

System Under Oath

Learning Microservice Architectures

Melle Koper

Delft University of Technology

System Under Oath

Learning Microservice Architectures

by

Melle Koper

Instructor: S. Verwer
Faculty: EEMCS, Delft

Style: TU Delft Report Style, with modifications by Daan Zwaneveld

Acknowledgements

I'm proud to present you my masters thesis. This project took 9 months to produce, but in reality it is the culmination of my entire academic career. I want to thank Dr. Sicco Verwer for his supervision throughout this process, I really enjoyed our weekly meetings to discuss, brainstorm and work through feedback. Next, I would like to thank Lars de Kroon and everyone at Ubiqu for giving me the opportunity to complete this project in collaboration with the company.

Finally, I want to thank my family and friends for the support not only throughout the course of writing this thesis, but throughout my academic career.

*Melle Koper
Delft, May 2026*

Abstract

Stateful learning techniques are crucial for the verification and validation of complex software systems. As the world becomes increasingly reliant on software, the need for secure and reliable applications continues to grow. Previous work on stateful learning has primarily focused on monolithic systems, leaving the application of these techniques to microservice architectures largely unexplored. This is despite their widespread use in modern software development.

This thesis investigates the challenges associated with learning systems based on microservice architectures. First, we present a framework for generating valid API requests across interacting services. Next, we address system flakiness during learning through the introduction of Two-Phase Equivalence Querying with adaptive sampling. Finally, to explain the non-deterministic behavior observed in microservice systems, we introduce the concept of State Drift, which captures state changes caused by internal communication between services.

To conclude this thesis, we apply the techniques developed throughout this work to a real-world software system. The system learned is an implementation of OAuth provided by Ubiq. This case study demonstrates how our proposed techniques can be used in industry and highlights the practical challenges that remain when applying stateful learning to microservice architectures.

Contents

Preface	i
Summary	ii
1 Introduction	1
1.1 Research Questions	2
1.2 Contributions	2
1.3 Outline	2
2 Background	3
2.1 Finite State Machines	3
2.2 Learning	4
2.3 Active Learning	4
2.3.1 Learning Algorithms	5
2.3.2 Equivalence Oracle	5
2.4 Passive Learning	6
2.4.1 FlexFringe	6
2.5 System Under Learn	7
2.5.1 Microservice architectures	7
2.5.2 REST API	8
2.5.3 SUL: OAuth 2.0	8
2.5.4 Functionality	9
3 Previous Works	10
3.1 Stateful Fuzzing	10
3.2 Automata Learning for Stateful Fuzzing	11
3.2.1 Active Learning	11
3.2.2 Passive Learning	11
3.3 Fuzzing Distributed Systems and Protocols	11
3.3.1 API and System-Level Approaches	11
3.3.2 Handling Protocol-specific Dependencies	12
3.4 Noise	12
3.5 Research Gap	12
4 Active Learning	13
4.1 Developmental SUL	13
4.2 Communication with SUL	14
4.2.1 Orchestrator	14
4.2.2 Corpus definition	14
4.3 Algorithm Selection	17
4.3.1 Equivalence Oracle	17
4.3.2 Learning Algorithm	18
4.4 Discussion	19
5 Handling Non-Determinism	21
5.1 Active Learning in a Non-Deterministic Setting	21
5.2 Query Cache	22
5.3 Experiment	23
5.4 Results	23
5.5 Discussion	25

6	Passive Learning to Catch State Drift	27
6.1	Purpose	27
6.2	State Drift	27
6.3	Methodology	28
6.3.1	Passive Learning and Active Learning	28
6.3.2	Log capturing	28
6.3.3	Log Processing	29
6.3.4	Passive Learner	29
6.3.5	Comparison of Machines	30
6.4	Results	30
6.4.1	Log Processing	30
6.4.2	Learning	30
6.4.3	State Drift	30
6.5	Active vs Passive machines	31
6.6	Conclusion	31
7	Learning OAuth	33
7.1	Application	33
7.2	Active Learning	34
7.2.1	Two Phase KV	34
7.2.2	Active Learning with Two Phase L*	36
7.2.3	Baseline Learning Campaign	36
7.3	Passive Learning	38
7.4	Findings	38
7.4.1	Race Condition	39
7.5	Discussion	39
8	Discussion	40
8.1	RQ: 1	40
8.2	RQ: 2	40
8.3	RQ: 3	41
8.4	Future Work and Limitations	41
8.4.1	Application	41
8.4.2	Learning Inefficiency	42
8.4.3	Passive Learning	42
8.4.4	State Machine Validation	42
9	Conclusion	43
	References	45
A	Corpus	48
B	Initial experimentation	51
B.1	Experiments	51
B.1.1	Learning on sandbox	51
B.1.2	Learning from a Single Trace	51
B.2	Passive Learning	52
B.2.1	Learning from a Single Trace	52

1

Introduction

As the world becomes more reliant on IT solutions for everyday tasks, outages become increasingly more impactful. In extreme cases, like the Amazon Web Services outage in 2025, much of the internet went down for several hours. Causing downtime for thousands of services, from shopping sites to social networks, also including more crucial systems such as banking services. Although systems such as Amazon Web Services are generally robust, outages continue to occur due to a variety of errors. These errors can include, faulty updates, injection of bad code, or third party reliance. [30]

In order to mitigate the risks of disruptions and security failures, validation and testing of the code within trusted services is a must. This is particularly important in foundational security mechanism, such as authentication and authorization protocols like OAuth. These mechanisms serve as the gatekeepers of user identity and sensitive data on the internet. Therefore, implementation flaws or invalidated edge case could potentially lead to data breaches or unauthorized system access. This means that the robust validation of these services is not only best practice, but a necessity to maintain system integrity and user trust.

Currently, it has become more popular for companies ranging from large enterprises to small business to transition their monolithic code bases to microservices to enhance the quality of their products. Microservice architectures allow for the decomposition of complex into independent services, which are loosely coupled to each other. This strategy has the advantage of faster application delivery and a boost in reliability, as each microservice can be developed, deployed and scaled independently. [19]

Although microservices can bring many advantages, they also introduce distinct vulnerabilities. Some of these vulnerabilities are the result of the increased complexity of service interaction and intricate dependency management. [19]

So clearly, when implementing a system with a microservice architecture on which other services, companies, and people rely, it is of great importance to thoroughly test it. Over the years, fuzzing has become a testing strategy that has grown in popularity. Relative to manual testing, which is labor intensive and prone to incomplete coverage, fuzzing can be an effective solution. More specifically, stateful fuzzing can be a viable alternative, as it allows for more effectively capturing complex interconnected behavior in the system which is being learned. [6]

This thesis is written in collaboration with Ubiqu, a scale up with a focus on digital identities and trust services. Ubiqu's products are used across sectors where digital identity and trust must operate with a high level of assurance. Like many other companies, Ubiqu's products consist of intricate integrated microservices.

Current active learning techniques are focused on learning singular systems. If one chooses to learn each microservice in isolation, it will ignore the state changes relevant to the system as a whole. When applying existing active learning techniques on deployed systems, they will also run into trouble due to various issues. One of these issues is the interconnected dependencies for state changes in the system as a whole can create race conditions affected by latency. Meaning that the system deployed

might not act as expected or in a deterministic manner due to internal requests that may or may not have been completed.

In addition to active learning, passive learning is an important tool that can be used to further broaden our abilities to learn the system as a whole. Through capturing all network communications and applying passive learning we can construct a more complete representation of the system as both calls made by the user and calls made by the individual services are included.

1.1. Research Questions

Previous fuzzing research has focused on stateful fuzzing, microservices and distributed systems, and fuzzing of API's. As far as we know, there is a research gap in the active learning of stateful microservice architectures. This is due to the fact that microservices are relatively complex to test. In order to fill this research gap, we want to answer the following research question:

How can stateful active learning techniques be applied to software systems developed as cryptographically dependent microservice architectures?

and the following potential sub research questions:

- **How can we apply learning methods to learn the expected behavior of a software system hindered by an unstable learning environment?**
- **How can passive and active learning techniques be applied in a complementary manner to improve the completeness of learning systems with microservice architectures?**

To show that these active and passive learning techniques are applicable to other systems, we would also like to experiment on a second stateful system.

1.2. Contributions

The contributions proposed by this thesis are the following:

- A method for learning microservices as one machine, the system is greater than the sum of its parts.
- A solution for learning state machines in a flaky deployment scenario.
- The application of passive learning combined with active learning to capture and explain race conditions.

1.3. Outline

Chapter 2 provides the basic background knowledge required to understand the basis of this thesis. In Chapter 3 we discuss prior research which motivates our research questions. Chapter 4 introduces our proposed solution for the active learning of microservices. Chapter 5 provides our solution for handling flakiness introduced by the setup of the system which we are learning. We then introduce our solution for explaining the unpredictable behavior displayed by the system through passive learning in Chapter 6. After which, we apply all of our proposed techniques on an actual developed system and discuss the results in Chapter 7. In Chapter 8 we provide a global discussion of our results and discuss any future work we would recommend. Finally in Chapter 9 we conclude this thesis.

2

Background

This chapter serves as an introduction to topics and algorithms referenced through the rest of the thesis. A basic understanding of these topics is necessary for the comprehension of the following chapters. First we introduce state machines and the concept of learning state machines. We then briefly describe how some of the learning algorithms and their differences. Finally we give a short introduction to the System Under Learn and its architecture.

2.1. Finite State Machines

The goal of this thesis is to learn finite state machines, more specifically, Mealy machines. A finite state machine is a mathematical model of computation in graph form. In a finite state machine, a finite number of nodes represent the possible system and a finite set of edges represent the transitions which may or may not change the state. [26]

Formally, a finite state machine is defined as follows:

- $F = (\Sigma, S, s_0, \delta, F)$ where:
- Σ is a finite set of input symbols, known as an alphabet
- S is a finite set of states
- s_0 is defined as the start state
- δ is the transition function which maps $S \times \Sigma \rightarrow S$
- F is the set of accepting states $F \subseteq S$

This thesis focuses on a subset of finite state machines called Mealy machine. A Mealy machine differs from a regular finite state machine because its outputs are determined by its current state and the input. [16]

Mealy machines are formally defined as follows: [16]

- $(\Sigma, O, S, s_0, \delta, \lambda)$ where:
- Σ is a finite set of input symbols, known as the input alphabet
- O is a finite set of output symbols, known as the output alphabet
- S is a finite set of states
- s_0 is defined as the start state
- δ is the transition function which maps $S \times \Sigma \rightarrow S$
- λ is the output function that maps $S \times \Sigma \rightarrow O$

Another important concept is the difference between deterministic and non-deterministic finite state machines. Given a word, a deterministic finite state machine processes each character sequentially. If the last character leads to the accepting state, the word is accepted. A non-deterministic finite state

machine differs in the fact that its transition function returns a subset of finite states instead of one state. Meaning that there can be multiple active states in parallel. [26]

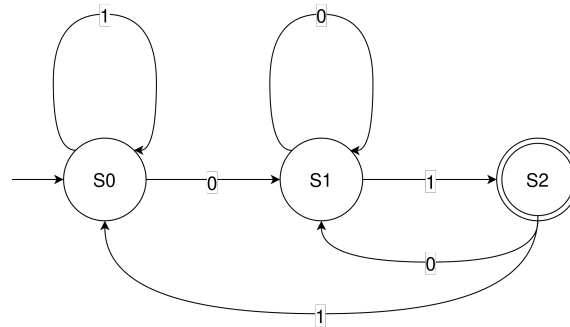


Figure 2.1: Example Deterministic Finite State Machine

Figure 2.1 is an example of a deterministic finite state machine that accepts any word that ends in "01". Take for example the word "01101", the starting state is S0, we first see a "0", which means we move to state S1, we then encounter a "1" which moves us over to S2, which is the accepting state. But we are not done yet as we have not traversed every letter in the word. The next "1" moves the machine to S0, then "0" to S1, and the final "1" moves the system to S2 which is the accepting state and the process is complete.

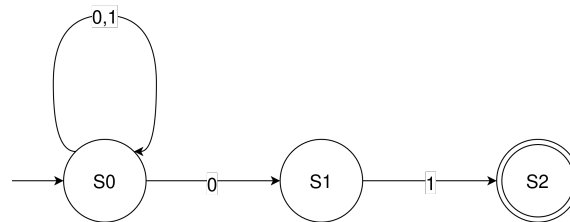


Figure 2.2: Example Non Deterministic Finite State Machine

For comparison, figure 2.2 is a non-deterministic finite state machine, which also accepts any word that ends with "01". When we take the same word, "01101", as before the traversal of the machine is different. when in the starting state S0, when a 0 is encountered, the machine can either stay in S0 or move to S1. This is the non-determinism represented. With one word we can end in multiple states. A word is accepted if at least one of the states which we end in is an accepting state.

2.2. Learning

The goal of learning techniques, both active and passive, is to reason about or infer the internal state machine of a system. The two main philosophies in learning are passive and active approaches. In passive learning there is no interaction with the system under learn, while in active learning, the learner gets to interact with the system.

2.3. Active Learning

Active state learning is the learning of a system's internal state machine through actively interacting with the system under learn a basic diagram is shown in Figure 2.3. In practice, this means generating a sequence of inputs, querying the System Under Learn with those inputs. The SUL will give outputs based on your defined sequence, the learner analyses these outputs and tries to generate a valid hypothesis state machine. If more information is necessary to generate a valid hypothesis, the learner will continue to generate sequences to query the SUL. This is called the learning phase, and queries to the SUL in this phase are called membership queries.

Once the learner has enough information to generate a valid hypothesis, this learning phase is complete.

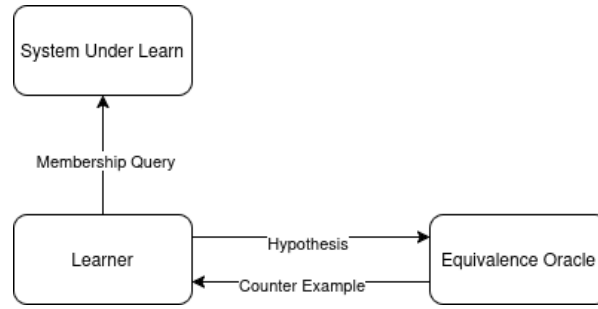


Figure 2.3: Basic Active Learning Diagram

The learner then submits its hypothesis to an equivalence oracle. The equivalence oracle determines if the generated hypothesis matches the internal state machine of the system. When the oracle cannot find a difference between the hypothesis and the internal state machine, the learning campaign is complete. However, if the two do not match, the oracle must return a counterexample to the learner. The learning algorithm will then incorporate the counterexample into its hypotheses and continue asking membership queries until its hypothesis is valid and complete again. This cycle repeats until the oracle cannot find a counterexample.

2.3.1. Learning Algorithms

There are various different learning algorithms that have been implemented for active automata learning. This thesis explores the use of algorithms such as L^* , $L\#$ and Kearns-Varzirani's algorithm. The intricate details of these algorithms are not necessary for understanding this thesis. For a more in-depth understanding of the behaviors depicted in the results, we recommend reading the original papers. [3][29] [20]

L^* , introduced by Angluin, is one of the original algorithms in automata learning from which other algorithms have grown. The algorithm learns a state machine through membership and equivalence queries. Membership queries ask the System Under Learn, for the output corresponding to a given input sequence, while equivalence queries determine whether the current hypothesis matches the actual system behavior. L^* maintains an observation table which is used to distinguish states and iteratively construct hypotheses until no counterexamples can be found. [3]

$L\#$, proposed by Vaandrager et al., differs from L^* by operating directly on an observation tree rather than maintaining a separate observation table. The observation tree closely resembles the generated hypothesis, simplifying the learning structure and reducing the need for large cumbersome data structures. This can improve scalability for larger systems while still using membership and equivalence queries to refine the hypothesis. [29]

Kearns-Vazirani's algorithm (KV), makes use of a discrimination tree instead of the observation table in L^* . The unique factor of the discrimination tree is that it allows for the learner to be more redundancy free, as only membership queries which contribute to the distinction of states have to be performed. [20]

It is important to note that learning algorithms can differ significantly in their internal data structures, methods of hypothesis generation, and reliance on equivalence queries.

2.3.2. Equivalence Oracle

In practice, we do not have an equivalence oracle, therefore we must determine if a hypothesis is equivalent to a system's state machine through systematically checking for counterexamples. One of these algorithms is Chow's W-method, in this paper however, we opt to use Wp-random (partial W method with random walk) an adaptation of the original W method.

The partial W method makes use of the following sets:

- The characterization set W is the set of input sequences which distinguish between the behaviors of every pair of states in the minimal state machine. [8]

- The transition cover set P is the set of sequences that ensure every transition in the hypothesis from one state to another is executed at least once. [12]
- The state cover set Q , set of input sequences that ensure every unique state in the hypothesis is visited at least once.
- The identification set W_i of state S_i , is the minimal set of input sequences required to create a unique output profile for state S_i , ensuring S_i is distinguishable from any other state in the system. The union of all identification sets is the characterization set.

