Analysis of IoT devices via API Exploitation and Model Extraction

Alexandre Guedes Pinto Eraclides

Thesis to obtain the Master of Science Degree in

Bologna Master Degree in Information Systems and Computer Engineering

Supervisor: Prof. Pedro Miguel dos Santos Alves Madeira Adão

Examination Committee

Chairperson: Prof. José Carlos Martins Delgado
Supervisor: Prof. Pedro Miguel dos Santos Alves Madeira Adão
Member of the Committee: Prof. João Carlos Serrenho Dias Pereira

November 2018
I dedicate this thesis to my parents and sister, that helped me on this journey.
Acknowledgments

First and foremost, I would like to thank my family for all the support and help they have always given me.

I would also like to thank my supervisor Professor Pedro Adão for allow me to work on this subject in Computer Science and for all guidance provided throughout the time that this work was developed.
Resumo

Ao longo dos últimos anos temos assistido a uma grande expansão dos dispositivos IoTs (Internet of Things) em diferentes áreas, levando a um aumento exponencial na produção de dispositivos como smartwatches, smart TVs, frigoríficos inteligentes, carros inteligentes entre outros. A grande maioria dos utilizadores de IoTs não está ciente, que o desenvolvimento e produção de muitos destes produtos são apressados de forma a chegarem ao mercado o mais rápido possível, pelo por vezes os testes poderão não ser efectuados de forma adequada. Este problema poderá ser exacerbado no futuro devido à previsível explosão na utilização dos IoTs e ao aumento da complexidade na sua utilização.

Nesta tese desenvolvemos uma ferramenta que vai efetuar testes automáticos aos dispositivos IoTs através de uma combinação de sequências de métodos com testes do tipo black-box. No final a ferramenta produz um modelo que contém as sequências onde foram detetados problemas. As sequências em conjunto com os logs produzidos pela ferramenta, ajudam a recriar os problemas.

Esta ferramenta testa todos os métodos presentes na API do IoT em diferentes estados internos. Testes black-box são utilizados para testar o último método da sequência, uma vez que o objetivo destas sequências é alterar o estado interno do dispositivo de modo a testarmos os métodos em diferentes condições. Esta metodologia de testes permite maximizar o número de situações que se possam expor comportamentos não desejados no dispositivo que está a ser testado.

Palavras-chave: IoT, API testing, black-box testing, model inference, fuzzing, REST
Abstract

In the last few of years there has been a great expansion in the development of the Internet of Things (IoT) on different areas, with a great increase in the production of new devices like smartwatches, smart TVs, Internet refrigerators, smart light bulbs and many more. The great majority of the IoT consumers are not aware that many of these products are rushed to the market and extensive testing is not always performed. This problem will get worse in the future, due to the foreseen increase of IoT devices and their complexity, which will also increase the possibility of errors in the software of these devices.

Over the course of this thesis we developed a new tool that automatically tests the IoT devices through a combination of method sequences and black-box testing. Also at the end of the testing, the tool generates a model that exhibits the sequences with problems. In order for the user to have a better understanding of these problems, the tool produces logs that together with the model allow him to recreate the problems found during the tests.

The tool tests every method in different internal states of the IoT device that it is able to reach. These sequences have the objective of setting the device into different internal states to exercise the methods in different environments. The goal of this test methodology is to attempt to maximize the number of scenarios that correspond to unintended behavior of the devices.

Keywords: IoT, API testing, black-box testing, model inference, fuzzing, REST
Contents

<table>
<thead>
<tr>
<th>Acknowledgments</th>
<th>v</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resumo</td>
<td>vii</td>
</tr>
<tr>
<td>Abstract</td>
<td>ix</td>
</tr>
<tr>
<td>List of Figures</td>
<td>xiii</td>
</tr>
</tbody>
</table>

1 Introduction                                 |       |
1.1 Thesis overview                           | 4     |

2 Related Work                                 |       |
2.1 IoT                                       | 5     |
2.1.1 IoT Wireless Technology                | 7     |
2.1.2 IoT and the Internet                   | 8     |
2.2 API Testing                               | 9     |
2.2.1 Sequence of Methods                    | 10    |
2.2.2 Black-box Testing                      | 12    |
2.3 Model Inference                           | 18    |

3 The Testing Tool                             |       |
3.1 SET-UP phase                              | 23    |
3.1.1 Configuration JSON                      | 25    |
3.1.2 Specification API JSON                  | 28    |
3.1.3 Reset API JSON                          | 32    |
3.2 Tool Architecture                         | 33    |
3.2.1 Jsonreader                              | 34    |
3.2.2 Sequence.producer                       | 35    |
3.2.3 Fuzzer                                  | 36    |
3.2.4 Caller                                  | 44    |
3.2.5 Logger                                  | 47    |
3.2.6 Model.producer                          | 50    |
3.2.7 Main                                    | 53    |
3.2.8 Tool Outputs                            | 56    |
4 Results

4.1 Device A

4.1.1 First issue

4.1.2 Second issue

4.1.3 Third issue

4.1.4 Fourth issue

4.1.5 Fifth issue

4.1.6 Sixth issue

5 Conclusions

5.1 Future Work

Bibliography

A Appendix Specification API JSON
## List of Figures

2.1 IoT prediction. Image from [5] ................................. 6
2.2 IoT market by 2025. Image from [6] ........................ 7
2.4 Randoop method sequence iteration. Image from [13] .................. 10
2.5 Java Path Finder. Each box represent an operation, where the objective is to generate long chains of operations. Image from [13] .............................. 11
2.6 Random executions versus systematic executions. Image from [13] ............... 12
2.7 Advantages and disadvantages of generation-based fuzzers vs mutation-based fuzzers. Image from [16] .................................................. 14
2.8 Random-based fuzzer input. Image from [19] .......................... 15
2.9 Template-based fuzzer input. Image from [19] .......................... 15
2.10 Block-based fuzzer input. Image from [19] ............................ 16
2.11 Pulsar overview. Image from [1] ...................................... 17
2.12 General workflow of CSight. Image from [23] ......................... 20
2.13 Synoptic input/output. Image from [24] .............................. 21
2.14 Different models comparison. Image from [26] .......................... 22

3.1 Fields of the configuration Json file ................................ 25
3.2 Hierarchical diagram for the Specification API Json file ..................... 28
3.3 Example of Specification API JSON file ................................ 29
3.4 Hierarchical diagram for the Reset API Json file .......................... 32
3.5 Architecture of the tool ................................................. 33
3.6 Example of the “setup” section ......................................... 35
3.7 Template of an error message. ........................................... 47
3.8 All logs ........................................................................ 48
3.9 Logs of Errors ................................................................ 49
3.10 Logs used in the Model-producer module ................................... 50
3.11 Model of the device A ..................................................... 51
3.12 Model of sequence with one method ...................................... 51
3.13 Model of sequence with two methods ..................................... 52
3.14 Total log file .................................................................. 56
Chapter 1

Introduction

The potential of the Internet of Things (IoT) is becoming increasingly more important in the world market. Initially it gained greater visibility in the domestic segment associated to electrical appliances and to personal use technology, but new emerging projects such as self-driving cars, health care monitoring, smart cities or home automation are generating new deployment opportunities.

In a near future, the rate that the IoTs will come up and their diversity will be so extraordinary that they will be present in almost every aspect of our lives, sharing and collecting data. The effect of the IoTs will go beyond the lives of each person and will have a great impact in every branch of our society, like the industry, health care, economy, cities and many others.

If we visualize the IoT network as a pyramid, at the top we have some devices that are used a lot and with a lot of computing power, such as laptops, smartphones, smartwatches and game consoles. On the middle, there are dozens of devices occasionally used, and that have moderate computational capacity such as thermostats, TVs, refrigerators, among others. At the base, there are hundreds of devices that people forget easily, such as air conditioning and heating systems (HVAC), implanted medical devices, digital photo frames, and electronic locks among many.

The new IoT solutions have an additional layer of architecture with software focused on data analytics, since it is not enough to have a mesh of sensors to collect data without being able to extract information and knowledge out of them. IoT networks offer the promise of a more meaningful interaction with users, but also with business enterprises, because real-time or near-real-time information will support a more timely and rigorous decision-making.

However, these projects may encounter difficulties nowadays due to a number of reasons, such as the fragmentation of equipment to implement a solution, technology standards, the volume of data collected, the privacy of the data collected and the important security issues that the IoTs will bring.

In the coming years, the IoT devices will start to be used in systems where the consequences of a hacking attack can be catastrophic. Since lives and infrastructures will be at risk, if these systems are hacked they can cause accidents on driverless cars, pacemakers malfunction, traffic jams in smart cities or fires at smart homes. These are some scenarios that are not difficult to imagine. Security breaches are probably the biggest concern and there is no doubt that risk levels will increase with the growth of
IoT deployments.

Big companies like Amazon, Microsoft, Google, among others, spend great sums of money to guarantee that their devices are extensively tested for all kind of problems, and they take special attention to problems related with security. These companies make sure that they employ experienced people on the security field to test and experiment the devices, looking for security vulnerabilities. The problem arises on middle and small companies that want to launch their IoT devices to the market, but do not possess the same knowledge, neither the budget nor the time (they want to sell their product as fast as possible) to do proper security testing.

The IoT is where the Internet meets the physical world and they open a completely new dimension to security. The IoTs will have a great impact on security as the attacks move from manipulating information to controlling actions. The explosion of IoTs will result in an exponential increase of the known attack vectors, since any device connected to the internet is a potential target. Nowadays a lot of IoT devices are vulnerable, either because the device was a simple sensor with low computing power and the security was not a concern, or because the developing team lacked security expertise.

For instance, Mirai is a malware that affects network devices and turn them into remotely controlled “bots”. All these “bots” together make a botnet that is used to control them simultaneously. Later this botnet can be used to create large-scale network attacks\(^1\). This is one of the dangers that can affect the IoT, since any insecure IoT can be infected to become a part of a botnet.

The OWASP (Open Web Application Security Project) foundation was created in 2001, to increase the world awareness to cybersecurity. The OWASP is a community dedicated to facilitate organizations to conceive, develop, acquire, operate, and maintain applications that can be trusted. All of the OWASP resources, that range from tools, to documents, to forums, etc, are free to anyone interested in improving its knowledge about security. In 2014 they published the top 10 most common vulnerabilities for the IoTs\(^2\), this ranking takes in consideration how easy it is to explore the vulnerability, how common it is to find the vulnerability and the impact that the vulnerability may have. The ranking is as follows:

1. Insecure Web Interface
2. Insufficient Authentication/Authorization
3. Insecure Network Services
4. Lack of Transport Encryption/Integrity Verification
5. Privacy Concerns
6. Insecure Cloud Interface
7. Insecure Mobile Interface
8. Insufficient Security Configurability
9. Insecure Software/Firmware

\(^1\)https://en.wikipedia.org/wiki/2016_Dyn_cyberattack access on 12/14/2017
\(^2\)https://www.owasp.org/index.php/Top_IoT_Vulnerabilities access on 12/14/2017
10. Poor Physical Security

Usually the IoT APIs either use REST or SOAP protocols to communicate over the internet. Although there are not much work done in the field of testing APIs of IoT devices through their respective protocols, testing has been largely studied for general software. One way that an API can be tested, is to individually test each method call. To this end black-box testing is an effective technique, that does not require access to the source code and has the goal of testing the program or method through its input. There are some tools like Pulsar[1], AFL[2], libfuzz3, Sulley4/Boofuzz5, Autofuzz[3], Syntribos6, PyJ-Fuzz7 and others, that were developed to test specific objectives like an internet protocol, local program coverage or even REST APIs.

Some of these tools such as zzuf8 and others are very easy to set up and use, but are limited in the way that they grasp the system, meaning that they only test the method itself without reasoning about the internal state of the system. Others like Autofuzz, Sulley/Boofuzz and Pulsar take a different approach, for instance some start by learning how the system interacts while others receive a file that details how to interact and test the system. While the latter appears to be more accurate and reliable, it has the disadvantage that it takes a lot of time to set up, or it requires a lot of knowledge about the tool in order to use it, or requires some extensive manual work to describe the communication or even to “feed” the tool with enough sample examples for it to learn how to interact with the system.

In order to test non-intended behaviours of IoT devices, we developed a tool that tests these devices through the API REST communication protocol, thus excluding any physical tests. Our tool tries to solve some of the existing problems in the current tools: It is open source, it requires minimal interactions with the user during the tests to the IoT device, it automatically generates tests and executes them on to the IoT and at the end a model is produced that becomes the “roadmap” to recreate the problems found. In order for this tool to work it requires three different files from the user: one file to describe the API of the IoT, another file to describe how to do the “reset” in the IoT device, and the last one to configure the tool running parameters to the user preferences.

The IoT devices may have several internal states depending on the requests/configuration provided by the user. For example, it is possible for a smart bulb to be on different states, examples are: state with the light “off”, or state with light “on”, or for the bulb to belong to a pool of devices, or even for pool to contain no devices. Several internal states can exist in these devices/systems and one of the prime objectives is to check how the device behaves in the different pool of existing internal states.

The testing of the IoT API device will be done in two major steps. The first step is the “setup”, on which the device is set in a certain internal state and some of the devices internal values are retrieved. Followed by the second step which runs the automatic tests, using all the available method calls in the API of the device. This way, it is possible to check the correct behavior of the methods. In addition, it is also possible to evaluate how the device behaves calling the methods in different environments (internal

3https://llvm.org/docs/LibFuzzer.html access on 12/14/2017
4https://github.com/OpenRCE/sulley on 12/14/2017
5https://github.com/jtpereyda/boofuzz on 12/14/2017
6https://github.com/openstack/syntribos access on 12/14/2017
7https://github.com/mseclab/PyJFuzz access on 12/14/2017
8http://caca.zoy.org/wiki/zzuf access on 12/14/2017
At the end, after all the testing is complete and the logs are recorded, it is possible to produce a model that helps the user to “clarify” what are the problems in the IoT and what are the methods that need to be called, in order to recreate the discovered problem in the logs.

This tool is only able to detect problems through the output produced by the IoT device, meaning that it can only deduce that a problem exists when the device outputs an exception, or when an input is accepted that shouldn’t be based on the specification of the IoT, or even if the device is on a state that shouldn’t be possible according to the sent payload.

1.1 Thesis overview

During the rest of this document, we will have the following sections. On the related work section, we will discuss all the researched material as well as the existing solutions and methodologies. On the tool section, we will discuss the architecture of our solution, explaining in detail the different modules, what were the reasons that led us to make the development in a certain way. On the results section, we will present our discoveries and discuss if the tool meets our expectations. On the last section, we have the conclusion, on which we will give an overview of our work and findings, together with what can be improved in the future.
Chapter 2

Related Work

In this related work we will start by describing the state of art of the IoTs (Internet of Things), the technology that these devices use and which protocols are available to connect them to the internet. We will then follow up by explaining how the API (Application Programming Interface) can be tested and we will also show some techniques to infer the internal model of the system using the system executions logs.

