attacker_token`.

First, since the user namespace checker is applied last, and it may be affected by other checkers applied previously, the input sequence is re-rendered (line 4) to prevent interference with previously applied checkers, and, in particular, ensure that consistent object ids exist in `global_cache`. Then, the algorithm records the object types produced by the last request of the current sequence, denoted `target_obj_types` (line 7). The object ids of the dynamic objects produced by the current sequence are then stored locally (lines 9 and 10). Afterwards, the outer loop (line 11) iterates over the current sequence until the first request, denoted `req`, which consumes some object type is found (lines 16 and 17).

Note the usual optimization to limit the size of the additional test cases generated for each input sequence (line 14). Once a request `req` that consumes some object type is found, the global cache of object ids is reset (line 20) and the current sequence is executed up to before `req` using the `attacker_token` (line 21). This constructs an attacker namespace containing predecessor dynamic objects. Afterwards, the object ids belonging to the victim user are restored (lines 25 and 26) and `req` is executed using the attacker's identity (line 27).

To hijack the objects belonging to a victim user, `req` is then executed with the attacker's identity after restoring the victim's object ids. If the request succeeds, a user namespace violation (see Chapter 4.2) has occurred for the current sequence (line 28).

Combining All Checkers

The four checkers defined in the previous section are executed as follows. Whenever the stateful REST API fuzzer reaches a new state (as defined in Chapter 4.1), its main driver calls the code shown in Figure 3.3. Depending on the last request executed, this code activates the checkers that are applicable to the current state. We now discuss important properties of these checkers and of their combination.

Extending stateful REST API fuzzing. The checkers *extend* the main driver of baseline *stateful REST API fuzzing* in two ways: (1) they extend the state space by executing additional tests and (2) they check for responses other than `5xx` and can

```
Inputs: seq, global_cache, reqCollection
# Execute the checkers after the main driver.
n = seq.length
if seq[n].http_type == "DELETE":
    UseAfterFreeChecker(seq, global_cache, reqCollection)
else:
if seq[n].http_response == "4xx":
    ResourceLeakChecker(seq, global_cache, reqCollection)
else:
    ResourceHierarchyChecker(seq, global_cache)
    UserNamespaceChecker(seq, global_cache)
```

Figure 4.5: **Checkers dispatcher.** Shows the implementation of the checkers dispatcher.

flag unexpected 2xx responses as rule-violation bugs. Thus, they clearly *increase* the bug-finding capabilities of the main driver: they can find bugs that the main driver alone would not find.

Active property checking versus passive monitoring. As discussed earlier, the checkers we define extend the search space explored by the main driver with additional test cases aimed at triggering and detecting specific rule violations. In contrast, passive runtime monitoring of these rules in conjunction with the main driver, i.e., without executing those new tests, would likely be unable to detect rule violations. Specifically, use-after-free and resource-leak rule violations would likely not be detected with passive monitoring alone because the default state space exploration, performed by the main driver, would likely not attempt to re-use deleted resources or resources after a failure, respectively. Similarly, resource-hierarchy and user-namespace rule violations would not be detected by passive monitoring either because the baseline main driver does not attempt to substitute object identifiers or authentication tokens, respectively. In other words, the additional test cases generated by the checkers are *necessary* to find rule violations and are *not redundant* with respect to non-checker tests.

Complementarity among the checkers. The four checkers we define complement each other: no two checkers will ever generate the same new tests, by construction, because their preconditions are all mutually exclusive. First, the use-after-free checker is the only checker activated by request sequences that end in a DELETE request. Second, the resource-leak checker is the only checker activated when the last request executed returns an invalid HTTP status code. Third, the resource-ownership checker is the only checker activated on request sequences with valid renderings that do not end in a DELETE request. Fourth and last, the user-namespace checker executed tests using an attacker token different from the authentication token used by the main driver and all other checkers, so it clearly extends the state space in an orthogonal dimension.

4.4 Search Strategies for Active Checkers

The main search strategy used for test generation in stateful REST API fuzzing is a Breadth First Search (BFS) in the state space defined by all possible request sequences. This search strategy provides full grammar coverage both with respect to all possible renderings of each individual request and with respect to all possible request sequence combinations of up to a given sequence length. However, since the search space explored by BFS is typically enormous, the search does not scale well as the sequence length increases. On the other hand, RandomWalk and BFS-Fast (discussed in Chapter 3.3) scale better than BFS (as shown in Chapter 3.6) but do not explore all request sequences of a given sequence length. Unfortunately, this limits the number of violations the security checkers can actively check for . To alleviate this, we introduce a new search strategy, called **BFS-Cheap**.

BFS-Cheap follows the inverse trade-off of BFS-Fast: it sacrifices full coverage of all possible request renderings at every state but explores all possible request sequences for a given sequence length, albeit not with all possible renderings. Specifically, given a set of sequences of length n, called `seqSet`, and a set of requests, called `reqCollection`, BFS-Cheap appends each `req` $\in$ `reqCollection` at the end of `seq`, executes the new sequence while considering the possible renderings of `req`, and adds to `seqSet` at most one valid (if any) and one invalid (if any) sequence renderings.

Valid renderings are used by the use-after-free, resource-hierarchy, and user-namespace checkers, while invalid renderings are used by the resource-leak checker. In practive, BFS-Cheap corresponds to one simple change in the algorithm of Figure 3.3: In line 31 of function RENDER we add a break statement that will stop the continuous rendering as soon as the last request of each test sequence has been executed with one valid and one invalid rendering.

BFS-Cheap thus provides a middle-ground between BFS and BFS-Fast (see Chapter 4.5 for experimental evaluation). It explores all possible request sequences up to a given sequence length (like BFS) and adds at most two new renderings for each sequence in order to avoid an enormous `seqSet` (like BFS-Fast). Two new renderings per sequence explored allow for active checking of all the security rules defined in Chapter 4.2 while maintaining a tractable number of sequences in `seqSet` as sequence length increases.

The suffix "cheap" comes from the fact BFS-Cheap is a cheaper version of BFS where at most one valid rendering is added to the BFS "frontier" `setSeq` for each new sequence. This leads to the creation of fewer resources than those created when all valid renderings of each request sequence are explored, as in BFS. For instance, imagine a request definition with an enum type describing ten different flavours of the same resource type. BFS-Cheap will stop creating resources once one resource of one flavour is successfully created. In contrast, BFS and BFS-Fast, will create ten resources of the same type with ten different flavours.

Bug Bucketization

In the context of active checkers, we define "bugs" as rule violations. Each bug is associated with the request sequence that was executed to trigger it. Given this property, we use the following procedure to create *per-checker* bug buckets:

> Whenever a new bug is found, compute all non-empty suffixes of the request sequence that triggers the bug, starting with the smallest one. If a suffix exists in a previously-recorded bug bucket, add the new sequence to that existing bug bucket. Otherwise, create a new bug bucket for the new sequence.

This bug bucketization scheme is the same as the one in Chapter 3 but, here, we maintain separate, *per-checker* bug buckets because the failure conditions are defined differently for each rule. Each bug will always be triggered by one checker for a specific sequence length (because of checker complementarity), except for "`500` Internal Server Error" bugs which may be triggered by both the main driver and checkers. For `500` bugs, each sequence will be added only once to the bug bucket of either the main driver or checker that triggered it first.

4.5 Evaluation

In this section, we report results of experiments with three proprietary and there open-source cloud services. First, we compare the three search strategies described in Chapter 4.4 and then we present results showing the number of rule violations reported by each checker on the three proprietary cloud services as well as the impact of various optimizations introduced in Chapter 4.3.

Experimental Setup

We experiment with two kinds of services, described next: three proprietary Azure and Office365 services, and three open-source cloud services.

Proprietary cloud services. We report results of experiments performed with three cloud services, whose names are anonymized (to avoid targeting them): Azure A and

Azure B are two Azure[12] resource management services, and O-365 C is an Office365[28] messaging service. The number of requests in the REST API of each of these three services ranges from 13 to 19 requests. We selected those three services because their size and complexity are representative among the cloud services we analyzed. So far, we have performed similar experiments with about a dozen production services, and our general experience with these other services is summarized in Chapter 4.6.

Every service discussed here has a publicly-available OpenAPI specification[13]. For each service, we compile its specification to produce a RESTler test-generation grammar. For a given service and API, the same grammar and fuzzing dictionaries were used across all the experiments reported in this section. There is no randomness in the renderings generated. Each fuzzing session lasts one hour and we use a PC connected to the internet and a valid service subscription that allows access to each service API. No other special test setup or service knowledge was required. As in Chapter RESTler-sec:restler:evaluation, we use a garbage-collector that deletes no-longer-used resources (dynamic objects) in order to avoid exceeding service quota limits.

Since we fuzz production services already deployed and accessible to anyone with a valid subscription, we have no visibility to what happens inside the backend of each service. Our fuzzing 1-h fuzzing sessions only observe the HTTP status codes of the responses it receives. All client-side requests are sent over the internet to the target services, and responses are parsed when received. We do not control the deployment of these services. Hence, the experiments reported are not fully controlled. Yet, we performed these experiments several times and the results did not vary significantly.

Open-source cloud services. We also report results on APIs of three open-source cloud services, namely GitLab, Mastodon, and Spree. The characteristics of these APIs are show in Table 3.1 of the previous chapter. Since these services are open-source, we use local deployments whose configuration is described in detail in Chapter 3.6 and we

API	Total Req	Search Strategy	Max Len	Tests	Main	Checker Stats			
						UseAftFr	Leak	Hier	NameSp
Azure A	13	**BFS**	3	3255	48.1%	11.5%	1.5%	0.1%	38.8%
		BFS-Chp	4	4050	55.0%	10.0%	0.8%	2.4%	31.8%
		BFS-Fast	9	4347	59.2%	15.5%	0.2%	0.1%	25.1%
Azure B	19	**BFS**	5	7721	46.4%	3.6%	0.4%	0.2%	49.4%
		BFS-Chp	5	7979	46.2%	3.5%	0.4%	0.2%	49.7%
		BFS-Fast	40	17416	65.3%	0.3%	0.0%	0.1%	34.3%
O-365 C	18	**BFS**	3	11693	89.4%	0.0%	1.0%	0.1%	9.5%
		BFS-Chp	4	10982	95.9%	0.0%	0.0%	0.1%	4.0%
		BFS-Fast	33	18120	66.9%	0.0%	0.0%	0.1%	33.0%

Table 4.1: **Comparison of *BFS*, *BFS-Fast* and *BFS-Cheap* on proprietary cloud services.** Shows the maximum sequence length (Max Len.), the number of requests sent (Tests), and the percentage of tests generated by the main driver (Main) and by the four checkers individually with each search strategy after 1 hour of fuzzing. The second column shows the total number of requests in each API family.

fuzz each one for one hour. Although we have not yet found any violations of security properties on the above open-source cloud services, we present experimental evaluation regarding various performance optimizations of active checkers in order to gain a more comprehensive insight across both proprietary and open-source services.

Comparing Search Strategies

In this subsection, we compare BFS-Cheap against and BFS-Fast. First, we report results on proprietary cloud services, and then, we complement these results with similar experiments on open-source cloud services.

Proprietary cloud services. Table 4.1 shows individual experiments with the three search strategies on each service, over a fixed time budget of one hour per experiment. For each experiment, we report the total number of requests in the API (Total Req.), the maximum sequence length generated (Max Len.), the total number of requests sent (Tests), the percentage of the requests sent by the main driver (Main) as well as the individual contribution of each checker.

API	**Total Req**	**Search Strategy**	**Max Len**	**Tests**	**Main**	**Checker Stats**			
						UseAftFr	**Leak**	**Hier**	**NmSp**
Commits	15 (*11)	**BFS**	4	8300	26.2%	45.7%	11.7%	2.6%	13.6%
		BFS-Chp	5	4867	55.5%	17.0%	7.47%	0.8%	19.0%
		BFS-Fast	7	3964	45.9%	11.2%	0.98%	15.1%	26.6%
Branches	8 (*2)	**BFS**	5	5848	42.2%	7.98%	11.3%	2.4%	35.9%
		BFS-Chp	5	5545	53.9%	4.47%	11.0%	1.7%	28.7%
		BFS-Fast	21	4657	54.5%	1.07%	0.27%	0.25%	43.8%
Issues/Nts	25 (*20)	**BFS**	2	1711	30.2%	6.31%	50.4%	0.0%	12.9%
		BFS-Chp	4	5519	18.8%	70.6%	0.41%	0.5%	9.58%
		BFS-Fast	3	3255	37.3%	3.3%	25.4%	9.03%	24.9%
Groups	53 (*2)	**BFS**	2	957	65.7%	0.10%	0.0 %	0.0%	34.1%
		BFS-Chp	3	1314	57.0%	1.90%	0.0 %	0.0%	41.0%
		BFS-Fast	4	1230	60.8%	0.48%	0.0 %	1.7%	36.9%
Projects	54 (*5)	**BFS**	2	856	99.1%	0.11%	0.0 %	0.0%	0.70%
		BFS-Chp	3	1379	57.3%	7.75%	0.0 %	0.0%	34.8%
		BFS-Fast	4	3173	55.0%	0.37%	0.0 %	6.7%	37.8%
Repos/Files	12 (*22)	**BFS**	3	16637	16.1%	8.76%	71.0%	0.0%	4.03%
		BFS-Chp	5	7219	65.7%	23.4%	2.46%	0.2%	8.13%
		BFS-Fast	4	2167	28.8%	11.8%	24.3%	4.9%	30.0%
Acc./Lists	26 (*3)	**BFS**	3	62322	21.3%	0.0 %	78.6%	0.0%	0.0 %
		BFS-Chp	5	22443	77.9%	0.10%	19.7%	2.2%	0.0 %
		BFS-Fast	14	28007	66.6%	0.06%	11.5%	21.%	0.0 %
Statuses	18 (*19)	**BFS**	2	96640	10.6%	0.0 %	89.3%	0.0%	0.0 %
		BFS-Chp	4	18247	96.1%	0.07%	2.93%	0.8%	0.0 %
		BFS-Fast	8	69280	24.1%	0.00%	75.3%	0.48%	0.0 %
Cart	8 (*11)	**BFS**	4	11673	98.3%	0.0 %	0.0 %	1.6%	0.0 %
		BFS-Chp	5	4527	92.0%	2.38%	0.0 %	5.5%	0.0 %
		BFS-Fast	6	1702	72.1%	9.51%	0.0 %	18.3%	0.0 %

Table 4.2: **Comparison of *BFS*, *BFS-Fast* and *BFS-Cheap* on open-source cloud services.** Shows the maximum sequence length (Max Len.), the number of requests sent (Tests), and the percentage of tests generated by the main driver (Main) and by the four checkers individually with each search strategy after 1 hour of fuzzing. The second column shows the total number of requests in each API family as well as the average number of primitive value combinations for the requests of each API family.

Table 4.1 clearly shows that, for all services, BFS reaches the smallest depth, BFS-Fast reaches the largest depth, and BFS-Cheap provides a trade-off between these two extremes, while being closer to BFS than BFS-Fast. The total number of tests generated varies across services, depending on the speed of the responses received from each service. For any given service, this number remains roughly similar except for BFS-FAST with Azure B and O-365 C where the total number of tests increases significantly. For O-365 C, this increase seems to be due to a significantly lower number of failed requests generated by BFS-FAST for these two services compared to BFS and BFS-Cheap. Such failed requests are sent back to the client (our fuzzer) with larger time delays. Delaying responses to failed requests is a well-known mechanism used by services to *throttle* future requests (i.e., to try to slow them down). On Azure B, BFS-Fast executes more tests because its request sequences are deeper but include many DELETE requests which are faster to execute (their responses are received almost instantly): BFS-Fast executes about 9 times more DELETE requests than BFS or BFS-Cheap.