The Partial W-method consists of two phases. Phase 1 verifies that every state defined in the specification exists in the System Under Learning and is uniquely distinguishable using its specific identification set (W_i). Simultaneously, the transitions leading from the initial state to these states are checked for correct output and state transfer. In Phase 2, the SUL is checked for all remaining transitions defined by the hypothesis, specifically those that were not checked during the first phase. [12]

Instead of implementing the vanilla partial W method, we have opted to take inspiration from LearnLib, a state of the art active learner, and implement the random Wp method. [17] Instead of sequentially running the test cases generated in the partial W method, we perform a random walk over the test sequences. This has shown to be more efficient in finding counterexamples. [17] We opted for the random Wp method due to its improved performance over the regular W method as well as the regular Wp method. [1]

2.4. Passive Learning

As opposed to active learning, passive learning attempts to learn the System Under Learning's internal state machine through passive analysis. This is usually done by processing the system's logs when it is under use. Passive learning requires for there to be interaction with the system, such that data is created from which state machines can be inferred.

2.4.1. FlexFringe

The passive learner utilized in this thesis is FlexFringe [32][31]. FlexFringe is a learning algorithm which learns probabilistic deterministic finite state automaton (PDFA) through a state merging algorithm.

A probabilistic finite automata (PFA) is similar to a finite state machine, with the biggest difference being the addition of a transition probability function. The inclusion of the transition probability function means that when in a specific state, any input can take multiple transitions with respective probabilities. The formal definition is as follows:

- $A = \{\Sigma, Q, I, T, F\}$
- Σ is a finite alphabet, Q is a finite set of state, $I: Q \rightarrow [0, 1]$ is the initial probability function, $T: Q \times \Sigma \times Q \rightarrow [0, 1]$ is the transition probability function and $F: Q \rightarrow [0, 1]$ is the final probability functions.
- the probability functions are defined such that $\sum_{q \in Q} I(q) = 1$, and for all $q \in Q : F(q) + \sum_{a \in \Sigma, q' \in Q} T(q, a, q') = 1$

A PDFA is a PFA with the additional requirement that the initial and transition probability functions are deterministic. Meaning that for each state and symbol, there exists only one transition to the next state. So unlike the the PFA where the transition is probabilistic, a PDFA could be defined using a symbol probability function.

Flexfringe can learn PDFA's from input traces produced from the system logs through state merging. Flexfringe cannot take raw logs as an input. The raw logs have to be converted to Abbadingo format, which is defined as follows:

- A header line containing two integers:
 - the number of samples N ,
 - the size of the alphabet $|\Sigma|$.
- N subsequent lines, each representing a single labeled sample.

Each sample line has the structure:

$$\langle \text{label} \rangle \langle \text{length} \rangle \langle s_1/r_1 \ s_2/r_2 \ \dots \ s_k/r_k \rangle$$

where:

- $\text{label} \in \{0, 1\}$ indicates whether the string is rejected (0) or accepted (1),
- length is the number of symbols in the string,
- each $s_i \in \Sigma$ is a symbol from the finite alphabet, typically encoded as integers.
- each r_i is the output following the input s_i

The basic state merging algorithm works as follows. There is an input sample which is a finite data set of example sequences (the traces generated from logs). The goal is to find a small PDFA which is consistent with the input sample mentioned. State merging start with a large tree shape model which captures the input data exactly. The the algorithm iteratively merges states together, which intuitively shrinks the model. The process ends when it is no longer possible to merge states, this happens when all possible left over merges result in an inconsistent model. [31]

Consistency in PDFAs is an important concept. A merge is consistent when for two state u and v when the future sequences in the input sample which occur after reaching u or v are similarly distributed. In essence, this is a Markov property, the future is independent from the past given the current state. This also means that States in a PDFa can be considered as clusters of prefixes with similarly distributed suffices. [31]

An important caveat is that even though states can be considered clusters, running a simple clustering algorithm will lead to non determinism. Therefore, a determination constraint is necessary. If two prefixes are clustered, then when they are extended by the same symbol from the input cluster, the respective resulting states must also be clustered. [31]

FlexFringe implements an evidence-driven state merging algorithm using the Red-Blue framework. In this framework, states are divided into two sets: red states, which represent stable states in the learned model, and blue states, which are candidate states for merging. Only merges between red and blue states are considered, which significantly reduces the number of possible merges and improves efficiency compared to unrestricted state merging. If no consistent merge can be found for a blue state, it is promoted to a red state. [21]

To efficiently perform and revert merges, FlexFringe uses the Union-Find data structure. This allows the algorithm to evaluate and score candidate merges before deciding whether they improve the resulting model.

2.5. System Under Learn

The system chosen for this thesis is OAuth 2.0. This implementation was chosen for the following reasons, its microservice architecture, the fact that it has an established code base that has been proven to work, and the importance of the system itself. In addition, if this implementation can be learned, hopefully the methods developed can be easily applied to other implementations of this same protocol.

2.5.1. Microservice architectures

Modern day large-scale software is commonly built using a microservice architecture. Unlike a more traditional molith structure, where the entire application is one unit, microservices break business logic into small, independent service that communicate over a network.

The modular approach of microservices allows teams to develop, deploy, and scale specific features simultaneously without impacting the system as a whole. A nice analogy to describe microservices is the following, consider a bureaucracy, each service is a specialized department. While every department performs one specific task, they must coordinate to ensure the entire bureaucracy functions as a whole.

2.5.2. REST API

Microservices need to communicate with one another, this communication can be done using REST APIs, which an acronym for Representational State Transfer Application Programming Interface. REST API is a communication protocol that allows for the communication between services over a network. A client is able to send a request, which is then processed by the application, the application can then return a response, usually with status codes attached.

1. **The Request:** The client sends an HTTP request using a specific method to define the intended action:
 - GET: Retrieve data from a resource.
 - POST: Create a new resource.
 - PUT/PATCH: Update an existing resource.
 - DELETE: Remove a resource.
2. **The Response:** The server processes the logic and returns a response, usually containing a **JSON** payload and an HTTP status code indicating the outcome.

Standard status codes are categorized by their numerical range:

- **2xx (Success):** Indicates the request was successfully received and processed (e.g., 200 OK).
- **4xx (Client Error):** Indicates a problem with the request, such as incorrect syntax or lack of permissions (e.g., 401 Unauthorized or 404 Not Found).
- **5xx (Server Error):** Indicates the server failed to fulfill an otherwise valid request (e.g., 500 Internal Server Error).

REST is inherently stateless, meaning each request from a client must contain all the information necessary for the server to understand and process it. The communication cycle follows a standard Request-Response pattern:

2.5.3. SUL: OAuth 2.0

The system under test chosen for the initial experiments is Ubiqu's implementation of the OAuth 2.0 framework, further referred to as OAuth [15]. OAuth is an authorization framework which allows a third-party application to access a user's protected resources on another service while keeping the login credentials secret. This implementation was specifically chosen due to the following facts: it has a microservice architecture, it is stateful, it requires certain cryptographic mappings in requests, and this system has been running on Ubiqu's servers allowing for there to be existing logs.

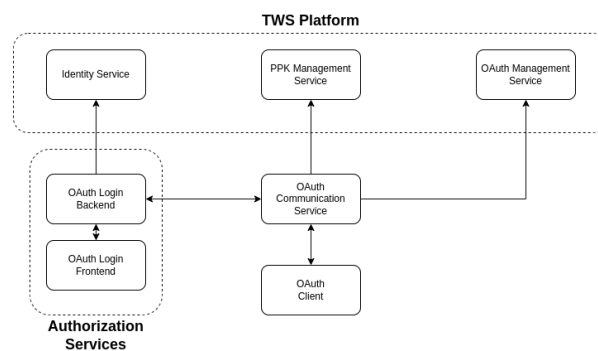


Figure 2.4: OAuth 2.0 architecture as implemented by Ubiqu. Each box represents a microservice, and the arrows represent communication flows.

The system architecture can be seen in figure 2.4. Each service is responsible for tasks in the OAuth flow. This diagram however, does not include all the necessary background services needed to run the system as a whole. Figure 2.4, includes the Authenticate App and the Identification and Transaction services. It was chosen to limit the initial experimentation to the services shown in both figures as otherwise the setup and testing would become too complex for our proof of concept experiment.

2.5.4. Functionality

OAuth is the industry standard protocol for authorization. OAuth distributes access tokens which represent the authorization to access specific resources on behalf of the end user. A real example would be granting a website personal information. As this protocol is widely used and involves the protection of data, it is crucial that the implementation does not contain flaws. Flaws, specifically related to granting access tokens could lead to data breaches.

3

Previous Works

In this chapter, we review the primary concepts and related research necessary for motivating and understanding our main research question. We begin by defining stateful fuzzing and its various approaches (Section 3.1). We then explore the role of automata learning in modeling system behavior for this context (Section 3.2). Finally, we discuss the specific challenges and techniques for fuzzing complex distributed systems and security protocols (Section 3.3).

3.1. Stateful Fuzzing

Fuzzing is a software testing technique that involves generating inputs to a program. These inputs can be invalid, random or unexpected, in addition to valid and expected data. A critical distinction exists between stateless and stateful fuzzing [9]. A **stateless** system, such as a file parser, takes a single input and produces an output. Fuzzing such a system involves generating many variations of that single input. In contrast, a **stateful** system takes a *sequence* of messages as input, where each input can cause an internal state change that affects the processing of subsequent inputs. Most network and security protocols are stateful [9].

A stateful system introduces a second level to fuzzing campaigns. The fuzzer must test not only the language of individual messages (the message format) but also the language of *traces* (the sequences of inputs). A comprehensive survey of stateful fuzzers [9] identifies several categories based on their approach:

- **Grammar-based:** Use a user-provided grammar to craft malformed messages.
- **Grammar-learner:** Automatically extract a grammar through passive or active learning. The fuzzing campaign consists of two phases, a learning phase and a fuzzing phase.
- **Evolutionary:** (e.g., AFL) Use feedback, typically code coverage, to guide the mutation of sample traces.
- **Evolutionary-grammar-based:** Combine a provided grammar with an evolutionary mechanism to mutate traces. They require a grammar as the starting point to generate messages.
- **Evolutionary-grammar-learner:** A complex, black-box approach that uses two feedback loops: one to mutate individual messages and another to infer a protocol state machine by mutating sequences.
- **Man-in-the-Middle (MITM):** Sits between a client and the System Under Learn to modify messages in transit.
- **Machine Learning:** Use an ML model trained on input traces to generate mutated traces. They require a set of traces to be used as a dataset to train the machine learning model, once the model is trained it can output mutated traces slightly different from the original dataset.

A significant challenge in this domain is the detection of logic-based vulnerabilities. Memory related vulnerabilities (like buffer overflows) often cause explicit crashes. In contrast, logic-based vulnerabilities

such as authentication bypass, downgrade attacks, or credential swapping, manifest as deviations from security guarantees and often lack externally recognizable effects [35]. This necessitates techniques that can understand and model the system's intended stateful behavior.

3.2. Automata Learning for Stateful Fuzzing

To find logical flaws, a fuzzer must first understand the system's state machine. Automata learning, or model inference, provides a set of techniques to construct these state models from observations. These techniques can be broadly divided into active and passive learning.

3.2.1. Active Learning

Active learning involves interactively probing the system under learn to build a model. A notable example is the use of **LearnLib**, an implementation of the L^* algorithm, to analyze TLS implementations [25]. This grey-box approach requires an alphabet of abstract messages and a method to reset the system under learn. L^* learns a system through actively sending sequences (membership queries) and proposing models (equivalence queries), it infers a state machine that can then be analyzed by hand or with a model checker to find logical flaws.

3.2.2. Passive Learning

Passive learning, in contrast, builds a model by observing existing system behavior without actively influencing it. This is useful for monitoring systems in production. **FlexFringe** is a framework that learns Probabilistic Deterministic Finite Automata from unlabeled trace data using a heuristic state-merging algorithm [31]. This approach has been used to monitor Kubernetes clusters by learning models from NetFlow data; new traces with a low likelihood probability in the learned model are flagged as anomalies [7]. These passively learned models can also be used for conformance analysis, comparing a model learned from a live deployment against a model derived from its implementation [7].

The models learned through these techniques can directly guide the fuzzing process. For example, **AGLFuzz** defines expected protocol properties using Linear Temporal Logic and fuzzes with the negation of these properties. It steers its test case generation based on the proximity of the current state to an accepting (error) state in the resulting automaton [35].

3.3. Fuzzing Distributed Systems and Protocols

Fuzzing microservices, REST APIs, and other distributed systems introduces challenges not present in monolithic applications. These include [10]:

1. **Inconsistency:** The distributed and asynchronous nature means execution paths may not be repeatable or predictable.
2. **Communication Overhead:** Feedback mechanisms that require extensive network communication can slow down the fuzzing process.
3. **Applicability:** Many fuzzers are not designed to handle the extensive codebases and complex interactions of industrial microservice environments.

3.3.1. API and System-Level Approaches

Several interesting approaches have been developed to tackle these issues. **SGPFuzzer** is a state-driven grey-box fuzzer that uses both code coverage and state feedback to autonomously learn and expand its state machine model [34]. For REST APIs, evolutionary algorithms are often used. **RESTier** is a stateful black-box fuzzer that performs static analysis on an OpenAPI (Swagger) specification to infer dependencies and generate test sequences [5]. Another api fuzzer is WuppieFuzz, which takes an initial alphabet as input. what makes WuppieFuzz different is that request dependencies can be defined in its corpus. [24]

A more advanced grey-box approach, **MISH** (Model Inference Search Heuristic), integrates passive learning directly into the fuzzing loop [6]. MISH is implemented within the **EvoMaster** evolutionary test generator. As EvoMaster executes test cases, MISH uses FlexFringe to passively learn a state machine from the system under learn's log statements in real-time. This model is then used as a novel

search heuristic: the fitness function rewards test cases that trigger deep or rarely-visited paths in the inferred state machine, guiding the fuzzer toward complex system-level behaviors [6].

3.3.2. Handling Protocol-specific Dependencies

A final challenge arises when fuzzing protocols with complex dependencies, such as encryption, session management, or domain-specific message formats. One effective strategy is the **Man-in-the-Middle (MITM)** architecture. **SECFUZZ**, a fuzzer for stateful security protocols like IKE, positions itself between a client and server to intercept and mutate traffic. This modular design allows it to handle encrypted messages without needing access to keys, focusing instead on the protocol's state logic [27].

When a state model is used, an abstract mapper is essential. A fuzzer for IPsec, which operates on hardware and uses encrypted tunnels, employed a black-box approach with a mapper. This component handles all encapsulation, transmission, and parsing, allowing the fuzzer's state-selection algorithm to operate on an abstract level [14]. AGLFuzz also relies on a protocol-specific mapper to convert abstract input/output symbols from its automaton into the concrete messages required by the system under learn [35]. This abstraction is key to creating scalable and maintainable stateful fuzzers.

3.4. Noise

When learning on a system which is deployed on a development cluster, the learner loses some control over the SUL. In addition, due to all communication with the SUL happening remotely, some noise or imperfection is introduced to the learning process. Both of these issues have been subject of previous works.

First, we address the lack of control over the SUL. Previous work has shown if the learner does not have control over the database, the learner must try and find a homing sequence that returns the system to its root node.[13] As we are not able to reset the database during the learning process, this concept is not applicable however, as we can take a different approach. Through the use of unique IDs while learning, we can ensure the state of each user and client interacting with the SUL is reset, due to the fact that we give the learner the ability to re-register with a new UUID, effectively resetting the state.

Next we can cover the presence of noise. Previous work suggests we can simply do a majority draw, which will mean the system learned is probably correct. [4] Majority draw implies that throughout the learning process each query is sampled multiple times in order to achieve a baseline behavior of the system. Which allows for the learning algorithm to learn the most probable behavior. This method is something we hope to adapt; however, due to the inefficiency in sampling traces over an HTTP network, this method must be adapted. In addition, the efficiency of this method is also reliant on the stability of the data, if the SUL becomes less stable overtime, this method will have to restart the learning process.

Noise has a significant negative impact on the efficiency of learning, as sampling needs to be introduced. The L^* algorithm, even though inefficient in the presence of noise, is relatively robust when using a probabilistic approximately correct substitute for equivalence querying. [33]

Previous research has also introduced methods for learning non deterministic real time automata through the introduction of a clock. [2] Although an interesting method, this is not applicable for our research. We know that in a perfect setting, the code should perform deterministically, and therefore believe there is an underlying reason such as internal communication that is the cause of the noise in the system.

3.5. Research Gap

From previous works, we conclude the following gap in the research. When learning microservices we believe they need to be learned as one system. The system itself is greater than the sum of its parts. This comes with a series of challenges, such communication with the SUL and potential noise introduced by the test setup. In order to handle the noise we propose a more efficient sampling technique which takes advantage of our REST API domain. Finally, in order to explain the origin of the noise, we propose the use of passive learning to learn a machine from a more complete alphabet.

Active Learning

4.1. Developmental SUL

For our developmental System Under Learn we created two simple services, one service serves as a user registration and user login. The second service is for requesting and submitting challenge once logged in. Crucially, the status of the user is communicated from the login service to the challenge service through an internal call. That internal call for the purposes of setting up the active learner is deterministic. However, to be able to mimic real word scenarios, we can introduce a non deterministic latency issue which delays the internal request being sent.



As mentioned for the purposes of creating a valid active learning setup, we enforce determinism on the SUL. The final learned model (Learned by L#) can be seen in Figure 4.1. The most simple happy flow is as follows, first the user must register and login, then the user will submit a 2fa request. That step will unlock the challenge endpoints in the second service. This is where the user can request a challenge. This challenge should be signed and then can be submitted again. The flow gets more complicated when logging out is also enabled.