2.1 IoT

One of the many definitions of IoT is as follows: “an interconnection of uniquely identifiable embedded computing devices within the existing Internet infrastructure, offering advanced connectivity of devices, systems and services that goes beyond machine-to-machine communications and covers a variety of protocols, domains, and applications”[4]. Where the word “Things” can range from small objects like sensors, smartwatches and medical devices to big objects like cars, fridges and houses/buildings. These devices interact with the users by generating and collecting information about its environment.

The number of devices that are connected to the Internet is growing at an unprecedented pace, for instance, the figure 2.1 shows an estimated number of 50 billion connected devices by 2021, which can range from computers, smartphones, tablets, gaming consoles, etc. A major contribution to this number will be from small devices with all the new type of gadgets and sensors that are emerging. Together they will create the environment of smart homes, smart hospitals and smart cities, in 2025 the figure 2.2 represents the market division of the IoTs.
The IoT allows physical objects to “interact” with the environment around them, to analyze and to communicate with other objects, to coordinate and take decisions. A world where these devices have a big impact in our lives is not too far from our current reality. Let us take an extreme example and assume that we have a house equipped with sensors like smart bed, smart coffee machine, smart shower, smart TV, smart HVAC, etc, which would be connected together to help with our daily lives forming the environment of a smart house. Then, it would be possible for the “house” to know when a person wakes up and prepares coffee, when would be the time to start warming the water for a shower (if that is the routine), when to turn the TV on the usual channel, when to inform the user of ingredients missing on the fridge, when to turn the light on if there is not enough ambient sunlight, when to open the garage, etc. We can see how much our lives would change and improve by the automatization of all these actions that are made possible by the interaction between different devices. Let us think of another example, the IoT could help to monitor a person’s health condition, by monitoring simple parameters like the person’s temperature, weight, food consumption, water consumption, heart rate, etc and establish a pattern. If there is any deviation from this pattern, it may mean that something can be wrong with the person’s health. This early detection of pattern deviation can help to detect early diseases.
2.1.1 IoT Wireless Technology

The figure 2.3 shows the technology network range, where WPAN - Wireless Personal Area Network operates until 100 meters and are usually associated to a smartphones connected over Bluetooth, to gadgets such as wireless headsets, to smartwatches or to fitness devices. WPAN devices rely on small batteries, so they usually have low transmission power. WLAN - Wireless Local Area Network (WLAN) operates until 1000 meters and are usually associated to home Wi-Fi networks. WNAN - Wireless Neighborhood Area Networks operates until 10 Km and an example is a smart grid that transmits electric readings from homes to power companies. Wireless Wide Area Network operates up to 100 Km and it is usually associated with mobile networks.
As we can see there are different types of wireless technologies that range from close range to many kilometers of distance, but the wireless networks can also be distinguished by their topology. There are two major types of topologies for networks. The Star topology is one of the most common network topologies. In a star network exists a central hub (can be a router, gateway, or a controller) which all the nodes communicate directly and then the hub retransmits the messages to the internet or to another hub. If a node wants to communicate directly with another node the message must be relayed through the hub. The mesh topology has no need for a central hub, since any node can transmit the message to any another node. The idea is if two nodes cannot reach each other, the message can be relayed through various nodes until it reaches the destination node. The mesh topology has the advantage of reaching further than other technologies by hopping a message between several nodes. One variation of a mesh topology is a gateway node that bridges the messages of the mesh network to the internet.

The IoT world is based on the WPAN and WLAN technologies with the following wireless protocols that dominate the IoT: Bluetooth, Zigbee\(^7\), Z-Wave\(^8\), 6LowPAN. On \(^9\) a solid review is made about all these technologies.

These protocols are quite used in IoT products. The Zigbee protocol is a high level WPAN communication protocol that provides wireless connection on home automation devices and other low-power devices and is used for instance by the Philips Hue bulbs\(^1\). Another example of usage of these protocols is the GoControl garage door that connects the device through the Z-Wave\(^8\) compatible smart home hub.

### 2.1.2 IoT and the Internet

IoT are associated with APIs (Application Programing Interface) because they act as an interface between different programs to ease their interaction. To connect these devices over the internet, it is normally used a Web service as the means of communication, which can either use SOAP, REST or XML-RPC over HTTP protocol.

In the last few years there has been a major shift from SOAP (Simple Object Access Protocol) web services, where the communication is done by using XML enveloped messages, to REST (Representational State Transfer) web services. This concept was first introduced by Roy Fielding in 2000 on his PhD thesis\(^{10}\), where it states that REST is not a protocol like SOAP, but rather a set of architectural guidelines, on how a web service should be developed over HTTP protocol. Rest APIs rely on the HTTP methods (POST, GET, PUT, DELETE, etc) to perform different actions and the response may be in different formats, being the most popular ones the JSON, XML or HTML.

As REST is based on HTTP, any device that is connected to the internet and is able to request a web page, it can also use a REST API. That is the reason why REST is suited for IoT, because a device can quickly make the information about its state available and also the REST model for an IoT has a faster development. For example, we can make request to know about the state of the device, followed by another request to change the state of the device.

\(^1\)https://www.developers.meethue.com/ on 12/14/2017
The problem lies in the trust that is necessary to have in these APIs, since the source code is unavailable and unknown, and many of these APIs are developed under extreme pressure to hit the market as soon as possible. Due to this, many times it results in unexpected errors or flaws in the code of the API, which may result in vulnerabilities and entry points for the hackers. Since APIs are used in other programs, the vulnerabilities may propagate to those programs, so there is a need for extensive testing in order to ensure that the API has a minimal number of bugs when its IoT goes to market.

2.2 API Testing

When testing an API - Application Programming Interface, we have to take into account that in the majority of cases, we have some constraints to run a complete set of tests to the API. One of the limitations is that the source code is not always available. Another big limitation is that the specification of the API is not always complete, or it does not represent the current state of the API. This can happen due to a number of reasons, one of the reasons is the API being updated but the specification is not, or due to last minute changes when the specification is already finished and published. This lack of knowledge/information about the internal workings of the API, makes it harder to develop tests for the APIs.

From our perspective the best way to test an IoT, is through a combination sequences of method calls to the DUT - Device Under Test, together with black-box testing (more specifically fuzz testing). With sequences of method calls we are able to “control/setup” the internal state of the device, and with black-box testing we can test the validations of the input parameters to the API methods. So the objective is to use sequences to put the DUT in different internal states, and then use black-box testing to see if some malformed input is accepted. With this technique we can detect problems involving specific internal states and malformed inputs, or just the cases where we check the correct validation of the input parameters in the API methods.
2.2.1 Sequence of Methods

The objective is to feed the DUT with sequences of method calls to ensure that the DUT does what it is suppose to do.

There are a lot of ways to create sequence of method calls, for example Randoop\cite{11} focus on the random generation of unit tests for JAVA programs. These tests usually consist of a sequence of method calls (figure 2.4) that create and mutate objects. At the end an assertion is made with the result of the sequence final method call. A test is created iteratively by randomly choosing a method or a constructor to invoke using previously computed values as inputs, also only legal sequences of methods are extended, (meaning that sequences that cause redundant states or states that do not respect the contract are not pursued). The feedback-directed generation algorithm creates a new sequence by concatenating its input sequences and appending a method call to the end starting from an empty set of sequences. This approach is also the base for the technique GRT - Guided Random Testing\cite{12} that tries to improve the quality of the test, by first doing a static analysis over the classes under test to extract some knowledge about it, such as possible constants during execution, method side effects and their dependencies. Then GRT uses this analysis to create a comprehensive pool of initial constant values and determines the properties of methods that form the basis of method sequence generation.

![Figure 2.4: Randoop method sequence iteration. Image from [13]](image)

Another way to generate method sequences is based on a model. JPF (Java Path Finder)\cite{14} started has a model checker, in which the feature “execution choices” is used to identify points where the program executes different parts of the code and then makes sequences to test all possible combinations. To create these sequences JPF relies on heuristics like BFS (Breadth-first search), DFS (Depth-first search) and others. Once again the objective is to test all possible executions of a program. The figure 2.5 shows an example of an execution of JPF.
Figure 2.5: Java Path Finder. Each box represent an operation, where the objective is to generate long chains of operations. Image from [13]

The tool TAUTOKO[15] is a typestate miner, which the objective is to create a specification based on program executions. To achieve this, TAUTOKO generates the first typestate model, only by analyzing the normal executions of the program. Then to enrich the model TAUTOKO will execute all methods in all states of the initial model. This approach is similar to the JPF because it tries to explore every possible combination, but afterwards TAUTOKO uses the typestate model to infer specifications about the program.

Making an evaluation of both techniques, it shows that random sequence executions have the advantage of being relatively fast, easily scalable, have a simplistic implementation and together with feedback testing is capable of going deep in testing the program executions. However, it has some problems because it doesn’t cover all possible executions, meaning that to find an interesting execution (an execution with an error) is a matter of “luck” and it also has the problem of converging to states that are “easy” to reach. Systematic sequence executions have one big advantage over random executions, it can cover all paths. However, the capacity of covering all paths also brings some drawbacks, like the problem of state explosion and the time that is needed to execute all paths.

Consider the grey triangle on the figure 2.6, the execution space of a program. On the left of the figure, we have the random sequences where we can see some executions go deep in execution space, but there are a lot of grey space between the executions. The middle and right of the figure represents respectively systematic sequences using BFS and DFS respectively and we can see a more compact explored space, although it still remains a lot of execution space to cover. Another point that we have to take in account is that to produce this kind of compact cover, the number of states that must be expanded can grow exponentially producing the problem of state explosion.
If we analyze this test approach from the point of view of a developer, it would be desirable to have a solution that can scale in a relatively easy way, a solution that can provide some good and different executions on the program, but above all, a solution that is fast because there are always concerns with the deadlines and with the budget. The random executions meets this criteria while the systematic executions causes stress on the time and budget. On the other hand, if we analyze the test approach from a hacking point of view, which time is not a concern, then the systematic executions would be a better solution because it presents a better coverage of all the executions of the program and it is more suitable to find the vulnerabilities that can be explored.

2.2.2 Black-box Testing

The source code behind the methods API usually is not available. So a good way to test the correct implementation of these methods is through Black-box testing, which is the type of testing that doesn’t know how the program works (because it doesn’t need to have access to source code) but knows what the response should be, so it relies on program inputs and outputs to test its functionality.

A way to test the correct functionality of the API methods is through the fuzz testing of the input parameters. The objective of fuzz testing is to call the method or program to be tested with all kind of different inputs. As an example, imagine that there is a method that receives an integer as input, then with this test we will execute the method with different types of inputs like strings, large integers, negative integers, special characters, etc. Through this way of testing, we can check if the method makes the correct verifications to its inputs, because if it is not the case, then it opens a way to explore the vulnerabilities of the method.

The difference between random testing and fuzz testing, is that the objective of random testing is not to crash the program. The randomized values can be inside the valid parameters and the primary objective is to find new information about the program or to increase coverage, while fuzz testing has the objective of crashing the program. This is done by sending all kinds of inputs to the program, where an evaluation is done to see if the program can deal with inputs that most of the time are meaningless.
**Fuzzers Definition**

A fuzzer can be defined based on three factors:

1. It can be a smart or a dumb fuzzer depending if it has knowledge or not about the input structure.
2. It can be a generation-based or mutation-based fuzzer depending on how the input is constructed.
3. It can be white-box, gray-box or black-box depending if it is aware of the program structure.

Starting on the first factor, a dumb fuzzer doesn’t know what the input should be, meaning that it is not aware how a valid input to the program must look to be accepted. So in a nutshell, a dumb fuzzer will generate/create random inputs regardless of the inputs which are used. This brings some advantages, a dumb fuzzer is extremely easy to develop and in theory it can be applied to any “situation”, but it has the disadvantage that most of the time, it will not produce a valid input specially in situations that require a “complex” parameter. For example, let us assume that we require to fuzz a field of an email address, but the program doesn’t check the syntax of the input, then a dumb fuzzer would be enough. On the other hand, if we assume that the program does some validation of the input (for example, the input must contain “@”), then a dumb fuzzer would not be appropriate, because a dumb fuzzer would still test every possible input and most of the time it would not produce an acceptable input.

To address the problem of “complex” inputs, we have the alternative of smart fuzzers, which have some kind of insight about what valid inputs are. These kind of fuzzers usually have some kind of documentation about the inputs or have config files to define how the input should be. This brings the obvious advantage of the smart fuzzer having a much higher efficiency over the dumb fuzzers, it also has some disadvantages like the time that is needed to develop a smart fuzzer, the time that is required for the fuzzer to learn how the input should be, the availability of the specifications and the time required to produce manually a config file. Returning to the example of the email address, if a specification was given stating that the email parameter needs to have the character “@”, a smart fuzzer would only fuzz interesting inputs to check the program, such as “@”, “a@”, “a@a”, “@a”, “-1@21AC”, etc.

The second factor is how a fuzzer constructs its inputs. From an abstract point of view a fuzzer can either be generation-based or mutation-based. Mutation-based fuzzers, as the name suggests, have the objective to mutate something. Usually this type of fuzzers receive a valid initial seed that they mutate, creating similar versions of the initial seed with minor changes and sometimes this is enough to detect some kind of vulnerabilities. This type of fuzzers have the advantage of being very easy to set up and most of the time, they do not require to have any knowledge about what they are testing, because they will only mutate a given seed. However it has some limitations, due to the fact that it will not make radical changes to the initial seed and mutation-based fuzzers are normally dumb fuzzers. These fuzzers change/mutate randomly, resulting frequently that the seed mutates to a not acceptable input. Let us use again the example of the email, if the following seed is given “xxx@something.com”, in one of the iterations the fuzzer may change the “@” to something else breaking the valid syntax of the input.
The generation-based fuzzer works by constructing the input from scratch, as a result these fuzzers are associated with smart fuzzing. The generation-based fuzzer must be able to generate a full valid input from scratch, so it must have some insight about what are valid inputs. The generation-based fuzzers have the advantage of being able to go beyond where a mutation-based fuzzer can go and have all the basic elements for that input, since generation-based fuzzers create their inputs from scratch with the basic syntax of what the program needs, to validate that input, the rest of the parameters can be fuzzed, while mutation-based are limited by the scope of the initial seed and have the risk to mutate a character of the basic syntax of the input. Nonetheless, there are some negative issues against the generation-based fuzzers, because they take time to set up, it normally requires intensive manual labor to describe what to fuzz in complex inputs and they require some specifications about the program. On the figure 2.7 we have a table describing some of the advantages and disadvantages of mutation-based fuzzers compared with generation-based fuzzers.