The total percentage of checker tests (Checkers) is the highest for BFS and the lowest for BFS-FAST, while BFS-Cheap is again in between. Indeed, while BFS-Fast generates the largest number of tests, its search space is pruned and activates checkers less often, as discussed in Chapter 4.4—this is the precise motivation for introducing BFS-Cheap in that section. An exception is the 33% spike in checker-generated tests by BFS-FAST for O-365 C. This spike seems to be due to a larger number of successful requests (see the previous paragraph), which in turn led to more checker tests.

From the individual checker statistics in Table 4.1, we observe that the number of tests they each generate varies from service to service. This number depends on the number of `DELETE` requests executed for the use-after-free checker, the number of failed resource-creation requests for the resource-leak checker, and the depth of the object hierarchy for the resource-hierarchy checker. In contrast, the user-namespace checker is triggered more consistently more often and contributes the largest percentage of checker-generated tests.

Open-source cloud services. Table 4.2 shows experiments with nine API families of GitLab, Mastodon, and Spree for the three search strategies over a fixed time budget of one hour per experiment. For each experiment, we report the total number of requests in the API (Total Req.) as well as the average number of available primitive value combinations for each request, the maximum sequence length generated (Max Len.), the total number of requests sent (Tests), the percentage of the requests sent by the main driver (Main) and the individual contribution of each checker.

First, across most APIs BFS-Cheap provides a middle-ground between BFS and BFS-Fast, except for Issues & Notes and Repos & Files. This is expected because of the design trade-off the two optimizations follow. Recall from Chapter 3.3 that BFS-Fast provides full grammar coverage with respect to all possible value renderings of each individual request but sacrifices full grammar coverage with respect to all possible request combinations of a given sequence length. Whereas, BFS-Cheap follow the inverse trade-off. The two particular APIs (i.e., Issues & Notes and Repos & Files) where BFS-Cheap manages to construct deeper sequences than BFS-Fast have one common characteristic: they both have a relatively higher average number of possible value combinations for each individual request type. Consequently, BFS-Fast, which provides full grammar coverage with respect to primitive value combinations of individual requests, ends up exercising a broader space and lacks in depth achieved. By contrast, the breadth of BFS-Cheap for each individual request is limited to two (one valid and one invalid rendering), and thus, a relative better depth is achieved.

Second, we compare the three search strategies regarding the total number of tests generated from the main driver versus from the checkers. BFS-Cheap, in comparison with BFS, allows for relatively more checker tests in Issues & Notes, User Groups, Projects, and Storefront Cart. In Issues & Notes BFS-Cheap leads to more use-after-free tests (70% versus 6%); in User Groups BFS-Cheap lead to more namespace tests (41% versus 34%); in Projects it leads to more use-after-free and more namespace tests (7% versus 0.1% and 34% versus 0.7% respectively); and in Storefront Cart it leads to

API	**Total Req**	**Mode**	**Statistics**		**Bug Buckets**				
			Tests	**Main**	**Main**	**UseAftFr**	**Leak**	**Hier**	**NmSp**
Azure A	13	optimized	4050	55.0%	4	3	0	0	0
		exhaustive	2174	44.5%	4	3	0	0	0
Azure B	19	optimized	7979	46.2%	0	0	1	0	0
		exhaustive	9031	36.1%	0	0	1	0	0
O-365 C	18	optimized	10982	95.9%	1	0	0	1	0
		exhaustive	11724	88.6%	0	0	0	1	0

Table 4.3: **Comparison of modes *optimized* and *exhaustive* of BFS-Cheap on proprietary cloud services.** Shows the number of requests sent in 1 hour (Tests) with BFS-Cheap, the percentage of tests generated by the main driver, and the number of bug buckets found by the main driver and each of the four checkers. *Optimized* finds all the bugs found by *exhaustive* but its main driver explores more states faster given a fixed test budget (1 hour).

more use-after-free and hierarchy tests (2% versus 0% and 5% versus 1% respectively). BFS-Cheap, in comparison with BFS-Fast, allows for relatively more checker tests in Branches, Issues & Notes, and User Groups. In Branches BFS-Cheap leads to more use-after-free and leakage checks (4% versus 1% and 11% versus 0.2% respectively); in Issues & Notes BFS-Cheap leads to more use-after-free checks (70% versus 3%); and in User Groups BFS-Cheap leads to more use-after-free and namespace checks (1.9% versus 0.4% and 41% versus 36% respectively);

Overall, the number of additional tests generated by the checkers is non-trivial since the checkers actively extend the state space explored by stateful REST API fuzzing. Yet, these numbers would have been much more imbalanced (more checker tests) without the optimizations discussed at the end of Chapter 3.3. Next, we present experimental results regarding the impact of these optimizations on the total number of tests produces by each checker versus the main driver.

API	Total Req	Search Strategy	Tests	Main	Checker Stats			
					UseAftFr	Leak	Hier	NmSp
Commits	15 (*11)	optimized	4867	55.5%	17.0%	7.4%	0.8%	19.0%
		exhaustive	6144	43.4%	15.3%	11.0%	0.6%	29.5%
Branches	8 (*2)	optimized	5545	53.9%	4.47%	11.0%	1.7%	28.7%
		exhaustive	6614	33.3%	6.1%	25.9%	0.5%	33.9%
Issues/Nts	25 (*20)	optimized	5519	18.8%	70.6%	0.4%	0.5%	9.5%
		exhaustive	6853	11.9%	78.2%	0.3%	0.4%	9.1%
Groups	53 (*2)	optimized	1314	57.0%	1.9%	0.0 %	0.0%	41.0%
		exhaustive	1426	45.8%	6.1%	0.0 %	0.0%	48.0%
Projects	54 (*5)	optimized	1379	57.3%	7.7%	0.0 %	0.0%	34.8%
		exhaustive	5072	15.0%	67.6%	0.0 %	0.0%	17.3%
Repos/Files	12 (*22)	optimized	7219	65.7%	23.4%	2.4%	0.2%	8.1%
		exhaustive	9363	41.1%	33.1%	8.1%	0.4%	17.1%
Acc./Lists	26 (*3)	optimized	22443	77.9%	0.1%	19.7%	2.2%	0.0%
		exhaustive	56900	26.7%	0.1%	72.1%	0.8%	0.0%
Statuses	18 (*19)	optimized	18247	96.1%	0.0%	2.9%	0.8%	0.0%
		exhaustive	21986	84.2%	0.8%	14.2%	0.7%	0.0%
Cart	8 (*11)	optimized	4527	92.0%	2.3%	0.0 %	5.5%	0.0%
		exhaustive	4527	92.0%	2.3%	0.0 %	5.5%	0.0%

Table 4.4: **Comparison of modes *optimized* and *exhaustive* of BFS-Cheap on open-source cloud services.** Shows the number of requests sent in 1 hour (Tests) with BFS-Cheap, the percentage of tests generated by all four checkers combined (Checkers), and the number of bug buckets found by the main driver and each of the four checkers. The second column shows the total number of requests in each API family as well as the average number of primitive value combinations for the requests of each API family. Overall, across all API families, in mode *optimized* the proportion of tests generated by the checkers decreases compared to the respective exhaustive explorations.

Comparing Checker Optimizations

Finally, We now compare the performance of the checker optimizations (two modes: *optimized* and *exhaustive*) discussed in Chapter 4.3. We report results on proprietary cloud services and on open-source cloud services.

Proprietary cloud services. Table 4.3 shows how many requests were sent in one hour of fuzzing with BFS-Cheap in the *Tests* column, and what percentage of those requests were generated by either the main driver of Chapter 3.3. The table also shows how many unique bugs (bug buckets) were found in one hour of search by the main driver and by each of the checkers. Results are presented for both the optimized and the exhaustive modes previously discussed.

We observe that the number of tests varies for different services and checker modes. However, the percentage of tests generated by the checkers is always higher with the exhaustive mode, as expected. Since in the optimized mode the checkers produce fewer tests per visited state, the main driver is allowed to explore more states faster. Yet, despite the lower number of checker tests per visited state, for all three services considered, the optimized mode finds all the unique bugs (bug buckets) found by the exhaustive mode. Also, for the O-365 C service, the main driver finds one more bug with the optimized mode compared to the exhaustive mode within one hour of search.

Table 4.3 reveals an interesting inversion that further demonstrates the value of the *optimized* checkers mode. In Azure A, we observe that the *optimized* mode produces almost twice as many tests than than the *exhaustive* mode (4050 versus 2174). At first sight, this is counter-intuitive. After a deeper investigation, we discovered that some of the tests produced by the *exhaustive* mode of the user-namespace checker have significantly larger response times for service Azure A. Indeed, this specific checker in *exhaustive* mode executes additional tests compared to the *optimized* mode, but containing expensive operations (i.e., high latency) that slow down the overall throughput.

During the course of all experiments with these three services, we found and reported a total of 7 unique bugs to the developers of those services, including 4 `500` bugs found by the main driver and 3 bugs found by each of the checkers except the user-namespace checker. In the next section, we discuss several interesting bugs found thanks to the checkers introduced in this paper.

Open-source cloud services. Table 4.4 shows how many requests were sent in one hour of fuzzing with BFS-Cheap in the *Tests* column, and what percentage of those requests were generated by either the main driver of Chapter 3.3 or by any of the four checkers. Unlike Table 4.3 of proprietary cloud service, Table 4.4 shows no bug buckets because the checkers found no confirmed bugs within one hour in none of the three cloud services. We make the following observations, regarding the impact of the optimizations on the number of additional tests generated by the checkers.

First, across all APIs, the optimized version of the use-after-free, the resource leakage, and the user-namespace checkers consistently lead to a lower percentage of additional tests generated by the checkers compared to the exhaustive exploration. This is the desired behaviour and is also expected. Recall from Chapter 4.3, that only the use-after-free, the resource leakage, and the user-namespace checkers have optimized versions, while there is no optimization performed on the resource-hierarchy checker. Therefore, as shown in the second column from the end, the difference between the two versions of the experiments (optimized versus exhaustive checkers) for the resource-hierarchy checker is either insignificant or there are some inversions that highlight the complementarity among checkers.

For example, in Commits the optimized version of the resource-hierarchy checker leads to more tests compared to the exhaustive one (0.8% versus 0.6%). Similarly, in Branches the optimized version of the resource-hierarchy checker again leads to more tests compared to the exhaustive one (1.7% versus 0.5%) and in Accounts & Lists the optimized version of the resource-hierarchy checker leads to more tests compared to the exhaustive one (2.2% versus 0.8%). This inversion, specifically for the resource-hierarchy checker (less tests in the exhaustive mode for some of the experiments) is a side-effect of the complementarity among checkers: less pressure added by one checker (i.e., less additional tests), more capacity for additional tests given to another checker in a given time frame.

Second, focusing on Table 4.4 and specifically on the use-after-free checker (first column of checker stats), we observe that the impact of the optimizations is more evident in API families with many requests, such as User Groups (from 6.1% to 1.9%) and Projects (from 67.6% to 7.7%). This happens because in the exhaustive mode of the use-after-free checker Figure 4.1, after deletion of a dynamic resource, each request available in `reqCollection` which consumes the same dynamic resource type (as the deleted resource), is invoked on the deleted request (line 12 to 19). Inevitably, more requests available in `reqCollecctin` will lead to more additional test. By contrast, in the optimized mode the use-after-free checker, the loop of lines 12 to 19 stops, in a partial search, once one request which consumes a matching resource type is executed.

Third, focusing on the resource-leakage checker (second column of checker stats), we observe the overall decrease in the number of test when using the optimized mode (Figure 4.2). However, it is more difficult to identify a more specific pattern based on the characteristics of each API. This is partially due to the nature of the BFS-Cheap search strategy used: the search stops once a valid and an invalid rendering has been found. This scheme makes the number of additional new tests generated from the leakage checker dependent on the ordering of the value combinations rendered for each grammar of each API. Obviously, such ordering is not consistent and comparable across APIs. That is, if the ordering of values is such that 1000 invalid renderings precede a valid one, then the leakage checker will be invoked 1000 additional times (see line 8 of Figure 4.5). On the other hand, if the ordering of values is such that 1 invalid renderings follows a valid one, then the leakage checker will be invoked 1 additional times. Therefore, fine-grained comparisons between APIs are inconsistent.

Finally, regarding the user-namespace checker (last column of checker stats), we observe that the specific property was not applicable on Accounts & Lists, Statuses, and Cart APIs since the checker never kicked in. Furthermore, we observe a relatively similar decrease in the number of checker tests when using the optimized mode across all APIs, excluding Projects. This inversion, on Projects API is, again, a side-effect of the complementarity between the checkers. In particular observe that in the optimized mode, the total number of tests fall from 5072 to 1379 and that, at the same time, the percentage drop in the use-after-free tests is from 67% to 7% while the user-namespace tests increase from 17% to 34%. Clearly, the exhaustive mode of the use-after-free checker was saturating the user-namespace checker (as well as the main driver) by creating too many additional tests.

4.6 New Bugs Found

At the time of this writing, we have used stateful REST API fuzzing extended with active checkers to test nearly a dozen production Azure and Office-365 cloud services of size and complexity similar to the three services used in the previous section. In almost all cases, our fuzzing was able to find about a handful of new bugs in each of these services. About two thirds of those bugs are "`500` Internal Server Errors", and about one third are rule violations reported by our security checkers. We have reported all these bugs to the respective service owners and *all have been fixed*.

We emphasize that, even when the security checkers do not find any bugs, they increase confidence that the rules they check cannot be violated and therefore they increase confidence in the overall service reliability and security.

This section presents examples of real bugs found in deployed Azure and Office-365 services and discusses their security relevance. We anonymize the name of those services and key details to avoid targeting any specific service.

Use-after-free violation in Azure. In an Azure service, we found the following use-after-free violation.

1. Create a new resource R (with a PUT request).
2. Delete resource R (with a DELETE request).

3. Create a new child resource of the deleted resource R and of a specific type (with another PUT request).

This sequence of requests results in a “`500` Internal Server Error”. The Use-after-free checker catches this as (1) it attempts to re-use in Step 3 the deleted resource in Step 2 and (2) the response of Step 3 is different from the expected “`404` Not Found” response.

Resource-hierarchy violation in Office365. In an Office365 messaging service where users can post messages and then reply and edit these, the resource-hierarchy checker detected the following bug.

1. Create a first message `msg-1` (with a request POST /api/posts/msg-1).
2. Create a second message `msg-2` (with a request POST /api/posts/msg-2).
3. Create a reply `reply-1` to the first message (with a request POST /api/posts/msg-1/replies/reply-1).
4. Edit the reply `reply-1` with a PUT request using `msg-2` as message identifier (with a request PUT /api/posts/msg-2/replies/reply-1).

Surprisingly, the last request in Step 4 returns a “`200` Allowed” response while it must have returned a “`404` Not Found” response. This rule violation reveals that the implementation of the API that posts a reply does not analyze the full hierarchy when checking permissions for a reply. Missing hierarchy validation checks are potential security vulnerabilities: an attacker might be able to exploit them to access child objects by bypassing the parent hierarchy.

Resource-leak violation in Azure. In another Azure service, the resource-leak checker triggered the following bug.

1. Create a new resource of type CM and of name X with a specific malformed body (with a PUT request). This returns a “`500` Internal Server Error”, which is already a bug.

2. Get a list of all resources of type CM: the returned list is empty.
3. Create a new resource of type CM with the same name X as in Step 1 with a well-formed body but in a different region (e.g., US-West versus US-Central) with a PUT request.