4.2. Communication with SUL

To facilitate querying the SUL, for both learning and equivalence queries, we implemented a JSON corpus structure, and an orchestrator to parse and handle the traces. The definition of the corpus is described in Section 4.2.2 and the orchestrator is defined in Section 4.2.1

4.2.1. Orchestrator

The learner creates traces based on the corpus defined previously. These traces contain the necessary information for the orchestrator to create valid inputs, through function definitions and mappings to previous requests or responses. If the orchestrator does not have the necessary information, for example when a request in a trace refers to a response that is non-existent, the orchestrator will simply give a 400 response without querying the SUL, this allows for the rest of the trace to be further queried.

The corpus/alphabet definition as defined in Section 4.2.2 gives the orchestrator enough information to transform all membership queries asked by a learner into valid requests for the System Under Learn. The communication flow of the learning process is pictured in Figure 4.2. As one can see the learner receives an alphabet from which membership queries are generated, these are passed to the orchestrator which turns the queries into valid HTTP requests. These requests are fed to the System Under Learn

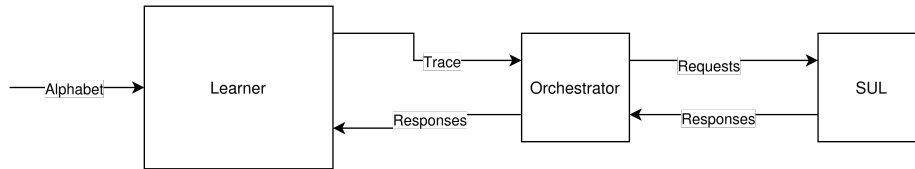


Figure 4.2: Figure representing the communication flow between learner and System Under Learn

4.2.2. Corpus definition

In an effort to generate traces, which can be directed at different microservices, we have to define an alphabet that contains the following information, endpoint, path, request type, request body, and headers. In order to deal with dynamic responses from the System Under Learn, the chosen implementation of the alphabet must also be able to dynamically reference previous responses within its requests. The alphabet must also allow for the ability to perform computation on the data dynamically, this is example useful for specific parsing or signing of cryptographic challenges. Easy definition of computational functions within the alphabet is crucial for applying this active learning method.

```

1 [
2   {
3     "template": "register_user",
4     "url": "HTTP://host.docker.internal:5001/",
5     "path": "api/register",
6     "method": "POST",
7     "data": {
8       "username": "user_{range(1,10000)|random}",
9       "password": "secure_password_123"
10    }
11  },
12  {
13    "template": "login",
14    "url": "HTTP://host.docker.internal:5001/",
15    "path": "api/login",
16    "method": "POST",
17    "data": {
18      "user_id": "{responses.register_user.JSON.uuid}",
19      "password": "secure_password_123"
20    }
21  },
22  {
23    "template": "request_challenge",
24    "url": "HTTP://host.docker.internal:5002/",
25    "path": "challenge/request",
  
```

```

26     "method": "POST",
27     "data": {
28         "user_id": "{{_responses.register_user.JSON.uuid_}}"
29     }
30 },
31 {
32     "template": "submit_challenge",
33     "url": "HTTP://host.docker.internal:5002/",
34     "path": "challenge/submit",
35     "method": "POST",
36     "data": {
37         "user_id": "{{_responses.register_user.JSON.uuid_}}",
38         "signature": "{{_sign_nonce(responses.request_challenge.JSON.challenge_nonce)_}}"
39     }
40 }
41 ]

```

Our corpus definition takes a lot of inspiration from WuppieFuzz, however it differs at a few crucial points. [24] Below are a few request definitions which display some of the design decisions made. Each request contains a url and a path, which tell the orchestrator where to send the generated request. The login template is a good example of referencing previous responses from the System Under Learn, it contains the user_id as generated by the system after a registration request. Importantly, the submit_challenge template includes the sign_nonce function mapping, which allows the orchestrator to dynamically apply computations to previous response data.

When the orchestrator is unable to produce valid input data based on its definition in the corpus, then it does not send the request to the System Under Learn. Instead, the orchestrator assumes the response from the system will be a 400 Bad Request. This design decision is made in order to minimize the amount of requests sent to the actual system, as each request is relatively time consuming and inefficient. In our setup it is possible to define default values such that requests can still be generated and sent, we however opt not to do this. An example of how to define default values is as follows:

```

1 [
2 {
3 {
4     "template": "submit_challenge",
5     "url": "HTTP://host.docker.internal:5002/",
6     "path": "challenge/submit",
7     "method": "POST",
8     "data": {
9         "user_id": "{{_responses.register_user.JSON.uuid_|_Missing_}}",
10        "signature": "{{_sign_nonce(responses.request_challenge.JSON.challenge_nonce)_}}"
11    }
12 }
13 ]

```

In this case the user_id is defined as missing when the register_user request has not been performed.

Defining endpoints/paths

We not only need to define request paths but also request endpoints as there are multiple domain names to which requests need to be sent to complete the OAuth flow.

Crucially, for fuzzing microservices, each request defined in the alphabet should be addressed to a specific service and endpoint, we don't believe this is possible in scratch WuppieFuzz. Having these endpoints be defined in the alphabet and handled in the orchestrator allows for the reuse of a code base and easy expansion or alterations to the alphabet, or when different projects are chosen as the System Under Learn.

To show that learning the systems separately does not necessarily result in a hypothesis matching the interaction as a whole, we learned the systems separately. The resulting machines are displayed in Figures 4.3 and 4.4. For learning the challenge service, we manually created a user that was already logged such that the learner could actually complete the process of requesting and signing a challenge.

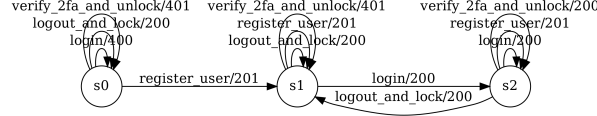


Figure 4.3: Final state machine of Login Service learned by L#

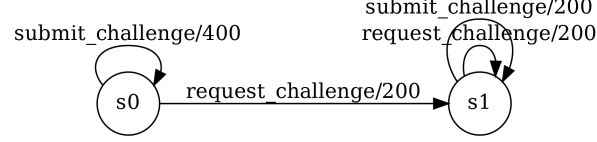


Figure 4.4: Final State Machine of Challenge Service learned by L#

Mapping Previous Requests and Responses

If previous responses or requests cannot be referenced, a lot of hard coding needs to be done, this could require editing the SUL, for example if the SUL generates a UUID, or a code challenge, that would have to be referenced to create valid requests in the future.

Referencing previous responses and requests possible in WuppieFuzz, similar system used in my method. What is limited in my research is that when we reference a previous response in a request, we choose to use the last occurring instance in the trace. This choice is made because we wanted consistency in the traces, which means choosing randomly is written off. We chose not to reference the first occurrence because we wanted to imitate user interaction as closely as possible, but the first occurrence is still possible with a simple adjustment, and we expect the resulting state machine to be different.

This design decision can be seen in Figure 4.1. A `register_user/201` transition leading to state s4 can immediately lead to a `verify_2fa_and_unlock/200` self loop. This 200 response is still referencing the previously logged in user.

Function Mapping

As mentioned at the start of this chapter, we need to map functions in requests to produce cryptographic inputs. not possible WuppieFuzz. For this we make use of the JSON data structure and the Jinja2 package, this allows us to define functions that can be mapped into the data.

Function mapping is important for dynamically creating valid inputs for the System Under Learn, think of performing computations, signing code challenges, or even parsing previous responses. As one can imagine, when statefully learning a system, it is vital that one can easily and in an automated fashion create valid inputs necessary for triggering state transitions.

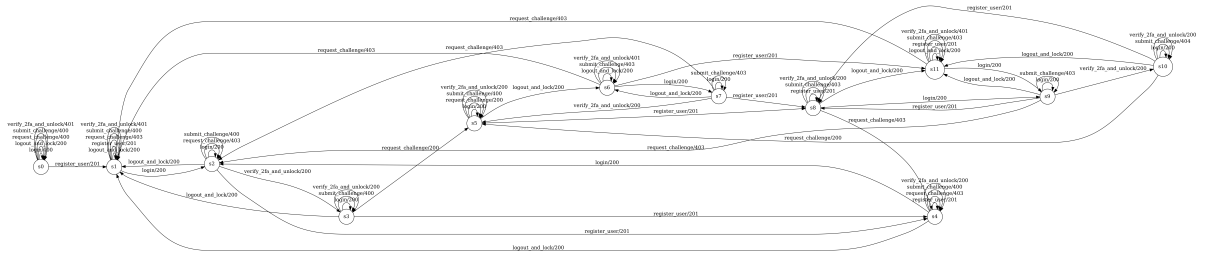


Figure 4.5: Final state machine of test system learned by L# without function mapping

In Figure 4.5, we show the result of a learning campaign excluding the function mapper for signing the challenge. Because the challenge endpoint expects a valid signature, as generated by the request challenge endpoint. When comparing this state machine to the one in Figure 4.1, the two look very similar except for the fact the machine learned without function mapping never succeeds in getting a

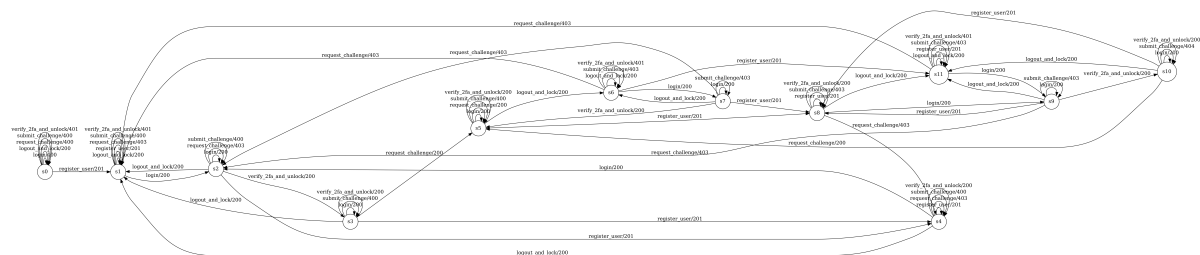


Figure 4.6: Final state machine of test system learned by L# without session management

successful submit_challenge/200 request. Which shows that function mapping is necessary when the learner needs to perform any type of computation to create valid requests for the System Under Learn.

Session Management

In addition to properly generating requests bodies, it is necessary for the orchestrator to dynamically set headers for session handling. Session handling through global session attribute, some requests require reusing and updating X-csrf tokens. This is another part where WuppieFuzz and my active learning implementation differ, as far as i know, with WuppieFuzz it is possible to login manually and insert necessary cookies at the start of a campaign, but it is not possible to dynamically receive and update these headers. When these tokens are updated to prevent replay attacks, they also need to be updated in the learner.

For all of our interactions with the SUL and the microservices we use an orchestrator that handles the session and header management. In addition, this orchestrator is capable of parsing the trace defined as a JSON, and sending each request to the necessary service for a response.

Figure 4.6 displays the machine learned by the campaign without session management. This machine looks the same as the one in Figure 4.5, which is expected. This is because only the challenge service utilizes session management through setting cookies, and without a valid cookies one cannot submit a valid challenge signature. Which shows when cookies are dynamically set, the learners cookies must be dynamic as well to create valid interactions.

4.3. Algorithm Selection

In this section, we discuss the algorithms chosen for our active learning campaigns. In the learning campaign two algorithms need to be selected, both a learning algorithm aswell as an algorithm for the equivalence oracle.

4.3.1. Equivalence Oracle

Importantly, for active learning we need to define an equivalence oracle. As the validity of the final state machine is dependent on the oracle, we prioritize efficiency. We want the oracle to determine whether the hypothesis matches the SUL using the least amount of resources. This ensures that counterexamples can be found in a reasonable amount of time and the learning process will not consume more resources than a developer is willing to invest.

These needs quickly eliminate Chow's W method as a viable equivalence oracle. Chow's W method constructs a characterization set, which is a selection of input sequences that can distinguish the behavior of every pair of states. This process can become very costly in time and space.

Instead of the W method, we chose to implement the W_p random method, due to its time advantage over the standard W method for equivalence. The standard W method simply produces too many queries for the SUL, and in our case the SUL is very slow, therefore too much time is lost in equivalence querying. W_p random is a variation of the W method, it is very similar, however it provides shorter test sequences. When randomness is introduced as well, finding counterexamples to the hypothesis becomes more efficient and therefore quicker. Which again is necessary when applying active learning in a small scale development environment.

4.3.2. Learning Algorithm

As previously discussed, we rely on Wp random to act as an equivalence oracle for the System Under Learn due to its major efficiency advantage over the original Chow's W method. The downside of this choice is that randomness makes it difficult to capture non-deterministic behavior. Therefore when selecting a learning algorithm, we want to have a good balance between the randomness of the equivalence oracle and the structural building of a cache to avoid too many hypothesis rebuilds.

Due to our learning setup it is possible to compare multiple learning algorithms without having to implement any major changes, besides of course the integration of the algorithm itself. In this Section we compare the following algorithms:

- L# [29]
- TTT [18]
- L* [3]
- ADT [11]
- Kearns-Vazirani [20]
- RS (Variant of L*) [23]
- DHC (Direct Hypothesis Construction) [22]

L# and L* were implemented by me, the rest of the algorithms were integrated from the LearnLib java library [17]. All algorithms were compared on the same equivalence oracle using the Wp Random method with a maximum budget of 5000 queries. RS and DHC were unable to find all twelve states with this budget, therefore their maximum equivalence testing budget was doubled.

For each algorithm we ran 30 campaigns on the developmental System Under Learn on which we enforced determinism. We chose for 30 campaigns in order to mitigate the effect of the counterexamples returned by the equivalence oracles. If the System Under Learn is not deterministic, all of the campaigns would fail.

The results of the campaigns are plotted in Figures 4.7 and 4.8. Figure 4.7 displays the average amount of membership queries asked to the oracle for the amount of states discovered by the learner. While Figure 4.8 shows the average amount of times the equivalence oracle was queried for each state found by learner.

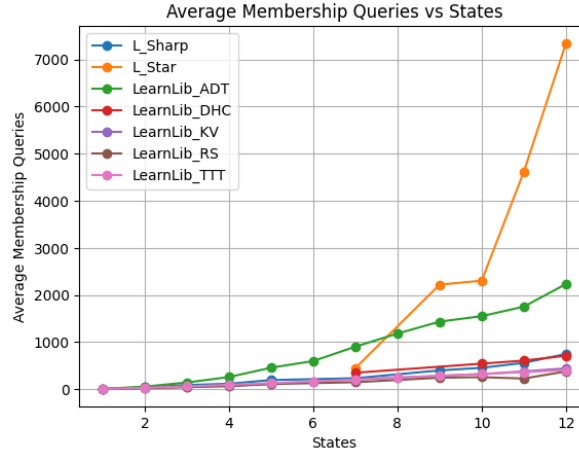


Figure 4.7: Graph showing average membership queries vs states found over 30 learning campaigns per learning algorithm

First we consider regular L*, although very consistent in building its cache, we determine it is not space or time efficient enough when learning systems with a large alphabet and possibly a very large amount of states. Therefore, we opted out of using L* simply because it is not efficient enough to be used outside of this experiment in this setting. L* creates an observation table instead of tree as described in background section for L#. The building of this observation table is relatively expensive

as it systematically eliminates all unknowns. Therefore, despite its advantages (consistent and widely proven, we chose not to use this algorithm for further experimentation, as it would not be viable in a production setting. As can be seen in figure 4.8, L^* does not perform many equivalence queries and is one of the top performers. However, when evaluating the membership queries performed, seen in figure 4.7, L^* needs to perform the most, which is expected.

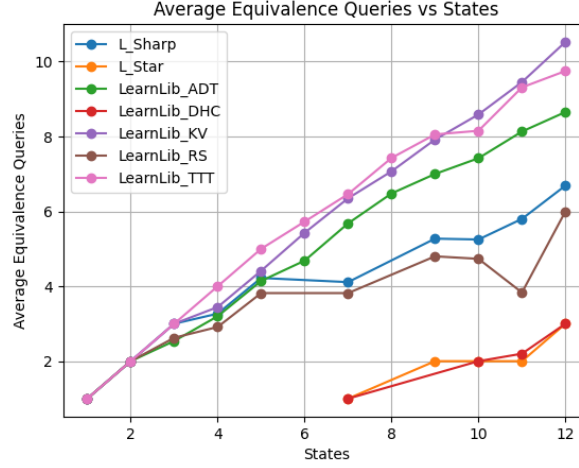


Figure 4.8: Graph showing average equivalence queries vs states found over 30 learning campaigns per learning algorithm

Next, we consider TTT [18]. TTT is highly efficient in a deterministic setting, as it relies heavily on the equivalence oracle, when that oracle is efficient so is TTT. In a non-deterministic setting however, this is its downfall. As our equivalence oracle introduces randomness, it becomes hard to distinguish results which should be ignored immediately. Which will cause a lot of costly hypothesis restructures in the learning process. TTT is fast and current state of the art, it relies heavily on a good counterexample oracle, as its philosophy is to fail quickly. Rapid iteration of hypotheses. Issue with TTT in our setting is an unstable environment. It is difficult to catch non-determinism caused by the environment, when there is a lot of randomness in our learning algorithm. When looking at the figures, the results are to be expected from TTT, it is among the top performers in membership queries, however the trade off is that it relies on asking many equivalence queries.

$L\#$ serves as a happy medium between the two algorithms. The replacement of the observation table with an observation tree is more time and space efficient, in addition the algorithm is less reliant on equivalence querying and therefore less randomness is introduced to in the learning campaign, some adjustments still need to be made to the original algorithm to ensure when instability goes uncaught it is fixed. TTT although more efficient, catches non determinism randomly and therefore is most likely less efficient.