![Figure 2.7: Advantages and disadvantages of generation-based fuzzers vs mutation-based fuzzers. Image from [16]](image)

The third and last factor depends on the knowledge about the structure of the program. The white-box fuzzers do some kind of analysis of the program source code, like static analysis, taint analysis or even symbolic executions to help the fuzzer to find new execution paths. SAGE[17] and Driller[18] are good examples of a white-box fuzzers. Gray-box fuzzers use some kind of instrumentation to glean some program structure, for example code coverage. Finally the black-box fuzzers have no access to the source code or any instrumentation of the application, so they can only reason or analyze the outputs of a given input. During the rest of this chapter we will introduce some combinations and tools that make use of these three definitions.
Dumb Fuzzers

The most simple fuzzer that can be developed is a dumb generation-based black-box fuzzer or Random-based fuzzer, the objective of this fuzzer is without any knowledge of the program/protocol being tested generate random inputs and analyze the outputs. We can see on figure 2.8 an example of an input that can be used in any kind of program input. It has the advantages of being very easy to develop and set up, but at the same time depending of the program/protocol being tested this kind of fuzzer can be useless.

**fuzzed input:**  \texttt{HdmxH&k dd**&&}%

Figure 2.8: Random-based fuzzer input. Image from [19]

Another combination that is widely used because of its simplicity and to a moderate extent with acceptable results is the dumb mutation-based black-box fuzzer or Template/Mutation-based fuzzer. As we discussed previously, this method consists of receiving a sample seed or a template and generate random mutations to feed the input of program/protocol. The advantages and disadvantages of this protocol were already discussed and on the figure 2.9 we can see an example of this application.

**template:**  \texttt{GET /index.html}
**fuzzed input:**  \texttt{GET /index.html, GET /inde?.html}

Figure 2.9: Template-based fuzzer input. Image from [19]

A broadly used fuzzer to test input files in a relatively easy way, is a dumb mutation-based black-box fuzzer like \texttt{Zzuf}[^2], that receives a sample of how the file should look and start to flip random bits on a valid file. In \texttt{Zzuf}, we can choose the percentage of bits in the file that we want to fuzz and see if program accepts the fuzzed file.

Smart Fuzzers

The next type of fuzzer is used a lot to fuzz local programs, it is a smart mutation-based gray-box fuzzer or Evolutionary-based fuzzer. Tools like \texttt{AFL}[2], \texttt{Libfuzz}[^3] and \texttt{VUzzer}[20] make use of this technique and the idea behind is quite simple. It starts by receiving a corpus of sample inputs or no samples at all (in this case it starts by generating a random input) and applying random mutations to these inputs. Then based on the feedback of the program and also with the help of instrumentation to detect block transitions, the tool can detect what inputs lead to new blocks in the code and retain those inputs to mutate them further. These fuzzers are largely used in a local environment because they need to be compiled with the program.

[^2]: http://caca.zoy.org/wiki/zzuf access on 12/14/2017
[^3]: https://llvm.org/docs/LibFuzzer.html access on 12/14/2017
White-box Fuzzers

SAGE[17] was one of the first implementation of White-box fuzzing. SAGE targets large applications where a single execution may contain many instructions. SAGE implements a directed-search algorithm, that maximizes the number of new input tests generated from each symbolic execution. Given a path constraint, all the constraints in that path are systematically negated one by one, placed in a conjunction with the prefix of the path constraint leading to it, and attempted to be solved by a constraint solver. This way, a single symbolic execution can generate thousands of new tests.

Driller[18] is another white-box fuzzer. This tool was built on top of AFL with the use of angr as a symbolic tracer. Driller traces inputs generated by AFL when AFL stops reporting any new interesting paths. Driller will take all untraced paths which exist in AFL’s queue and look for basic block transitions that AFL failed to satisfy. Driller will then use angr to synthesize inputs for these basic block transitions and present it to AFL. From here, AFL can determine if any paths generated by Driller are interesting, it will then go ahead and mutate these as a normal seed in an attempt to find more paths.

Block-Based Fuzzers

The following type of fuzzer is extensively used to test network protocols, it is a smart generation-based black-box fuzzer, which usually receives as an input a configuration file or a script that explicitly states how a request should be and what to fuzz and not to fuzz, a good example is the figur 2.10. There are a lot of fuzzers of this type, but we will only mention in more detail: Sulley⁴/Boofuzz⁵, Syntribos⁶, Pulsar[1] and Autofuzz[3].

Sulley was dropped and then picked again under the name of Boofuzz. Boofuzz is a network fuzzer framework based on the fuzzer SPIKE, which was the first fuzzer to come up with the idea of block-based fuzzing. It uses a python script, in which the protocol is described in a block-based way, where then it is used to test the protocol. Boofuzz also has the capacity of linking several requests together in order to form a session, this is useful to go through a process of authentication. The phase of authentication has to be manually described on the script or it can also be the target of the fuzzing process. This process of linking requests can be used to test a certain sequence of requests or to guide the protocol to a certain state where the fuzzing will be happening.

Syntribos is also a fuzzer that receives a configuration file and an example of a HTTP request, Syntribos can then replace any parameter of an API URL, HTTP header and request body with a given set of strings. Syntribos works by iterating through each position of the request automatically and in

---

⁴https://github.com/OpenRCE/sulley on 12/14/2017
⁵https://github.com/jtpereyda/boofuzz on 12/14/2017
⁶https://github.com/openstack/syntribos access on 12/14/2017
this way it is capable of finding vulnerabilities like SQL injection, LDAP injection, buffer overflow, etc. *Syntribos* was developed with the idea of testing API's, so having this in mind the developers defined minimal effort to make a configuration file.

**Model-Based Fuzzers**

The next tools are a variation of the block-based fuzzing because they first analyze the network traffic, to infer a model about that protocol, then it uses the model to understand the protocol to be fuzzed and to gather information about the parameters. By doing this, it minimizes the use of the configuration file to “educate” the fuzzer on how to fuzz the protocol, instead it takes the fuzzer to learn through the real example of the protocol in use.

Another fuzzer network protocols tool is *Pulsar*[1], which has the objective of testing and fuzzing the implementation of protocols that do not have specifications exits and a source code which is difficult to analyze. This tool uses a combination of fuzz testing concepts with reverse engineering and simulation. It works by observing the network traffic of an unknown protocol and infers the general model (in a markov representation) for the message formats and protocol states. By using this model it can effectively explore the protocol state space during the fuzzing process and direct the analysis to states which are rarely visit, so it is more likely to expose vulnerabilities under fuzz testing. *Pulsar* extracts a template from the inferred model that uses to define a set of primitives that can be applied to message fields at specific stages of the communication. The general process of *Pulsar* is shown in the figure 2.11.

![Figure 2.11: Pulsar overview. Image from [1]](image)

*Autofuzz*[3] also fuzzes network protocols, that learns the protocol implementation by constructing a finite state automaton (FSA) from the captured network traffic. By applying bioinformatic techniques *Autofuzz* learns about individual message syntax, including the fields and the types that are associated to each field. In a general way *Autofuzz* mutates in a “smart” way the communication sessions between server and client using the FSA as a guide.

*IxFIZZ*[21] uses the modeling language *Promela*, and its interpreter *SPIN*, to describe the the protocol state space, after this description of the protocol it uses the model to do an automatic generation of stateful fuzzing scripts on the fuzzing framework *Sulley*. The interesting thing about *IxFIZZ* is that they use a block-based fuzzer that requires a script to do its fuzzing process, but this script is generated in an automated way based on the description of the protocol.
After all the testing is done, we are left with an immense number of logs to process and it is very difficult to analyze logs and to try and keep a mental map of what's happening in that specific log or what was the state of the device when the a certain call was made. Due to these limitations the model inference can be used to generate the internal model from the logs and give a better understanding of the test results.

Paid Fuzzers

There are also some paid fuzzers on the market. The most known paid solution is the Peach fuzzer\(^7\). Peach is a smart fuzzer that combines generational based and mutational based fuzzing. Peach makes use of what they call peach pits, these are templates of different services (health care pit pack, industrial control system pit pack, internet-of-things pit pack and many others) that were previously used and in constant improvement. Some features of Peach are extensibility to keep up with the scale and complexity of the system, advanced monitoring capabilities, improved components for logging and publishing, robust support administered by security specialists, among many others.

Another paid solution specialized in testing APIs is SoapUI Pro\(^8\), which is a testing tool for REST APIs, SOAP APIs and other popular APIs. Some of the main features are creating tests directly from Open API Specification and other popular API description formats, functionality tests to prove that the API does what is supposed to do and returns the data it’s expected to return, create API test scenarios by providing design tools that use point-and-click shortcuts to simplify advanced functionality down to a few clicks, the use of realistic dynamic data to simulate user behavior and test boundary conditions, security scans and automated testing.

2.3 Model Inference

Understanding the system behavior can be a difficult task for developers that try to make changes to code which was created by someone else, or for developers that try to fix some bad behavior of a system, or for testers that try to produce tests and require some knowledge of the system (white-box) to understand a protocol interaction or even to infer some invariants/specifications behind the system.

The best way to deal with the problems mentioned above is to analyze the model of the system. It holds all right/valid transitions between states, all the existing states in the system and also the correct workflow that the system must take to have a normal/correct behavior. So, if a model of the system can help us to have a greater understanding of that same system and also help us to verify if the logic behind it is correct, it would be expected that every single program/system would have a model. So why don’t we see all the systems and programs with their corresponding model that are so helpful “outside” of the development phase to understand the “internal” work of the system? From what we could understand there are two main reasons, the first is quite simple to understand, to produce an accurate model of a complex system it requires a lot of effort and coordination between all the people involved in the project,

\(^7\)https://www.peach.tech/ on 12/14/2017
\(^8\)https://smartbear.com/product/ready-api/soapui/overview/ on 12/14/2017
so in many cases there is no time to do it and there is also the problem of updating the model every time the system has a major update or a new feature. The second reason is related with a security concept that many companies adopt - “security through obscurity”, this means that if no one knows how the system works, the vulnerabilities associated with the system are a lot harder to find and for this same reason many companies don’t disclose the specifications of their programs or API.

In order to overcome these problems a solution is the automatic inference of system models, in which the inferred model algorithms rely mostly on execution logs to infer their models. This model inference tools have a major limitation with the way that these models are built, because if the execution logs fail to cover some aspect of the system, it will not be represented on the model, also the problem of inferring a model is computable but is NP-complete. We will now make a brief description about some tools that infer models from execution traces. One point to take into account is that the base for modern model-inference algorithms is an algorithm called $K$-Tails[22], which is a state-merging algorithm that takes a log and a $k$ parameter. The logs are represented as a DFA (Deterministic Finite Automata) composed of linear sub-DFAs, one linear DFA per trace, that are arranged in a parallel fashion with a single initial and end state common to all sub-DFAs. The $k$ parameter represents the size and generality of the inferred model, a smaller $k$ represents a more merged states and produces a more solid model, while a bigger $k$ restricts the state equivalence. The idea is, if two executions have a similar $k$-long sequences of observed events, then those points most likely represent the same program state. The process only stops when all equivalent points are merged.

**CSight**

$CSight$[23] is a tool to infer a CFSM (Communication Finite State Machine) model from execution logs, this tool receives as an input a set of execution logs, a set of use-defined regular expressions and a set of channel definitions. The input log must contain error-free executions, the regular expressions are required to know how to parse the log, for example to know which part of the log corresponds to the timestamp, sender, receiver, events, etc. $CSight$ works in three stages, the first stage corresponds to the temporal property mining, meaning that $CSight$ mines properties or invariants from the logs, the invariants that $CSight$ can mine are “always followed”, “never followed by” and “always precedes”. The second stage is the creation of the initial model, this initial model tries to generalize the log executions allowing some executions to violate the invariants mined in the stage 1. Finally the third stage refines the model in order to satisfy all the mined invariants in stage 1, $CSight$ operates by starting from the initial model and changing this model until all the invariants are satisfied, in other words whenever a counter-example exists (an execution that violates the model), the model is refined in order to eliminate that counter-example. A general workflow of $CSight$ can be seen in the figure 2.12.
Synoptic Synoptic[24] is another tool that processes the logs from the system executions similar to CSight. Synoptic receives as input a set of logs and a set of regular expressions to parse the logs and extracts from the logs triplets containing the trace identifier, the timestamp and an event type, since Synoptic requires that the event instances in a trace to be totally ordered. Like CSight, Synoptic mines the same kind of temporal invariants from the logs, “always followed”, “never followed by” and “always precedes”. Synoptic use the BisimH algorithm to generate the model from the logs, where the objective of this algorithm is to produce a graph that satisfies all the invariants. The figure 2.13 represents a set of logs, a set of regular expressions and the output as a model.
InvariMint

InvariMint[25] is an approach to specify model inference algorithms declaratively. Under the experiments performed on InvariMint a conclusion was reached that led to new insights and a better understanding of existing algorithms, simplifying the creation of new ones by including extensions of previous algorithms. InvariMint presents two key features, the first one explicitly specifies the types of properties that will be enforced in the final model and second decouples the mechanism of property mining from property specification. InvariMint takes as input log traces and outputs a model of those traces. The objective of this tool is not to create models, but to provide a common language for expressing, or specifying model inference algorithms. If we specify two different algorithms with same language, we can have a better understanding of one versus the other, allowing us to combine and even extend these algorithms. When InvariMint was applied to Synoptic it solved two problems the first one was the non-determinism, meaning that if a user changes the input log and Synoptic produces a different model, the user does not know if the difference in the log explains the changes in the returned model. The second one was the performance of the inference algorithm, meaning that the current algorithm of Synoptic maintains all the parsed log traces in memory.
Perfume

*Perfume* [26] is a model inference algorithm that extends *Synoptic*, in a manner to take into account the resource usage (CPU resource usage and other resources) of the system in which logs are produced. *Perfume* infers the model directly from the logs with the requirement that the user must provide a set of regular expressions. A good feature of this inference algorithm is that it is predictive, it observes the executions and predicts unobserved combinations of behavior that is likely to represent possible system executions. The way that Perfume infer its model is the same as the others.

In the figure 2.14 we can see a comparison of the models produced with same input.

![Figure 2.14: Different models comparison. Image from [26]](image)
Chapter 3

The Testing Tool

In order to test IoT devices through their API’s we decided to develop a new tool. For this tool we wanted something different from what the existing solutions do, but at the same time something that was capable of producing good and new results, from the perspective of discovering problems in the software behind the IoT devices. Some of the established objectives for this tool were: it had to be easy to use, the testing process should be automated, it should be open source, and at the end it should produce a model. The model produced has the objective of helping the user to understand and recreate the error.