Unexpectedly, the last request in Step 3 returns a response "`409` Conflict" instead of an expected "`200` Created". This behavior means that the service has reached an inconsistent state: the failed request in Step 1 has left unintended side-effects on the service state. Indeed, the GET request in Step 2 shows that the user view is correct: the CM resource named X attempted to be created in Step 1 has not been created. However, the second PUT request in Step 3 proves that the service still remembers the failed creation of the CM resource named X attempted in the first PUT request of Step 1. This bug is potentially dangerous: an attacker could create an unbounded number of such "zombie" resources by repeating Step 1 using many different names, and exceed his/her official quota since such failed resource creations are (correctly) not counted towards the user' resource quota. Yet, they are clearly remembered (incorrectly) somewhere in the backend service.

Anecdotal Experiences: Eager Resource-Accounting DoS Attack. During the course of our experiments, we found another type of cloud-service security vulnerability by accident. Specifically, we planned to fuzz an Azure service overnight. However, after fuzzing that service for about five hours, we were contacted by the Azure team owning that service: they had detected unusual traffic created by our fuzzing tests and asked us to stop those tests immediately. Indeed, they told us our experiments had unintentionally caused serious health issues to this service. We summarize the security and reliability vulnerability that was determined to be the root cause of the incident which we accidentally triggered.

Our fuzzing system uses a garbage collector to avoid exceeding quotas for the cloud resources created during fuzzing. For instance, if the default quota for a resource type Y is 100, at most 100 resources of that type can be created at any time, and our garbage collector makes sure that the number of live resources never exceeds such quotas by deleting (using a DELETE request) resources that are no longer used. Without garbage collection, our fuzzing tool would typically reach quota limits in minutes and would not be able to continue its state-space exploration.

In the case of the specific Azure service, it turns out that any PUT request to create a resource of a specific type, let us call it IM, returns a response quickly (nearly instantaneously) but actually also triggers other tasks that take minutes to complete in the service backend. Similarly, a DELETE request for a resource of that same type IM also returns quickly but also trigger delete tasks that also take minutes to complete.

The health issue we triggered was due to the way internal service counters, tracking resource usage, were updated when creating and deleting resources of type IM. Specifically, PUT and DELETE requests that create and delete resources of type IM updated counters towards quotas *eagerly* (i.e., without waiting for the several minutes actually needed to fully complete these actions). As a result, an attacker could create-then-delete quickly many resources of type IM without exceeding his/her quota while triggering a huge number of backend tasks—orders-of-magnitude more than the official quota—hence, literally flooding the backend service with an enormous number of tasks. Such a Denial-of-Service attack was accidentally triggered by our fuzzing tool and its garbage collector and is relevant to the intuition behind the Use-after-free checker: that is, once a resource is deleted no side-effect should impact the state of the backend service. However, since such delete operations must usually be asynchronous, similar errors, that can even lead to security vulnerabilities, are difficult to foresee.

A fix to this vulnerability is to update usage counters towards quotas for DELETE requests *only when all delete backend operations have been completed* i.e., minutes later in the case of IM resources. This way, the amount of backend tasks is still linearly bounded by the official quota, since subsequent IM resource-creation PUT requests will be blocked until preceding DELETE requests have been fully completed.

4.7 Summary

We introduced four security rules that capture desirable properties of REST APIs and services, and showed how stateful REST API fuzzing can be extended with active property checkers that automatically test and detect violations of these rules. We implemented active checkers following a modular design and evaluated various performance optimization on three open-source and three proprietary cloud services. Using active property checkers we reported bugs related to rules violations of security properties on proprietary Azure and Office365 services, which were all fixed. Indeed, violations of the four security rules, introduced earlier, are clearly potential security vulnerabilities. So far, all reports have been taken seriously by corresponding service owners.

Chapter 5

Learning-based Mutations and Coverage-guided Feedback for Stateful REST API Fuzzing

In the previous chapters, we described systems that automate stateful REST API fuzzing and extended it with checkers that detect violations of desirable security properties. These systems, in principal, implement grammar-based fuzzing and inherit some of its limitations. In particular, the automatically generated fuzzing grammar rules usually include few values for each primitive type, like strings and numeric values, in order to limit the combinatorial explosion of the fuzzing space. These primitive-type values are either obtained from the API specification itself or from a user-defined dictionary of values. Furthermore, all these values remain static over time, and are not prioritized in any way. These limitations (fuzzing rules with predefined sets of values and lack of feedback) are typical of grammar-based fuzzing beyond REST API fuzzing.

In this chapter, we introduce Pythia[1], a new fuzzer that *augments grammar-based fuzzing with coverage-guided feedback and a learning-based mutation strategy for stateful REST API fuzzing.* Pythia's mutation strategy helps generate grammatically-valid test

[1] Pythia was an ancient Greek priestess who served as oracle, commonly known as the Oracle of Delphi, and was credited for various prophecies.

```
POST /api/projects HTTP/1.1
Content-Type: application/json
PRIVATE-TOKEN: DRiX47nuEP2AR
{"name":"21a8fa"}

HTTP/1.1 201 Created
{"id":1243, "name":"21a8fa", created_at":"2019-11-23T20:57:15",
"creator_id":1, "forks_count":0, "owner":{"state":"active"}}

POST /api/projects/1243/repository/branches HTTP/1.1
Content-Type: application/json
PRIVATE-TOKEN: DRiX47nuEP2AR
{"branch":"feature1"}

HTTP/1.1 201 Created
{"branch":"feature1", "commit":{"id":"33c42b", "parent_ids":[],
"title":"Add README.md", "message":"Add README.md",
"author_name":"admin", "authored_date":"2019-11-23T20:57:18"}

POST /api/projects/1243/repository/commits HTTP/1.1
Content-Type: application/json
PRIVATE-TOKEN: DRiX47nuEP2AR
{"branch":"feature1", "commit_message":"testString",
"actions":[{"action":"create", "file_path":"admin\xd7@example.com"}]}

HTTP/1.1 500 Internal Server Error
{"message":"internal server error"}
```

Figure 5.1: **Pythia test case and bug found.** The test case is a sequence of three API requests testing commit operations on GitLab. After creating a new project (first request) and a new branch (second request), a commit with an invalid file path triggers an unhandled exception.

cases and coverage-guided feedback helps prioritize the test cases that are more likely to find bugs.

5.1 Background and Motivation

To motivate the benefit of Pythia over the purely grammar-based systems discussed earlier, we present a sample REST API test case which, in fact, uncovers a previously-unknown bug in GitLab and explain why such test cases are beyond reach for the tools discussed earlier

Example REST API test case and detected bug. Figure 5.1 shows a sample Pythia test case for GitLab. The test case consists of three request-response pairs and exercises functionality related to version control commit operations. The first request of type POST (line 1) creates a new GitLab project. It has a path without any resources and a body dictionary with a parameter specifying the desired name of the requested project ("21a8fa"). In response, it receives back metadata describing the newly created project, including its unique id (line 6). The second request, also of type POST, creates a repository branch in an existing project (line 10). It has a path using the previously created resource of type "project" and id "1243", and a body dictionary setting the target branch name ("feature1"), such that, the branch can be created within the previously created project. In response (line 15), it receives back metadata describing the newly created branch, including its designated name. Finally, the last request (line 20) uses the latest branch (in its body) and the unique project id (in its path) and creates a new commit. The body of this request contains a set of parameters specifying the name of the existing target branch, the desired commit message ("testString"), and the actions related to the new commit (i.e., creation of a file). However, the relative path of the target file contains an unexpected value "admin\xd7@example.com", which triggers a 500 Internal Server Error (line 26) because the unicode 'x7' is unhandled in the ruby library trying to detokenize and parse the relative file path. We treat "500 Internal Server Errors" as bugs. To generate similar test cases, with unexpected values, one has to decide which requests of a test case to mutate, what parts of each individual request to mutate, and what new values to inject in the mutated parts.

Complexity of REST API testing. The example of Figure 5.1 shows the sequence of events that need to take place before uncovering an error. It highlights the complexity of REST API testing due to the structured format of each API request and because of producer-consumer dependencies between API requests. For example, the second request of Figure 5.1 (line x) must include a properly structured body payload and also use the project id “1243” created by the first request. Similarly, the third request (line x) must include a properly structured body payload and use resources produced by the two preceding requests (one in its path and one in its body). Overall, a REST API test case is *syntactically valid* regarding the syntax of internal parts of individual requests (i.e., request type, path, header, and body) and *semantically valid* regarding producer-consumer dependencies across requests. A syntactically and semantically valid test case is a *grammatically valid*, or just *valid*, test case. Furthermore, each valid test case is a *stateful* sequence of requests, since resources produced by preceding requests may be used by subsequent requests.

Limitations of stateful REST API fuzzing. Stateful REST API fuzzing, introduced in Chapter 3, is a grammar-based fuzzing approach that statically analyzes the documentation of a REST API and generates a *fuzzing grammar* for testing a target service through its REST API. A RESTler fuzzing grammar contains rules describing (i) how to fuzz each individual API request; (ii) what the dependencies are across API requests and how can they be combined in order to produce longer and longer test cases; and (iii) how to parse each response and retrieve ids of resources created by preceding requests in order to make them available to subsequent requests. During fuzzing, each request is executed with various value combinations depending on its primitive types, and the values available for each primitive type are specified in a user-provided fuzzing dictionary. In the example of Figure 5.1, the value of the field “action” in the last request (line 24) will be one of “create”, “delete”, “move”, “update”, and “chmod” (i.e., the available mutations for this enum type) and the value of the field “commit_message” will be one of “testString” or “nil” (the default available mutations for string types). In contrast, the value of the field “branch,” which is a producer-consumer dependency, will always have the value “feature1” created by the previous request. By construction, the set of grammar rules driving stateful REST API fuzzing leads to grammatically valid test cases.

However, stateful REST API fuzzing, and more broadly grammar-based fuzzing, has two limitations. First, the available mutation values per primitive type are limited to a small, fixed number in order to bound an inevitable combinatorial explosion in the number of possible fuzzing rules and values. Second, these static values remain constant over time and are not prioritized in any way.

In the next section, we introduce Pythia, a new fuzzer that augments grammar-based fuzzing with coverage-guided feedback and a learning-based mutation strategy for stateful REST API fuzzing. Pythia’s mutation fuzzing strategy generates grammatically-valid new test cases and coverage-guided feedback is used to prioritize test cases that are more likely to find new bugs.

5.2 Overview of Pythia

Pythia is a grammar-based fuzzing engine to test cloud services through their REST APIs. Since these APIs are structured (see Chapter 5.1), generating meaningful test cases (a.k.a. mutants) is a non-trivial task—the mutants should be grammatically valid to bypass initial syntactic and semantic checks; yet, they must contain some ill-formed inputs to uncover errors. Randomly mutating seed inputs often results in invalid test cases, as we will see in Chapter 5.6. One potential solution is to sample the mutants from the large state space defined by the fuzzing grammar and inject errors to them. However, for complex grammars, like those defined for REST APIs, exhaustively enumerating all valid mutants is infeasible. As a workaround, Pythia first uses a statistical model to learn common usage patterns of target REST APIs from valid seed inputs.

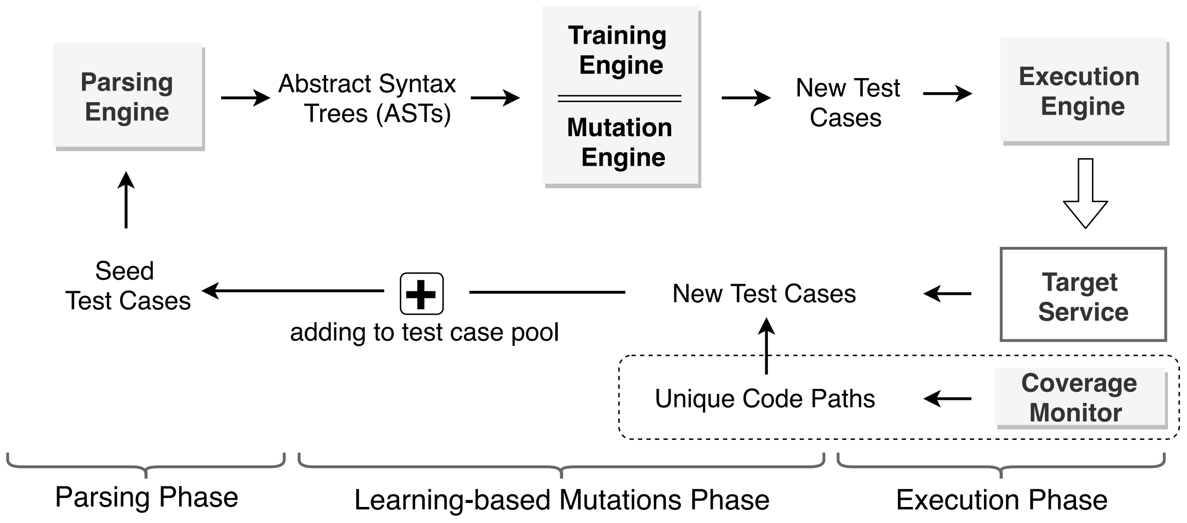


Figure 5.2: **Pythia architecture.** The steps inside the dotted box show optional, add-on features.

Then, it generates new mutants by injecting a small amount of noise causing the learnt model to deviate from the common usage patterns. This small noise results in mutants that deviate from common usage pattens, yet are mostly grammatically valid.

Figure 5.2 presents a high-level overview of Pythia. It iteratively operates in three phases: parsing, learning-based mutations, and execution. Initially, the parsing phase (Chapter 5.3), parses the input test cases using a regular grammar and outputs the corresponding Abstract Syntax Trees (ASTs). Initial input test cases can be generated either by using RESTler to fuzz the target service or by using actual production traffic of the target service. Next, the learning-based mutations phase (Chapter 5.4), operates on ASTs. The training engine trains a sequence-to-sequence (seq2seq) autoencoder[67;170] in order to learn common AST structures of seed test cases. This includes the syntax of individual requests (i.e., primitive types and values) and the dependencies across requests of a given test case. Then, the mutation engine generates new test cases by injecting a small amount of random noise in the trained autoencoder causing it to slightly deviate from common patterns and generate new test cases.

Finally, in the execution phase (Chapter 5.5), new test cases are executed by the

$$
\begin{aligned}
&S = sequence \\
&\Sigma = \Sigma_{http-methods} \cup \ \Sigma_{resource-ids} \cup \ \Sigma_{enum} \\
&\ \cup \ \Sigma_{bool} \cup \ \Sigma_{string} \cup \ \Sigma_{int} \cup \ \Sigma_{static} \\
&N = \{request,\ method,\ path,\ header,\ body,\ \beta_1,\ \beta_2,\ \beta_3, \\
&\quad producer,\ consumer,\ fuzzable,\ enum, \\
&\quad bool,\ string,\ int,\ static\} \\
&R = \{sequence \rightarrow request + sequence \ \mid \ \varepsilon\textbf{,} \\
&\ request \rightarrow method + path + header + body\textbf{,} \\
&\ method \rightarrow \Sigma_{http-methods}\ \textbf{,}\ path \rightarrow \beta_1 + path \ \mid \ \varepsilon\textbf{,} \\
&\ header \rightarrow \beta_1 + header \ \mid \ \varepsilon\textbf{,}\ body \rightarrow \beta_1 + body \ \mid \ \varepsilon\textbf{,} \\
&\ \beta_1 \rightarrow \beta_2 \ \mid \ \beta_3\textbf{,}\ \beta_2 \rightarrow producer \ \mid \ consumer\textbf{,} \\
&\ producer \rightarrow \Sigma_{resource-ids}\textbf{,}\ consumer \rightarrow \Sigma_{resource-ids}\textbf{,} \\
&\ \beta_3 \rightarrow static \ \mid \ fuzzable\textbf{,}\ static \rightarrow \Sigma_{static}\textbf{,} \\
&\ fuzzable \rightarrow string \ \mid \ int \ \mid \ bool \ \mid \ enum \ \mid \ uuid\textbf{,} \\
&\ string \rightarrow \Sigma_{string}, \ldots\}
\end{aligned}
$$

Figure 5.3: **Regular Grammar $\mathcal{G}$ for REST API test case generation.** The production rules of $\mathcal{G}$ with non-terminal symbols capture the properties of any REST API specification, while the alphabet of terminal symbols is API-specific since different APIs may contain different values for strings, integers, enums, and so on.

target service. Pythia uses a simple oracle (tracking "500 Internal Server Errors") to identify which test cases trigger bugs and retain them to facilitate further manual inspection. Moreover, code coverage feedback (when available) obtained by the coverage monitor is used to distinguish test cases that activate unique code paths and prioritize them for further mutations and to augment the initial corpus of seed test cases. Yet, when coverage feedback is not available, Pythia operates as a blackbox fuzzer and all test cases are treated equally.