DHC excels when cost of membership queries are low and the amount of states are very high. Not the case which is not the case in the SUL. It performs similarly to both ADT and KV. Finally, LearnLib RS seems to perform well, however it failed to find all 12 states two of its 30 runs, making it not as reliable as L^* for example which found all twelve states in all 30 learning campaigns, making it efficient, but unreliable.

4.4. Discussion

This chapter serves as a validation of our design decisions for learning microservice architectures. Learning the services as one system provides significant insights compared to learning each service in isolation, as shown by comparing the standalone state machines (Figures 4.3 and 4.4 with the complete system model (Figure 4.1). We outline the necessary mechanisms for this approach, specifically the ability to reference previous requests and responses. In addition, custom function mapping is essential for generating valid requests, as demonstrated in Figure 4.5. Finally, Figure 4.6 highlights the importance of session and header management when learning stateful REST APIs.

Our learning framework allows for the easy switching between learning algorithms, enabling a direct performance comparison on the SUL. We see that the algorithms perform as expected based on their respective reported results. A general observation about the algorithms is the trade off between membership and equivalence queries. With the exception of DHC, most algorithms seem to either rely on a high number of membership queries for each state found or a high number of equivalence queries for learning.

The results from this chapter bring us one step closer to learning microservices, the learning algorithms mentioned are inherently not robust against unexpected flakiness or non-deterministic responses. As mentioned earlier, when learning on a live system over an internet connection, unexpected errors can occur. In order to apply the methods derived in this chapter on a real world scenario, we first must ensure the learning algorithms do not break when they receive an unexpected input.

In summary, in this chapter we introduce a learning framework which allows for the active learning of microservices architectures. The framework dynamically produces valid API requests through response and function mapping into a predefined template. This allows for learning deeper states for which actions such as signing data is necessary. We show the framework works through applying it to a test setup with various different existing learning algorithms.

5

Handling Non-Determinism

As mentioned in the Chapter 4, there is non-determinism in the OAuth test setup. This non-determinism can have very different causes, we believe one of these causes includes internal state drift. Internal communication between microservices is in various stages of completion when a follow up request is sent. In an effort to observe and explain this occurrence, we propose the use of passive learning in the setup.

5.1. Active Learning in a Non-Deterministic Setting

In order to learn a deterministic system in a non-deterministic setting, an expected behavior must be established. If expected behavior is not established before being introduced to the learner, the learning algorithms will crash when differing behavior is introduced. This is because the default learning algorithms cannot handle non-determinism.

For the expected behavior we prioritize success responses from the System Under Learn and otherwise the most common negative response. We determine the expected behavior has changed when the original expected behavior is negative and we receive a positive response, or when the original behavior's stability deviates from other behaviors by a specific threshold. If the expected behavior is updated, and it has already been committed to the learning algorithm, the learning attempt is deemed corrupted, and the attempt is abandoned. Then a new learning attempt is initialized with the existing cache.

When the SUL behavior deviates from the expected behavior, but not enough to change the cache, we retry the trace. This becomes an efficiency issue, as if a certain call is unstable, then repeating it multiple times compounds the instability of the trace. To mitigate this, we considered snapshot learning which makes snapshots of the SUL check pointing specific prefixes making querying more efficient. [28] Snapshots are copies of the System Under Learn when in a specific state. The use of snapshots allows the learner to initialize the System Under Learn repeatably and consistently without having to query the prefix which leads to the necessary state in question. This solution was however not applicable to our testing setup, as it would require taking snapshots from many microservices, including background systems for which access is limited. Initializing the snapshots would then also be prone to error as it is unknown if certain race conditions could still be active in the snapshots, potentially invalidating future traces. Meaning snapshot learning is infeasible for the scope of this project and the goal of learning a state machine for the OAuth implementation.

Since snapshot learning is not possible, we explore a different method to increase the querying efficiency. We propose a two phase learning protocol driven by the equivalence oracle. In the first phase, the equivalence oracle only returns counterexamples deemed to be stable by the cache defined in section 5.2. This will hopefully stimulate the learning algorithms to generate membership queries with more stable prefixes, allowing the cache to be expanded, while also exploring the stable part of the SUL first. When no counterexample can be found in the first phase, the equivalence oracle enters a second phase where it allows unstable counterexamples. Even though counterexamples can now be

unstable, the equivalence oracle ranks the found counterexample and returns the most stable one.

Algorithm 1 Deferred-Verification Active Learning for a Nondeterministic SUL

Require: Stability threshold τ , learning algorithm $\mathcal{A}$, happy-flow flag h

Ensure: Stable cache and learned model

Load alphabet Σ and symbol metadata

```

1:  $C \leftarrow \text{Cache}(\tau, h)$ 
2:  $S \leftarrow \text{InitLearningState}()$ 
3:  $\text{SUL} \leftarrow (\Sigma, C, S)$ 
4:  $\text{stable} \leftarrow \text{false}$ 
5:  $\text{attempts} \leftarrow 0$ 
6: while  $\text{stable} = \text{false}$  do
7:    $\text{attempts} \leftarrow \text{attempts} + 1$ 
8:    $H \leftarrow \mathcal{A}(\text{SUL}, \Sigma)$ 
9:    $\text{stable} \leftarrow \text{VerifyCache}(\text{SUL})$ 
10: end while
11: return  $(H, C, \text{attempts})$ 

```

Each learning algorithm is integrated in such a way that the equivalence oracle is queried in two phases, the equivalence oracle keeps track of which phase it is in. This allows for easy integration with the existing LearnLib algorithms with no adjustments necessary in their existing code base. The construction of our implementation also allows for the easy swapping between algorithms as well as injecting new ones.

5.2. Query Cache

In order to efficiently compare prefix results, we introduce a query cache. The cache is constructed as follows, each transition is defined as a symbol in the alphabet, and each node contains the result of executing that symbol on the SUL. Each node contains the expected result, as well as all results recorded. For each possible result, we also track the number of times it has occurred. Similarly to previous works, we determine the expected result to be the most common one. [4] We do make a slight adjustment to this strategy. As we are learning REST APIs, we take advantage of the HTTP protocol in the results. Instead of simply selecting the most common result, we give a bias towards results with 2xx results. Successful responses are usually not triggered accidentally and therefore our desired behavior. We allow for success responses to be less consistent, as we are interested in the traces which elicit this positive behaviors. We also believe positive behavior can be more linked to information leak, and unwanted successes can be more critical than unexpected failures.

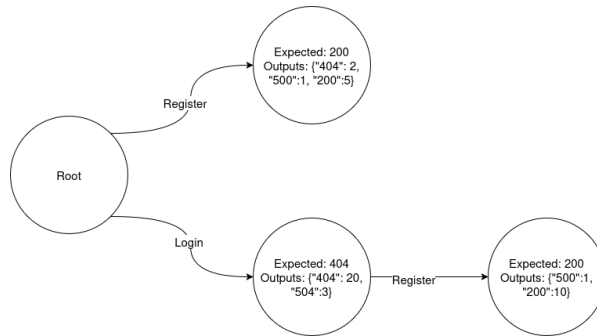


Figure 5.1: Representation of the implemented cache.

To avoid introducing non-deterministic traces in our chosen learning algorithm, we introduce a dynamic sampling method. Whenever a trace is queried from the learner which extends the cache by some suffix, we compare the suffix with previous occurrences stored in the cache. When a suffix is deemed unstable, meaning that the sequence of API requests defined in the suffix has multiple occurrences

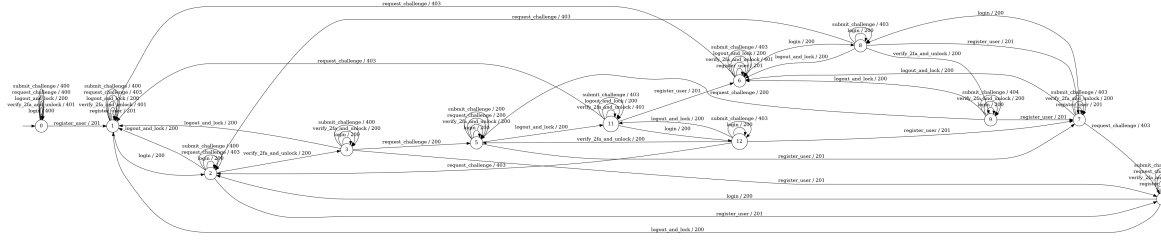


Figure 5.3: State machine learned through KV, with unreachable states removed

by the two phase Wp random method. From the figure it can be seen that KV has the lowest median query count with 7296 queries. L* has the lowest variation, with a standard deviation of 829 queries. DHC, L# RS and L* all have a similar median query count between ten and twelve thousand queries.

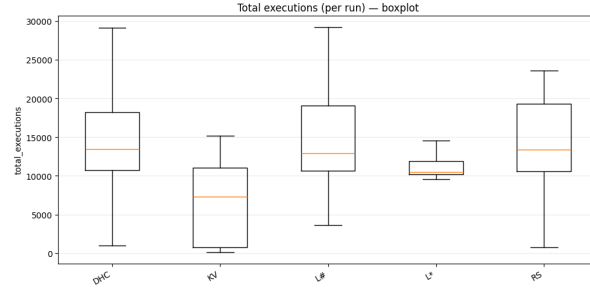


Figure 5.4: Box plot representing total executions per run for all tested algorithms

The next metric is the amount of equivalence queries performed by the learning algorithm, which means how many times the learning algorithm asks for a counterexample from the equivalence oracle. Figure 5.5a shows all the distributions of equivalence queries performed by each of the learning algorithms. Remarkably, KV has the highest medium by a relatively large margin at 89 equivalence queries. KV also has the highest standard deviation at 47 queries.

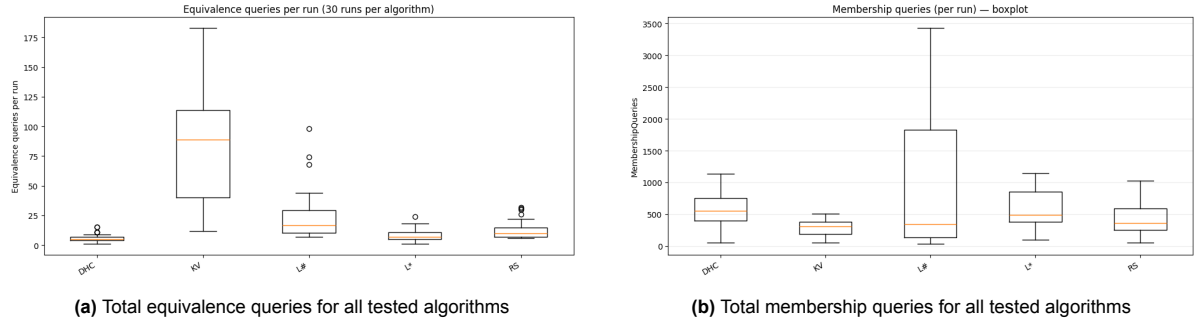


Figure 5.5: Comparison of query counts across algorithms.

In contrast to equivalence queries, KV outperforms the other algorithms in relation to membership queries with a median of 306. L# has a relatively similar median, however the standard deviation of L# is the largest at 3129 queries. The other algorithms have a relatively similar performance when purely compared on the membership queries asked.

Figure 5.6a shows the distribution of learning attempts per run. Each learning attempt implies the need for a restart due to non-determinism being introduced to the learner or when the SUL is too inconsistent for learning. When comparing the algorithms it is clear DHC has the fewest learning attempts, with RS having a similar performance. However, RS has a larger standard deviation when compared to DHC. KV needs the most learning attempts with a median of 8 attempts and a standard deviation of 4.35 attempts.

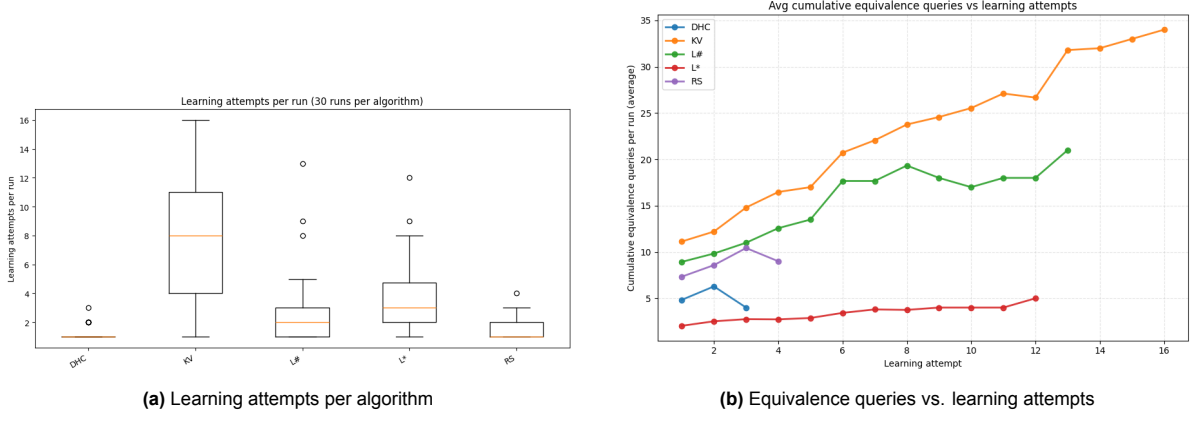


Figure 5.6: Relationship between learning attempts and equivalence queries across algorithms.

In an effort to better understand differing needs for learning attempts per algorithms, we plot the equivalence queries vs the amount of learning attempts in figure 5.6b. As can be seen, for L# and KV the average amount of equivalence queries increases with the amount of learning attempts.

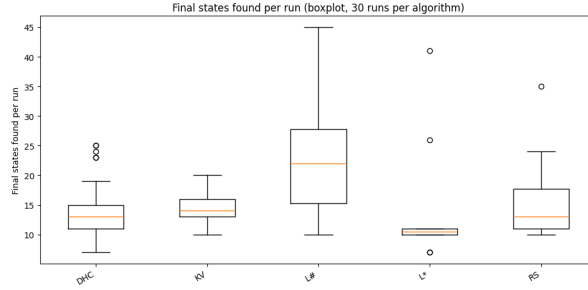


Figure 5.7: Final state machines found

Importantly, the accuracy of the algorithms is also important, in order to compare this, we look at the amount of states represented in the final models of each run displayed in figure 5.7. The SUL has 12 states in a deterministic setting, therefore we use that number as a baseline. L* is closest with a median of 10.5. DHC and RS both have a median of 11, and KV a median of 13. L# is the largest outlier as it has a median of 22 states, L# also has the largest standard deviation at 9.65 states. KV has the smallest standard deviation at 2.24 states.

Algorithm	Cache Hit Rate Avg	Cache Hit Rate Std	n_runs
KV	70.52	3.42	30
RS	69.06	3.40	30
DHC	67.72	8.81	30
L#	66.13	9.90	30
L*	64.38	17.02	30

Table 5.1: Average cache hit rate and standard deviation per algorithm over 30 runs.

Finally we look at cache hit rate, displayed in Table 5.1, cache hit rate gives insight in repeated queries over learning attempts. These repeated requests are answered from the cache and not the SUL. We can see that KV has the highest cache hit rate with RS being close behind. Notably L* has a very high standard deviation of cache hit rate.

5.5. Discussion

In this chapter, we show that it is possible to extract a deterministic state machine from a system which at times can be flaky. Through our adaptive sampling method, we are able to relatively efficiently

extract the expected behavior from the SUL and ensure that the learning algorithms always receive deterministic inputs. When the expected behavior of the SUL changes, or we deem that the system is too inefficient to learn in its current state, the learning process is restarted. The cost of restarting the learning process is mitigated as much as possible through maintaining a persistent cache.

Similarly to Chapter 4, we are able to learn using various different learning algorithms. When evaluating the performance of these algorithms, DHC and RS proved unreliable for this setup. Both algorithms exhibit a high variability in the total number of executions required and the final number of states discovered. Similarly, L# shows a high standard deviation in the number of membership queries asked relative to the states found. We suspect L#, at times, may have gotten stuck attempting to resolve deep changes in its observation tree, as both L# and KV showed a correlation between the number of equivalence queries and the number of learning attempts.

An important observation is that the learning algorithms which rely heavily on equivalence queries also have a tendency to needing more learning attempts. Meaning that the expected behavior of the learner is deemed to change more frequently or the system is too inefficient to learn. In our test setup, we know the latter is not the case. We speculate the changes of the expected behavior to be due to the reliance of equivalence queries, as the Wp random method does not systematically build the cache. Meaning randomness is introduced and behavior cannot be as easily verified. Which leads us to believe the higher the instability of the SUL, the more we want to rely on membership queries.

This leaves KV and L* as our most viable options when applying our learning methods on OAuth. KV shows a high execution efficiency, however it is burdened by the high restart rate. While L*, although relying on many executions, is very consistent and reliable.

In conclusion, we have now introduced a learning framework which allows for the learning of microservice architecture which sporadically act in a non-deterministic manner. We show that the Two Phase counterexample and adaptive sampling techniques work through applying the framework on a flaky System Under Learn.

6

Passive Learning to Catch State Drift

In this chapter, we propose our solution for combining the use of passive and active learning in order to gain a more complete understanding of the System Under Learn. In Section 6.3, we describe the reasoning and methodology behind our learning process. In Section 6.2, we introduce the concept of state drift, and its importance in analyzing the machines learned through both active and passive learning methods. Finally, in Section 6.4, we display and analyze the results of our proposed learning methods.