The testing process of this new testing tool will be based on two methodologies. The first methodology is fuzzing, which will be applied to the payloads and the *urls* of the API methods, to guarantee that nothing that is outside of the parameters stated in the API specification is accepted by the IoT device. This situations may range from inputs that are outside of the parameters described, special characters to the software in general, numbers in the place of strings and vice-versa, etc. For the fuzzing process of our tool we use three different tools, two are open source while the last was developed by us. The first tool is *Radamsa*, that is a specialized fuzzing program that receives a seed (it can be a normal/valid value) and returns the fuzzed seed. This program is useful to do some general fuzzing on the API method parameters, it allows us to generate random values to verify if something that is outside of the valid values is accepted. The second tool is *Pyjfuzz*, that is specialized in fuzzing payloads in the JSON format. This is helpful, because not only it allows to fuzz the parameters in the payload but it also fuzzes the structure of the JSON, it allows to verify if a sent payload that has an invalid structure for JSON is rejected or accepted by the IoT device. The reason why we used two tools for the fuzzing (the *Radamsa* and the *Pyjfuzz*) is that the *Pyjfuzz* is ideal to target the structure of a JSON and to produce some general attacks, however we were not happy with the way the random results were produced in order to test problems like *overflows*. For that reason we added the tool *Radamsa* to the mix to do specific fuzzing to the parameters of the payload. The last program was developed by us and it has the objective of testing the limits of the parameters in the payload, based on information provided by the specification. This feature of testing the limits of the payload cannot be found on the tool *Radamsa* or the tool *Pyjfuzz*.

The second methodology consists of using sequences of method calls to put the IoT device in different internal states. Our intention is to check if certain internal states can have an effect in accepting
or rejecting API calls that have invalid parameters. These internal states can influence the API calls in two ways. The first is that a specific parameter in the internal state of the IoT device, may be required to have a certain value in order for the API call to be accepted. The second is a quantitative internal state, which means that the internal state has an internal variable that is dynamic, and only when this variable meets some kind of criteria is the API call accepted (for example counter needs to be below a certain value or above a certain value). If these dependencies are not properly checked and the API calls are accepted regardless of the fact of the device being in the correct state or not, then a behavior which was not foreseen may happen.

Before we can run the tool, it is necessary to prepare and configure three JSON files. One to configure the tool running parameters to the user preferences, another to detail the API of the IoT device, and the last is to inform our tool of how to put the IoT device in the initial state. These files will be crucial for the tool to know how to interact with the IoT device using the correspondent API, to set up the initial state before every test and also to allow the user to set up some parameters that may help the tool in the fuzzing process, like the number of requests that the IoT device accepts per minute.

The general architecture of this tool is composed by seven modules. The first module is the Json-reader, that is responsible for parsing the JSON files, in order to extract some vital information for the operation of the tool, for example the API methods (name of the methods, input parameters, ranges of the inputs, type of the inputs, etc) to be called in the testing phase. The second module is the Fuzzer, that is responsible for all the testing process of the IoT, using tools and functions to generate the fuzzing of the payload and the url in order to test in the IoT device. The third module is the Caller, that is responsible for all the communications between the program and the IoT device and also for resetting the device to the initial state. The fourth module is the Logger, that is responsible for logging the response of the IoT device, and filtering the responses that are interesting and discarding the ones that are not important for the testing context. The fifth module is the Sequence Producer, as the name implies this module is responsible for building the sequence that will be used by the tool to test the IoT device. The sixth module is the Model Producer, that will use some specific logs to produce the model. The last module is the main module that runs the programs and orchestrates all the previous modules.
3.1 SET-UP phase

Before we can start using the tool, there are three JSON files that the tool requires in order to function. The first one is the Configuration JSON file that indicates to the tool some working parameters. The second is the Specification API JSON file that indicates what are the methods present in the API of the device, what is the security token for that API, how to construct the payloads for the API calls, what are the valid value ranges for the parameters on those payloads, etc. The third is the Reset API JSON file that keeps all the information in order to set the IoT device to its initial state.

3.1.1 Configuration JSON

The configuration JSON file exists to configure some operation parameters of the tool. These operation parameters are useful for the tool to know where the other two files are located and also to define some internal values related to the execution of the tool. We will now explain in detail each variable in the configuration JSON file, its purpose and in the Figure 3.1 we present the fields of the file.

![Configuration Json](image)

Figure 3.1: Fields of the configuration Json file

```json
{
  "file":"Joins/device.json",
  "request_times":"no limit",
  "dfs_level":2,
  "starting_method":"put /lights/:light_id/state",
  "first_set_up":"put /lights/:light_id/state",
  "method_type":"all",
  "reset_method":"resetdevice .json",
  "skip_fuzzing":"no",
  "total_number_of_requests":"no limit",
  "get_variables":":device",
  "produce_model_on_interrupt":"no"
}
```
File

The File variable has the type String and exists to indicate to the tool the path to the Specification API JSON file, that contains the specification of the API. This was built as a configuration variable, to give the user the liberty to decide where this file should be placed. This variable is then used by the module Jsonreader to load all the relevant information of the API to the tool.

Request_times

The request_times variable has the type String and its default values is “no limit”. It is an internal variable that informs the tool of how many calls can be made per minute to the IoT device. This helps the tool to avoid problems with IoT devices that limit the number of calls that can be made per minute. With this variable whenever the tool reaches that that limit in less than a minute, it will wait for the completion of that time in order to continue with the testing and by doing so it will avoid an error that would halt the testing process. The value “no limit” is used when the user wants to disable this feature.

Dfs_level

The Dfs_level variable has the type Integer and its an internal variable that notifies the tool about the level needed for the DFS algorithm. This variable is used by the module Sequence Producer. Before the module begins to produce the sequences this variable is used to determine how many levels will the DFS tree have. This was set as a variable to give the user flexibility to increase the number of levels in the DFS tree for longer sequences resulting in a more detailed testing or to decrease the number of levels in the DFS tree for a faster run of the tool, resulting in shorter sequences. In the limit the sequences are such that every method is called one time.

Starting_method

The Starting_method variable has the type String and its a variable that indicates to the tool the first method to be called in the sequences. The idea is to give the user the possibility to choose the first method of the sequences, this can be used in a debug situation to force the tool to test a specific method in the beginning of test procedure. It doesn’t mean that there will be no sequences that start with other methods, its just that the first produced sequences produces will have that method in the beginning of their sequences.

First_setup

The first_setup variable has the type String and it has the goal of setting the initial state of the IoT device before any testing procedures start. The variable receives a list of methods (it can be more than one, if its necessary to set up the state of groups, scenes, colors, etc.), and this list of methods indicates to the tool which methods to call from the Reset API JSON file (that contains all the methods with their
respective payloads, that can be called to put the device on the \textit{initial state} so that the \textit{initial state} is achieved on the IoT device before any testing starts.

\textbf{Method\textunderscore type}

The \textit{method\_type} variable has the type \textit{String} and it works as a filter, meaning that every time a sequence is generated, this variable informs the tool which methods can be used for the sequences, based on their operation. This variable can receive as input the following values: \textit{all} (if all the methods can be present in the sequences), \textit{get} (only methods that have the \textit{get} operation may be present), \textit{put} (only methods that have the \textit{put} operation will be present), \textit{post} (only methods that have the \textit{post} operation may be present), \textit{delete} (only methods that have the \textit{delete} operation may be present) or combinations of the values that were just described.

\textbf{Reset\_method}

The \textit{Reset\_method} variable has the type \textit{String} and it exists to inform the tool where is the file with the methods and payloads necessary to put the IoT device into the \textit{initial state}. Before the tool starts to run and after every time a call is accepted, it is necessary to put the IoT device back in its initial state. This \textit{Reset API} JSON file has all the information necessary for that to happen, it has the paths of the methods (or the urls), the operations and the payloads with values.

\textbf{Skip\_fuzzing}

The \textit{Skip\_fuzzing} variable has the type \textit{String} and its a simple flag to inform if it is necessary to go through the testing process. This can be useful in situations where the logs already exist or were imported and the user only wants to produce the model.

\textbf{Total\_number\_of\_requests}

The \textit{Total\_number\_of\_requests} variable has the type \textit{String} and it is used to set a maximum number of requests that can be performed. This variable accepts a number or the value “\textit{no limit}” (to disable this feature). In the cases that a number is set, every time a call is made to the IoT device it will decrement the value on this variable, when it reaches zero the tool will stop the testing process. The idea behind this variable is to give the user the ability to limit/control the number of calls made to the IoT device. It can also be applied in situations when there is no limit for the number of calls per minute but there is a limit for the number of calls per day.

\textbf{Get\_variables}

The \textit{Get\_variables} variable has the type \textit{String} and it is an internal flag for the tool. This exists to let the tool know how it should deal with the responses of the devices, or if the payload for that device can be empty or not, among other small changes that allow the tool to perform a better testing to the device.
Produce_model_on_interrupt

The `Produce_model_on_interrupt` variable has the type `String` and it is a flag that informs to produce the model if the tool is interrupted/aborted in the middle of the testing process. This variable gives some flexibility to the user when he needs to interrupt the tests and if the tool finds any problem the model will be produced. Just to clarify, no matter if the tool is interrupted or not the logs are always kept and the model produced, but every time the tool is re-run the previous logs and model are deleted.

### 3.1.2 Specification API JSON

The *specification API JSON* file supplies the tool with some information about how to interact with the API of the IoT device. This information can range from, what are the methods to be used by the tool to test the API, how to construct the payloads to test those methods, what are the valid ranges that the parameters can take to be accepted by the IoT device, etc. We will now describe the various fields that can be used in this file to describe the API. All the variables on the *url* are composed by the character “::” followed by the name of the variable. In the following Figure 3.2, we can see the hierarchical diagram of this file.

![Hierarchical diagram for the Specification API Json file](image)

A detailed description of all the components present on this file can be found in the “Appendix Specification API JSON”, while in this section we will only describe the more important components.
Figure 3.3: Example of Specification API JSON file

Header

The header is where all the information necessary for the parameter header in the call is described. This can be empty, or it stores security credentials or security tokens. An example can be seen in the Figure 3.3.

Basepath

The basepath as the name implies is the part of the url that is common to all calls, usually this never changes since the methods are in the last part of the url, so this variable abstracts the common part. Since this base path can also have variables, their value is indicated in the path.parameters of methods
description. An example of a basepath with variable is “http://domain.com/api/:token”, in this example the variables are the token.

Paths

The paths is the section of the file that informs the tool what are the methods that exist on the API, how they are built, what are the ranges for the parameters, what is the operation, etc. The next dependency after paths are the methods that are to be used by the tool and then comes the correspondent operation. For some methods it is possible to have more than one operation. An example can be seen in the Figure 3.3.

Parameters

The parameters section is where all the information about the payloads of the calls and the variables of the path calls are described. This section is composed by the following two sections the path.parameters and the body.parameters/query.parameters, since a method sometimes has a query on the payload, the method can either have a body.parameters or query.parameters depending on the situation. An example can be seen in the Figure 3.3.

Monitor

The monitor section has the objective of informing the tool of the method and operation that needs to be used, in order to retrieve the state of the IoT device that allows the tool to compare the changes caused by the call. This helps the tool to know how to collect the state before and after the call is made.

Setup

Before detailing about what goes on inside the parameters section, we will discuss the setup section. The setup section has the function of informing the tool that the described method is necessary to populate some variables related to the IoT device. It also has a list of methods that are necessary to call in order to put the IoT back in its initial state, this for the cases that the call of the method is accepted. It also has one action parameter (different from the action that can be found in the sections following the action section) that can take the values create or delete, to help the tool to track the goal of the method. Sometimes when putting the IoT device back into its initial state it may happen that a resource needs to be created, like a new group and the ideal scenario is to create the one that was just deleted.

Type

The type variable has the objective of indicating to the tool the type of the variable. The type can be string, integer, float, boolean, array, object, etc. This is then used by the tool to check if the parameter in the fuzzed payload has the correct type, and if it has not it implies that the payload shouldn’t be accepted. If it is accepted something went wrong. It is also used in other steps during the tool execution.
This variable can be found in all the positions that API variables can be described, path.parameters, body.parameters and query.parameters. An example can be seen in the Figure 3.3.

Requirement

The Requirement parameter informs the tool, if the variable always needs to be present in the payload of this method or if it is optional. The parameter can have two values, the first is optional, which will mean that the variable may or may not be present on the payload (random choice) and required. Required as the name implies the variable has always to be present in the payload of this call. This was developed to give the tool the opportunity of constructing different sets of payloads, but at same time have the confirmation that the required items for the call to be accepted are always present. An example can be seen in the Figure 3.3.

Range

The range parameter gives the possibilities for the values of the variable. The range parameter can take one of three set of values. It can take the no info value, for a specific value like a name or a device id. It can take a list of strings for different specific values that can be used by the parameter. And it can take an interval of valid values that the parameter may hold. An example can be seen in the Figure 3.3.

Example

The Example section is the section in control for passing information to the tool, in order for the tool to build a valid value for the variable. The following sections are the available sections in the Example section. An example can be seen in the Figure 3.3.

Value

The value parameter is used when the user has absolute certainty of what value is needed to be used in that parameter. For instance, if an operation requires variables username and a password, then the value of these variables is fixed and given by the system. In this case the parameter value in the example section will inform to the tool of the specific value of this variable.

Choice

The choice parameter is a list of all the valid choices that the parameter can take. This gives the tool the possibility of choosing at random one of the values of this list. This parameter is used usually when the variable takes boolean values or different string choices (for example “enabled” or “disabled”). An example can be seen in the Figure 3.3.
3.1.3 Reset API JSON

The *reset API JSON* is a file that has all the methods that the tool may need to put the IoT device back into the *initial state*. Whenever it is necessary to put the IoT back into the *initial state*, the tool needs to know what are the existing methods and the correspondent payloads. These situations are usually before the testing phase and every time a call is accepted, resulting in a change of the IoT device internal state. The description of this JSON file is similar to the specification API JSON. In the Figure 3.4 we can see a hierarchical diagram for this file.

![Hierarchical diagram for the Reset API Json file](image)

Figure 3.4: Hierarchical diagram for the Reset API Json file
3.2 Tool Architecture

This section aims to discuss the architecture of the tool. We will discuss the different modules that compose the tool, why we decided to develop the modules in a certain way, what were the main objectives, what are the most important functions of each module. We will start by describing the modules in the following order, first the \textit{jsonreader}, then the \textit{sequence_producer}, the \textit{fuzzer}, the \textit{caller}, the \textit{logger}, the \textit{model_producer}, and last the \textit{main}.

The tool was developed in \textit{python 3} with the help of its libraries. It uses the following tools, the \textit{Pyjfuzz}, the \textit{Radamsa} and the \textit{Synoptic}. During this section, we will mention why we decided to use these programs, what do they contribute and what obstacles we encountered by using these tools.

In the Figure 3.5 we can see the diagram of the architecture. How the tool interacts with all its components: the user, the IoT device, the outputs it produces, the logs and the model.

![Diagram of the architecture](image)

**Figure 3.5: Architecture of the tool**
3.2.1 Jsonreader

The Jsonreader module is the first module to be used by the tool. In a general way this module is responsible for parsing the three JSON files (the configuration JSON, the specification API JSON and the reset API JSON), in order for the tool to have all the information necessary about the IoT device.