5.3 Parsing REST API Test Cases

In this phase, Pythia infers the syntax of the seed inputs by parsing them with a user-provided Regular Grammar (RG) with tail recursion. Such an RG is defined by a 4-tuple $\mathcal{G} = (N, \Sigma, R, S)$, where N is a set of non-terminal symbols, Σ is a set of terminal symbols, R is a finite set of production rules of the form $\alpha \rightarrow \beta_1\beta_2 \ldots \beta_n$,

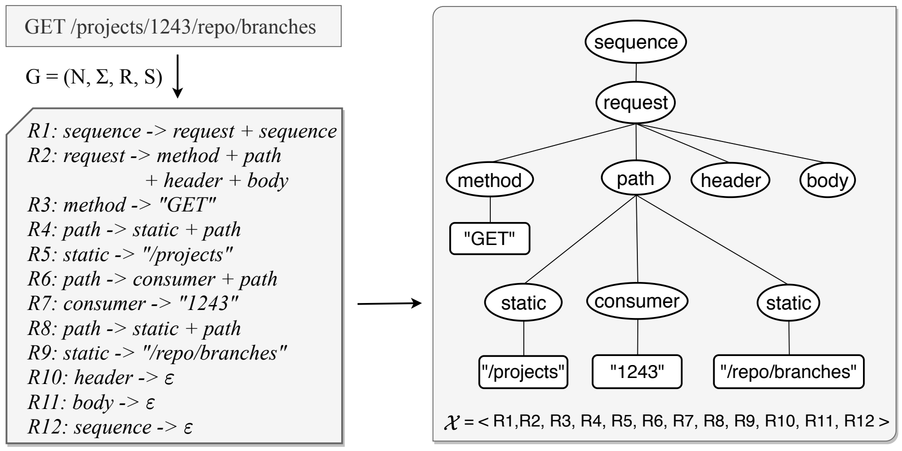

GET /projects/1243/repo/branches
G = (N, Σ, R, S)
R1: sequence -> request + sequence
R2: request -> method + path + header + body
R3: method -> "GET"
R4: path -> static + path
R5: static -> "/projects"
R6: path -> consumer + path
R7: consumer -> "1243"
R8: path -> static + path
R9: static -> "/repo/branches"
R10: header -> ε
R11: body -> ε
R12: sequence -> ε
sequence
request
method
path
header
body
"GET"
static
consumer
static
"/projects"
"1243"
"/repo/branches"
𝒳 = < R1,R2, R3, R4, R5, R6, R7, R8, R9, R10, R11, R12 >

Figure 5.4: **Parsing RESTler seed test case into an AST.** Show how Pythia parses an RESTler seed test case (left) into an AST (right) following rules from grammar $\mathcal{G}$.

where $\alpha \in N, n \geq 1$, $\beta_i \in (N \cup \Sigma), \forall 1 \leq i \leq n$, and $S \in N$ is a distinguished start symbol. The syntactic definition of $\mathcal{G}$ looks like a Context Free Grammar, but because recursion is only allowed on the right-most non-terminal and no other cycles are allowed, the grammar is actually regular. Figure 5.3 shows a template $\mathcal{G}$ for REST API test case generation. A test case that belongs to the language defined by $\mathcal{G}$ is a sequence starting with the symbol *sequence* followed by a successions of production rules (R) with non-terminal symbols (N) and terminal symbols (Σ).

Theorem 1. *The language described by grammar $\mathcal{G}$ (Figure 5.3) is regular.*

Proof. We will prove Theorem 1 by writing a regular expression. We see that $\mathcal{G}$ is equivalent to the regular expression $(\langle A \rangle\ \langle B \rangle\ \langle B \rangle\ \langle B \rangle^*)+$, where

- $\langle A \rangle = (POST \mid PUT \mid GET \mid DELETE \mid PATCH)$
- $\langle B \rangle = (\Sigma_{static} \mid \Sigma_{string} \mid \Sigma_{int} \mid \Sigma_{bool} \mid \Sigma_{enum} \mid \Sigma_{uuid} \mid \Sigma_{resource-ids})$

$\langle A \rangle$ corresponds to all available HTTP request type values; the first $\langle B \rangle$ corresponds to all primitive type values available request headers of the target API; the second $\langle B \rangle$ corresponds to all primitive type values available request paths of the target API; and

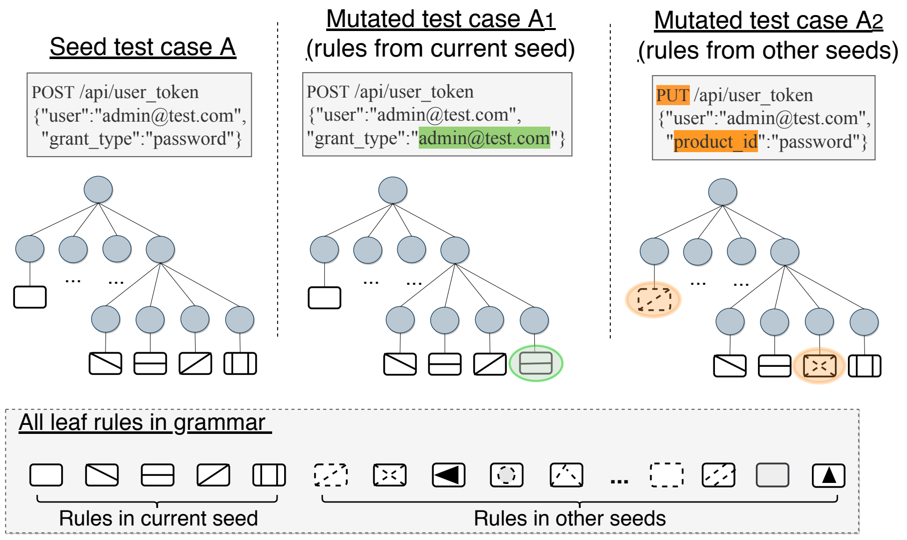

Seed test case A
POST /api/user_token {"user":"admin@test.com", "grant_type":"password"}
Mutated test case A1 (rules from current seed)
POST /api/user_token {"user":"admin@test.com", "grant_type":"admin@test.com"}
Mutated test case A2 (rules from other seeds)
PUT /api/user_token {"user":"admin@test.com", "product_id":"password"}
All leaf rules in grammar
Rules in current seed
Rules in other seeds

Figure 5.5: **Seed test case and new test cases with mutations using values from other seeds (center) or from the original seed (right).** Shows the two cases of learning-based mutations performed by Pythia.

the last (optional) $\langle B \rangle$ corresponds to all primitive type values available request bodies of the target API. Therefore, the language described by $\mathcal{G}$ is indeed regular. $\square$

Figure 5.4 shows how seed RESTler test cases are parsed by Pythia's parsing engine. A successions of production rules in $\mathcal{G}$ are applied to infer the corresponding ASTs; the tree internal nodes are non-terminals and the leaves are terminals of $\mathcal{G}$. Pythia then traverses the tree in Depth First Search (DFS) order and obtains a sequence of grammar rules. For example, a simple test case `X=``GET /projects/1243/repo/branches"` will be represented as a sequence of grammar production rules $\mathcal{X} =< R_1, R_2, \ldots, R_{12} >$ shown at the bottom of the Figure. Given a set of seed inputs, the output of this phase is a set of abstracted test cases, $\mathcal{D} = \{\mathcal{X}_1, \mathcal{X}_2, \ldots, \mathcal{X}_N\}$, which correspond to ASTs of the respective seed inputs. Ultimately, the set of abstracted seed test cases, $\mathcal{D}$, is passed to the training and mutation engines.

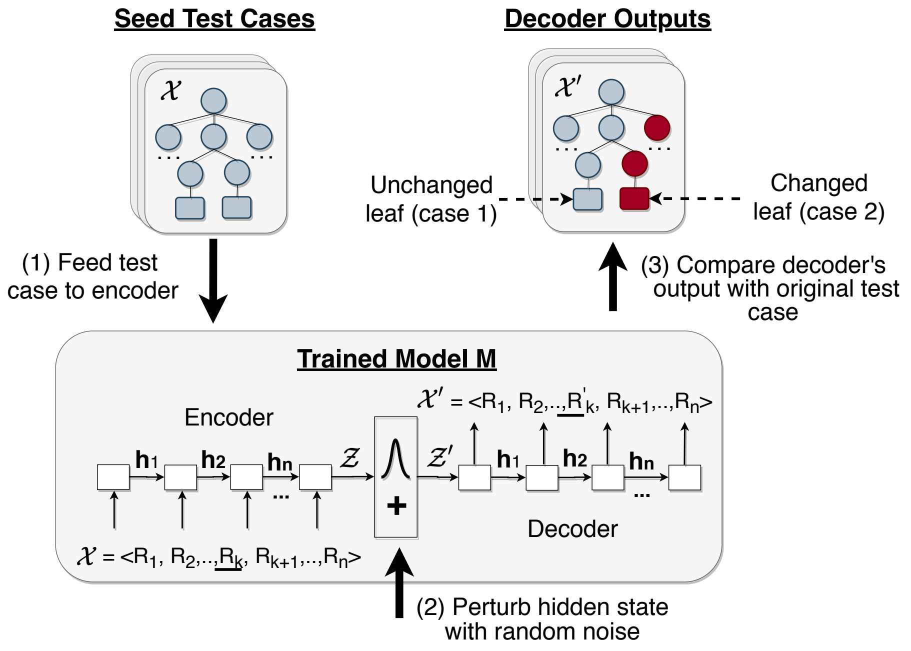

Seed Test Cases
Decoder Outputs
$\mathcal{X}$
$\mathcal{X}'$
Unchanged leaf (case 1)
Changed leaf (case 2)
(1) Feed test case to encoder
(3) Compare decoder's output with original test case
Trained Model M
Encoder
Decoder
$\mathcal{X}' = <R_1, R_2,..,R'_k, R_{k+1},..,R_n>$
$\mathcal{X} = <R_1, R_2,..,R_k, R_{k+1},..,R_n>$
h_1 h_2 h_n
$\mathcal{Z}$ $\mathcal{Z}'$
(2) Perturb hidden state with random noise

Figure 5.6: **Overview of Pythia mutation engine.** Shows an overview of Pythia's mutation engine

5.4 Learning-based REST API Mutations

The goal of this phase is first to use the abstracted seed test cases $\mathcal{X} \in \mathcal{D}$ and learn common AST structures of the target APIs, and then to mutate these common structures and generate new test cases. An autoencoder type of neural network is particularly suitable for this purpose, as it can learn embedded representations of the input AST structures. Then, we can add random noise to the learnt representations, decode them back to the original AST formats, and get new test cases.

Pythia uses an autoencoder model $\mathcal{M}$, which is trained on $\mathcal{D}$. The autoencoder model $\mathcal{M}$ consists of an encoder and a decoder (see Figure 5.6). $\mathcal{M}_{encoder}$ represents each abstracted test case $\mathcal{X}$ to an embedded representation $\mathcal{Z}$ which captures the latent dependencies of $\mathcal{X}$. $\mathcal{M}_{decoder}$ decodes $\mathcal{Z}$ back to $\mathcal{X}'$. To generate learning-based mutations, Pythia minimally perturbs the embedded representation $\mathcal{Z}$ of $\mathcal{X}$ and decodes it back to $\mathcal{X}'$. Our *key insight* is that, since the autoencoder is initially trained on

Algorithm 1: Learning-based Pythia mutations

```
Input: seeds $\mathcal{D}$, grammar $\mathcal{G}$, model $M_{\theta,\mathcal{D}}$, batch $N$
1  while $time\_budget$ do
2      $\mathcal{X} \leftarrow get\_next\_seed(\mathcal{D})$
3      $\mathcal{Z} \leftarrow M_{\theta,\mathcal{D}}.encoder(\mathcal{X})$
4      $new\_sequences \leftarrow \emptyset$
       // Perturbation: Exponential search on random noise scale
5      for $j \leftarrow 0$ to $N$ do
           // Noise draw from normal distribution
6          $\delta_j \leftarrow random.normal(\mathcal{Z}.shape, 0)$
           // Bound and scale random noise
7          $\delta_j \leftarrow 2^j * \delta_j / \|\mathcal{Z}\|_2$
           // Add noise on decoder's starting state
8          $\mathcal{X}'_j \leftarrow M_{\theta,\mathcal{D}}.decoder(\mathcal{Z} + \delta_j)$
9          $new\_sequences.append(\mathcal{X}'_j)$
10     end
       // Select the prediction with smallest noise scale
11     $\mathcal{X}'_{min} \leftarrow \arg\min_{scale} new\_sequences$
       // Case 1: Grammar rules from current seed
12     $rules \leftarrow terminals(\mathcal{X})$
13     foreach $index$ in $get\_different\_leaves(\mathcal{X}, \mathcal{X}'_{min})$ do
14         foreach $rule$ in $rules$ do
15             $mutation \leftarrow rule + random\_bytes$
16             $\mathcal{X}[index] \leftarrow mutation$
17             $EXECUTE(\mathcal{X})$
18         end
19     end
       // Case 2: Grammar rules from other seeds
20     $rules \leftarrow terminals(\mathcal{G}) - terminals(\mathcal{X})$
21     foreach $index$ in $get\_common\_leaves(\mathcal{X}, \mathcal{X}'_{min})$ do
22         foreach $rule$ in $rules$ do
23             $mutation \leftarrow rule + random\_bytes$
24             $\mathcal{X}[index] \leftarrow mutation$
25             $EXECUTE(\mathcal{X})$
26         end
27     end
28 end
```

grammatically valid test cases, most of the generated ASTs will remain grammatically valid even after adding small perturbations on $\mathcal{Z}$. Furthermore, the generated output ASTs will have fewer discrepancies (i.e., differ less) from the respective input ASTs at places where the structure is common across many training inputs and, conversely, more discrepancies (i.e., differ more) from the respective input ASTs at places where the structure is less common across training inputs. Next, we explain how we leverage this insight to generate API-specific learning-based mutations that lead to new grammatically valid new test cases, with sufficiently ill-formed parts which exercise rare code paths and uncover bugs.

Training engine.

Given the abstracted test cases in $\mathcal{D}$, the training engine learns their encoded vector representations using an autoencoder type of neural network[108], which is realized with a simple seq2seq model $\mathcal{M}_{\mathcal{D}}$ trained over $\mathcal{D}$. Usually, a seq2seq model is trained to map variable-length sequences of one domain to another (e.g., English to French). By contrast, we train $\mathcal{M}$ only on sequences of domain $\mathcal{D}$ such that $\mathcal{M}_{\mathcal{D}}$ captures the latent characteristics of test cases.