6.1. Purpose

Due to the intertwined dependencies of the microservices used in Ubiqu's OAuth implementation, internal communication could affect the state of the system as a whole. This means that active learning is limited in the transitions that can be learned. In reality, the alphabet of the System Under Learn is larger than its outside interactions. We therefore want to simulate and test these flaky dependencies through the development of our learning solution.

6.2. State Drift

After active learning is complete, we have a state machine that models the system as a whole as it would work ideally, this however is not the reality. As described in Chapter 4, we extract the expected behavior from the System Under Learn through ignoring the unexpected data. Through the use of passive learning we can explain these unexpected results through capturing internal race conditions.

Through issues like latency, there is the possibility for race conditions in internal requests made by the microservices. This influences requests made by the active learning algorithm, which in turn makes the System Under Learn seem to behave unexpectedly from the perspective of the learner.

Through our log processing we can observe all internal communications between the microservices, and are therefore able to expand the learning alphabet. The increase of the alphabet can then help explain the observed instability in the system. Any other causes of instability are not able to be observed in our set up, such as database transactions or system downtime caused by outside influences.

We refer to transitions that are unique to the passive learner, meaning the difference in the alphabet between the passive learner and the active learner, as internal transitions. From the perspective of the active learner these transitions are either traversed or not by the time it sends its next request, creating the appearance of non-determinism. These transitions can be traversed freely as the active learner does not have a direct influence on their execution.

We call this phenomenon state drift. Which refers to the possible states a machine can be in when only traversing these internal paths, which leads to different results.

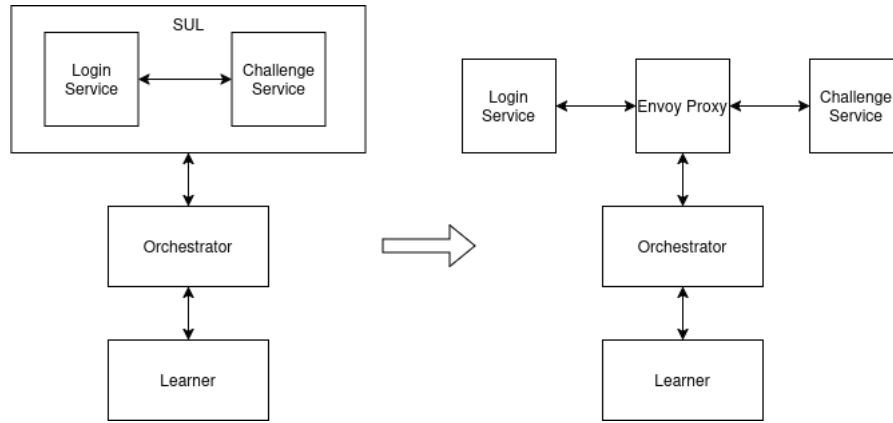


Figure 6.1: Diagram displaying setup to capture the logs of the learning process.

6.3. Methodology

This section describes the methodology and testing for the passive learning of non-deterministic microservice architectures.

6.3.1. Passive Learning and Active Learning

We choose to make use of passive learning for this task, as the internal communication between microservices is indirectly triggered by the active learner. Since the active learner does not have these internal communications as part of its alphabet, we must passively observe these interactions, which is why we proceed as follows. In order to create the logs necessary for applying a passive learner, we make use of the active learning setup defined in Chapter 4. This allows for the generation of logs without needing user interaction, which is necessary for codebases that do not yet have a large user base. In addition, user interaction with the application has a tendency to be representative of the expected behavior of the system. Specifically in the case of the OAuth systems implemented by Ubiqu, most interactions are done through a front end, limiting a user's ability to perform unexpected actions. Therefore, generating the logs through an active learning process produces more diverse and unexpected traces.

6.3.2. Log capturing

In order to capture all communications in logs, including all internal communications, we route all communications through an envoy proxy. The envoy proxy serves as a man in the middle for both communication between services and all communication from the active learner. The envoy proxy captures and logs the following information for each request:

- Request URL
- Request path
- Request method
- HTTP status code
- Request timestamp

This information is enough to differentiate between requests and identify which log belongs to which symbol in the alphabet. In addition, the timestamp allows us to retroactively identify which log line belongs to which membership or equivalence query asked by the learner or equivalence oracle. In addition to logging all incoming and outgoing communications of the SUL, we also maintain which traces have been sent from the side of the learner. For each trace, the learner notes the following:

- The trace queried to the SUL
- The start time of querying the SUL
- The end time of querying the SUL

Combining the data provided by the learner and the log data provided by the envoy proxy provides enough information to process the logs into valid traces necessary for passive learning.

6.3.3. Log Processing

In order to apply FlexFringe on our logs, we first need to convert said logs to Abbadingo as described in Section 2.4.1. An important note is that our machines do not have an accepting or a rejecting state, therefore all labels are set to one. To transform the logs to valid traces for FlexFringe, we group the logs by time windows which are documented by the active learning process.

It is however not as simple as simply taking all of the events that happen within a window and converting them to logs due to the following reasons:

- Events can be missing due to our active learning techniques.
- Internal calls can be irrelevant and considered noise. This is because events from previous windows may overlap with later windows because asynchronous internal requests can continue executing after the learner has already moved to the next query.
- Due to abstraction or crashes, non-determinism can still occur.

The first issue are missing events, which are seen in the active learner but not in the logs. The SUL Orchestrator infers HTTP result 400 for request in which it cannot infer the defined data in the alphabet. This is done as an efficiency method to minimize the amount of interaction with the actual SUL. This means however that the active learner has more data than is represented in the logs, as the requests are never sent. In an effort to give the passive learner as much information as the active learner has, we inject these events with the HTTP status code 400 retroactively in the traces. This ensures that the state merger has enough information to more accurately differentiate between states and avoids false mergers.

The next issue are noisy internal events. Since we are working with time windows, it can be the case that internal events triggered by previous traces can bleed into the logs of following traces. If these events are not filtered correctly, it can lead to unexpected outcomes.

For example, consider the following two traces:

External Event 1/200 → Internal Event/200 → External Event 2/200

External Event 1/200 → Internal Event Previous Trace/200 → External Event 2/400

External Event 2 has two different outputs. This is because, in the second trace, the internal event does not actually occur. The Internal event is bled over from a previous interaction with the System Under Learn. If both traces are kept as is, it will seem as if the system is behaving non-deterministically.

To avoid this scenario, we remove internal events in two cases. In the first case, if the defined trace has leading internal events, we assume internal events will only be triggered by interaction with the System Under Learn. When there is no interaction, internal events should not occur that affect the queries asked by the learner. The second case is when there are two traces that only differ by one internal call. In this scenario we deem the internal call to be noise if all the outcomes of the two traces are the same. If the outcomes differ the internal call is maintained.

Finally, the passive learner used in these experiments is not capable of handling any non-deterministic traces, therefore any duplicate trace must match its outputs. We enforce this through a majority voting mechanism. Meaning that any time duplicate traces have a mismatch, we remove the traces that are in the minority. In the case of a tie, we choose to keep the successful response. In the case the non-determinism is caused by downtime in the server, meaning we observe a HTTP 504 status code (Gateway timeout), we assume some outside influence has caused a lapse in deterministic responses. To counteract this this downtime, we inject a CRASH symbol in the learning alphabet.

6.3.4. Passive Learner

The passive learner we use is FlexFringe as implemented at the TU Delft. We used the standard running settings for mealy data, meaning the "heuristic-name" is set to "mealy" and the "data-name" variable is set to "mealy_data". No additional settings or customization are applied in this experiment.

For the experimental setup, we use the same System Under Learn as the one used in the experiment of active learning non-deterministic microservices. The active learning algorithm we use to produce the necessary logs for the passive learner is our two phase implementation of Kearns-Vazirani.

6.3.5. Comparison of Machines

We expect the language defined by the machine learned by our active learning process to be a subset of the language accepted by the machine learned through passive learning. We assume this, because the passive learner is fed the behaviors triggered through our active campaign. To confirm this, we want to compare the machine to the machines learned by the active learner in both the deterministic and non-deterministic test setup. However, when applying the passive learner on the logs produced by the system, the state machine learned will have a larger alphabet relative to the state machine learned by the active learning algorithms. This is due to the fact that the internal calls are not included in the active learning process. Therefore, in order to compare the behavior of the machines we use two methods. The first method we refer to as a strict graph traversal. In the strict traversal we only cover transitions of the shared alphabet, this effectively means we ignore the requests that lead to internal transitions. The second method, we refer to as a transient graph traversal. In transient graph traversal, we traverse over internal transitions as if they were not there, meaning that during the comparison multiple paths and outputs are possible. We consider two machines to be behaviorally similar if there is at least one path in which the two machines have the same outputs.

- Strict Traversal: Traverse both machines with the shared alphabet.
- Non Strict Traversal: Free traversal over non overlapping alphabet, if one of the possible outputs matches, then we deem the behavior to be the same.

6.4. Results

In this section, we describe the results of our experiment testing the feasibility of utilizing active learning to create data for the application of passive learning.

6.4.1. Log Processing

The active learning process resulted in 27868 traces. When processing the logs we have to remove 120 of them, due to those traces mismatching the the majority vote of the duplicates. In this experiment we do not have to inject any CRASH events, as the System Under Learn has remained stable.

The resulting alphabet of the processed traces contains 7 symbols, which is exactly one larger than the alphabet used for the active learning process. This is expected, as there is only one API call that the two services use to communicate with one another.

6.4.2. Learning

In our experiment, FlexFringe is able to learn the state machine including the internal alphabet, this state machine is displayed in Figure 6.2. The resulting machine has 32 states and 193 transitions. The transitions that are triggered by the internal API calls are represented by the dotted line.

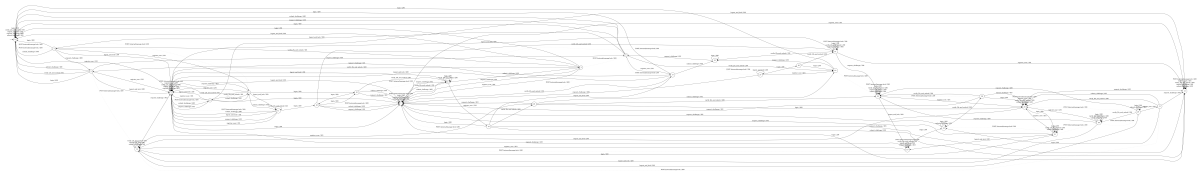


Figure 6.2: State Machine learned by FlexFringe, dotted lines represent transitions triggered by internal requests.

6.4.3. State Drift

When analyzing the machine displayed in Figure 6.2, we find that 22 states which are influenced by state drift. Meaning that an internal API call can trigger a transition to a new state that is unobserved by the active learner.

Figure 6.3 shows the potential state drifts of state sixteen from the machine displayed in Figure 6.2.

We can make these trees for all states. What can be derived from the tree is the possible difference in outcomes when the internal drift occurs vs when it does not. Specifically, this transition tree shows that when the internal/manage-lock request does not occur, then the request_challenge call will return a 403 HTTP response.

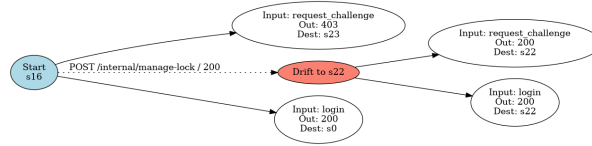


Figure 6.3: Representation of state drift present in the test system

The state drift shown in Figure 6.3, highlights the fact that internal communication is necessary to properly change the state of the system as a whole. The behavior displayed is expected, as the Login Service must communicate to the Challenge service that the challenge endpoints are open to the user. Therefore if the internal call is not completed, we expect the request_challenge call to return a 403, as seen in the figure. When the internal call does complete, the system moves to a new state, and the request_challenge call returns a 200.

The presence of this state drift is evidence that the state machine of the System Under Learn is larger than the machine learned through the active learning campaigns, as internal communication between microservice can affect the outcome of the interactions.

6.5. Active vs Passive machines

To compare the state machine learned through active learning, displayed in Figure 5.3, with the machine learned by FlexFringe, displayed in Figure 6.2, we use a strict and the non strict traversal. First we can observe several structural differences. The machine learned by the active learner only has 13 states and 78 transitions. The machine learned through passive learning is much larger. This is to be expected due to the larger alphabet in passive learning.

The behavioral differences are also important to note. Through our strict traversal, only traversing transitions of the shared alphabet of the machines, we find that there are 103 behavioral differences. Then when we allow for the free traversal of transitions which are not shared between the machines, then we find that the behavior does indeed match exactly.

To be clear, we are unable to create a direct state mapping between the two machines, this is because we treat the machine learned by the passive learner as a non-deterministic machine. In non-deterministic machines, multiple states can represent a single state in a deterministic machine. In this case meaning we cannot make a distinct one to one mapping.

What we can conclude from this experiment is that it is indeed possible to capture and model the non-determinism ignored in the active learner, when that non-determinism is caused by the presence or lack of internal communications between the machines. We also show that the behavior in the actively learned machine is included in the machine found through our passive learning methods.

6.6. Conclusion

In our case, passive learning is advantageous due to the fact that it can observe more communication/logs when compared to the active learning setup. Active learning is limited to input output, but passive learning has full access. This full access is achieved through routing all communication between microservices through a proxy. The proxy logs the time stamp of the request, the type of request, the destination service, and the path. While running the active learning campaign we keep a log file of all the traces ran, where we keep track of the start and stop times such that we can convert the logs collected by the proxy to traces.

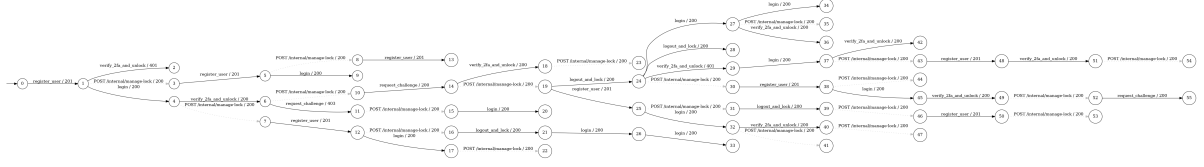


Figure 6.4: Internal data structure created by FlexFringe when learning from the logs

Importantly, because FlexFringe works through merging two states when there is no evidence that differentiates the two, it can make mistakes. As the logs are created by the active learner, there is enough evidence to differentiate states on the limited alphabet. Since the passive learning process has a larger alphabet, there might not be enough evidence that include these new transitions. Which could cause the premature merger of states. That is why we believe it is important to also analyze the internal FlexFringe data structure displayed in Figure 6.4, as it displays the observations made without performing any mergers.

In this chapter, we introduce the use of passive learning in combination with active learning techniques to capture a more complete representation of the System Under Learn. We introduce the concept of state drift, which is the shifting of states caused by internal communications between services. Although the state merging through passive learning will not lead to perfect hypotheses due to premature state merging, the presence of state drift does show that the non-determinism experienced by the active learner can be explained. The state machine representing the System Under Learn which includes the internal communications between microservices is more intricate than can be represented through outside interaction with the system.

7

Learning OAuth

In this chapter we apply the learning techniques developed in the previous chapters on a real world example. In Section 7.1, we present the learning setup for the System Under Learn and the algorithms chosen. In Section 7.2, we display the results of the active learning campaigns, and in Section 7.3 we display the results of applying our passive learning techniques on the system. In Section 7.4 we discuss the bugs found by the learning process. Finally in Section 7.5, we discuss the findings of the application of our methods on a real life system.

7.1. Application

As discussed in Chapter 2, the chosen System Under Learn is Ubiqu's implementation of the OAuth protocol. Not only does this include the protocol itself, but also all the services necessary for extracting a valid login token. OAuth is chosen due to its high complexity, its stateful nature, and the fact that it is implemented as a series of interconnected microservices.

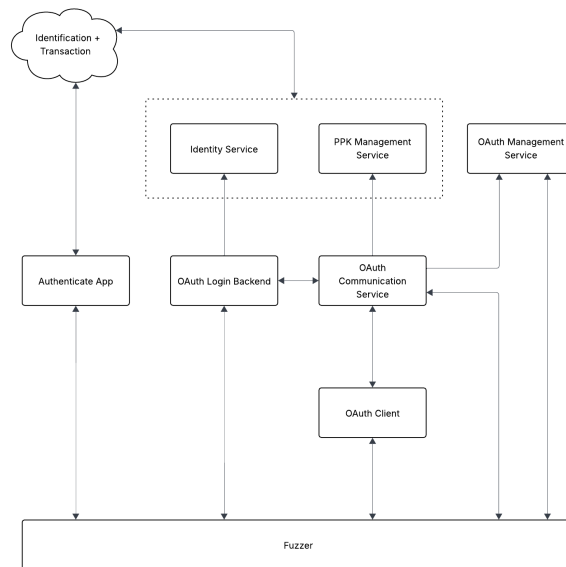


Figure 7.1: Learning Architecture, each box represents a Microservice and the arrows represent the lines of communication.

The experimental setup is displayed in Figure 7.1. Due to the reliance on Hardware Secure Modules, we cannot run all the services locally. Therefore, we are reliant on Ubiqu's testing cluster for part of the interactions. The services that are deployed remotely are the following, the Authenticate App, the

Identification and Transaction Service, the Identity Service and the PPK Management Service. The rest of the necessary services are run locally, including the learner itself. This reliance on the testing environment creates some instability in the learning setup, however our learning pipeline is designed to handle this flakiness.