The parsing part is composed of functions that will be able to read a JSON file and interpret the information described in it. The first important function, is a function that receives a path to the JSON file and then loads all its contents to a variable. Then we have several functions that use the variable with all the contents of the file to extract the following informations, the base path of the IoT device API, the header that may contain important information in order for the tool to be able to make the calls, like security tokens, usernames, etc. All the methods that the tool is able of call to the IoT device API, the operations that are related to the paths and the parameters for each call (this parameter is composed by the path parameters and body parameters or query parameters) of the IoT device API. These are some of the basic functions that allow the tool to extract the information necessary in order to produce and execute the calls to the IoT device.

The other important function in this module, is the variable_retriever function. The variable_retriever function has the goal of informing the tool about the variables that are used in this specific IoT device API. This is important because during the testing phase and depending on the methods, it is possible to change the value of these ids for certain functionalities. For example in some systems, every time a device is added it generates a new id in an incremental way. So if we had a device with id 1 and we delete it and add it again, it will get id 2. From this example we can see that if the tool is not able to keep up with the progression of the values during the testing phase it would be impossible to reach the end of the testing process.

These variables are used by the tool to store unique ids, the name of the variable of these ids is then used on the path of the call for the tool to know where to replace it. For example the tool is required to know the id of the device, if the call to the API interacts with that specific device. The same can be said if the call needs the group id, or the scene id, or other unique identifier that may be necessary to access a certain resource. So, in order for the tool to know what are the names for these variables and how to discover their values, the function variable_retriever exists. What it does is to go to the setup sections of all the methods (in the specification API JSON file) to verify what is the value on the variable_retriever. If the value is false then it means that the method does not return the id of a variable. If it is true then that section also has another field called variable_id (an example can be seen in the Figure 3.6). It is telling the tool, that by calling that method the IoT device will return the id or a value to some variable and the name of the variable (given by the parameter variable_id) associated to this returned value (that will be stored in the tool). So, with this function it is possible to know the name of the variable for the ids and what is the resource associated with the returned ids.
The sequence producer is the module responsible for producing all the sequences of methods, that will be used in the testing process of the tool. This module has two important functions: the new_paths function and the sequence_production function.

The new_paths function is the first function to be used in the module. The function has the objective of filtering the methods that are not relevant for that test to run. The function uses the field method_type in the configuration JSON file. If the value of the variable is all then nothing is filtered and all the methods described are used if the value of the field is get then only the methods that have the get operation will be used, if the value of the variable is "post, put" then only the methods that have the operation of post and put will be used.

The other important function and perhaps the most important function of this module is the sequence_production function, which is in charge of producing the sequences to be used in the test process of the tool. This function receives the filtered methods from the previous function, and it also receives the starting method given by the field starting_method in the configuration JSON file, and the DFS level, that is given by the field dfs_level in the configuration JSON file.

The dfs_level contains the number of levels that the sequences are required to have. The user can choose to have longer and more sequences or shorter and less sequences by regulating this parameter in the configuration JSON file. This function uses the DFS (Depth-first search) algorithm to produce its sequences. The algorithm will have as a root node all the methods that are available for this function, but if a starting_method is given it will be the priority on the list. The algorithm will have as many levels as the ones described on the variable dfs_level. The objective is to have almost all combinations of method calls (as sequences) that are possible inside of number of the levels for the sequences.
3.2.3 Fuzzer

The fuzzer is one of the most important modules in this tool. The main objective behind this module, is to use several functions and external tools, to fuzz a valid payload to test the IoT device. This module contains a lot of small functions, so we will only describe the most important functions of the module, and we will also describe what external tools this module uses.

The two tools that are used in this module are the Radamsa and the PyjFuzz.

Radamsa is a side-product of OUSPG’s Protos Genome Project. Radamsa is a specialized program that fuzzes the input received in a string format, it is a dumb black-box fuzzer. The idea behind this fuzzer is to generate random inputs, without any knowledge of the program nor the format of the data used by the program. Radamsa was developed with the idea of being a general purpose fuzzer, in other words the objective was to build a fuzzer that would discover problems, independently of the data used by the program being tested. The way it works is, it receives an input as a seed/base for the fuzzing and then it outputs the fuzzed result. Radamsa accomplishes its fuzzing process by having several heuristics and change patterns applied to the input received that can produce several results, it can have one small change, or a lot of distinct changes, or bit flips or even more complex changes. An example of the fuzzing process of the Radamsa tool, is given by the input False it can output False, or Fa1se, or Falseeeeeeeeeeeeeee. The same can happen with numbers, given the input 1 it can output 11111111111111111111111111111111, or -2334564 or 0. The probability of the tool producing any of these outputs is completely random.

PyjFuzz is another fuzzing tool, but PyjFuzz specializes in fuzzing JSON payloads. This is useful for our tool because in the REST API the payloads are sent in a JSON format. PyjFuzz was developed by Daniele Linguaglossa and it was based on the fuzzer Radamsa. PyjFuzz works by receiving an input (in the JSON format) as a seed/base and outputs the input fuzzed (also in JSON format). PyjFuzz also has a table technique, this table has the objective of molding the fuzzing, in order to test the techniques in the table. The table has the following techniques. Polyglot in this case means, that it executable within various injection contexts in its raw form.

1. XSS injection (Polyglot)
2. SQL injection (Polyglot)
3. LFI attack
4. SQL injection polyglot (2)
5. XSS injection (Polyglot) (2)
6. RCE injection (Polyglot)
7. LFI attack (2)
8. Data URI attack
9. LFI and HREF attack
10. Header injection

11. RCE injection (Polyglot) (2)

12. Generic template injection

13. Flask template injection

14. Random character attack

**Why do we use two fuzzing tools on our tool**

From the various experiments done during the development of this tool we noticed that the tool *PyjFuzz*, was good to test situations when we wanted to check if the IoT device accepted a payload with a bad JSON structure, but it didn’t match our criteria in terms of creating similar incorrect payloads to the ones given and payloads that could produce an overflow in the IoT device.

In order to solve this problem we decided to introduce another fuzzing tool *Radamsa*. This tool can create good payloads to test the boundaries of a possible overflow and also can create a more diverse set of fuzzed parameters, based on the input given. Of course it has the downside of being incapable of working over a JSON payload (since the input needs to be a string), it doesn’t use any of techniques mentioned in the list above and it doesn’t test the JSON structure.

A problem that both the Radamsa program and the PyjFuzz program couldn’t solve is the boundary test for the parameters in the payload that are integers or lists that require a specific number of elements. So to solve that problem we developed the *off_by_one* function test the upper boundaries and the lower boundaries of the parameters present in the payload. As the name implies it will try to add one or remove one depending of the situation.

**Fuzzing process**

All the testing to each method is done in fifty calls, the first thirty calls are to test the method only fuzzing the payload. The first ten using the tool *PyjFuzz*, the next ten using the tool *Radamsa*, and the ten after using the function *off_by_one*. The last twenty calls are to test the *url* of the method using fuzzing. The first ten of the twenty calls are to fuzz using the tool *Radamsa*, and the last ten using the tool *PyjFuzz*. The function *off_by_one* is not used to fuzz the *url* because it makes no sense to use it there since there are no limits so to speak.

*Fuzzing_pyjfuzz* and *Fuzzing_radamsa* are the functions in charge of calling the tools outside of our tool. The *pyjfuzz* as previously said only receives as input a JSON data structure and outputs a JSON data structure, this is fine since the payload of a call to a REST API is in JSON format. On the other hand in order to send inputs to the *Radamsa* tool and to retrieve them, in both situations need to be in a string format (the fact that the input is a string has no impact in the fuzzing process), but when the payload is built in a JSON format these strings need to be converted to their original type. To achieve this objective we use the function *conversion_func*, which receives the string output from the *Radamsa*
tool and the type that the parameter should be, then converts the string to the appropriate type, so that it may be used in the construction of the JSON payload.

Now we will start describing and discussing the functions. The functions in question are the parser_url, change_current_path, change_variable_in_body, and the change_base_path. We grouped these three functions together because all three have the same objective, but act on different places. So the idea is to locate the variables known to the tool in the url, or in a parameter of the payload about to be sent, or in the base path. This is possible because all the variables are prefixed with a ":" before the name of the variable.

The parser_url function has the objective of identifying a variable in the url and return the name of the variable being used. For example let’s say that the function receives as input the /lights/:light\_id, then it would return the variable light\_id.

The change_variable_in_body function has the objective of detecting and replacing the parameters in the payload, that needs a value used by these variables.

Let us take for example a method that aggregates devices in a group and for that one of the parameters of the payload is the parameter ligths, that corresponds to the list of the devices ids that would be aggregated in this new group. So in the specification API JSON file the value of the parameter ligths would be [\_id] with the type array. This informs the tool that this parameter is a list of the values stored in the variable id, the values are chosen in an arbitrary way.

The function change_base_path has the objective of identifying the variables in the base path and replacing them by the known values (for cases where the base path has such variables).

The change_current_path function is the more complex function of this group. The function is in charge of replacing the variables in the current path and the current path is everything that comes after the base path. This function also receives the name of the tool that is going to be used, for cases that are necessary to fuzz the values of the variables in the current path.
Function: off_by_one_fuzzing

```python
for parameter in payload.parameter:
    z in random(0,2)
    if parameter is required in the payload:
        payload.add(parameter);
        parameter_data=info_of(parameter);
    elif ( z < 2 ):
        payload.add(parameter);
        parameter_data=info_of(parameter);

    p,q,n in random(0,1)
    if (parameter IN payload AND type(parameter_data)==String):
        if (p<0.5 AND q<0.5):
            value=parameter_data.length + 1;
            payload.parameter.value=generate_string(value);
        elif (p<0.5 AND q>=0.5):
            value=parameter_data.length;
            payload.parameter.value=generate_string(value);
        elif (p>=0.5 AND q<0.5):
            value=1;
            payload.parameter.value=generate_string(value);
        else:
            value=0;
            payload.parameter.value=generate_string(value);
    else:
        for the cases that the parameter isn’t a string or a integer the value is generated normally with the data of the parameter
        payload.parameter.value = generate_value(parameter_data);

return payload;
```
The off_by_one_fuzzing function, is a function as previously mentioned to generate parameters that aim to test the boundaries stated by the specification, for those parameters in question.

This function starts by checking if the method in question will have a normal payload or a query in the payload, the difference is that if it is the latest then only one string will be provided in the payload. After the type of payload is discovered, the function will do the same action for each parameter. First it will check if that parameter is required to be in the payload, if it isn’t required then that parameter has two thirds of probability to be added to the payload. If the parameter is added to the payload (because it is required or because it is chosen by the probabilities), it will extract the information stated in the example section of the specification API JSON. If the parameter type is a string, the length field exists and the example section has the value field. After, there is a fifty percent chance of going to either the upper limit of the length value or to the lower limit. If the upper limit is chosen then it will have again a fifty percent chance of either choosing the upper limit value or the upper limit value plus one. Once the choice is made it will add the character “a” to the string until the chosen length is reached. If the lower limit is chosen since the tool is working with strings the only value that can be tested is the empty string. If the range parameter is in the example section of the parameter and the length field exists, then a choice will be made. Again it will be a fifty percent chance, between the limits for the choice being the ones stated on the on the field range or the ones stated on the field length (difference between the range and the length in the parameters with numbers, is that the range is the valid interval accepted by the IoT device for that parameter, while the length is the valid interval that the parameter can take, usually the length interval field is the same or bigger than the range interval field). Once this choice is made there will be another fifty percent choice between the upper limit and the lower limit, if the upper limit is chosen there will be yet another fifty percent choice between the final value being the upper limit value or the upper limit value plus one. If the lower limit is chosen there will be, again another fifty percent choice between the final value being the lower limit value or the lower limit value minus one. If any other option appears in the example section, the parameter for the payload will be constructed based on that information to build a valid parameter.
The body\_payload\_parser function is in charge of building valid payloads for the calls based on the information extracted from the specification API JSON file. It works on a similar way of the previous function. It will go through each of the parameters for the payload stated in the specification API JSON file for this call. It will first check if the parameter is a query or not, if it is a query the parameter will be built in a normal way and then converted to string with the necessary template. If it’s not then it will check if the parameter is required for it to be on the payload to be accepted, if yes it will be added. If it is optional then it will have three fourths of probability for this parameter to be added to the payload. Before the parameter is added to the payload it is necessary to build a valid value, so it will be generated with the information extracted from the example section for this parameter in the specification API JSON file. If the field choice is in the example section, then the parameter will take the value of one of the several values presented in the list of the field choice. If the field range is in the example section, then it will use the interval given in this field to chose a number integer or float (depending on the type of the parameter) that is inside of this given interval, and by doing so it will be a valid value that will be accepted by the IoT device. If the field template is in the example section, then will have another field called method which has the objective of referencing another method. This function will be called again with the objective of doing the payload for the reference method. The output of this parameter will be a completely valid payload to the method reference. This type of parameters also support another feature, for the cases that it is necessary an array of different payloads for the same call. In these cases the field times is also given, to state how many payloads will be need to generate. When the parameter is of the
The fuzzing function is the function in charge of producing the url (fuzzed or not) and the payload (fuzzed or not), with the help of the functions mentioned above. The first step is to know at what state of the method testing is the tool currently at, the function knows this through a variable received as an input, that tells the iteration of the method testing. Each time a call is made it adds one to the iteration.

If the iteration is below zero, then it means that it is a normal call and nothing needs to be fuzzed, this iteration is normally used to do valid requests in order to reach to the right position in the sequence for the tests to take place. This is what we call the control call, where no fuzzing tool or fuzzing function is used. If the iteration is between zero and nine, then it means that the body will be fuzzed. The url of the request will be a valid one (it doesn’t necessarily needs to be the same in each iteration), a valid

```python
return call;
```
payload of the method being tested will be requested to the function `body(payload, parser)` and then this same payload will be given to the tool `Pyj fuzz` in order to have the fuzzed version of that payload. The next step before this function returns the payload and the `url`, is to check if the payload despite being a fuzzed payload is still a valid one, in order to do that we call the function `check_if_payload_is_fuzzed` with the payload, it returns true if the payload is not a valid one or false if the payload is a valid one. If the iteration is between ten and nineteen, then it means that the body will be fuzzed. The same process as before will take place, but this time the tool used will use the `Radamsa` tool and at the end the payload will be checked if it is a valid one or not despite being fuzzed. If the iteration is between twenty and twenty-nine, it means that the body will be fuzzed. Yet again the same process as before will happen but this time the function used is `off_by_one` and at the end the payload will be checked to see if it is a valid one or not. If the iteration is between thirty and thirty-nine, it means that the `url` will be fuzzed and the payload will be a valid one for the method being tested. The function `change_current_path` will be used in order to fuzz the current path of the `url` (the base path is always a valid one and it is never fuzzed) with the tool `Radamsa`. The function `body(payload, parser)` will be used to generate a valid payload for the method. At the end the function `check_if_payload_is_fuzzed` will be called in order to check if the `url` (despite being fuzzed) is a valid `url`. If the iteration is between forty and forty-nine, then it means that the `url` will be fuzzed and the payload will be a valid one for the method being tested, the procedure is the same as before, but the tool used to fuzz the current path of the `url` is the `Pyj fuzz`. At the end this function returns several values, but the important ones are the `url` that will be used in the call, the payload that will be used in the body of the call and the `fuzz_flag` variable to indicate if the payload is a valid one or not. This variable is useful because if the call is accepted and the variable is true then it means that something went wrong on the IoT device. On the other hand we can discard the cases where the payload is accepted and the variable is false, or the request is not accepted despite the fact the `fuzz_flag` variable being false or true.