A typical seq2seq model consists of two Recurrent Neural Networks (RNNs): an encoder RNN and a decoder RNN. The encoder RNN consists of a hidden state $\mathbf{h}$ and an optional output $\mathbf{y}$, and operates on a variable length input sequence $\mathbf{x} =< x_1, \ldots, x_n >$, augmented with two auxiliary tokens `<SOS>` and `<EOS>` marking the beginning and the ending of each sequence respectively. At each time t (which can be thought of as position in the sequence), the encoder reads sequentially each symbol x_t of input $\mathbf{x}$, updates its hidden state $\mathbf{h}_t$ by $\mathbf{h}_t = f(\mathbf{h}_{t-1},\ x_t)$, where f is a non-linear activation function, such as a simple REctified Linear Unit (ReLU) or a more complex a Long Short-Term Memory LSTM) unit [109], and calculates the output y_t by $y_t = \phi(\mathbf{h}_t)$, where ϕ is an activation function producing valid probabilities[57]. At the end of each input sequence, the hidden state of the encoder is a summary $\mathbf{z}$ of the whole sequence. Conversely, the decoder RNN generates an output sequence $\mathbf{y} =< y_1, \ldots, y_{n'} >$ by predicting the next symbol y_t given the hidden state $\mathbf{h}_t$, where both y_t and $\mathbf{h}_t$ are conditioned on y_{t-1} and on the summary $\mathbf{z}$ of the input sequence. Hence, the hidden state of the decoder at time t is computed by $\mathbf{h}_t = f(\mathbf{h}_{t-1},\ y_{t-1}, \mathbf{z})$, and the conditional distribution of the next symbol is computed by $y_t = \phi(\mathbf{h}_t,\ y_{t-1}, \mathbf{z})$, given functions f and ϕ.

We train the seq2seq model $\mathcal{M}$ on $\mathcal{D}$ to maximize the conditional log-likelihood $\arg\max_{\boldsymbol{\theta}} \frac{1}{N}\sum_{i=1}^{N} logp_{\boldsymbol{\theta}}(\mathbf{y}_i|\mathbf{x}_i)$, where $\boldsymbol{\theta}$ is the set of the learnt model parameters and each $\mathbf{x}_i, \mathbf{y}_i \in \mathcal{D}$. As explained earlier, $\mathcal{M}_{\theta,\mathcal{D}}$ is trained on sequences of one domain (i.e., $\mathbf{y} = \mathbf{x}$) and is then used by the mutation engine.

Mutation engine.

For each test case $\mathcal{X} \in \mathcal{D}$, the goal of the mutation engine is to decide how to mutate each input location of $\mathcal{X}$. Since $\mathcal{X}$ is a sequence of grammar rules $< R_1, R_2, \ldots, R_n >$ (see Chapter 5.4), the mutation strategy determines how to mutate each rule. Our learning-based mutation engine sees a rule R_i (in its context) in one of the following ways: either it *has* or *has not* seen R_i's alternatives in the training corpus. In the first case, Pythia mutates R_i with alternative rules from the context of the current seed (Chapter 5.5; center); otherwise, Pythia mutates R_i with randomly selected new rules from other seeds in the training corpus (Chapter 5.5; right). Both cases lead to mutants that are grammatically valid as Pythia operates on the leaf nodes of AST structures.

This mutation strategy is realized by the auto-encoder. To mutate a seed test $\mathcal{X}$, Pythia perturbs its embedded vector representation ($\mathcal{Z}$) by iteratively adding random noise of increasing scale, and then decodes it back to a new test case $\mathcal{X}' =< R'_1, R'_2, \ldots, R'_n >$. The minimum perturbation that creates differences between $\mathcal{X}$ and $\mathcal{X}'$ is selected, and the respective differences determine how to mutate each rule of the seed test case $\mathcal{X}$ as follows:

1. Rules where $\mathcal{R}'_i$ and $\mathcal{R}_i$ differ indicate locations where the model has seen more variance during training and mutations with the rules from the context of the current seed should be used (Figure 5.5; center).
2. Rules where $\mathcal{R}'_i$ and $\mathcal{R}_i$ are the same after the perturbation indicate locations

where the model has not seen many variations in those rules and their context during training, and thus, mutations with new rules, not in the original in input/output sequences, should be used (Figure 5.5; right).

Algorithm 1 implements Pythia's learning-based mutation strategy and Chapter 5.6 pictorially illustrates it. The algorithm takes a set of abstracted test cases $\mathcal{D}$, a regular grammar $\mathcal{G}$, a trained autoencoder model $\mathcal{M}_{\theta,\mathcal{D}}$ and its batch size N as inputs, and iterates over $\mathcal{D}$ until the time budget expires (Line 1). At a high level, the algorithm operates in 2 steps: perturbations and comparison-mutations.

• *Perturbations* (lines 5 to 11): For each test case $\mathcal{X}$, the encoder of model $M_{\theta,\mathcal{D}}$ obtains its embedding $\mathcal{Z}$ and Pythia perturbs it with random noise. Pythia draws N noise-values $\{\delta_0, \delta_1, \ldots, \delta_{N-1}\}$ from a normal distribution, bounded by $2-$norm of $\mathcal{Z}$ and scaled exponentially in the range $\{2^0, 2^1, \ldots, 2^{N-1}\}$. The N noise values are used to perturb $\mathcal{Z}$ independently N times and get different perturbed vectors $\{\mathcal{Z}+\delta_0,\ \mathcal{Z}+\delta_1, \ldots,\ \mathcal{Z}+\delta_{N-1}\}$, which serve as N different starting states of the decoder. In turn, they lead to N different outputs $\{\mathcal{X}'_0,\ \mathcal{X}'_i, \ldots,\ \mathcal{X}'_{N-1}\}$ for each input $\mathcal{X}$. From these N outputs, Pythia selects $\mathcal{X}'_{min}$ which differs from $\mathcal{X}$ and is obtained by the smallest (2-norm) perturbation δ_{min} on $\mathcal{Z}$.

The N-step exponential search for the smallest perturbation that leads to a new output $\mathcal{X}'_{min}$ helps avoid pervasive changes that completely destroy the embedded representation $\mathcal{Z}$ of $\mathcal{X}$. In fact, perturbing the embedded representation $\mathcal{Z}$ of $\mathcal{X}$ with noise and decoding it back to $\mathcal{X}'_{min}$ forces the model to act as a denoising autoencoder[176]. Such models are robust to partial noise destructions, since the learnt representation is expected to capture stable structures of common dependencies in the observed inputs[175]. Hence, the outputs $\mathcal{X}'_{min}$ will be close to the respective inputs $\mathcal{X}$, and many will remain grammatically valid because the model is trained on grammatically valid inputs. Furthermore, because of the variance-bias trade-off[89], the recovered output ASTs will have less discrepancies (lower error) from the respective input ASTs at places where the structure is common across many training inputs (low variance); and conversely, the recovered output ASTs will have more discrepancies (higher error) at places where the structure is less common across training inputs (high variance). We leverage these insights and generate learning-based mutations by comparing $\mathcal{X}$ with $\mathcal{X}'_{min}$ for each test case $\mathcal{X} \in \mathcal{D}$.

- *Comparison & Mutations* (lines 12 to 27): The two groups of nested for-loops in Algorithm 1 implement the two different mutation cases explained earlier. The first group of nested for-loops (lines 13 to 19) targets leaf locations where $\mathcal{X}$ and $\mathcal{X}'_{min}$ differ (high variance, case 1). In such locations of $\mathcal{X}$, new mutations are generated by iteratively applying leaf grammar rules (i.e., terminal symbols) in $\mathcal{X}$. The second group of nested for-loops (lines 21 to 27) targets leaf locations where $\mathcal{X}'_{min}$ and $\mathcal{X}$ are the same (low variance, case 2). In such locations of $\mathcal{X}$, new mutations are generated by iteratively applying leaf grammar rules originally not in $\mathcal{X}$. In both cases, the new grammar rules are augmented with auxiliary random byte alternations on the byte representation of rule terminals to avoid repeatedly exercising identical rule payloads.

5.5 Execution Phase

In this phase, the execution engine takes as input new test cases generated by the mutation engine and executes them to the target service. Executing a test case includes sending its requests to the target service over http(s) and receiving the respective responses back. During such interactions, Pythia uses a simple oracle. to identify which test cases lead to responses indicating bugs to retain them for further manual inspection. Currently, Pythia uses a simple oracle that captures 500s, which are a generic class of errors indicating internal server errors. Yet, in principle, Pythia could replace its oracle with recent tools that capture other types of failures[46].

During testing, Pythia leverages code coverage information, obtained by the coverage monitor, to distinguish test cases that activate unique code paths. To track covered code, Pythia first statically analyzes the target service and extracts basic block locations. Then, the target service is configured to produce code coverage information, which is collected by the coverage monitor. Given the basic block locations extracted and the coverage information collected by the coverage monitor during testing, each test case is mapped to a bitmap describing the respective code path activated. Such code coverage information (feedback) helps distinguishing test cases that reach new code paths and ultimately minimize an initially large corpus of many likely-redundant test cases to a smaller set that entirely consists of test cases activating unique code paths. Yet, if it is infeasible to collect code coverage information for a target service, Pythia operates as a purely blackbox fuzzer and still outperforms prior approaches both in code coverage achieved and in bugs found (see Chapter 5.6).

Implementation of Pythia

Pythia follows a single-threaded *Python* implementation. Although a multi-threaded implementation may increase testing throughput, Pythia targets server-side code and in a multi-threaded environment it would be challenging to disentangle requests from concurrent sequences and reconstruct the exact test cases triggering a bug. We use an off-the-shelf seq2seq RNN with input embedding, implemented in tensorflow[172]. The model has one layer of 256 Gated Recurrent Unit (GRU)[69] cells in both encoder and decoder. Dynamic input unrolling is performed using `tf.nn.dynamic` RNN APIs and the encoder is initialized with a zero state. We train the model by minimizing the weighted cross-entropy loss for sequences of logits using the Adam optimizer[117]. We use batches of 32 sequences, iterate for $2k$ training steps with a learning rate of 0.001, and initial embedding layer of size 100. The vocabulary of the model depends on the number of production rules in the fuzzing grammar of each API family and ranges in couple of hundred “words”. Training such a seq2seq model in a CPU-only machine takes no more than two hours. All the experiments discussed in our evaluations were run on Ubuntu 18.04 Google Cloud VMs[22] with 8 logical CPU cores and 52GB of physical memory. Each fuzzing client is used to test a target service deployment running on the same machine with the fuzzing client.

5.6 Evaluation

In this section, we report results of experiments obtained with Pythia on there open-source cloud services. We answer the following questions:

Q1: How do the three baselines compare with Pythia in terms of code coverage increase over time?

Q2: How does initial seed selection impact the code coverage achieved by Pythia?

Q3: How does code coverage feedback impact the code coverage achieved by Pythia?

Q4: Can Pythia detect new bugs in production-scale cloyud services?

Experimental setup.

In total, we tested six API families of GitLab[93], two of Mastodon[131], and one of Spree[168]. These API families and services are the same with the ones used the previous chapters (see Table 3.1). We also use here the same configurations as the ones described in Chapter 4.5. In principal, the total number of requests in each API family along with the average number of available primitive value combinations for each request indicate the size of the state space that needs to be tested.

Monitoring framework & initial seeds. We statically analyze the source code of each target service to extract basic block locations and configure each service, using

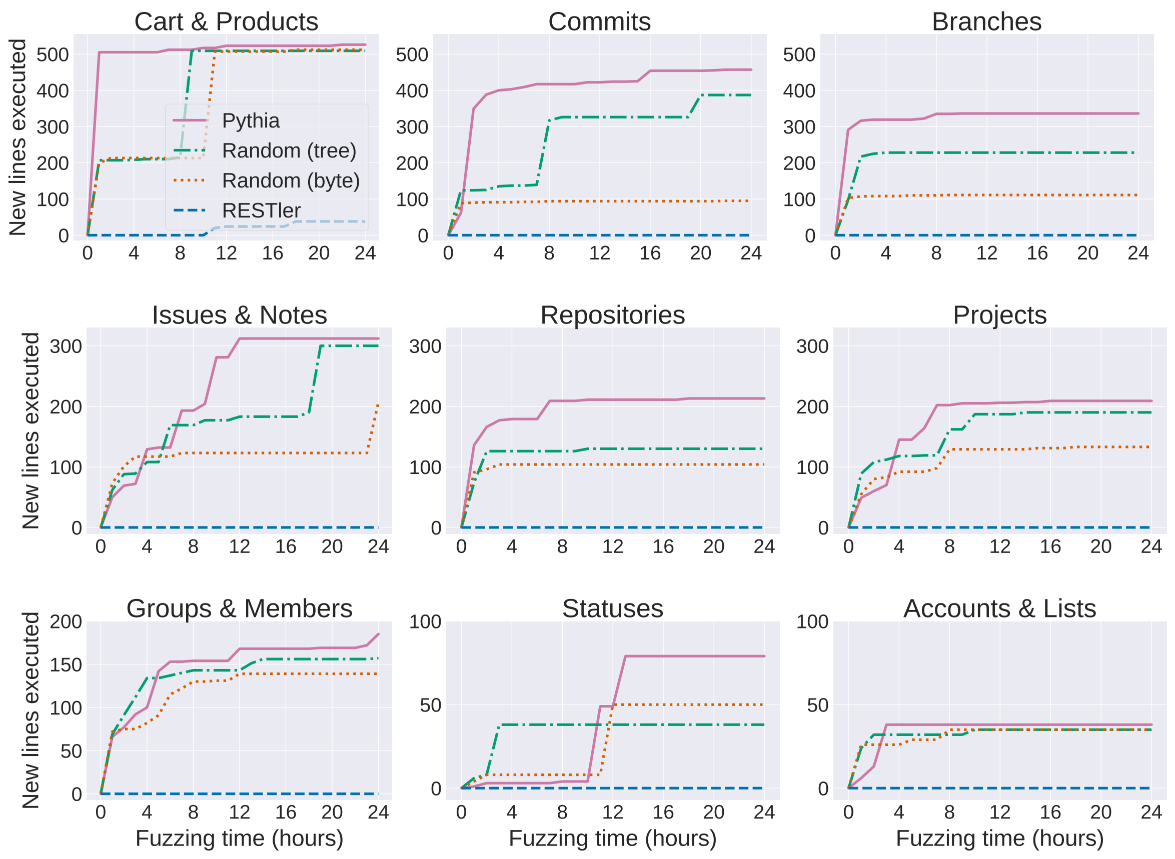


Figure 5.7: **Comparison of Pythia mutations strategies with respect to other baselines.** Seed collection phase: Run RESTler on each API family to generate seed corpora. Fuzzing phase: Use the seed corpora generated in the seed collection phase to perform three individual 24h fuzzing sessions per API family. Comparison: Shows the number of new lines executed during the fuzzing phase, excluding those executed during the seed collection phase. Pythia performs best compared to all baselines.

Ruby's `Class:TracePoint` hooks, to produce stack traces of lines of codes executed during testing. During testing, all target services are being monitored by Pythia's coverage monitor which converts stack traces to bitmaps of basic block activations corresponding to the test cases executed. In total, our static analysis extracts $11,413$ basic blocks for GitLab, $2,501$ basic blocks for Mastodon, and $2,616$ Spree.

Unless otherwise specified, we obtain initial seed corpora by running RESTler for 24h on each API family using its default Breadth First Search (BFS) fuzzing mode, its default fuzzing dictionary (i.e., two values for each primitive type), and by turning off its

Garbage Collector (GC) to obtain deterministic results. We choose BFS mode because in this mode RESTler conducts a systematic state space exploration and generates balanced test case sequences, with enough variation and reasonable lengths to train seq2seq RNNs. Depending on the target API, a test-case length ranges from 3 to 6 requests, or equivalently from 505 to 825 production rules, or from 907 to 1430 raw bytes. In contrast, RESTler's BFS-Fast and RandomWalk fuzzing modes lead to test cases that are unusable to train seq2seq RNNs. (BFS-Fast leads to few, very long test cases and RandomWalk leads to entirely random test cases.) Yet, when we compare the number of bugs found by Pythia versus RESTler, we run RESTler in all its fuzzing modes and report its best numbers.