The alphabet used for our experimentation is defined in Appendix A. This alphabet is selected as it is the minimum set of requests necessary to complete a successful OAuth flow. Extra requests were left out to speed up the learning process, as the goal of this experiment is to prove it is possible to learn a system with a microservice architecture.

7.2. Active Learning

In this section, we discuss the results of the learning campaigns driven by two phase KV and two phase L*. From our experiments in Chapter 5, we conclude that these algorithms are the two most viable options. We chose KV due to its high querying efficiency, which is necessary due to the high time cost for each HTTP request to the System Under Learn. We also chose to apply L* on OAuth due to its high consistency and predictable nature, which is necessary when the system is inconsistent and learning restarts are required. In addition, we have the results of a cumulation of learning efforts driven by L* with retry logic, human intervention, and a cache.

7.2.1. Two Phase KV

The application of KV on the OAuth System Under Learn did not result in a learning campaign that terminated naturally. The learning setup was unable to learn a complete Mealy machine within a reasonable amount of time. We cut off the experiment after 92 hours due to the frequency of learning restarts and the lack of speed up created by the growing cache. In its best learning attempt, KV was able to learn 22 states before restarting the learning process. In total, KV had 3 learning attempts, which consisted of 42 iterations, 17666 queries, of which 5478 were unique.

We believe that the reliance on equivalence querying, in combination with the use of Wp random as the equivalence oracle, introduces too much randomness in the learning process. Since the learner relies on counterexamples, each learning attempt, KV produces different traces due to the lack of consistency in which counterexamples are presented by the equivalence oracle. Since the learner will take unpredictable paths each learning attempt, the efficiency gain of the cache is lost. In addition, because we rely on the cache to enforce stability when querying the System Under Learn, when the learning algorithm deviates from previously seen paths, it becomes harder to verify the stability of the query. When unstable queries are uncaught and non-determinism has been injected into the learning process, the learning process will have to eventually restart.

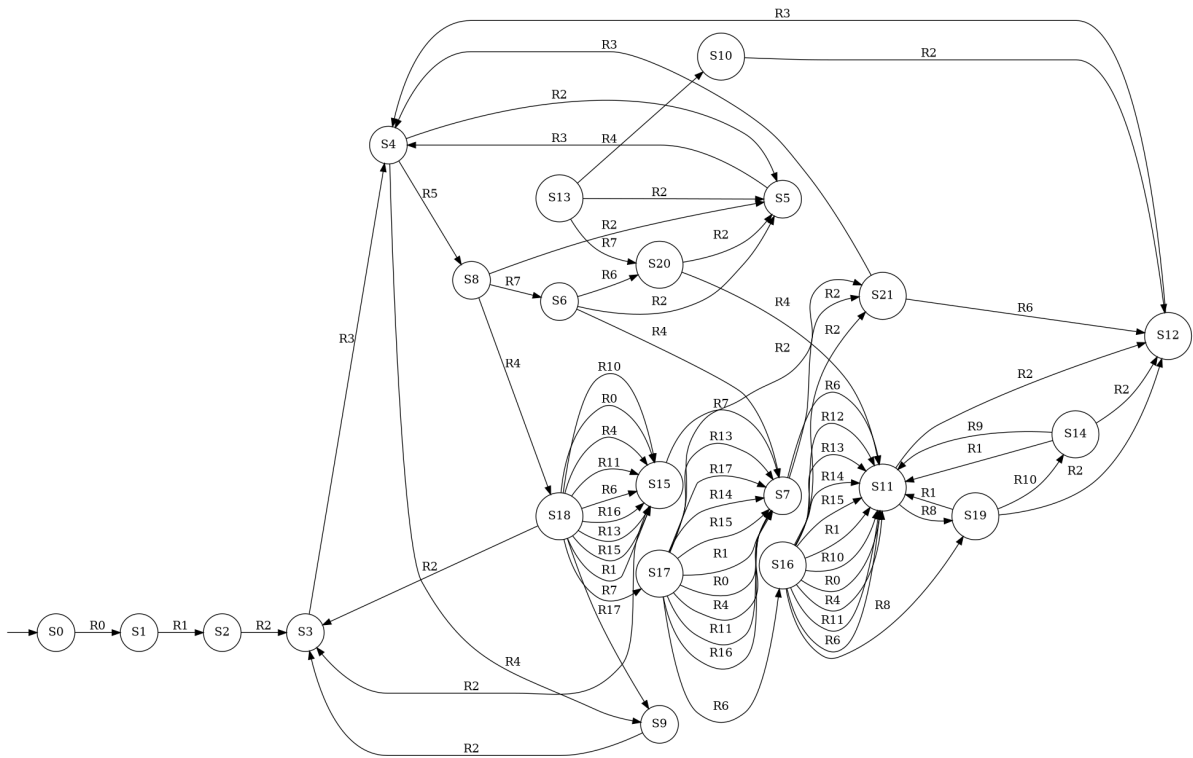


Figure 7.2: Largest hypothesis found by KV on OAuth during its learning campaign

The largest machine found by KV can be found in Figure 7.2 and the legend can be found in Table 7.1. For legibility, we have removed all of the self loops in the hypothesis. Although KV was unable to find a complete machine, as we cut its run time short, it is important to note that KV was able to successfully retrieve a token from the OAuth system.

Table 7.1: Transition legend

ID	Request Name	Status
R0	register_client	201
R1	authorize_request	200
R2	register_user	201
R3	register_auth_app	200
R4	register_auth_app	500
R5	unlock_auth_app	200
R6	get_user_info	200
R7	get_registration_status	200
R8	user_login	200
R9	oauth_callback	200
R10	init_client	200
R11	unlock_auth_app	400
R12	logout_user	200
R13	get_authentication_status	200
R14	get_registration_status	404
R15	oauth_callback	500
R16	logout_user	400
R17	user_login	400

7.2.2. Active Learning with Two Phase L*

The initial learning attempts using Two Phase L* were unsuccessful due to setting the base values to one. This caused many restarts in the learning process due to an unstable system under learn. To counteract this, we set the minimum samples per query to three. In addition, we found that when we require the counterexamples found in phase 1 to be 100% stable, the returned counterexamples are unnecessarily complex purely to find traces which have not yet been seen, this is because nearly no events in the cache are fully stable. Therefore, we lowered the minimum required stability for a counterexample to be 80%. This change resulted in the hypothesis as seen in Figure 7.3. An important note is that for legibility, the self loops are removed, and the definitions for the transitions can be found in Table 7.2.

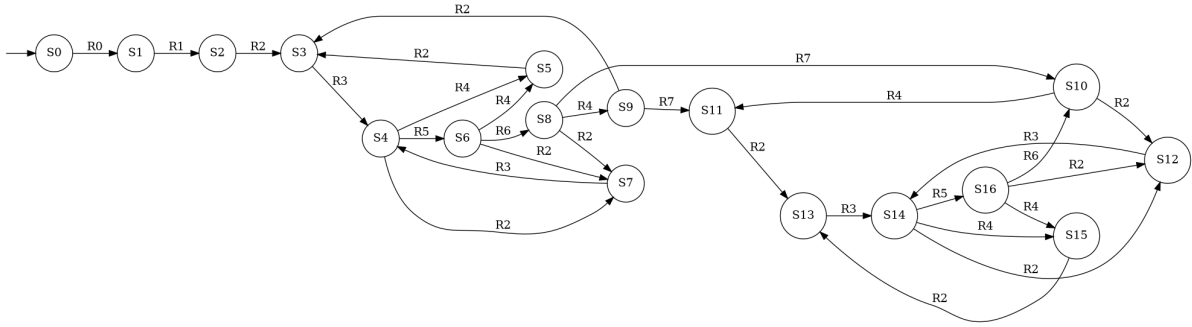


Figure 7.3: Hypothesis achieved through two phase L*

The learning campaign was cut short, and the hypothesis that was found was the first hypothesis found by the learner. The learner found this hypothesis in its first attempt. The hypothesis was found in 1700 membership queries. The campaign was stopped after 103 hours and the learner was on its 3rd learning attempt. The reason it was cut short was due to the high querying cost and high instability of the learning setup. In total, the learning campaign queried the system under learn 20813 times, of which 5321 were unique, 13659 were sampling queries. The unique queries consist of membership queries, counterexample querying, and counterexample processing. The average querying time was 17.9 seconds.

Table 7.2: Legend for Figure 7.3

ID	Request Name	Status
R0	register_client	201
R1	authorize_request	200
R2	register_user	201
R3	register_auth_app	200
R4	register_auth_app	500
R5	unlock_auth_app	200
R6	get_registration_status	200
R7	get_user_info	200

The final found hypothesis does not show any of the deeper behavior expected in the OAuth state machine, this is due to the learner not fully processing any of the counterexamples. We believe that if we had given the learner more time, it would eventually grow its hypothesis.

7.2.3. Baseline Learning Campaign

As we do not have an actual state machine for the system under learn, we attempt to create one such that we can compare our methods. To create a baseline, we ran L* with a simple retry method for multiple learning campaigns. These learning campaigns produced a cache of responses which we manually ensure does not contain any conflicts.

The experimental setup is the same as the experiment described in Section 7.1 and pictured in Figure 7.1. Meaning there is still some reliance on services running on Ubiqu's staging environment. Not having complete control over the system under learn introduces some instability to the experimental setup. For example, when the Identity Service becomes unhealthy, the OAuth Login Backend will reject all requests. In this case, that instability can lead to non-determinism, which the L^* algorithm cannot handle. To avoid this, non-determinism is checked by caching traces. If a cached trace matches the prefix of the current trace queried to the system under learn, the outputs are compared. When these outputs do not match, the query is rerun with delay until it does. This allows for the systems on which the setup relies to become healthy again. If non-determinism remains, the learning process is restarted and we manually update the cache to match the expected behavior.

After populating the cache through active learning campaigns using L^* combined with the W method for finding counterexamples, we achieved the hypothesis shown in Figure 7.4. The labels for the transitions can be found in table 7.4. An important note about the state machine displayed, this is the machine after all of the self loops have been filtered out. This means that each state in the final hypothesis has self loops consisting of the entire alphabet minus the outgoing edges. Self loops have been filtered out to increase the legibility of the figure.

Another important note is the amount of queries required by our learning campaign. To learn the system, L^* needed 3757 membership queries.

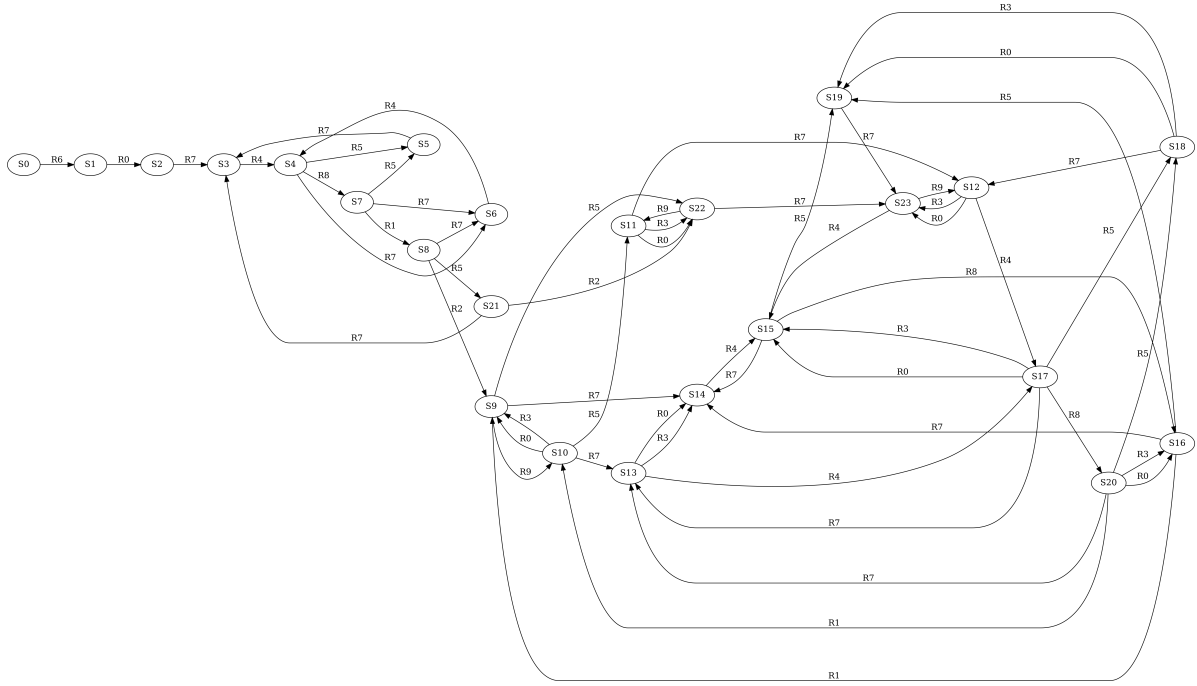


Figure 7.4: Hypotheses achieved through active learning campaign

The final hypothesis achieved consists of 24 states and has successfully completed the OAuth flow, depicted by the presence of the `oauth_callback 200`. The final hypothesis was also the first one found when the observation table became closed and consistent, as the W method with w set to one was not able to find a counterexample in the time allowed for the experiment. This means that the final hypothesis might still have some inconsistencies with the System under Learn, however we were not able to find a counterexample given the current experiment limitations.

Upon further inspection of the final hypothesis, a bug can be spotted. Transition R3 `oauth_callback` with status 200 should only be accessible once per authentication session. Therefore, transitions such as the one between S11 and S22 are invalid. This is a trigger to check the code base to verify if the state machine is correct, and indeed the status of the authentication session is never updated when the `oauth_callback` is successfully made, matching the hypothesis generated through our fuzzing campaign.

Table 7.3: Legend for Figure 7.4

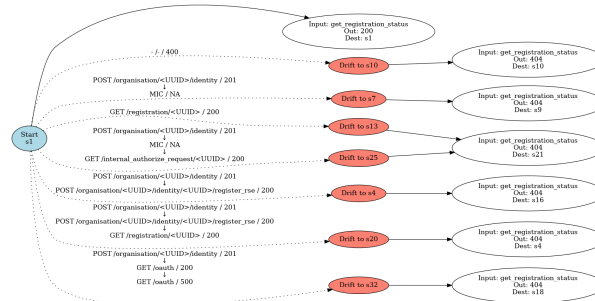
ID	Request Name	Status
R0	authorize_request	200
R1	get_registration_status	200
R2	get_user_info	200
R3	oauth_callback	200
R4	register_auth_app	200
R5	register_auth_app	500
R6	register_client	201
R7	register_user	201
R8	unlock_auth_app	200
R9	user_login	200

This bug has been reported and fixed.

7.3. Passive Learning

To further investigate our System Under Learn, we applied the learning techniques described in Chapter 6. We choose to use the logs produced by the Two Phase KV method, as it resulted in the largest hypothesis in a single learning campaign. After rendering valid traces and running FlexFringe, the resulting state machine is too large to properly display, as it contains 57 states and 160 outgoing edges. From our hypothesis we can construct the conflict trees which represent the state drift caused by internally triggered events.

We want to focus on the unstable `get_registration_status` found in our learning campaigns, we find various conflict trees. As discussed in Chapter 6, the passively learned machine is not perfect due to the possibility of premature mergers, the presence of state drift does point to internal interactions that can cause unexpected results. One of these cases is represented in Figure 7.5.

**Figure 7.5:** Conflict Tree representing state drift in the state machine learned by FlexFringe

From this conflict tree, we can derive that the active call `get_registration_status` can result in either a 200 or 404 based on the internal interactions that precede it. Which points to the possibility of internal race conditions affecting the outcome of the active learning process.

7.4. Findings

Through our learning campaigns we found one bug and one race condition. The first bug we find through analyzing the state machine learned by L* displayed in Figure 7.4. When following the transitions of the machine, we find that transition R3, `oauth_callback` with status 200 can be accessed multiple times with one authentication session. This can be observed for example in the transitions between S11 and S22. These transitions should not be possible, as the authentication session should be deemed finalized after the completion of one successful OAuth callback. When analyzing the code, when the OAuth callback completes, the state update of the authentication session is missing. This bug has

been reported and fixed on all environments.

7.4.1. Race Condition

The second bug we found is not present in the state machine, as it is a spurious success. We find that when querying `get_registration_status` twice in a trace after unlocking the authenticate app, we spuriously get two 200 responses. When analyzing the codebase, this should not be possible as the first call should alter that state of the registration status such that when the second call comes in, the registration can no longer be found. Therefore in the sequence `get_registration_status` $\rightarrow$ `get_registration_status` should result in 200 $\rightarrow$ 404. The cause for the spurious 200 $\rightarrow$ 200 responses is a race condition caused by a delayed callback which means the registration status is not properly updated before being queried.

To further analyze this race condition and the instability it causes in the learning process, we ran the following trace 100 times:

```
register_client -> init_client -> authorize_request -> register_auth_app -> unlock_auth_app ->
get_registration_status -> get_registration_status
```

The http status code for the final `get_registrations_status` call was 404 57 out of 100 runs and 200 for 43 out of those 100 runs. This causes major instability in our learning campaigns. When analyzing the code base, it becomes clear that the call is important for creating a state change in the system. This state change can only occur when the internal race condition has been completed. Ubiqu's frontend avoids this issue through polling the endpoint until the expected result has occurred. This ensures the necessary state transitions have been made.