Function: `check_if_payload_is_fuzzed`

```python
if (testing_target == 'disable') {
    return false;
} else if (testing_target == 'url') {
    variables_in_the_url = diff(call.url, data.url.current_path);
    for var in variables_in_the_url {
        if (var NOT IN stored_valid_values) {
            return true
        }
    }
    return false;
} else if (testing_target == 'body') {
    for parameter in call.payload {
        if (parameter_is_invalid(parameter, data.payload)) {
            return True;
        }
    }
    return False;
}
```

The function `check_if_payload_is_fuzzed` is in charge of checking if the fuzzed payload is still a valid payload or not. To be a valid payload every parameter in the payload must be inside the valid values stated by the specification of the API, that is described in the `specification API JSON file`. The function starts by seeing what is the target of the fuzzing in the method, the body or the `url`. If the `url` is the
target of the fuzzing this function also has access to the template of the current path used and by doing the “difference” (between the template and the current path used with the values) it can figure what is the value used to replace the variables in the template of the current path. Then all that is left to do, is to check if that value is on the values stored by the tool for the variables present in template of the current path. If the value is present then the function returns False, which means that nothing is different from valid values. If the value is not present then the function returns True, which means that at least one value is not present in the values stored by the tool for the variables in question. If the body of the request was the target of the fuzzing process, then each parameter of the fuzzed payload will be checked to confirm its values. When a parameter is selected this function access to the information that was extracted from the specification API JSON file about this parameter on the correspondent method. Then with the information extracted all that is left to do, is to check if the value is inside of the extracted valid values. For example let’s say that there is a parameter power that has two valid options on and off. If the parameter takes one of these two values the function returns False, if it is anything else besides these two values the function returns True. It is a little bit more complex when working with parameters that are payloads for other methods or have the objective of making other method calls, but the function is still able to show if those payloads in the parameters have invalid values or not despite being fuzzed. The importance of this function is for the cases that a call is accepted. If that call had a fuzzed url or a fuzzed body, then it is an indication that something isn’t working right. This function indicates to the tool if what was fuzzed is still valid or not, then the tool with this information and with the information of the acceptance of the call, can infer if something went wrong, based on the information extracted from the specification API JSON file.

3.2.4 Caller

The module caller has the following goals: to do the calls to the IoT device, to revert the IoT device back to its initial state once a call is accepted, to get the state of the IoT device before a method can begin its testing phase, to compare the state before and after the cases that the call is accepted, in order to confirm if what changed in the internal state was supposed to change or not. During this section we will discuss about the functions that allow this module to do what was described above.

Starting with the function get_values_for_state, as the name of the function implies it has the objective of fetching the current state of the IoT device. It works by making a get call with the method stated in the monitor section of the method being tested in the specification API JSON file. The number of methods stated in the monitor section should be only one, that returns certain informations about the resources in the IoT device, that once the call of the method being tested is accepted it may be possible to see the changes. This is important for two reasons, the first one is that we want to be able to “track” what was changed by doing that call. For example to detect situations where parameters on the internal state of the IoT device were modified without being the target of the call. The second reason, is that the state of the IoT before the method testing starts is not always the initial state. For example if we are in a sequence with more than one method and the method being tested is not the first one on the sequence,
then the state of the IoT device before the testing to the target method starts, is not the initial state. So in order to be able to do the comparison of what change in the state of the IoT device, it is necessary to have the updated information about the state, before the target method starts being tested.

The function populate_vars has the objective of fetching the values of the variables that are used through out the testing phase in the urls. This function will go through all the methods in the specification API JSON file and look at the setup section, if the method is a method that will return a value to a variable, then that method will be called and the returned value will be associated to the name of the variable that is also present in that same section. As mentioned before, these variables have the objective of producing a valid url, since the value of the variables can change depending on the calls that are made during the testing phase.

The first_reset_state is a function that has the objective of putting the IoT device in the initial state before any testing starts, since the device before the testing starts may not have all the components that the user considers to be part of the initial state. It uses the methods described on the reset API JSON file and the list of methods in the variable first_set_up on the configuration JSON file, so this function will call the methods on that list and it will use the information about those methods described on the reset API JSON file, the information is related to how to construct the urls and the payload for those calls/methods on the list. This function is only called one time and it is before the testing process begins. Once this function finishes we can guarantee that the device is on the initial state chosen by the user to initiate the testing phase.

The function call execution is in charge of making the calls to the IoT device. Every time there is a need to get or send something to the IoT device this is the function used. It is a simple function that makes use of the python “request” library to produce its requests, and is a bridge between the tool and the IoT device, since the requests are sent from the tool to the IoT device trough this function and the response is transmitted from the device to the tool through this function. Every request has a timeout of four seconds, meaning that after four second if the device didn’t respond it will jump to the next test.

The function revert_state is another important function of this module, this function is in charge of reverting the IoT device back to the initial state described in the file reset API JSON file. The idea behind this is, during the testing phase of a method, if a call with a payload that may or may not be fuzzed is accepted, then the internal state of the IoT device will change and we don’t want to keep testing the same method on top of that state for two reasons. The first reason is that it becomes impossible to compare states, since the collection of the state to compare is done prior to the testing of the method. If this was allowed to happen then we wouldn’t know what payload was responsible for the changes in the internal state of the IoT device and would become impossible to connect the payload to the resulted action. The second reason is that if a request with a payload that is not valid is accepted, we are working on top of a state that has a high probability of being an incorrect state and lead to problems. The way that this function works is by collecting the methods that allow the device to go back to the initial state if the method being tested is accepted. The collection of states is in the variable reverse in the setup section of the method being tested. Then it will extract the values of the variables (if they exist) from the url, so that the reverse calls can affect the same resource in the IoT device. After the values are
extracted from the url of the accepted call, the function will produce the payloads for the collection of methods, as stated on the file reset API JSON file. For the cases that the collection of methods have a url with variables, the values used will be the ones that were extracted from the url of the accepted call. An example of this is when the request creates a new device. In this case the tool needs to know the id of that device in order to later delete it. Another example is when the request alters something in the state of one resource and there are multiple resources. In this case the tool needs to know which resource was accessed in order to put it back in as initial state. There are also cases where no id is used on the url, but the IoT device returns the id as a value of the resource created, the tool also stores this value in order to delete it.

The reposition_of_method_calls function, is a function that has the objective of preparing the IoT device for sequences that have more than one method and the target method for the testing is not the first in the sequence. If we have a sequence with two methods, where the target for the testing is second method and there is a request during the tests that is accepted, the tool will automatically put the IoT device in the initial state. The problem is that to be able to test the second method in the sequence it is necessary to be in a state produced by a valid request brought by the first method in the sequence and not the initial state, so what this function does is after the function of revert_state is called, it generates a valid payload and url for the first method.

The compare_get_states function has the objective of every time that a call is accepted to compare the states, these states are composed of the one gathered by the function get_values_for_state that returns the state before the method starts being tested and the state right after the call was accepted. The idea is to see what is difference from one state to another and also use the payload to justify that changes. If something changed in the state and it isn’t in the payload or if a payload with an invalid parameter caused an unforeseen change, then something is wrong. This function returns the name of the parameters that were different, if the result of the comparison is different from zero (meaning that at least one difference was found) then a detail log will be produced about the problem found with the used information: the payload, the url, the sequence, the position in the sequence, the state before the request, the state after the request, etc.

To help do this comparison of the different states we developed the function compare_dictionares, as the name implies the function will compare both the dictionaries that possess the states of the IoT device prior to the request being accepted and after the request being accepted. The function will go through each of the parameters on both dictionaries and check the following situations. If for the same parameter both the values are equal and the payload doesn’t have that parameter then there is no problem; if they are equal and the payload has that parameter with the same value then also there is no problem; but if they are equal and the payload has the parameter with a different value, then the request failed to update the value and something must be wrong. If they are not equal and the payload doesn’t have that parameter, it may or may not be a problem, depending of the functionality of the IoT, but for the our tool it will be marked as a problem. If they are not equal and the payload has the parameter of the last state, then there was a successful update of the parameter and there is no problem.

There was a problem while developing this function when there was a need to compare the numeric
values from the state with the ones in the payload, because more often than not the values on the state were round or the decimals were round, which created situations where the value in the parameter payload was different from the state, but in fact it wasn’t. So to resolve this problem we first check what are the limits of the parameters, if it is between zero and one, we give a zero point two margin on the difference of values, meaning when we did the difference of the number in the state by the number in the payload and the difference couldn’t be higher then zero point two. When the limit was anything else (between zero and three hundred and sixty, or zero and one hundred, etc) then the difference couldn’t be higher than ten. By taking this solution we could see that only the abnormal cases were reported by the tool, while the rounding cases stooped being detected as errors.

3.2.5 Logger

The logger module has three main objectives. The first goal is to process the response from the IoT device, in order to check if it is a response that was accepted or not by the IoT. The second goal is to log all the calls done to the IoT device and the last goal is to keep a detailed log whenever a problem is discovered in the device. We will now give a detailed description of the functions responsible for these three objectives and how they make it possible to be achieved.

Process Response

Starting with the first objective we have the function text_response. There are multiple ways for an IoT device to report that an input was not correct. One is the response with a value outside of the status code 200 (this code usually means that the request was accepted by the IoT device), where there are already several status code responses defined for specific situations, like the status code 404 that represents that the resource does not exist, or the status code 422(unprocessable entity) with a message saying that some parameter is wrong or missing, etc. Another way to inform the user that the request was wrong, or that it had something that could not be processed by the IoT device, is to give the status code 200 as response, and to describe in the text field what the IoT device couldn’t process. Thus, this function looks not only for an error in the response code returned by the IoT device, but also searched for an error field in the text (if this is not empty). We show one such template in Figure 3.7.

```json
{
   "error": {
      "type": "<ID>",
      "address": "<resource/parameteraddress>",
      "description": "<description>"
   }
}
```

Figure 3.7: Template of an error message.
Log Activity to the IoT

The second goal is to log all the activity done to IoT device, which is done by the function $\text{total.log}$. This function uses the call number to keep an ordered record of events. It uses the type of call to record the action that it was done to the device. It uses the operation to indicate the operation used (get, put, etc). It assists the url to indicate what method was used and what was the url (with variables, if it had any). It uses the payload, for the cases where the call had a payload, with the values for the parameters of that payload. The last field is the status code returned by the IoT device. In the following image (Figure 3.8) we can see a small example of what was just described.

```
Call number: 1 Type of call: Populate the API variables
Operation: get
Url: https://domain.com/v1/scenes
Payload: {}
Response code: 200
-----------------------------------------
Call number: 2 Type of call: Populate the API variables
Operation: get
Url: https://domain.com/v1/lights/all
Payload: {}
Response code: 200
Call number: 3 Type of call: first reset
Operation: put
Url: https://domain.com/v1/lights/id/state
Payload: {
    'color': 'green', 'duration': 0, 'power': 'on', 'brightness': 1
}
Response code: 207
-----------------------------------------
Call number: 4 Type of call: Populate the API variables
Operation: get
Url: https://domain.com/v1/scenes
Payload: {}
Response code: 200
```

Figure 3.8: All logs

Detailed Log

The last goal is to keep a detailed log of the problems that were found in the IoT device. The function in charge, is the function $\text{log}$. Whenever this function receives an indication from the main module to log that sequence of calls, it means that something that shouldn’t be accepted was accepted or a problem was detected in the state of the IoT device. When the sequences have more than one method, this function also logs all the calls, prior to the call that triggered the problem. This gives all the information about the previous calls and the state that the IoT device was before it called the method where the problem was detected. We will now describe the structure of the detailed logs that is illustrated as an example in the Figure 3.9.

First the logs have the test number correspondent to that call, the time that the call was placed, the sequence that is being tested, the place in the sequence correspondent to the failed request, the operation, the used url (with the variables replaced by their values whenever it was the case), the returned code of the IoT device, the response text that represents the text field of the response of the IoT device, the fuzz type done to the IoT device either url fuzzing (to fuzz the url), body fuzzing (to fuzz the payload sent to the IoT device) or control fuzzing either (when no fuzzing is applied) It also has the tool that was used to do the fuzzing (Radamsa, Pyfuzz or off_by_one), the base payload (used in the
control calls or the payload used has the seed for the fuzzing), the **fuzzed payload** (is the fuzzing done to the base payload), the **process payload** (what the IoT device accepts in the payload as there are cases where the IoT device can reject some parameters in the payload), the **state before** the request is sent and the **state after** the request is accepted, and finally the **differences found between the state** before the request is sent and after the request is accepted.

Figure 3.9: Logs of Errors
3.2.6 Model producer

The model producer is the module responsible for generating the model, after the testing phase is completed and all the logs produced. The logs have the following format, “operation:method, sequence_number, testing_number” and an example of this is “put:/ligths/:id/state, 2, 129”. This log represents the call to the method “/ligths/:id/state”, with the operation “put”, it is on the second sequence, and its test number is 129. The value chosen for the test number is the value of the last call of that method, and for that reason we can have more than one log with the same test number. All the calls with the same test number are part of the same sequence. In Figure 3.10 we present an extended example of this type of logs.

![Logs used in the Model producer module](image)

Figure 3.10: Logs used in the Model producer module
The *Synoptic*[24] program as mention in the *Related Work* section, is a tool specialized in producing models from logs. We decided to use the tool *Synoptic* to produce the model and that was the reason why we structured the logs in the same way compatible with the tool *Synoptic*. The module has a function *model_inference_default*, where it makes a call to the *Synoptic* tool with these logs. The *Synoptic* requires as input the logs and a set of regular expressions for it to be able to parse the logs. Also the logs are required to be in an ordered way and to have a unique trace identifier (in the case of our tool this identifier is the sequence number) in order for it to be able to produce the model. Figure 3.11 is an example of the model of the *device A*.

![Figure 3.11: Model of the *device A*](image)

Evaluating the model produced from the logs that found problems in the device, we identified a weak spot, because it was not possible from looking at the model to recognize where the error happened for cases when the sequences had more than one method. This type of sequences (with more than one method) were represented in the model by one good call to a method where no problem was detected, followed by another call with a problem detected. It didn’t mean that the first call was free of any problem, it meant that the model represented all the methods present in the sequences where problems were detected. To try to solve this issue we tried different approaches and one solution was to create an individual model for each problem found, resulting in models like the Figure 3.12 (the problem was in the first node) or the Figure 3.13 (the problem was in the second node).