Baselines. We evaluate Pythia against three blackbox baselines.

(i) RESTler: We use RESTler both for seed test case generation and for comparison. On each family of target APIs, we run RESTler for 2 days. The first day, *seed collection phase*, is used to generate seed test cases. The second day, *fuzzing phase*, is used for comparison. We compare the incremental coverage achieved by RESTler versus Pythia over the initial coverage achieved in the seed collection phase.

(ii) Random byte-level mutations: This is the simplest form of mutations. As suggested by their name, byte-level mutations are random alternations on the bytes of each seed test case. In order to produce byte-level mutations, the mutation engine selects a random target position within the seed sequence and a random byte value (in the range $0 - 255$), and updates the target position to the random byte value. Naturally, this type of mutations is not designed to generate grammatically valid mutants.

(iii) Random tree-level mutations: In order to produce random tree-level mutations, the mutation engine selects a random leaf of the respective AST representation and a random rule from $\mathcal{G}$ with a terminal symbol, and flips the target leaf with using the random rule. The mutations are exclusively performed on the tree leaves, and not in

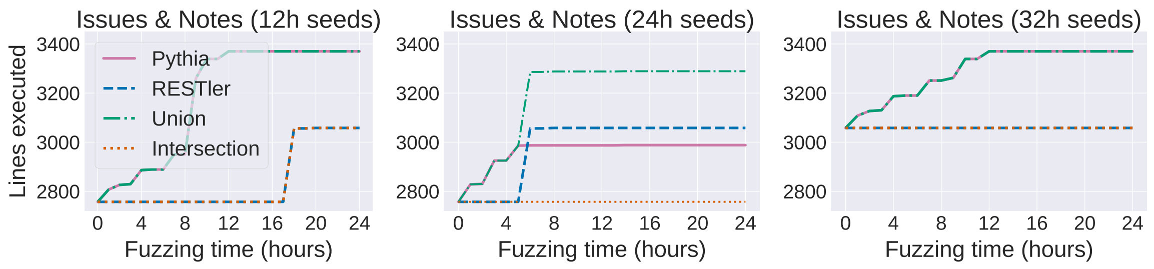


Figure 5.8: **Impact of initial seed collection.** Seed collection: Run RESTler for 12h on each API. Fuzzing: Use each corpus to perform three individual 24h guided tree-level Pythia mutation sessions. Moreover, let RESTler run for 24h additional hours (32h in total). Comparison: Shows the number of new lines executed after the initial 12h of seed collection.

internal nodes, in order to largely maintain the grammatical validity of each test case. Leaves correspond to primitive values and substituting a primitive value with a value of another type will most likely maintain grammatical validity. However, since the target leaves and the new rules (mutations) are selected at random for each test case, the target state space for mutations on realistic tests cases is quite large. For example, the test case shown in Figure 5.1 corresponds to a tree consisting of 73 leaf nodes and the grammar used to produce it has 66 rules with terminal rules. This, defines a state space with approximately $5,000$ feasible mutations only for one seed — let alone the total size of the state space defined by the complete corpus of seeds. Next, we evaluate Pythia's learning-based mutation strategy which considers the intrinsic structure of each test case and significantly prunes the size of the search space.

Code Coverage Achieved by Pythia (Q1)

We investigate Pythia's code coverage compared to the three blackbox baselines introduced above. Since all baselines are blackbox, here, we run Pythia *without* using code coverage feedback. RESTler is initially run for 24 hours to generate seed corpora (seed collection phase) across each API family. The seed collection phase is extended to 32 hours only in "Issues & Notes" API family due to a late plateau, explained in Q3. Then, RESTler runs for 24 additional hours for comparison (fuzzing phase), and all other fuzzers, including Pythia, are only run for the 24h fuzzing phase, as they mutate seed corpora generated by RESTler. Figure 5.7 shows the results.

In comparison to RESTler, during the 24h fuzzing phase Pythia discovers new lines of code (LOC) across all APIs and services, ranging from 43 lines (for "Accounts & Lists") to 500 lines (for "Cart & Products"). In contrast, RESTler plateaus after the initial seed collection phase and discovers no new lines during the 24h fuzzing phase. The only exception is "Cart & Products" where RESTler discovers 30 lines after 16 hours, while Pythia discovers 500 new lines.

Pythia discovers more LOC than RESTler in the fuzzing phase, because the latter navigates extremely large search spaces. For example, during fuzzing "Commits" API, RESTler explores a search space of 19K sequences of length five and 11 possible value combinations each (on average). Such search-space explosion is similar across all APIs and once RESTler plateaus, it is then challenging to further increase code coverage. In contrast, Pythia performs learning-based mutations and, as shown in Figure 5.7, in the fuzzing phase it outperforms RESTler and always finds new LOC across all APIs. The percentage improvement of Pythia over the initial code coverage achieved by RESTler in the seed collection phase ranges between 1% and 15% (depending on the target APIs). This increase, although relatively small is the result of learning-based mutations performed only by Pythia that allows exercising rare code paths, never reached by RESTler, and uncovers bugs.

We also compare Pythia with two random baselines. Across all APIs the relative ordering of all fuzzers remains consistent: Pythia outperforms the random tree-level baseline, which, in turn, outperforms the random byte-level baseline. Such ordering is expected. As explained earlier, raw byte-level mutations tend to violate both semantic and syntactic validity of the seed test cases and consequently underperform compared to tree-level mutations that mostly obey grammatical validity. Although the latter produces syntactically valid mutations, it mutates without API-specific guidance and, thus, cannot target its mutations to locations that have a higher impact on code coverage. In contrast, Pythia produces API-specific learning-based mutations that increase code coverage faster and higher than random tree-level mutations.

Impact of Seed Selection (Q2)

Previously, we saw that RESTler plateaus in the initial seed collection phase and then discovers almost no newlines while fuzzing across all API families. In contrast, Pythia uses RESTler seeds and further increases code coverage in the fuzzing phase. We will now investigate how RESTler and Pythia compare before RESTler plateaus and examine the impact of initial seed selection on the line coverage achieved by Pythia. We consider that RESTler plateaus if it discovers no new lines during the 24h fuzzing phase. We select "Issues & Notes" because RESTler takes the longest time to plateau among all API families (discovers new lines up until 32h) and compare three RESTler seed collection configurations: RESTler run for 12h (generates 5K seeds), for 24h (generates 12K seeds), and for 32h (generates 15K seeds). Figure 5.8 shows the cumulative increase in the number of lines executed by RESTler and Pythia during the fuzzing phase (on top of those executed during the seed collection phase) as well as the union and intersection of lines executed by both tools.

In the 12h setting, the lines discovered by Pythia are a superset of those discovered by RESTler because the intersection overlaps with RESTler while the union overlaps with Pythia. By contrast, in the 24h setting, the two tools discover diverging sets of lines because the intersection remains constant while both tools discover new lines and the union increases. In the 32h setting, the lines discovered by Pythia are also a superset of

Target APIs	**New LOC executed by Pythia**		
	Blackbox	Greybox	Improvement
Commits	457	457	-
Branches	336	340	1%
Issues & Notes	312	384	23%
Groups & Mmbrs	185	220	19%
Projects	209	292	40%
Repos & Files	213	240	13%
Accounts & Lists	38	38	-
Statuses	79	79	-
Storefront Cart	526	614	17%

Table 5.1: **Impact of code coverage feedback.** Shows the total number of lines executed by Pythia after 24h of fuzzing each API family without and with using code coverage feedback.

those discovered by RESTler. As explained in Chapter 5.4, Pythia generates mutations using many new values for each primitive type (including random payload alternations), whereas RESTler uses a predefined set of values and conducts a systematic state space exploration. Because RESTler search spaces are typically large, given few seeds (e.g., 5K in the 12h settings), Pythia may outperform RESTler and explore new lines faster. However, Pythia is not designed to perform mutations that extend sequences with additional requests and instead focuses on targeted AST-leaf-level mutations. Thus, if the initial seed corpus is large and RESTler has not plateaued (e.g., $12K$ seeds in the 24h setting), Pythia and RESTler may discover diverging sets of new lines.

Overall, when Pythia is run after RESTler plateaus (few hours in most API families except “Issues & Notes”), the lines discovered by Pythia are a superset of those discovered by RESTler because the former performs new mutations that exercise new LOC and discover new bugs. Yet, even when RESTler has not plateaued (e.g., in the 12h setting RESTler keeps discovering new lines in the fuzzing phase), Pythia still discovers new LOC, on top of those discovered by RESTler. The conclusions drawn for “Issues & Notes” generalize across all API families tested so far.

Impact of Code Coverage Feedback (Q3)

As explained in Chapter 5.5, Pythia can optionally use code coverage feedback to select only test cases that increase code coverage and prioritize them for further mutations. When a target service can be instrumented to produce coverage information, Pythia's coverage monitor collects this information and selects for further mutations only test cases that exercise unique code paths Thus, Pythia avoids mutating redundant seeds that activate identical code paths, and thus, more efficiently increases code coverage.

Table 5.1 shows the total number of new lines executed by Pythia after 24h of fuzzing each API family without (blackbox) and with (greybox) using code coverage feedback as well as the respective improvements. We observe that the best percentage improvement from code coverage feedback is obtained in GitLab's "Issued & Notes", "Groups & Members", and "Projects" API families (23%, 19%, and 40% respectively). These improvements are desirable and are more expected in API families with relatively more requests (see Table 3.1: e.g., 54 for "Projects") where using code coverage to prioritize test cases (i.e., request combinations) that exercise unique code paths leads to better improvement. Code coverage also leads to a 17% improvement in Spree's Storefront Cart. On the other hand, there is marginal or no improvement in GitLab's "Branches" and "Commits" API families that contain relatively less API requests (e.g., 8 for "Branches"). There is also no improvement in Mastodon's API families. Particularly in Mastodon, Pythia's coverage monitor discovers very few unique basic blocks activated by the test cases ("32" for "Accounts & Lists" and 4 for "Statuses"), while the same number ranges in the hundreds of basic blocks for all other API families. Consequently, since the coverage signal is too sporadic, there is no added value when using coverage feedback in Mastodon. Overall, using code coverage is an optional feature, which usually helps improve code coverage.

Target APIs	RESTler			Pythia		
	Tests	**500s**	**Bugs**	**Tests**	**500s**	**Bugs**
Commits	7.3K	0	0	10.7K	132	6
Branches	4.9K	0	0	12.3K	135	3
Issues & Notes	8.1K	0	0	11.1K	246	2
Groups & Mmbrs	9.1K	0	0	15K	234	4
Projects	6.5K	0	0	18.3K	185	3
Repos & Files	7.1K	0	0	14.9K	79	5
Accounts & Lists	10.6K	0	0	63.5K	1307	4
Statuses	26K	336	1	56K	962	2
Storefront Cart	13.7K	2018	1	18.7K	401	4
Total	-	-	**2**	-	-	**33**

Table 5.2: **New bugs found.** Shows the number of test cases generated, "500 Internal Server Errors" triggered, and new bugs found by RESTler and Pythia after 24h of fuzzing each API family.

Number of Bugs Found (Q4)

Although code coverage is an indicative proxy to measure the effectiveness of bug finding tools, the ultimate metric is the number of new bugs found. Pythia finds new bugs across all the tested APIs. In total, Pythia found 33 new bugs.

During fuzzing, a high number of "500 Internal Server Errors" is usually triggered by Pythia and different instances of the same bugs may be reported repeatedly. Since "500 Internal Server Errors" are potential server state corruptions with unknown consequences in the target service health, it is desirable to avoid duplication and report unique instances of each bug in order to facilitate further manual inspection. We use code coverage information and group unique bugs according to the following rule: *count as unique bugs the "500 Internal Server Error" instances that emanate from test cases exercising unique code paths.* Nonetheless, if code coverage information is not available, Pythia can group bugs using the structure of the types of the non-rendered request sequence (i.e., the same bucketization scheme followed by RESTler).

Table 5.2 compares Pythia with RESTler and shows the total number of test cases generated, 500s errors triggered, and unique bugs found after 24h of fuzzing each API family. To ensure fair comparison between the two tools, Table 5.2 reports results where Pythia runs in blackbox mode (i.e., without using code coverage feedback). Furthermore, we cross-checked that total number of 2 bugs reported for RESTler remains the same for all RESTler configurations (including BFS, BFS-Fast, and RandomWalk) even after fuzzing for 48h.

We observe that Pythia generates more test cases than RESTler across all APIs—220K versus 93K respectively. This is because Pythia's learning-based mutations do not lead to new request sequence combinations. (i.e., constant number of request per test case). In contrast, RESTler's mutations continuously increase the number of requests per test case by appending new requests. Hence, the target service test case throughput is better for Pythia than for RESTler. Most importantly, we observe that Pythia's learning-based mutations trigger 500s across all API families, whereas RESTler's mutations trigger 500s only in two API families. Using the bug grouping rule described earlier, we group Pythia's 500s to 33 unique bugs. Independently (i.e., using its own bug grouping methodology), RESTler also found 2 unique bugs, which we do not count in the column reported for Pythia.

New Bugs Found

During our experiments with Pythia on local GitLab, Mastodon, and Spree deployments we found 33 new bugs. All bugs were easily reproducible and were all confirmed by the respective service owners. We describe a subset of those bugs to give a flavor of what they look like and what test cases uncovered them.

Example 1: Bug in Storefront Cart. One of the bugs found by Pythia in Spree is triggered when a user tries to add a product in the storefront cart using a malformed request path `` ``/storefront/|add_item?include=line_items'' ``. Due to erroneous input sanitization, the character ``|'' is not stripped from the intermediate path parts. Instead, it reaches the function `split` of library `uri.rfc3986_parser.rb`, which treats it as a delimiter of the path string. This leads to an unhandled *InvalidURIError* exception in the caller library `actionpack`, and causes a “500 Internal Server Error” preventing the application from handling the request and returning the proper error, i.e., “400 Bad Request”. This bug can be reproduced with a test case with two requests: (1) creating a user token and (2) adding a product in the chart using a malformed request path. Bugs related to improper input sanitization and unhandled values passed across multiple layers of software libraries are usually found when using fuzzing. Pythia found bugs due to malformed request paths in all the services tested.

Example 2: Bug in Issues & Notes. Another bug found by Pythia in GitLab’s Issues & Notes APIs is triggered when a user attempts to open an issue on an existing project, using a malformed request body. The body of this request includes multiple primitive types and multiple key-value pairs, including `due_date, description, confidentiality, title, asignee_id, state_event`, and others. A user can create an issue using a malformed value for the field title, such as `{"title":"DELE\xa2"}` which leads to a “500 Internal Server Error”. The malformed title value is not sanitized before reaching the fuction `create` of `<class:Issues>` that creates new issues. This leads to an unhandled `ArgumentError` exception due to an invalid UTF-8 byte sequence. This bug can reproduced by (1) creating a project and (2) trying to post an issue with a malformed title in the project created in (1).

Interestingly, adding malformed values in other fields of the request body does not necessarily lead to errors. For instance, the fields `confidentiality` and `state_event` belong to different primitive types (boolean and integer) which are properly parsed and sanitized. Furthermore, mutations that corrupt the json structure of the request body or that do not use existing project ids lead to no such errors. Brute-forcing all possible ways to break similar REST API request sequences is infeasible. Instead, Pythia learns common usage patterns of the target service APIs and then applies learning-based mutations breaking these patterns and generating many grammatically valid test cases. Pythia found such input sanitization bugs, due to malformed request bodies, in all services tested. A similar bug is also shown in Figure 5.1.

Other examples of unhandled errors found by Pythia are due to malformed headers and request types. All the bugs found during this work are currently being reported to the respective service owners, and some of those previously-unknown errors have already been confirmed.