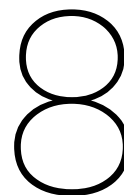
7.5. Discussion

The current learning setup, although successful in learning a machine, is not optimal for application in real development environments. We would recommend running all services, including the learner on the same cluster. This would minimize the instability in the setup, and therefore speed up the learning process. In addition, when we cannot explain non-determinism through our passive learning method, it means there is a condition that is not caught through API calls. Therefore, we recommend also including database transactions in the learning alphabet. We would also recommend combining the logs of all services, not only the services which are deployed locally, this can ensure the all interactions are properly displayed.

When comparing the methods used, each has their successes and drawbacks. Our Two Phase L* method was able to find a hypothesis, however it was unable to refine on the counterexamples found within a reasonable time. Similarly, our Two Phase KV method was able to refine its hypothesis multiple times, however it suffered from costly learning restarts, which meant the learning campaign could not naturally conclude in the given time frame. Finally, our baseline method, which learned its initial hypothesis from a cache developed over various attempts and simply retried when it encountered unexpected behavior, was able to find the largest hypothesis and even a bug. However, it too was unable to refine its hypothesis based on the counterexample in a reasonable time.

The troubles in our learning campaigns are because OAuth as a System Under Learn is far from ideal. The application itself has a high reliance on many microservices which are difficult to reconfigure while maintaining its functionality. The difficulty in learning OAuth is exacerbated by the high query cost. Some queries to the System Under Learn take multiple seconds, due to the internal computation necessary for creating assets. This querying expense slows down learning dramatically, which means we need to keep our local setup running for multiple days. This is a challenge in itself.

In this chapter, we show that despite the challenges given by the OAuth implementation, we are able to apply the techniques developed in this thesis to a real world codebase. Even though we had to preemptively stop the learning campaigns, we do show that the application is indeed possible given more time or a more stable System Under Learn.



Discussion

In this chapter we discuss the contributions and interesting findings of the research performed in this thesis. In addition, we cover any limitations and future work we would recommend to further develop and utilize the findings we have made.

8.1. RQ: 1

In Chapter 4, we show how we can learn a microservice architecture as one application. Through learning the system as a whole instead of individual services closer reflects the real life workings of the system. Learning the system as a whole can reveal new and unique insights in the system that classic learning does not. The system is greater than the sum of its parts.

Our solution allows for the easy application on new systems, the only adjustments a user would have to make is defining the alphabet for the System Under Learn and define any necessary mapping functions. In addition, through our integration with LearnLib, it becomes easy to switch learning algorithms to best match ones learning setup.

Although our definition of the alphabet and generation of valid HTTP requests through a query orchestrator allows for the learning of microservice architectures, it is not yet ready to deal with all challenges introduced with this goal. In order to apply this learning method on real world examples, the learner must be able to effectively deal with unexpected responses from the System Under Learn.

8.2. RQ: 2

The most monumental challenge with learning microservices is the introduction of non-determinism in the learning process. In order to counteract this, we introduce a dynamic sampling method, which determines how often to query specific traces based on the predicted stability of the suffix. We find that this method decreases the amount of sampling queries necessary, however the possibility of needing to restart the learning process is not eliminated.

We find that splitting the equivalence querying process in two phases can speed up the the learning process through first only returning stable counterexamples. This triggers learning algorithms to first explore more stable requests. Once the stable counterexamples are exhausted, we move onto returning unstable counterexamples.

An important observation is that not all learning algorithms are made equal in our learning setup. We find that the learning algorithms which more heavily rely on the equivalence oracle for hypothesis refinement tend to need more learning restarts. As we use the Wp random method for equivalence queries, this conclusion is not unexpected. Instead of structurally building a cache on which the learning pipeline can rely to verify the outputs of the trace, the equivalence oracle introduces many random traces into the learning process. Not only does this increase the possibility of an error being introduced to the learner, the reliance of counterexamples to further push the learning process also means the learner does not act in a predictable manner after each restart. We would recommend experimenting with

counter example oracles which search for a counterexample in a consistent and repeatable manner, such that throughout each learning attempt, the cache can be better utilized.

As the learners that rely heavily on equivalence queries restart more often, and each learning iteration is different due to the variation of counterexamples, the cache becomes less efficient. Algorithms like L^* which act in a predictable manner and slowly build their cache are able to recover much faster from learning restarts, as it is less reliant on the equivalence oracle for building its hypotheses.

Stability of the System Under Learn is a major factor for the learning efficiency in a non deterministic setting, however we find that some learning algorithms are more adept than others in this learning environment. We do show that learning these systems is possible.

8.3. RQ: 3

We show passive learning and active learning can be used symbiotically. Through using active learning techniques to produce logs for passive learning, we can get a more accurate insight of the System Under Learn. Logs produced through interaction with the system have a tendency to bias the normal happy flow. Therefore, when applying passive learning on those logs, algorithms like Flexfringe can be more likely to perform premature state mergers. In addition, as the happy flow is biased, one is far less likely to learn illegal state transitions in the final learned machine. We also show that active learning in isolation is not enough to gain a full understanding of the System Under Learn. When solely learning through API calls, microservice can perform internal communication which is not captured by active learning algorithms.

We introduce two manners of comparing the machines learned by the active and passive learners. A strict traversal only comparing transitions of the overlapping alphabet as well as a transient traversal. The transient traversal allows for the free traversal of transitions which are not in the shared alphabet. Through this comparison we derive the term state drift. State drift refers to the drift a state machine can make from one state to the other only following transitions defined by the internal calls of a system. We found that in our experimental setup, that these state drifts can account for non-determinism experienced in the active learning process caused by internal API requests.

8.4. Future Work and Limitations

Through our research, we have found some limitations of our current setup, as well as defined some areas for future research.

8.4.1. Application

The application of our learning pipeline, although possible, is highly impacted by the stability of the System Under Learn and the cost of querying traces. In order to actually utilize the learning techniques defined in this thesis, we recommend deploying the learner on the same cluster as the System Under Learn in order to minimize any possible instability caused by network flakiness.

Some future improvements would also include writing the cache to memory. Currently there is a hardware limitation as the cache is not stored in a database, therefore if the query count explodes, performance of the learner decreases. Storing the cache in memory will also allow for easier recovery after unexpected failures or experimenting with differing learning algorithms. Crucially however, when writing the cache to memory, one will have to make sure to remove the cache manually whenever the System Under Learn is updated. Another future improvement we recommend is storing the returned counterexamples, this will allow the learner to more quickly recover on restarts when the returned counterexample matches previous learning attempts.

In addition, although possible, applying these learning techniques on the full spectrum of Ubiqu services would be very difficult due to the hardware bottleneck in the creation of identities in the Hardware Security Module. This slows down learning immensely due to the high query costs. As this is difficult to speed up, we recommend either setting up a testing environment which works without the HSM or learning different internal systems. For applications that do not have such a high computational bottleneck, we recommend the learning techniques in this paper.

8.4.2. Learning Inefficiency

As mentioned, a major cause for the inefficiency in learning is any instability in the System Under Learn. We find that stable predictable learning algorithms such as L^* in combination with a cache and adaptive sampling are able to learn unstable systems, but with relatively bad efficiency.

We recommend future research look at developing a bespoke algorithm which can easily alter its internal data structure without invalidating all results, avoiding the need for learning restarts. Which can hopefully be more space and time efficient than L^* if the data structure is similar to $L^\#$ discriminant tree.

In the presence of instability and high query costs, we also recommend future research explore snapshot learning for microservices. We determined this was out of scope in our research due to the complexity this can introduce and the reliance of remote hardware in our System Under Learn. In scenarios where the learner can have full control over the microservice deployments however, this method could be feasible. Through snapshot learning, it would be possible to decline the cost of querying and ensuring stability in the learning process.

8.4.3. Passive Learning

We believe the use of passive learning to learn from both the external and internal communications with microservices creates more insightful Mealy machines. In our current learning pipeline however, the active learner can only guarantee enough evidence to distinguish states based on the alphabet exposed for external communications. It would be beneficial to allow Flexfringe to drive the learning process in order to generate enough data to also differentiate using the internal transactions. For this to be possible, Flexfringe would need to generate traces with the external alphabet which could trigger internal transitions.

Our current learning pipeline is limited to only collecting the logs from API transactions. This does not capture all seemingly non-deterministic occurrences in the System Under Learn. We also recommend capturing and centralizing all database transactions as part of the alphabet for the passive learner. This can capture a more detailed depiction of the System Under Learn and also further explain any instability in the system.

8.4.4. State Machine Validation

Currently, in order to validate the state machine, it requires domain knowledge and human oversight. In order to more efficiently utilize these learning technologies, we recommend the following:

We can integrate learning into a CI/CD pipeline, which will ensure that new code will not introduce or remove any new states or state transitions. This can be vital for maintaining a secure and validated codebase in production environments. This strategy does, however, rely on manually verifying the original state machine.

Our second recommendation is researching the automatic generation of Linear Temporal Logic based on design or protocol specifications. Being able to effectively produce Linear Temporal Logic would allow another step in the verification pipeline to be automated without the direct need for domain experts. This in turn allows for the more detailed auditing and validation of vital services.

9

Conclusion

The primary goal of this thesis is to develop and apply active learning techniques to software systems developed using a microservice architecture. Microservices work together to create functionality for an end user. We find that current research focuses on learning each system in isolation. We show that the system is greater than the sum of its parts. Therefore, in certain cases, learning the services in isolation will not reveal the entire workings of the system. This mirrors the distinction between unit and integration testing. While individual services may behave correctly in isolation, their emergent behavior when deployed together can only be captured through treating the architecture as a unified system.

In order to learn microservices as one system, we develop a learning pipeline in which an orchestrator converts the defined alphabet into valid HTTP requests. These requests can include references to previous responses, requests, and even custom function mapping. The interpreted requests are then sent to the appropriate microservice as defined by the alphabet.

We find that learning microservices is not that simple, due to the communication over the internet, and the dependencies on internal communication, there is flakiness present in the learning process. Traditional active learning algorithms are not developed to be able to handle these inconsistencies in the System Under Learn and will terminate prematurely. In order to counter act this, we set up a learning process in which we verify the queries with a cache, as well as introduce dynamic sampling based on dangerous suffixes. This minimizes the necessary interactions with the System Under Learn to extract the expected behavior.

Through learning unstable systems, we find that not algorithms are created equal. Learning algorithms which use equivalence querying as a driving factor usually introduce more instability into the learning process and therefore require more restarts. This behavior is expected because unlike traditional learning, we do not have an oracle for equivalence querying. Therefore we utilize Wp random as the equivalence querying method, which means we have less control over the returned counterexamples. Which in turn introduces variability in each learning attempt.

Finally, we suggest that in order to learn a a more representative Mealy machine one must apply active and passive learning techniques harmoniously. Due to internal communication between microservices, the active learner is not able to capture all transitions. We also introduce the concept of state drift. State drift is defined by the internal unobserved communication between microservice causes the systems state to change independently of the active learner's queries, which leads to observed non-determinism. Which means that instability is caused by the lack of or presence of internal communication within the services. Meaning, that transitions hidden from the active learner can affect the state machine as a whole.

After developing these theories, we apply these methods on an actual codebase, OAuth. Although learning is possible, it is tedious. Due to the instability of the learning setup and the high cost of querying the System Under Learn, we recommend some future work. We also show the importance of the application of stateful learning techniques, as through our experimentation we were able to highlight

a bug aswell as the presence of race conditions. The methods we developed can be directly applied, however we recommend deploying the learner alongside its System Under Learn which will minimize latency and instabilities.

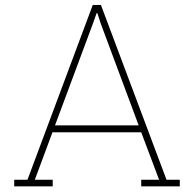
In summary, in this thesis we set out to learn software systems developed using microservice architectures. We developed a learning pipeline which can easily be applied to many different systems, with only the need of defining an alphabet. When learning these architectures, several challenges arise, the largest of them is instability. Instability can be caused by many different factors and active learning methods are not developed to handle this. We developed a method through adaptive sampling and query validation to counteract the affects of this instability. Finally, in order to explain the instability, as well as gain a more complete understanding of the System Under Learn, we apply passive learning and introduce the concept of state drift. To conclude our research, we successfully apply the developed methods on a real world System Under Oath.

References

- [1] Bernhard K. Aichernig, Martin Tappler, and Felix Wallner. “Benchmarking Combinations of Learning and Testing Algorithms for Automata Learning”. In: *Form. Asp. Comput.* 36.1 (Mar. 2024). issn: 0934-5043. doi: 10.1145/3605360. url: <https://doi-org.tudelft.idm.oclc.org/10.1145/3605360>.
- [2] Jie An et al. “Learning Nondeterministic Real-Time Automata”. In: *ACM Trans. Embed. Comput. Syst.* 20.5s (Sept. 2021). issn: 1539-9087. doi: 10.1145/3477030. url: <https://doi-org.tudelft.idm.oclc.org/10.1145/3477030>.
- [3] Dana Angluin. “Learning regular sets from queries and counterexamples”. In: *Information and Computation* 75.2 (1987), pp. 87–106. issn: 0890-5401. doi: [https://doi.org/10.1016/0890-5401\(87\)90052-6](https://doi.org/10.1016/0890-5401(87)90052-6). url: <https://www.sciencedirect.com/science/article/pii/0890540187900526>.
- [4] Dana Angluin and Philip Laird. “Learning From Noisy Examples”. In: *Mach. Learn.* 2.4 (Apr. 1988), pp. 343–370. issn: 0885-6125. doi: 10.1023/A:1022873112823. url: <https://doi-org.tudelft.idm.oclc.org/10.1023/A:1022873112823>.
- [5] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. “RESTler: Stateful REST API Fuzzing”. In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 2019, pp. 748–758. doi: 10.1109/ICSE.2019.00083.
- [6] Clinton Cao, Annibale Panichella, and Sicco Verwer. *Automated Test-Case Generation for REST APIs Using Model Inference Search Heuristic*. 2025. arXiv: 2412.03420 [cs.SE]. url: <https://arxiv.org/abs/2412.03420>.
- [7] Clinton Cao et al. “Learning State Machines to Monitor and Detect Anomalies on a Kubernetes Cluster”. In: *Proceedings of the 17th International Conference on Availability, Reliability and Security*. ARES 2022. ACM, Aug. 2022, pp. 1–9. doi: 10.1145/3538969.3543810. url: <http://dx.doi.org/10.1145/3538969.3543810>.
- [8] T. S. Chow. “Testing Software Design Modeled by Finite-State Machines”. In: *IEEE Trans. Softw. Eng.* 4.3 (May 1978), pp. 178–187. issn: 0098-5589. doi: 10.1109/TSE.1978.231496. url: <https://doi-org.tudelft.idm.oclc.org/10.1109/TSE.1978.231496>.
- [9] Cristian Daniele, Seyed Behnam Andarzian, and Erik Poll. “Fuzzers for Stateful Systems: Survey and Research Directions”. In: *ACM Comput. Surv.* 56.9 (Apr. 2024). issn: 0360-0300. doi: 10.1145/3648468. url: <https://doi-org.tudelft.idm.oclc.org/10.1145/3648468>.
- [10] Peng Di, Bingchang Liu, and Yiyi Gao. “MicroFuzz: An Efficient Fuzzing Framework for Microservices”. In: *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice*. ICSE-SEIP ’24. Lisbon, Portugal: Association for Computing Machinery, 2024, pp. 216–227. isbn: 9798400705014. doi: 10.1145/3639477.3639723. url: <https://doi-org.tudelft.idm.oclc.org/10.1145/3639477.3639723>.
- [11] Markus Theo Frohme. *Active Automata Learning with Adaptive Distinguishing Sequences*. 2019. arXiv: 1902.01139 [cs.LG]. url: <https://arxiv.org/abs/1902.01139>.
- [12] S. Fujiwara et al. “Test selection based on finite state models”. In: *IEEE Transactions on Software Engineering* 17.6 (1991), pp. 591–603. doi: 10.1109/32.87284.
- [13] Roland Groz, Nicolas Bremond, and Adenilso Simao. “Using Adaptive Sequences for Learning Non-Resettable FSMs”. In: *Proceedings of The 14th International Conference on Grammatical Inference 2018*. Ed. by Olgerd Unold, Witold Dyrka, and Wojciech Wieczorek. Vol. 93. Proceedings of Machine Learning Research. PMLR, Feb. 2019, pp. 30–43. url: <https://proceedings.mlr.press/v93/groz19a.html>.