![Figure 3.12: Model of sequence with one method](image)
This approach told us exactly where the problem was found, it is always in the last node of the model, but it had a bigger problem compared with the previous solution. The models were produced in a much bigger number and most of them were repetitions of each other. We came to the conclusion that the initial approach together with the detailed log was easier for the user to identify where the problem was found and also got detailed information of what caused it.

Figure 3.13: Model of sequence with two methods
3.2.7 Main

We could say that the main module are the brains of the tool, because it connects and uses all the other modules in order to produce the tests and execute them on the IoT device. The main module is basically what fuels the tool and makes it possible to work. Through the description of this module, we can practically understand the normal operation of the tool, since we will start by describing from the beginning, how the tool initiates the test to the end how the tool produces the model of the IoT device.

Module: main

```plaintext
variables initialization ;
validation of the values inside the configuration JSON file;

specification_data=JSONreader.readjson(specification API JSON);
methods=JSONreader.paths(specification_data);
reset=JSONreader.readjson(Reset API JSON);
sequences=SequenceProducer.newpaths(methods,operations_allowed);
id_and_method_extractor=JSONreader.variable_retriever(specification_data);
for populate IN id_and_method_extractor{
  var_store[populate.id]=populate.method;
}
var_store=caller.populate_vars(var_store);
log files creation and initialization ;
Caller.first_reset(first_setup);
for s IN sequences{
  var_store=caller.populate_vars(var_store);
  for m IN s{
populating some variables of m with resource to the specification_data.paths like : current_path, operation, parameters, etc.
test_number_to_the_method=fuzzer.number_of_times_to_fuzz(current_path);
  if (monitor IN specification_data.m and m.operation!=get){
    state_before=caller.get_values_for_state(monitor);
  }
  for t IN test_number_to_the_method{
    call=fuzzer.fuzzing(specification_data.m,t);
    response=Caller.call_execution(call);
    if(response.accepted(response)){
      if (fuzz_target='url'){
        logger.log(call,state_before,response);
        logger.model_log(m);
      } else if (fuzz_target='body'){
        caller.compare_get_state(call,state_before,response,monitor)
      }
    }
    caller.revert_state(reset,call);
    caller.reposition_of_method_calls(t)
  }
}
model_producer.model_inference_default(model_log);
```

The module starts by initializing all the variables that will be used and it uses the function `read_json` of the module `jsonreader` to load all the data of the `configuration JSON` file to a variable. Then, it checks the values present in the file (`configuration JSON`) to make sure that they are according to the expected parameters. For example in the `starting_method` variable, the tool expects a value (that is a method) that is present in the pool of methods. Another example is the variable `reset_method` that is expected to have the path to a valid file (that would be the path to the file `reset API JSON`), so it checks if the value given is a valid path to a file. The last example is the variable `request_times`, that is expected to have the number of requests that the tool can make to the IoT device per minute, this value must be an integer and also it must be above zero.

After all the variables are initiated and have their imported values checked, the module produces the sequences that will be used in the testing process of the IoT device, but first, in order to produce
these sequences, it is necessary to extract all the methods that exist on the API of the IoT device. They are specified in the specification API JSON file. After we have all the methods the tool will filter them in the function new_paths of the module sequence_producer, with the help of the variable method_type described on the file configuration JSON. At the end of this function we have the pool of methods that the tool uses to test, so the next step is to take those methods and call the function sequence_production in the module sequence_producer, where it will find all the sequences necessary to test the IoT device. The next step is to figure what are the methods and the names of the variables that will contain the values of the identifiers of the groups, devices or other values that may be necessary. In order to discover these values the tool uses the function variable_retriever of the module jsonreader. The results are the names of the variables and the methods necessary to call in order to populate those variables. The next step is to make use of the module caller and function populate_vars to populate those variables. The variables and the correspondent values will be stored in the variable var_store that is accessed every time the tool requires one of those values. In the next phase, the tool makes sure that the folders to store the logs exist (if it doesn’t, they will be created) and proceed to initialize all the log files, so that it is possible to store the information in them. At last, before the fuzzing process starts, the tool is required to put the IoT device in the right state for the testing (the initial state). The tool uses the function first_reset of the module caller, this function will put the device in the initial state defined by the user in the file reset API JSON. With the information of this file, the tool is able to generate several calls with correspondent payloads that ultimately will leave the device in the initial state.

All the preparations are now complete so that the tool can proceed to the testing process, the variables were initialized with the right values, the sequences were produced, the logs folder and files were created and the IoT device is in the initial state. The testing phase starts with a cycle that will go through each sequence that was generated previously, this cycle starts by repopulating all the variables in the var_store, since it cannot guarantee that no value has changed as a result from a method call being accepted in the sequence being tested or in previous sequence. To guarantee that the tool has always the right values it calls the function populate_vars in the module caller. Then, it goes into another cycle that will test the methods in the sequence, this cycle has the objective of advancing in the methods of the sequence being tested. Here the tool starts by getting the values to the following variables, the current_path, the operation, the parameters that contain all the information about how to construct the payload and the url. Then, it will also retrieve the number of times that the method in question will be fuzzed. It can be twenty times if the method only has the variables in the url and no body on the method, it can be thirty times if the method only has a body that can be fuzzed and no variables in the url, or fifty times if it has a body that can be fuzzed and variables in the url, or only one valid call when there are no body to fuzz or variables in the url. This number will be calculated by the function number_of_times_to_fuzz in the module fuzzer. After this phase, the tool will check if that method has a monitor (for some methods it is irrelevant to try and get the state before the testing) defined and if the operation is different from get (if the operation is get then we know that, there will be no change in the state of the IoT device). If it has a monitor method and the operation is different from get then the state that the device is currently in will be retrieved using the function get_values_for_state in the module caller
using the method defined on the monitor field.

Then it comes the cycle that will test the method chosen in the sequence. This cycle will use the values described before to know how many tests it will do to the method. This cycle starts by calling the function fuzzing in the module fuzzer, in order to get the url and the payload for the call of the method being tested, the function also gives some other informations that is useful in the cases that is necessary to do a detailed log. These informations can be the tool used for the fuzzing, what was the target of the fuzzing (position in the sequence), what was the fuzz_flag, what was the payload used as the template, etc. Then with that information the tool will call the function call_execution of the module caller, in order to place this call with the received information (the url and the payload) onto the IoT device. Next, it checks the cases that the response has a status code of 200, in order to verify if the call was really accepted by the IoT device, or if it has an error message in the text field. In these cases, the status code will be changed into an irrelevant one. For the cases that the url was the fuzzing target and the call was accepted, the tool proceeds to log the call, if the payload was the target of the fuzzing another step is necessary before the log can happen. The additional step is the comparison between the states and the payload, which is done through the function compare_get_states of the module caller. If any irregularity is detected, then the function will proceed to the log of the call. After the states are compared and all the logs are done, it will start the process of reverting the IoT device back to the initial state, this process was already described in the function revert_state and reposition_of_method_calls that are in the module caller. Once this is done, the IoT device is at a state that it is possible for another sequence or method to be tested and the cycle continues to the next sequence. When all the sequences are tested and the testing process finishes, the tool will call the function model_inference_default in the module model_producer that will use the logs model_log to produce the model of the sequences with the errors detected.
3.2.8 Tool Outputs

This section will describe the outputs that are produced by the tool, some of them were already mentioned in the previous section, but they were not the focus of the discussion. The tool produces four files as outputs, three of them are log files (the total log, the detailed log and the model log) and the last is the model of the sequences that resulted in an error.

So, the first output is a log file, that logs all the activity done by the tool to the IoT device. This has the objective of allowing the user to recreate the exact same calls that were done in that test run. It includes the call number, in an ordered fashion from the first call made until the last. It has the operation of the call, it has the url used on the request, the payload used by the request and on the last parameter it registers the status code returned by the server. The requests may be: the get variable values; the get state of the IoT, the reset of the IoT, the Method call with/without fuzzing, and any other request that may be used by the tool to the IoT. In the Figure 3.14 we can see an example of what was just described.

```
Call number: 1 Type of call: Populate the API variables
Operation: get
Url: https://domain.com/v1/scenes
Payload: {}
Response code: 200

Call number: 2 Type of call: Populate the API variables
Operation: get
Url: https://domain.com/v1/lights/all
Payload: {}
Response code: 200

Call number: 3 Type of call: first reset
Operation: put
Url: https://domain.com/v1/lights/id/state
Payload: {'color': 'green', 'duration': 0, 'power': 'on', 'brightness': 1}
Response code: 207

Call number: 4 Type of call: Populate the API variables
Operation: get
Url: https://domain.com/v1/scenes
Payload: {}
Response code: 200
```

Figure 3.14: Total Log file

The second output is also a log file, that has the objective of logging all the sequences necessary to generate the model of the sequences that had an error. This file logs one line per request and each line is composed by the operation together with template of the current path, the sequence number and the test number. The sequence number is then used by the model producer as the unique identifier of the trace. In the Figure 3.15 we can see an example of what was just described.
Figure 3.15: Total log file

The third output file is the last log file, which has the objective of giving a detailed description of what went wrong in the call/sequence. It enumerates several parameters, that makes the analysis of the call easier. It also logs the parameters of the call that were used on the same sequence, but were not the target of the test, so they will be all valid calls, but sometimes it is the state of the IoT prior to the testing of the method, that affects the outcome of the testing. First the logs have the test number correspondent to that request, it has the time that the request was done, then it has the sequence that is currently being tested, what is the place in the sequence correspondent to the target of the testing, what is the operation of the target of the test, what is the used URL (what are the values used in the variables, if they had any), what is the response status code of the IoT device, what is the response text of the IoT device and what is the type of fuzzing done to the IoT device. It can be URL fuzzing (to fuzz the path/URL), it can be body fuzzing (to fuzz the payload sent to the IoT device) and it can be the control fuzzing (when no fuzzing is applied). It also has the tool that is used to do the fuzzing (Radamsa, Pyfuzz, off_by_one), the base payload is the payload used in the control calls or the payload used has the seed for the fuzzing, the fuzzed payload is the payload after the fuzzing was done to the base payload, the processed payload represent what the IoT device accepted in the payload, because there are cases in which the IoT device can reject some parameters in the payload and accept others. The state before the request was sent and the state after the request was sent, and the last parameter represents the differences found between the state before the request was sent and after the request was sent. The Figure 3.16 is a practical example of this parameters that were just described.
Test number: 126
Sequence: ["put /lights/id/state", "post /lights/id/state/delete"]
Place in sequence: 0
Operation: put
Url: https://domain.com/v1/lights/id/state
Returned code: 202
Returned text:
Fuzz type: control call
Fuzz tool: no tool used
Base payload:
{"color": "rgb:247,168,207", "duration": 1.43990342662084722, "power": 'off', 'fast': True}

Test number: 129
Sequence: ["put /lights/id/state", "post /lights/id/state/delete"]
Place in sequence: 1
Operation: post
Url: https://domain.com/v1/lights/id/state/delete
Returned code: 207
Returned text: {
  "results": [
    {
      "status": "ok"
    }
  ]
}
Fuzz type: body fuzzing
Fuzz tool: off_by_one
Base payload:
{"power": 'on', 'duration': 693198653.3711875, 'hue': 233.56922640251548, 'brightness': 0.8683929147983231}
Fuzzed payload:
{"power": 'on', 'duration': 3155768896.8, 'hue': 0.3, 'brightness': 1.1}
Processed payload:
{"brightness": 1.1, 'duration': 3155768896.8, 'hue': -0.1, 'power': 'on'}

Figure 3.16: Detailed log file

The last output file is the model that is produced by the module model-producer with the help of the model-logs and the tool Synoptic. So the Synoptic tool takes the logs and a set of regular expressions (to parse the log), in these set it also has to be indicated what should be the unique identifier to distinguish the traces, for the tool it will be the sequence number. As referred before, the information of this model has limitations and sometimes it can be misleading, because it models the sequence until the point of the target method of the testing, meaning that not all the methods registered in this file have problems. Even with this limitation, the model is useful for the user to see the different possibilities and interactions that may be possible between the methods/sequences with problems.
Chapter 4

Results

This section is reserved to discuss the results produced by the tool. Our tool was tested with multiple IoT devices available in the market and in this section we discuss in detail the results obtained for a specific device. The issues found are non-conformance issues with respect to the specification (and not security vulnerabilities) and were communicated to the developers and fixed in due time. The assignment that takes longer is the generation of the three configuration files, which is the only task that requires human intervention in our tool. The tool generates a high amount as well as diversified tests and the time it takes to test the devices is proportional to the depth set to the DFS algorithm, and above all it can incorporate into the testing the use of sequences composed of the methods that belong to the API of the IoT.

4.1 Device A

The API of this equipment is composed of eleven methods, which were all described in specification API JSON file. Based on these methods and with a depth of two in the DFS algorithm the tool produced one hundred and thirty-two sequences. At the end of the testing process the tool discovered six issues on this equipment and produced the correspondent model. As mentioned before all the discovered issues were minor as we flag as an issue anything that does not match the public description of the device API. In the Figure 4.1 we can see the model that was produced by the tool for this equipment. Next we will describe in detail, each one of the six issues.
4.1.1 First issue

The first issue that we encountered on this IoT device, was related to how the IoT device processed the *duration* parameter. The valid values for this parameter (*duration*) are between “0.0” and “3155760000.0”. Every request that had this parameter in its payload, with the value “-1” (which is an invalid value), the payload was accepted by the IoT device, but the device wouldn’t update its state. For example if the IoT device was on the color green, and we sent a call to change its state to blue, with a payload that had the parameters “duration: -1” and “color: blue”, the IoT device would accept this payload, but the color would remain green. This issue doesn’t have a negative impact on the device beyond the fact, that it accepted an invalid value and despite accepting a payload it didn’t update its state.

4.1.2 Second issue

The second issue is related to the IoT device returning the error status code 500. This happens when the tool did the fuzzing to the url of the request and resulted in *internal server error*. In the list below there are several examples of *url's* requests from different methods that led to this error.

- https://domain.com/lights/d073d53d5322321"xcaf%d%nNaN\r
- https://domain.com/v1/lights/d073d214747$65537;xcalc%d'xcalc!xcalc!xcalc%p%s\n%n%s\x00%nNaNaa aa%d%n\x00\n$PATH\u0000-10784fa
- https://domain.com/v1/lights/d073d5%#x%s\x0d%naaa%d%nxcalc%naaaa%d%n\x0d$"xcalc$\NaN$&3213fa
- https://domain.com/v1/lights/d073d5073d5\x1a\1!lxcalc\r\n\n$1\3d5129f%dNaNkcalc\r\n\n$1\32 767%n\u
4.1.3 Third issue

The third issue is related to a specific method of this IoT device. The method in question has the following current path “/lights/:id” and the operation get. The objective of this call is to return the state of all the IoT devices if the “:id” variable has the value “all”, or the state of a specific device if the value is the id of such device. By the specification the variable “:id” can never be empty; it either has the value “all”, or the value of a specific device.