5.7 Summary

Pythia is the first fuzzer that augments grammar-based fuzzing with coverage-guided feedback and a learning-based mutation strategy for stateful REST API fuzzing. Pythia uses a statistical model to learn common usage patterns of target REST APIs from grammatically valid seed inputs. It then generates mutations by injecting a small amount of noise in the learnt model, causing it to deviate from common usage patterns. Pythia's learning-based mutation strategy helps generate new, grammatically valid test cases and coverage-guided feedback helps prioritize the test cases that are more likely to find bugs. We presented detailed experimental evidence—collected across three productions-scale, open-source cloud services—showing that Pythia outperforms prior approaches both in code coverage achieved and in new bugs found. Pythia found new bugs in all services tested so far. In total, Pythia found 33 bugs which were all confirmed by the respective service owners.

Chapter 6

Conclusions

Modern cloud services are complex conglomerates: they consist of multiple layers of software components that are usually written by multiple parties, span different languages and run-time environments, and continuously interact with each other. This model of multiple entangled layers of abstraction, which is typical in modern applications beyond cloud services, inevitably imposes increased complexity with performance, security, and reliability ramifications[43]. Indeed, many of the bugs discovered in cloud services in the context of this dissertation were due to invalid inputs that traversed different layers of abstraction and led to unforeseen errors. A seemingly valid input for one layer of abstraction, may constitute an invalid input with detrimental consequences for an underlying layer of abstraction. How to effectively test entire cloud service stacks for such unforeseen errors is still an open research challenge. This dissertation took a novel step forward.

We investigated the hypothesis that we can leverage the structured usage of cloud services through REST APIs and feedback obtained during interaction with such services (e.g., in the form of responses and HTTP status codes received or in the form of service-side code coverage achieved) in order to build systems that test cloud services in an automatic, efficient, and learning-based way through their APIs.

First, we introduced RESTler, a pioneer system using stateful REST API fuzzing to test cloud services through their APIs. RESTler statically analyzes the API documentation of a RESTful cloud service (given in an API specification language, such as OpenAPI[29]) and then generates tests by (1) inferring dependencies among request types, and by (2) learning invalid request combinations from the service's past responses. We presented empirical evidence showing that these two techniques are necessary to thoroughly test a target cloud service through its REST API, while at the same time pruning the large search space of possible request sequences. We evaluated various search strategies on three open-source, production scale cloud services and found tens of previously-unknown bugs. Moreover, we used RESTler to test four proprietary Azure and Office365 cloud services, and found several bugs in each of them. All bugs found by RESTler were confirmed and fixed by the corresponding service owners.

Although these results are encouraging, baseline stateful REST API fuzzing can only detect the generic class of "`500 Internal Server Errors`." Thus, we, then, described how stateful REST fuzzing can be extended with active checkers capturing desirable REST API security properties beyond "`500 Internal Server Errors`." Specifically, we introduced four security rules that capture desirable properties of REST APIs and services, and implemented active property checkers that automatically test and detect violations of these rules. We implemented all active checkers following a modular design and evaluated various performance optimizations on three open-source and three proprietary cloud services. Using active property checkers we reported bugs related to rule violations of security properties on proprietary Azure and Office365 services, which were all confirmed and fixed. Our reports were taken seriously by the respective service owners, since violations of the four security rules were potential security vulnerabilities and it was safer to fix these bugs rather than risk a live incident with unknown consequences.

Finally, we introduced Pythia, the first fuzzer that augments grammar-based fuzzing with coverage-guided feedback and a learning-based mutation strategy for stateful REST API fuzzing. Pythia uses a statistical model to learn common usage patterns of target REST APIs from grammatically valid seed inputs. It then generates mutations by injecting a small amount of noise in the learnt model, causing it to deviate from common usage patterns. Pythia's learning-based mutation strategy helps generate new, grammatically valid test cases and coverage-guided feedback helps prioritize the test cases that are more likely to find bugs. We presented experimental evidence showing that Pythia outperforms prior approaches both in code coverage achieved and in new bugs found. Pythia found new bugs in all services tested so far (33 in total) which were confirmed by the respective service owners.

Overall, in this dissertation we focused on bugs that were camouflaged behind the REST APIs of cloud services. This is a new class bugs and unlike buffer overflows in binary-format parsers, or use-after-free bugs in web browsers, or cross-site-scripting attacks in web pages, it is still largely unclear how severe these errors are. However, our work already has significant industrial impact: "Over the last 16 months, RESTler (including new extensions) has progressively been deployed more broadly inside Microsoft, and its application directly contributed to finding and fixing several hundred new bugs in Azure, Office365, and Bing services."[31]

Our world keeps changing fast. Research organizations, educational institutions, and other companies are increasingly switching from the complexity of owning and maintaining their own, on-premise computing infrastructure to instead access and pay on demand cutting edge cloud technologies. Without a doubt, systems for testing cloud services will have a broad real-life impact. This dissertation marks a clear path forward to test cloud services in *automatic*, *efficient*, and *learning-based* way for reliability, scalability, and performance issues through their APIs.

Bibliography

[1] Sample bug-4. https://gitlab.com/gitlab-org/gitlab-ce/issues/50949.

[2] Peach Fuzzer. http://www.peachfuzzer.com/. (Accessed on 06-02-2020).

[3] SPIKE Fuzzer. http://resources.infosecinstitute.com/fuzzer-automation-with-spike/. (Accessed on 06-02-2020).

[4] "A collection of vulnerabilities discovered by the AFL fuzzer". https://github.com/mrash/afl-cve. (Accessed on 06-02-2020).

[5] AgitarOne: Putting Java to the Test. http://www.agitar.com/solutions/products/agitarone.html. (Accessed on 06-02-2020).

[6] APIFuzzer. https://github.com/KissPeter/APIFuzzer. (Accessed on 06-02-2020).

[7] Application Verifier. https://docs.microsoft.com/en-us/security-risk-detection/concepts/application-verifier, . (Accessed on 06-02-2020).

[8] AppSpider. https://www.rapid7.com/products/appspider, . (Accessed on 06-02-2020).

[9] Address Sanitizer. https://clang.llvm.org/docs/AddressSanitizer.html. (Accessed on 06-02-2020).

[10] Amazon Web Services (AWS). https://aws.amazon.com/, . (Accessed on 06-02-2020).

[11] Amazon API Gateway. https://aws.amazon.com/about-aws/whats-new/2018/09/amazon-api-gateway-adds-support-for-openapi-3-api-specification/, . (Accessed on 06-02-2020).

[12] Microsoft Azure. https://azure.microsoft.com/en-us/, . (Accessed on 06-02-2020).

[13] Azure REST API Specifications. https://github.com/Azure/azure-rest-api-specs, . (Accessed on 06-02-2020).

[14] GNU Binutils. https://www.gnu.org/software/binutils/. (Accessed on 06-02-2020).

[15] BooFuzz. https://github.com/jtpereyda/boofuzz. (Accessed on 06-02-2020).

[16] Burp Suite. https://portswigger.net/burp. (Accessed on 06-02-2020).

[17] Clang: a C Language Family Frontend for LLVM. https://clang.llvm.org/. (Accessed on 06-02-2020).

[18] "Size of the public cloud computing services market from 2009 to 2022". https://www.statista.com/statistics/273818/global-revenue-generated-with-cloud-computing-since-2009/. (Accessed on 06-02-2020).

[19] ELF format of Executable and Linking Format (ELF) files. http://man7.org/linux/man-pages/man5/elf.5.html. (Accessed on 06-02-2020).

[20] American Fuzzy Lop. https://github.com/google/AFL. (Accessed on 06-02-2020).

[21] GCC, the GNU Compiler Collection. https://gcc.gnu.org/. (Accessed on 06-02-2020).

[22] Google Cloud. https://cloud.google.com/, . (Accessed on 06-02-2020).

[23] Google API Discovery Service. https://developers.google.com/discovery/, . (Accessed on 06-02-2020).

[24] Overview of JPEG. https://jpeg.org/jpeg/. (Accessed on 06-02-2020).

[25] LibFuzzer: A Library for Coverage-guided Fuzz Testing. http://llvm.org/docs/LibFuzzer.html. (Accessed on 06-02-2020).

[26] MPEG Layer III Audio Encoding). https://www.loc.gov/preservation/digital/formats/fdd/fdd000012.shtml. (Accessed on 06-02-2020).

[27] MPEG-4: The Moving Picture Experts Group. https://mpeg.chiariglione.org/standards/mpeg-4. (Accessed on 06-02-2020).

[28] Microsoft 365. https://www.microsoft.com/microsoft-365. (Accessed on 06-02-2020).

[29] OpenAPI Specification. https://swagger.io/specification/. (Accessed on 06-02-2020).

[30] Adobe Systems Incorporated. PDF Reference, 6th edition, Nov. 2006. http://www.adobe.com/content/dam/Adobe/en/devnet/acrobat/pdfs/pdf_reference_1-7.pdf. (Accessed on 06-02-2020).

[31] Personal Communication. Patrice Godefroid, Microsoft, August 2020.

[32] PROTOS - Security Testing of Protocol Implementations. https://www.ee.oulu.fi/roles/ouspg/Protos. (Accessed on 06-02-2020).

[33] CPython. `https://github.com/python/cpython`. (Accessed on 06-02-2020).

[34] Qualys Web Application Scanning (WAS). `https://www.qualys.com/apps/web-app-scanning/`. (Accessed on 06-02-2020).

[35] Sulley. `https://github.com/OpenRCE/sulley`. (Accessed on 06-02-2020).

[36] TnT-Fuzzer. `https://github.com/Teebytes/TnT-Fuzzer`. (Accessed on 06-02-2020).

[37] Extensible Markup Language (XML). `https://www.w3.org/TR/xml/`. (Accessed on 06-02-2020).

[38] S. Allamaraju. *RESTful Web Services Cookbook*. O'Reilly, 2010.

[39] S. Anand, C. S. Păsăreanu, and W. Visser. JPF-SE: A Symbolic Execution Extension to Java Pathfinder. In *Proceeding of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2007.

[40] S. Anand, C. S. Păsăreanu, and W. Visser. Symbolic Execution with Abstraction. *International Journal on Software Tools for Technology Transfer*, 11(1), 2009.

[41] D. Angluin. Learning Regular Sets from Queries and Counterexamples. *Information and computation*, 75(2), 1987.

[42] T. Arts, J. Hughes, J. Johansson, and U. Wiger. Testing Telecoms Software with Quviq QuickCheck. In *Proceedings of the 2006 ACM SIGPLAN workshop on Erlang*, 2006.

[43] V. Atlidakis, J. Andrus, R. Geambasu, D. Mitropoulos, and J. Nieh. Posix abstractions in modern operating systems: The old, the new, and the missing. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys)*, 2016.

[44] V. Atlidakis, P. Godefroid, and M. Polishchuk. RESTler: Stateful REST API Fuzzing. In *Proceedings of the 41st IEEE/ACM International Conference on Software Engineering (ICSE)*, 2019.

[45] V. Atlidakis, R. Geambasu, P. Godefroid, M. Polishchuk, and B. Ray. Pythia: Grammar-Based Fuzzing of REST APIs with Coverage-guided Feedback and Learning-based Mutations. *arXiv preprint arXiv:2005.11498*, 2020.

[46] V. Atlidakis, P. Godefroid, and M. Polishchuk. Checking Security Properties of Cloud Services REST APIs . In *Proceedings of the 13th IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2020.

[47] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley. Enhancing Symbolic Execution with Veritesting. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, 2014.

[48] T. Back. *Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms.* Oxford university press, 1996.

[49] C. Baier and J.-P. Katoen. *Principles of Model Checking.* MIT press, 2008.

[50] O. Bastani, R. Sharma, A. Aiken, and P. Liang. Synthesizing Program Input Grammars. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2017.

[51] S. Bhansali, W.-K. Chen, S. De Jong, A. Edwards, R. Murray, M. Drinić, D. Mihočka, and J. Chau. Framework for Instruction-level Tracing and Analysis of Program Executions. In *Proceedings of the 2nd International Conference on Virtual Execution Environments*, 2006.

[52] C. M. Bishop. *Pattern Recognition and Machine Learning.* springer, 2006.

[53] M. Böhme, V.-T. Pham, and A. Roychoudhury. Coverage-based Greybox Fuzzing as Markov Chain. *IEEE Transactions on Software Engineering*, 45(5), 2017.

[54] P. Boonstoppel, C. Cadar, and D. Engler. RWset: Attacking Path Explosion in Constraint-based Test Generation. In *Proceedings of the 14th International conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008.

[55] E. Bounimova, P. Godefroid, and D. Molnar. Billions and Billions of Constraints: Whitebox Fuzz Testing in Production. In *Proceedings of the 35th IEEE/ACM International Conference on Software Engineering (ICSE)*, 2013.

[56] R. S. Boyer, B. Elspas, and K. N. Levitt. SELECT: a Formal System for Testing and Debugging Programs by Symbolic Execution. *ACM SigPlan Notices*, 10(6), 1975.

[57] J. S. Bridle. Training stochastic model recognition algorithms as networks can lead to maximum mutual information estimation of parameters. In *Advances in neural information processing systems*, pages 211–217, 1990.

[58] S. Bucur, V. Ureche, C. Zamfir, and G. Candea. Parallel Symbolic Execution for Automated Real-world Software Testing. In *Proceedings of the 6th Conference on Computer Systems*, 2011.

[59] J. Burnim and K. Sen. Heuristics for Scalable Dynamic Test Generation. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2008.

[60] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A Static Analyzer for Finding Dynamic Programming Errors. *Software: Practice and Experience*, 30(7), 2000.

[61] C. Cadar. Targeted Program Transformations for Symbolic Execution. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*, 2015.

[62] C. Cadar and D. Engler. Execution Generated Test Cases: How to Make Systems Code Crash Itself. In *International SPIN Workshop on Model Checking of Software*, 2005.

[63] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically Generating Inputs of Death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS)*, 2006.

[64] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[65] P. Chen and H. Chen. ANGORA: Efficient Fuzzing by Principled Search. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (S&P)*, 2018.

[66] V. Chipounov, V. Kuznetsov, and G. Candea. The S2E platform: Design, implementation, and applications. *ACM Transactions on Computer Systems (TOCS)*, 30(1), 2012.

[67] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. Learning Phrase Representations Using RNN Encoder-decoder for Statistical Machine Translation. *arXiv preprint arXiv:1406.1078*, 2014.

[68] Y. Choi, H. Kim, H. Oh, and D. Lee. Call-flow aware api fuzz testing for security of windows systems. In *2008 International Conference on Computational Sciences and Its Applications*, 2008.

[69] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio. Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling. *arXiv preprint arXiv:1412.3555*, 2014.

[70] K. Claessen and J. Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. *ACM SIGPLANT NOTICES*, 46(4), 2011.

[71] E. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded Model Checking Using Satisfiability Solving. *Formal methods in system design*, 19(1), 2001.

[72] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic Verification of Finite-state Concurrent Systems Using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(2), 1986.

[73] J. Clarke, J. J. Dolado, M. Harman, R. Hierons, B. Jones, M. Lumkin, B. Mitchell, S. Mancoridis, K. Rees, M. Roper, et al. Reformulating Software Engineering as a Search Problem. *IEEE Proceedings-software*, 150(3), 2003.

[74] E. M. Clarke Jr, O. Grumberg, D. A. Peled, and G. Holzmann. *Model Checking*. MIT press, 2018.

[75] D. M. Cohen, S. R. Dalal, J. Parelius, and G. C. Patton. The Combinatorial Design Approach to Automatic Test Generation. *IEEE Software*, 13(5), 1996.

[76] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton. The AETG System: An Approach to Testing Based on Combinatorial Design. *IEEE Transactions on Software Engineering*, 23(7), 1997.