- [14] Jiaying Guo et al. "Stateful black-box fuzzing for encryption protocols and its application in IPsec". In: *Computer Networks* 251 (2024), p. 110605. issn: 1389-1286. doi: <https://doi.org/10.1016/j.comnet.2024.110605>. url: <https://www.sciencedirect.com/science/article/pii/S1389128624004377>.
- [15] D. Hardt. *RFC 6749: The OAuth 2.0 Authorization Framework*. USA, 2012.
- [16] M. Holcombe. *Algebraic Automata Theory*. Cambridge Studies in Advanced Mathematics. Cambridge University Press, 1982.
- [17] Malte Isberner, Falk Howar, and Bernhard Steffen. "The Open-Source LearnLib". In: *Computer Aided Verification*. Ed. by Daniel Kroening and Corina S. Păsăreanu. Cham: Springer International Publishing, 2015, pp. 487–495. isbn: 978-3-319-21690-4.
- [18] Malte Isberner, Falk Howar, and Bernhard Steffen. "The TTT Algorithm: A Redundancy-Free Approach to Active Automata Learning". In: vol. 8734. Sept. 2014. isbn: 978-3-319-11163-6. doi: 10.1007/978-3-319-11164-3.
- [19] Raveen Kanishka Jayalath et al. *Microservice Vulnerability Analysis: A Literature Review with Empirical Insights*. 2024. arXiv: 2408.03960 [cs.CR]. url: <https://arxiv.org/abs/2408.03960>.
- [20] Michael J. Kearns and Umesh V. Vazirani. *An introduction to computational learning theory*. Cambridge, MA, USA: MIT Press, 1994. isbn: 0262111934.
- [21] Kevin Lang, Barak Pearlmutter, and Rod Price. "Results of the Abbadingo One DFA Learning Competition and a New Evidence-Driven State Merging Algorithm." In: vol. 1433. Jan. 1998, pp. 1–12. isbn: 978-3-540-64776-8. doi: 10.1007/BFb0054059.
- [22] Maik Merten et al. "Automata Learning with On-the-Fly Direct Hypothesis Construction". In: *Leveraging Applications of Formal Methods, Verification, and Validation*. Ed. by Reiner Hähnle et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 248–260. isbn: 978-3-642-34781-8.
- [23] R.L. Rivest and R.E. Schapire. "Inference of Finite Automata Using Homing Sequences". In: *Information and Computation* 103.2 (1993), pp. 299–347. issn: 0890-5401. doi: <https://doi.org/10.1006/inco.1993.1021>. url: <https://www.sciencedirect.com/science/article/pii/S0890540183710217>.
- [24] Thomas Rooijackers et al. *WuppieFuzz: Coverage-Guided, Stateful REST API Fuzzing*. 2025. arXiv: 2512.15554 [cs.SE]. url: <https://arxiv.org/abs/2512.15554>.
- [25] Joeri de Ruiter and Erik Poll. "Protocol State Fuzzing of TLS Implementations". In: *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, Aug. 2015, pp. 193–206. isbn: 978-1-939133-11-3. url: <https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/de-ruiter>.
- [26] Michael Sipser. *Introduction to the Theory of Computation*. Third. Boston, MA: Course Technology, 2013. isbn: 113318779X.
- [27] Petar Tsankov, Mohammad Torabi Dashti, and David Basin. "SECFUZZ: Fuzz-testing security protocols". In: *2012 7th International Workshop on Automation of Software Test (AST)*. 2012, pp. 1–7. doi: 10.1109/IWAST.2012.6228985.
- [28] Frits Vaandrager. "Model learning". In: *Commun. ACM* 60.2 (Jan. 2017), pp. 86–95. issn: 0001-0782. doi: 10.1145/2967606. url: <https://doi-org.tudelft.idm.oclc.org/10.1145/2967606>.
- [29] Frits Vaandrager et al. *A New Approach for Active Automata Learning Based on Apartness*. 2022. arXiv: 2107.05419 [cs.FL]. url: <https://arxiv.org/abs/2107.05419>.
- [30] Jordan Valinsky. *The internet just had another global outage. why does this keep happening?* | CNN business. Oct. 2025. url: <https://edition.cnn.com/2025/10/20/tech/aws-why-internet-outages-keep-happening>.
- [31] Sicco Verwer and Christian Hammerschmidt. *FlexFringe: Modeling Software Behavior by Learning Probabilistic Automata*. 2024. arXiv: 2203.16331 [cs.LG]. url: <https://arxiv.org/abs/2203.16331>.
- [32] Sicco Verwer and Christian A. Hammerschmidt. "flexfringe: A Passive Automaton Learning Package". In: *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2017, pp. 638–642. doi: 10.1109/ICSME.2017.58.

- [33] Lina Ye et al. "Analyzing Robustness of Angluin's L^* Algorithm in Presence of Noise". In: *Logical Methods in Computer Science* Volume 20, Issue 1 (Mar. 2024). issn: 1860-5974. doi: 10.46298/lmcs-20(1:22)2024. url: [http://dx.doi.org/10.46298/lmcs-20\(1:22\)2024](http://dx.doi.org/10.46298/lmcs-20(1:22)2024).
- [34] Yingchao Yu et al. "SGPFuzzer: A State-Driven Smart Graybox Protocol Fuzzer for Network Protocol Implementations". In: *IEEE Access* 8 (2020), pp. 198668–198678. doi: 10.1109/ACCESS.2020.3025037.
- [35] Dongliang Zhao et al. "AGLFuzz: Automata-Guided Fuzzing for detecting logic errors in security protocol implementations". In: *Comput. Secur.* 149.C (Feb. 2025). issn: 0167-4048. doi: 10.1016/j.cose.2024.103979. url: <https://doi-org.tudelft.idm.oclc.org/10.1016/j.cose.2024.103979>.



Corpus

```
1 [
2   {
3     "template": "init_client",
4     "url": "http://oauth-proxy.local:9501/",
5     "path": "init-client/",
6     "method": "POST",
7     "data": {
8       "client_id": "{{_responses.register_client.json.client_id_}}",
9       "client_secret": "{{_responses.register_client.json.client_secret_}}",
10      "code_verifier": "{{_code_verifier_}}"
11    }
12  },
13  {
14    "template": "register_client",
15    "url": "http://oauth-management-service.local:9501/",
16    "path": "oauth_client/",
17    "method": "POST",
18    "headers": {
19      "X-issuer": "com.ubiqu.authenticate.staging",
20      "X-subject": "ubiqu-oauth-v2",
21      "X-token-type": "service_account"
22    },
23    "data": {
24      "name": "fuzzing_client",
25      "redirect_uris": [
26        "http://oauth-proxy.local:9501/oauth/"
27      ]
28    }
29  },
30  {
31    "template": "register_auth_app",
32    "url": "http://oauth-proxy.local:9501/authenticate-app/",
33    "path": "register/",
34    "method": "POST",
35    "data": {
36      "nonce": "{{_responses.register_user.json.nonce_}}"
37    }
38  },
39  {
40    "template": "unlock_auth_app",
41    "url": "http://oauth-proxy.local:9501/authenticate-app/",
42    "path": "unlock/",
43    "method": "POST",
44    "data": {
45      "rse_token": "{{_responses.register_auth_app.json.rse_token_}}",
46      "puk": "{{_responses.register_user.json.puk_}}",
47      "new_pin": "12345"
48    }
49  },
50 ]
```

```

50 {
51   "template": "get_user_info",
52   "url": "http://oauth-login-backend.local:9501/",
53   "path": "user/",
54   "method": "GET",
55   "data": {}
56 },
57 {
58   "template": "logout_user",
59   "url": "http://oauth-login-backend.local:9501/",
60   "path": "user/{{_responses.get_user_info.json[0]['uuid']_}}/logout/",
61   "method": "POST",
62   "data": {}
63 },
64 {
65   "template": "get_authentication_status",
66   "url": "http://oauth-login-backend.local:9501/",
67   "path": "authentication_session/{{_parse_redirect_url(responses.authorize_request.json)_}}/",
68   "method": "GET",
69   "data": {}
70 },
71 {
72   "template": "user_login",
73   "url": "http://oauth-login-backend.local:9501/",
74   "path": "authentication_session/{{_parse_redirect_url(responses.authorize_request.json)_}}/login/",
75   "method": "POST",
76   "data": {
77     "uuid": "{{_responses.get_user_info.json[0]['uuid']_}}"
78   }
79 },
80 {
81   "template": "get_registration_status",
82   "url": "http://oauth-login-backend.local:9501/",
83   "path": "registration/{{_responses.register_user.json.uuid_}}/",
84   "method": "GET",
85   "data": {}
86 },
87 {
88   "template": "register_user",
89   "url": "http://oauth-login-backend.local:9501/",
90   "path": "registration/",
91   "method": "POST",
92   "data": {
93     "identity": {
94       "organisation": {
95         "name": "test_org",
96         "id": "{{_str(uuid.uuid4())_}}"
97       },
98       "sub": "{{_str(uuid.uuid4())_}}",
99       "name": "Test_User",
100       "preferred_username": "testuser"
101     }
102   }
103 },
104 {
105   "template": "oauth_callback",
106   "url": "http://oauth-login-backend.local:9501/",
107   "path": "oauth_callback/?session_id={{_parse_redirect_url(responses.authorize_request.json)_}}",
108   "method": "GET",
109   "data": {}
110 },
111 {
112   "template": "authorize_request",
113   "url": "http://oauth-communication-service.local:9501/",
114   "path": "authorize/",
115   "method": "GET",
116   "data": {},
117   "params": {

```

```
118     "client_id": "{{_responses.register_client.json.client_id|_default('MISSING')}}",
119     "code_challenge": "{{_generate_code_challenge(code_verifier)}}",
120     "code_challenge_method": "S256",
121     "redirect_uri": "http://oauth-proxy.local:9501/oauth/",
122     "response_type": "code",
123     "scope": "openid"
124 }
125 }
126 ]
```

B

Initial experimentation

B.1. Experiments

B.1.1. Learning on sandbox

The first goal is to prove that the OAuth system is indeed stateful and that it is possible to apply state learning on the system. Therefore, as a first experiment passive learning is applied on the existing logs from Ubiqu's pre-production environment. The pre-production environment, from now on referred to as sandbox, is a Kubernetes cluster on which all necessary microservices for OAuth are present and used for testing purposes. This setup allows for system logs, which are created through real interactions with the system, to be extracted.

For experimentation purposes logs were collected from the following microservices, OAuth Management Service, OAuth Communication Service, OAuth Login Backend, and OAuth Login Frontend. The Identity Service and PPK Management Service were left out because their states are indirectly reflected in the OAuth Login Backend and OAuth Communication Service. In addition, the initial fuzzing campaign will not aim to directly fuzz these services, this will be a future goal. The Authenticate App and the Identification and Transaction services were also left out as they are represented indirectly in other microservices.

The logs are gathered into one file, from which the important information is then extracted for passive learning. The information extracted is as follows, the request method, request host, the path, the return status and the trace id. The traces are then clustered by trace id, and abstracted to create a readable format for the learning algorithm. Finally, actual learning of the system is done by FlexFringe. [31]

B.1.2. Learning from a Single Trace

Once it is proven that the system is indeed stateful and we can learn a state machine from the logs, the goal is to setup an active learning implementation. For that to be possible we must first be able to create valid traces that can complete the entire OAuth flow. In addition to being valid, these traces must be mutable for future experimentation. To achieve this, we use a json model structured such that it allows for the dynamic definition of HTTP requests, as we can insert responses from previous requests into future requests. In addition, we can define functions outside of the json that can map transform that can map the data. For example the code_challenge is generated dynamically. The entire definition of the alphabet can be found in Appendix A.

As active learning involves actual interaction with the system under learn, an experimental setup where we have control over the system is necessary. For the purposes of our initial experiments, we run most services locally in Docker containers. The following services are running on a remote server, the Authenticate App, the Identification and Transaction Services, the Identity Service, and the PPK Management Service. These services were left out of the initial learning campaign due to their added complexity, and the fact that outside users will never have direct communication with these services. The experimental setup can be seen in figure 7.1.

A key difference from the first experiment is that we introduce the Authenticate App to active learning, this is necessary as without explicit interaction with the Authenticate App the OAuth Flow should not be able to complete successfully.

To gather the logs while running the defined trace, we use an Envoy proxy through which the communication between all services is routed. As we want the results to be comparable to those to previous experiments we extract the same information from the logs and also use FlexFringe to learn a model.

B.2. Passive Learning

The application of FlexFringe (a passive learning method) on the existing logs of the OAuth system successfully generated a state machine. Therefore, showing that Ubiqu's implementation of OAuth is indeed stateful. The resulting state machine is pictured in figure B.1 and the corresponding legend for the transitions can be found in table B.1.

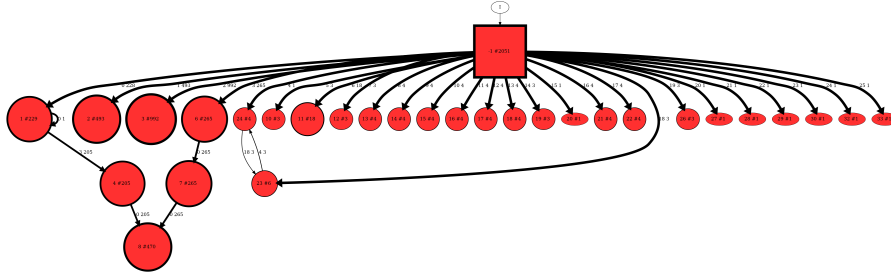


Figure B.1: State machine achieved through passive learning.

The final machine consists of 34 states, this count includes the initial state, and 33 transitions. Notably, there is one self loop when transition 0 occurs at least twice. In addition to the self loop, there is another loop in the system, with transition 4, 18, 4, this loop can also be reached through transitions 18, 4, 18. Transition 4 importantly is the redirect with the OAuth access token, and therefore when this transition is reached the OAuth protocol has been executed successfully.

The state machine also shows that not all possible requests are conducted on the OAuth system, the trace mostly consists of successful attempts for retrieving access tokens. This gives more reason for active learning, as not all actions are conducted with a high enough frequency to effectively learn from real use case logs.

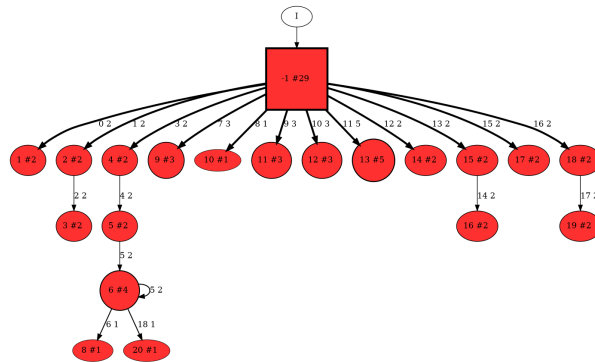
Crucially, not only does the resulting state machine show that OAuth is stateful, the success of this experiment shows that a state machine can be learned over multiple microservices through combining their logs.

B.2.1. Learning from a Single Trace

We successfully created a trace that dynamically generated cryptographic fields and mapped inputs dependent on prior requests properly. In addition, the trace was able to successfully complete a single OAuth flow. The state machine learned from the resulting logs is displayed in figure B.2 and the legend for the transitions can be found in table B.2.

Table B.1: Legend for figure B.1

ID	Method	Path	Status
0	GET	/internal_authorize_request/<UUID>/	200
1	GET	/authorize/	302
2	GET	/bind/	200
3	GET	/oauth/authorize/	302
4	GET	/oauth_callback/	302
5	OPTIONS	/authentication_session/<UUID>/login/	200
6	GET	/authentication_session/<UUID>/	200
7	POST	/authentication_session/<UUID>/login/	200
8	GET	/user/	200
9	OPTIONS	/authentication_session/<UUID>/	200
10	GET	/authorize/	200
11	OPTIONS	/user/	200
12	POST	/bind_ack/	200
13	GET	/bind_start/	302
14	GET	/	200
15	GET	/login/oauth/	302
16	POST	/token/	200
17	POST	/ppk/<UUID>/sign/	200
18	PATCH	/internal_authorize_request/<UUID>/	200
19	GET	/callback/<UUID>/	302
20	POST	/transaction/<UUID>/callback/	200
21	POST	/discover_account/<UUID>/callback/	200
22	OPTIONS	/authentication_session/<UUID>/identify/	200
23	POST	/authentication_session/<UUID>/identify/	200
24	GET	/authentication_session/<UUID>/	404
25	GET	/login/<UUID>	200

**Figure B.2:** State Machine resulting from single trace

The resulting model has 20 transitions and 21 states, including the root. The model includes only one self loop occurring after transition 3, 4, 5 with transition 5 and has a depth of 4. Even though the resulting model is relatively simple, it shows active learning on the system is possible, as we can dynamically create traces which can complete the OAuth flow. This linear structure is the expected outcome of learning from a single, successful execution trace. It effectively captures the "happy path" of the OAuth flow, serving as a crucial baseline for comparison. This successful trace generation and model creation is the necessary foundation for the full active learning campaign.

Table B.2: Legend for figure B.2

ID	Method	Host	Path	Status
0	GET	oauth-communication-service.local:9501	/authorize/	302
1	GET	oauth-communication-service.local:9501	/internal_authorize_request/<UUID>/	200
2	GET	oauth-login-backend.local:9501	/oauth/authorize/	302
3	GET	oauth-login-backend.local:9501	/bind_start/	302
4	POST	oauth-login-backend.local:9501	/bind_ack/	200
5	GET	oauth-proxy.local:9501	/login-frontend/authorize/	200
6	GET	oauth-proxy.local:9501	/login-frontend/bind/	200
7	GET	oauth-login-backend.local:9501	/user/	200
8	POST	oauth-login-backend.local:9501	/registration/	201
9	GET	oauth-login-backend.local:9501	/registration/<UUID>/	200
10	POST	oauth-login-backend.local:9501	/authentication_session/<UUID>/login/	200
11	GET	oauth-login-backend.local:9501	/authentication_session/<UUID>/	200
12	POST	oauth-login-backend.local:9501	/user/<UUID>/logout/	200
13	PATCH	oauth-communication-service.local:9501	/internal_authorize_request/<UUID>/	200
14	GET	oauth-login-backend.local:9501	/oauth_callback/	302
15	GET	oauth-communication-service.local:9501	/callback/<UUID>/	302
16	POST	oauth-communication-service.local:9501	/token/	200
17	GET	oauth-proxy.local:9501	/oauth/	200
18	GET	oauth-proxy.local:9501	/login-frontend/	200