The issue is when the value empty is given to the variable “:id”. According on the specification this shouldn’t work, but in reality it returns a response equal to when the value of the variable is “all”. This is contrary to the specification and so we flagged it as an error.

4.1.4 Fourth issue

The fourth issue is also specific to a certain method in the IoT device. Before we explain what the issue is, we will explain what the method is supposed to do and what it can receive as parameters. The method in question has the following current path “/lights/:id/state/delta”, where the variable “:id” informs about the devices that will receive this request. This method has the objective of giving an additive value to certain parameters in the IoT device. This parameters can be, the infrared, the hue, the saturation, the brightness and the kelvin. All of this parameters when they receive a valid value, will add that value to the current value in the state for that parameter, until the upper limit for that parameter is reached, with the exception of the parameter hue, that works in a cyclic way in angle degrees, meaning that after it reaches the number “360” it will go to “0” again. This method can only add values. Now that we described the method, the issue is related to the update of the parameter hue. So if the value for the hue in the state of the IoT device is “10” and in this method the value for the parameter hue is “380”, the resulted value in the IoT device state should be “30”. This is correct and it is how the method should work. The issue is when a value big enough is given. Lets take the same example and for the parameter in the method lets give the value “400000000”, it should give “1111111111” cycles and the state should have the value “50”, but in reality when a value big enough is given the result for the parameter hue in the IoT state is always “0”. So the issue is that the IoT device is not doing the right calculus and just assuming that when the value is big enough the result will be zero.

4.1.5 Fifth issue

The fifth issue that we found is related to the method that has the following current path “/lights/:id/toggle” (again the variable “:id” informs on the IoT devices that will be affected by this method). This has the objective of every time that the IoT device receives this method and the parameter power (it describes if the device is on or off) and the state of the IoT device is on it changes to off and vice-versa. The issue is in the parameter duration, that this method can have on its payload. If that parameter takes certain values it produces an abnormal behavior in the IoT device. For example if the device on the parameter power has the value off and this method is called, with the value for the parameter duration set to “-16792465476215.0” or “-1679251545.4534614” then we end up with the value of on for the
parameter power and the value “1” for the parameter brightness (which indicates that the lamp should be on and with full brightness) on the state of the device. But in reality the physical state of the device is off (no light is being sent). This issue was detected when the IoT device accepted this request with an invalid value (since the valid values for the parameter duration are between “0.0” and “3155760000.0”), then when manually testing and recreating the issues found we observed that in this issue there was a discrepancy on the state that was returned by the IoT device and the physical state observed for the device. Another detail is that this issue can only be observed when the value for the parameter power in the state of the IoT device prior to the sending of the request is off.

4.1.6 Sixth issue

A feature of this IoT device is whenever the brightness parameter takes the value zero, the device changes the value of the power parameter to off and the value of the brightness parameter to one. The valid values for this brightness parameter is between zero and one. The issue happens when the input to the brightness parameter is a big number “555555555555555”, the state of IoT device takes the value of the power parameter to on and the value of the brightness parameter to zero and this should never happen.
Chapter 5

Conclusions

In this thesis we foresee that the IoTs will have a great impact in our world, controlling all sorts of devices on different aspects of our lives, from our work, to our personal lives, to our "society". For the present and for the future, we have to increase the testing of the software behind these devices, since the "virtual world" is having more and more impact in our physical world and personal lives and if we are to believe and trust the IoTs with our personal information, then we need to guarantee a certain quality of the software that runs on these devices.

We started this thesis by studying the existing literature on API black-box testing. We realized that there were two major testing fields on how to test an API. The first field was fuzzing that allows to intensively test the input parameters of an API methods without having access to the source code. The other field was the test of how a sequence of method calls can influence the internal state of a device in order to show abnormal behavior. There was also a study of the bibliography on how to create a system model from the data accumulated from the tests, in other words, how we could create an accurate model that represents the internal model based on the inputs and outputs of the tests.

Only after the evaluation of the previous mentioned topics, we started the development of a new testing tool for the APIs of IoT devices that use REST protocol. The tool has three main goals: 1) to do fuzz testing on all the methods that belong to the API; 2) to test sequences that are combinations of these methods and; 3) to generate the model based on the results of these tests that represent sequences of the problems detected. For this testing to be possible the tool needs three files that describe the initial state of the IoT device, the configuration for the usage of the tool, and the methods available in the API together with all the security information necessary to produce the requests. The sequences are produced before the testing process starts and all the possible combinations are generated based on the level provided by the DFS algorithm that the tool uses to generate the sequences. After that the tool will automatically go through each one of these sequences testing the last method in the sequence. The sequence is used to exercise the testing of the methods in different internal states of the IoT device. During the fuzzing process the tool uses PyjFuzz, Radamsa and off_by_one tools to produce different request of the methods and the objective is to test the most diverse values for the different parameters present in the payload and also to test the url used by that method. After the testing process is finished
we are left with three different kinds of logs. The detailed log that details various informations about the state of the IoT device and the calls that originated an error. The total log that contains all the calls made to the IoT device. The model log that contains the sequences that had problems and that are used to produce the model. The model production starts after the testing process is finished, it uses the tool Synoptic and the model log to produce this model which together with the detailed log can show to the user different interactions between methods that have problems and help the user to understand how certain combinations of the problems may cause a catastrophic error in the IoT device.

We improved the state of the art in IoT testing, by creating a new tool that automatically tests the IoT devices through a combination of method sequence production with black-box testing, that will also generate the model that exhibits the problematic sequences of these IoT devices.

To evaluate the potential of this tool, we tested it in real scenarios with multiple IoTs that are already available in the market, in order to determine if the tool is capable of detecting errors/bugs or to detect abnormal behaviors in the IoT. In our tests we were able to detect several non conformance issues that contradicted the specification of the API of the devices being tested.

At the end, we accomplished to develop a tool, that is able to tackle some of the problems that exist on the other tools available in the world market. It fulfills all the objectives that we had set at the beginning: to be open source and based on tools that are also open source, to be intuitive, to do automatic testing with the minimum action of the user, to create a diversity of tests for each of the methods being tested, to produce a model of the sequences for the problems detected, and to be able to produce results (find problems) in IoT devices which are in production in the market. The only aspect that we weren’t able to fully fulfill, is the time that is necessary to produce the files that are used as input by the tool. The tool uses these files to learn how to interact with the IoT, how to reset the device to the initial state, and to configure some parameters.

5.1 Future Work

Some ideas about the future work for this tool may pass through a new way to pass information to the tool of the specification of the API, either by learning from the interaction between the user and the IoT or by the software community reaching a consensus about a file that is used to describe an API. Another point that could be improved is the production of the model, the user could identify the methods that have problems from the ones that didn’t had any problem. The last point that could also be improved is to use the problems found as a base for new sequences, and by doing so, trying to figure out a new way of producing abnormal behaviors in the devices.
Bibliography


[8] www.z wave.com. ZWave Home Control...


Appendix A

Appendix Specification API JSON

Header

The header is where all the information necessary for the parameter header in the call is described. This can be empty, or it stores security credentials or security tokens. For example the device A device requires a security token in the header of the call in order for the call to be accepted. An example can be seen in the Figure A.1.

Basepath

The basepath as the name implies is the part of the url that is common to all calls, usually this never changes since the methods are in the last part of the url, so this variable abstracts the common part. Since this base path can also have variables, their value is indicated in the path.parameters of methods description. An example of a basepath is “http://domain.com/v1”, in the Figure A.1.

Paths

The paths is the section of the file that informs the tool what are the methods that exist on the API, how they are built, what are the ranges for the parameters, what is the operation, etc. The next dependency after paths are the methods that are to be used by the tool and then comes the correspondent operation. For some methods it is possible to have more than one operation. An example can be seen in the Figure A.1.

Parameters

The parameters section is where all the information about the payloads of the calls and the variables of the path calls are described. This section is composed by the following two sections the path.parameters and the body.parameters/query.parameters, since a method sometimes has a query on the payload, the method can either have a body.parameters or query.parameters depending on the situation. An example can be seen in the Figure A.1.
Monitor

The monitor section has the objective of informing the tool of the method and operation that needs to be used, in order to retrieve the state of the IoT device that allows the tool to compare the changes caused by the call. This helps the tool to know how to collect the state before and after the call is made.

Setup

Before detailing about what goes on inside the parameters section, we will discuss the setup section. The setup section has the function of informing the tool that the described method is necessary to populate some variables related to the IoT device. It also has a list of methods that are necessary to call.

Figure A.1: Example of Specification API JSON file
in order to put the IoT back in its *initial state*, this for the cases that the call of the method is accepted. It also has one *action* parameter (different from the *action* that can be found in the sections following the *action section*) that can take the values *create or delete*, to help the tool to track the goal of the method. Sometimes when putting the IoT device back into its *initial state* it may happen that a resource needs to be created, like a new group and the ideal scenario is to create the one that was just deleted.

**Variable retrieval**

The *variable retrieval* parameter has the objective of informing the tool if the method and operation in question are necessary to populate a variable. Meaning that the tool will know the values of the variable used to represent the resource in the IoT device. For example what are the id's of the light resources for the IoT devices, what are the id's used for the groups, what are the id's used for the scenes, etc. To get these id's the tool needs to know what methods and operations to call in order to retrieve them.

This variable is a flag that instructs the tool if the response of calling the method will return the id's that in conjunction with the field *get variables* on the configuration file (to interpret the response of the IoT device) it is possible for the tool to have all of the id's that it may need for the testing process.

**Variable_id**

For the cases where the *variable retrieval* is true it is necessary to have the parameter *variable_id*. This informs the tool of what is the name of the variable for that id. This is important because every time the tool is constructing a *url* and meets a variable, it verifies if the variable is present (in the tool's memory) and what are the values available to use in that variable. Suppose that the path has a variable *device_id* ("http://.../:device_id/..."). If the tool has the variable *device_id* it will use the stored value to construct a legitimate path.

**Reverse**

The *reverse* parameter has the purpose of instructing the tool about the methods and operations to be called in order to put the IoT device back on its *initial state* if the method call in question is accepted. There are cases where there are more than one method and operation. For example, let us assume that we deleted one device in the system. Then it is necessary to do multiple calls in order to guarantee that the device is back to the system and it is in the initial state.

**Path.parameters**

Going back to describing the *parameters* section we will start by the *path parameters*. The *path parameters* has the objective of describing the variables present in the methods path. The *path parameters* have the following parameters that will be described in detail in the *body parameters*: The *name_of_the_variable* with the following fields, the *type*, the *requirement*, and the *example*. An example can be seen in the Figure A.1.
**Query parameters**

The `query_parameters` section has the objective of building a query for the payload. The parameters will be the same as described in the `body_parameters`, the only difference being that the result will be a string composed of those parameters.

**Body parameters**

The `body_parameters` is where a great part of the API description goes. This section has the objective of helping the tool to construct a valid payload and at the same time, after the valid payload is fuzzed, to check if the fuzzed payload is still a valid one (based on the type and range parameters). An example can be seen in the Figure A.1.

**Name of the variable**

The `name_of_the_variable` section contains all the information relevant to that variable in the payload. An example can be seen in the Figure A.1.

**Type**

The `type` variable has the objective of indicating to the tool the type of the variable. The type can be `string`, `integer`, `float`, `boolean`, `array`, `object`, etc. This is then used by the tool to check if the parameter in the fuzzed payload has the correct type, and if it has not it implies that the payload shouldn’t be accepted. If it is accepted something went wrong. It is also used in other steps during the tool execution. This variable can be found in all the positions that API variables can be described, `path_parameters`, `body_parameters` and `query_parameters`. An example can be seen in the Figure A.1.

**Requirement**

The `Requirement` parameter informs the tool, if the variable always needs to be present in the payload of this method or if it is optional. The parameter can have two values, the first is `optional`, which will mean that the variable may or may not be present on the payload (random choice) and `required`. `Required` as the name implies the variable has always to be present in the payload of this call. This was developed to give the tool the opportunity of constructing different sets of payloads, but at same time have the confirmation that the required items for the call to be accepted are always present. An example can be seen in the Figure A.1.

**Range**

The `range` parameter gives the possibilities for the values of the variable. The `range` parameter can take one of three set of values. It can take the `no info` value, for a specific value like a name or a device id. It can take a list of strings for different specific values that can be used by the parameter. And it can take an interval of valid values that the parameter may hold. An example can be seen in the Figure A.1.
Length

The *length* is the parameter that dictates the number of characters that the value of the variable may take. For example if *type* of the variable is string and the *length* is between zero and thirty-two, it means that the variable accepts all strings that have between zero and thirty two characters.

Action

The *action* parameter has the objective of giving more the details to the tool about what the variable will do. For instance, it can be used to inform the tool that the variable will *add* or *decrease* a certain value, it can inform that the variable can *cycle*, meaning that when it reaches one of the limit values it will go in the direction of the other limit. It can inform the tool on which variable it is represented and on which variable the action will have an effect. It can also have the information that the variable is a time related variable, meaning that the variable is a timer or it informs at which time the action will take place. An example can be seen in the Figure 6.2.

```
"action":{
    "operation":"add",
    "parameter":"bri",
    "override_by":"bri"
},
```

Figure A.2: Example of the Action section.

Example

The *Example* section is the section in control for passing information to the tool, in order for the tool to build a valid value for the variable. The following sections are the available sections in the *Example* section. An example can be seen in the Figure A.1.

Value

The *value* parameter is used when the user has absolute certainty of what value is needed to be used in that parameter. For instance, lets take an hypothetical case where we have the variables *ip* and *username*, for this case the value of their variables is something very specific like an *ip* address and a *username* given by the system, in this case the parameter *value* in the *example* section will inform to the tool of the specific value of this variable.

Choice

The *choice* parameter is a list of all the valid choices that the parameter can take. This gives the tool the possibility of choosing at random one of the values of this list. This parameter is used usually when the variable takes boolean values or different string choices (for example “enabled” or “disabled”). An example can be seen in the Figure A.1.
Call

The call parameter is when the valid value for this parameter is a call to another method, present in the same specification of the same API. What this parameter does is to give a list of operations to chose from and the basepath, and it will choose a random existing method that fits the given conditions. This is more commonly used in methods that have triggers, because when that trigger is activated it will perform an action, one example is the alarm feature.

Times

The times parameter is used to inform the tool how many times it will generate a value for this parameter. In other words this is used for arrays that have more than one entry, it informs the tool if the variable needs an array of size two, than it will generate a value twice and then it creates an array with those two values.

Template

The template parameter is used to reference another specific method. These parameter informs the tool that this parameter will have a specific call to another method. The difference between the template and the call, is that the call chooses a random method while the template receives a specific method to call. An example can be seen in the Figure A.1.