[77] D. Coppit and J. Lian. Yagg: An Easy-to-use Generator for Structured Test Inputs. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, 2005.

[78] D. R. Kuhn and D. R. Wallace and A. M. Gallo. Software Fault Interactions and Implications for Software Testing. *IEEE Transactions on Software Engineering*, 30(6), 2004.

[79] O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare. *Structured Programming*. Academic Press Ltd., 1972.

[80] L. De Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008.

[81] R. Emek, I. Jaeger, Y. Naveh, G. Bergman, G. Aloni, Y. Katz, M. Farkash, I. Dozoretz, and A. Goldin. X-Gen: A Random Test-case Generator for Systems and SoCs. In *Proceedings of the 7th IEEE International High-Level Design Validation and Test Workshop, 2002.*, 2002.

[82] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking System Rules Using System-specific, Programmer-written Compiler Extensions. In *Proceedings of the 4th conference on Symposium on Operating System Design and Implementation (OSDI)*, 2000.

[83] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. *ACM SIGOPS Operating Systems Review*, 35(5), 2001.

[84] R. T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. University of California, Irvine Irvine, 2000.

[85] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended Static Checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI)*, 2002.

[86] Flask. Web development, one drop at a time. `http://flask.pocoo.org/`.

[87] G. Fraser and A. Arcuri. 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 (FSE)*, 2011.

[88] V. Ganesh, T. Leek, and M. Rinard. Taint-based Directed Whitebox Fuzzing. In *Proceedings of the 31st International Conference on Software Engineering (ICSE)*, 2009.

[89] S. Geman, E. Bienenstock, and R. Doursat. Neural Networks and the Bias/Variance Dilemma. *Neural computation*, 4(1), 1992.

[90] GitLab. Sample bug-1. `https://gitlab.com/gitlab-org/gitlab-ce/issues/50955`, .

[91] GitLab. Sample bug-2. `https://gitlab.com/gitlab-org/gitlab-ce/issues/50265`, .

[92] GitLab. Sample bug-3. `https://gitlab.com/gitlab-org/gitlab-ce/issues/50270`, .

[93] GitLab. Gitlab api. `https://docs.gitlab.com/ee/api/`, .

[94] GitLab. Hardware requirements. `https://docs.gitlab.com/ce/install/requirements.html`, .

[95] GitLab. Statistics. `https://about.gitlab.com/is-it-any-good/`.

[96] P. Godefroid. Model Checking for Programming Languages Using VeriSoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1997.

[97] P. Godefroid. Compositional Dynamic Test Generation. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (PLDI)*, 2007.

[98] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *Proceedings of the 26th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[99] P. Godefroid, A. Kiezun, and M. Y. Levin. Grammar-based Whitebox Fuzzing. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2008.

[100] P. Godefroid, M. Levin, and D. Molnar. Automated Whitebox Fuzz Testing. In *Proceedings of the 15th Network and Distributed System Security Symposium (NDSS)*, 2008.

[101] P. Godefroid, M. Y. Levin, and D. A. Molnar. Active Property Checking. In *Proceedings of the 8th ACM International Conference on Embedded Software (ICESS)*, 2008.

[102] P. Godefroid, H. Peleg, and R. Singh. Learn&Fuzz: Machine Learning for Input Fuzzing. In *Proceedings of the 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2017.

[103] D. E. E. Goldberg. Genetic Algorithms in Search, Optimization & Machine Learning.

[104] Goodfellow, Ian and Pouget-Abadie, Jean and Mirza, Mehdi and Xu, Bing and Warde-Farley, David and Ozair, Sherjil and Courville, Aaron and Bengio, Yoshua. Generative Adversarial Nets. In *Proceedings of Advances in Neural Information Processing Systems (NIPS)*, 2014.

[105] A. Hagerer, H. Hungar, O. Niese, and B. Steffen. Model Generation by Moderated Regular Extrapolation. In *Proceedings of the International Conference on Fundamental Approaches to Software Engineering*, 2002.

[106] H. Han and S. K. Cha. IMF: Inferred Model-based Fuzzer. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[107] K. V. Hanford. Automatic Generation of Test Cases. *IBM Systems Journal*, 9 (4), 1970.

[108] G. E. Hinton and R. R. Salakhutdinov. Reducing the Dimensionality of Data with Neural Networks. *science*, 313(5786), 2006.

[109] S. Hochreiter and J. Schmidhuber. Long Short-term Memory. *Neural computation*, 9(8), 1997.

[110] F. Howar, B. Jonsson, and F. Vaandrager. *Combining black-box and white-box techniques for learning register automata*. Springer, 2019.

[111] M. Howard and S. Lipner. *The Security Development Lifecycle*. Microsoft Press, 2006.

[112] J. Jaffar, V. Murali, and J. A. Navas. Boosting Concolic Testing via Interpolation. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, 2013.

[113] S. Jana, Y. J. Kang, S. Roth, and B. Ray. Automatically Detecting Error Handling Bugs Using Error Specifications. In *Program of the 25th USENIX Security Symposium*, 2016.

[114] Y. Kang, B. Ray, and S. Jana. Apex: Automated Inference of Error Specifications for C APIs. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2016.

[115] R. E. Kass, B. P. Carlin, A. Gelman, and R. M. Neal. Markov Chain Monte Carlo in Practice. *The American Statistician*, 52(2), 1998.

[116] J. C. King. Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7), 1976.

[117] D. P. Kingma and J. Ba. Adam: A Method for Stochastic Optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[118] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by Simulated Annealing. *Science*, 220(4598), 1983.

[119] B. Korel. Automated Software Test Data Generation. *IEEE Transactions on software engineering*, 16(8), 1990.

[120] Kuo-Chung Tai and Yu Lei. A Test Generation Strategy for Pairwise Testing. *IEEE Transactions on Software Engineering*, 28(1), 2002.

[121] R. Lämmel and W. Schulte. Controllable Combinatorial Coverage in Grammar-Based Testing. In *Proceedings of TestCom*, 2006.

[122] V. Le, C. Sun, and Z. Su. Finding Deep Compiler Bugs via Guided Stochastic Program Mutation. *ACM SIGPLAN Notices*, 50(10), 2015.

[123] C. Lemieux and K. Sen. Fairfuzz: Targeting Rare Branches to Rapidly Increase Greybox Fuzz Testing Coverage. *arXiv preprint arXiv:1709.07101*, 2017.

[124] G. Li, I. Ghosh, and S. P. Rajan. Klover: A symbolic execution and automatic test generation tool for c++ programs. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV)*, 2011.

[125] Y. Li, B. Chen, M. Chandramohan, S.-W. Lin, Y. Liu, and A. Tiu. Steelix: Program-state Based Binary Fuzzing. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*, 2017.

[126] O. Lichtenstein and A. Pnueli. Checking that Finite State Concurrent Programs Satisfy their Linear Specification. In *Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 1985.

[127] H. Lin, J. Zhu, J. Peng, and D. Zhu. Deity: Finding Deep Rooted Bugs in JavaScript Engines. In *Proceedings if the 19th International Conference on Communication Technology (ICCT)*, 2019.

[128] M. B. Cohen and P. B. Gibbons and W. B. Mugridge and C. J. Colbourn. Constructing Test Suites for Interaction Testing. In *Proceedings of the 25th International Conference on Software Engineering (ICSE)*, 2003.

[129] R. Majumdar and K. Sen. Hybrid Concolic Testing. In *Proceedings of the 29th International Conference on Software Engineering (ICSE'07)*, 2007.

[130] R. Majumdar and R.-G. Xu. Directed Test Generation Using Symbolic Grammars. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2007.

[131] Mastodon. Mastodon. `https://mastodon.social/about`, .

[132] Mastodon. Statistics. `https://joinmastodon.org/`, .

[133] P. M. Maurer. Generating Test Data with Enhanced Context-free Grammars. *Ieee Software*, 7(4), 1990.

[134] P. McMinn. Search-based Software Test Data Generation: A Survey. *Software testing, Verification and reliability*, 14(2), 2004.

[135] Microsoft. Azure dns zone rest api. `https://docs.microsoft.com/en-us/rest/api/dns/zones/get`.

[136] B. P. Miller, L. Fredriksen, and B. So. An Empirical Study of the Reliability of UNIX Utilities. *Communications of the ACM*, 33(12), 1990.

[137] M. Mitchell. *An Introduction to Genetic Algorithms*. MIT press, 1998.

[138] D. A. Molnar and D. Wagner. Catchconv: Symbolic Execution and Run-time Type Inference for Integer Conversion Errors. 2007.

[139] M. Musuvathi, D. Y. Park, A. Chou, D. R. Engler, and D. L. Dill. CMC: A Pragmatic Approach to Model Checking Real Code. *ACM SIGOPS Operating Systems Review*, 36(SI), 2002.

[140] N. Havrikov and A. Zeller. Systematically Covering Input Structure. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019.

[141] Y. Naveh, M. Rimon, I. Jaeger, Y. Katz, M. Vinov, E. s Marcu, and G. Shurek. Constraint-based Random Stimuli Generation for Hardware Verification. *AI magazine*, 28(3), 2007.

[142] N. Nethercote and J. Seward. Valgrind: A Program Supervision Framework. *Electronic notes in theoretical computer science*, 89(2), 2003.

[143] S. Newman. *Building Microservices: Designing Fine-grained Systems*. O'Reilly Media, Inc., 2015.

[144] N. Nichols, M. Raugas, R. Jasper, and N. Hilliard. Faster Fuzzing: Reinitialization with Deep Neural Models. *arXiv preprint arXiv:1711.02807*, 2017.

[145] OAuth. Oauth 2.0. `https://oauth.net/`.

[146] S. Ognawala, T. Hutzelmann, E. Psallida, and A. Pretschner. Improving Function Coverage with Munch: A Hybrid Fuzzing and Directed Symbolic Execution Approach. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, 2018.

[147] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed Random Test Generation. In *Proceedings of the 29th International Conference on Software Engineering (ICSE)*, 2007.

[148] R. Padhye, C. Lemieux, K. Sen, M. Papadakis, and Y. Le Traon. Semantic Fuzzing with zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2018.

[149] R. Padhye, C. Lemieux, K. Sen, L. Simon, and H. Vijayakumar. FuzzFactory: Domain-Specific Fuzzing with Waypoints. *Proc. ACM Program. Lang.*, 3(OOPSLA), 2019.

[150] R. P. Pargas, M. J. Harrold, and R. R. Peck. Test-data Generation Using Genetic Algorithms. *Software testing, verification and reliability*, 9(4), 1999.

[151] H. Peng, Y. Shoshitaishvili, and M. Payer. T-Fuzz: Fuzzing by Program Transformation. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (S&P)*, 2018.

[152] V.-T. Pham, M. Böhme, A. E. Santosa, A. R. Caciulescu, and A. Roychoudhury. Smart Greybox Fuzzing. *IEEE Transactions on Software Engineering*, 2019.

[153] V.-T. Pham, M. Böhme, and A. Roychoudhury. AFLNET: A Greybox Fuzzer for Network Protocols. In *Proceedings of the 13th IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2020.

[154] P. Purdom. A Sentence Generator for Testing Parsers. *BIT Numerical Mathematics*, 12(3), 1972.

[155] J.-P. Queille and J. Sifakis. Specification and Verification of Concurrent Systems in CESAR. In *International Symposium on Programming*, 1982.

[156] H. Raffelt, B. Steffen, and T. Margaria. Dynamic Testing via Automata Learning. In *Haifa Verification Conference*, 2007.

[157] M. Rajpal, W. Blum, and R. Singh. Not All Bytes are Equal: Neural Byte Sieve for Fuzzing. *arXiv preprint arXiv:1711.04596*, 2017.

[158] D. A. Ramos and D. Engler. Under-constrained symbolic execution: Correctness checking for real code. In *Proceedings of the 24th USENIX Security Symposium*, 2015.

[159] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos. VUzzer: Application-aware Evolutionary Fuzzing. In *Proceedings of the 24th Network and Distributed System Security Symposium (NDSS)*, 2017.

[160] C. R. Reeves. *Modern Heuristic Techniques for Combinatorial Problems.* John Wiley & Sons, Inc., 1993.

[161] M. J. Renzelmann, A. Kadav, and M. M. Swift. Symdrive: testing drivers without devices. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[162] D. M. Ritchie and K. Thompson. The unix time-sharing system. *Bell System Technical Journal*, 57(6), 1978.

[163] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, 2002.

[164] P. Saxena, P. Poosankam, S. McCamant, and D. Song. Loop-extended Symbolic Execution on Binary Programs. In *Proceedings of the 18th International Symposium on Software testing and analysis (ISSTA)*, 2009.

[165] K. Sen, D. Marinov, and G. Agha. CUTE: A Concolic Unit Testing Engine for C. *ACM SIGSOFT Software Engineering Notes*, 30(5), 2005.

[166] D. She, K. Pei, D. Epstein, J. Yang, B. Ray, and S. Jana. NEUZZ: Efficient Fuzzing with Neural Program Learning. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (S&P)*, 2019.

[167] E. G. Sirer and B. N. Bershad. Using Production Grammars in Software Testing. *ACM SIGPLAN Notices*, 35(1), 1999.

[168] Spree. Spree. `https://spreecommerce.org/`.

[169] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *Proceedings of the 23th Network and Distributed System Security Symposium (NDSS)*, 2016.

[170] I. Sutskever, O. Vinyals, and Q. Le. Sequence to Sequence Learning with Neural Networks. *Advances in NIPS*, 2014.

[171] M. Sutton, A. Greene, and P. Amini. *Fuzzing: Brute Force Vulnerability Discovery.* Addison-Wesley, 2007.

[172] Tensorflow. Tensorflow. `https://www.tensorflow.org/`.

[173] N. Tillmann and J. De Halleux. PEX: Whitebox Test Generation for .net. In *International Conference on Tests and Proofs*, 2008.

[174] M. Utting, A. Pretschner, and B. Legeard. A Taxonomy of Model-based Testing Approaches. *Software testing, verification and reliability*, 22(5), 2012.

[175] P. Vincent, H. Larochelle, Y. Bengio, and P.-A. Manzagol. Extracting and Composing Robust Features with Denoising Autoencoders. In *Proceedings of the 25th International Conference on Machine learning*, 2008.

[176] P. Vincent, H. Larochelle, I. Lajoie, Y. Bengio, and P.-A. Manzagol. Stacked Denoising Autoencoders: Learning Useful Representations in a Deep Network with a Local Denoising Criterion. *Journal of machine learning research*, 11(Dec), 2010.

[177] J. Wang, B. Chen, L. Wei, and Y. Liu. Skyfire: Data-driven Seed Generation for FXuzzing. In *Proceedings of the 38th IEEE Symposium on Security and Privacy (S&P)*, 2017.

[178] J. Wang, B. Chen, L. Wei, and Y. Liu. Superion: Grammar-aware Greybox Fuzzing. In *Proceedings of the 41st IEEE/ACM International Conference on Software Engineering (ICSE)*, 2019.

[179] T. Wang, T. Wei, G. Gu, and W. Zou. TaintScope: A Checksum-aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection. In *Proceedings of the 31st IEEE Symposium on Security and Privacy (S&P)*, 2010.

[180] J. Yang, C. Sar, and D. Engler. EXPLODE: A Lightweight, General System for Finding Serious Storage System Errors. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[181] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using Model Checking to Find Serious File System Errors. *ACM Transactions on Computer Systems (TOCS)*, 24(4), 2006.

[182] M. Yannakakis and D. Lee. Testing Finite-State Machines. In *Proceedings of the 23rd Annual ACM Symposium on the Theory of Computing (STOC)*, 1991.

[183] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim. QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *Proceedings of the 27th USENIX Security Symposium*, 2018.