hochschule mannheim

Conception, Implementation and Postmortem Documentation of a Modular Proxy Application for Testing Internet of Things Applications

Moritz Laurin Thomas

Master Thesis

for the acquisition of the academic degree Master of Science (M.Sc.)

Course of Studies: Computer Science

Department of Computer Science

University of Applied Sciences Mannheim

31.05.2021

Tutors

Prof. Dr. Thomas Specht, Hochschule Mannheim

Pierre-Alain Mouy, M.Sc., NVISO GmbH

Thomas, Moritz Laurin:
Conception, Implementation and Postmortem Documentation of a Modular Proxy Application for Testing Internet of Things Applications / Moritz Laurin Thomas. –
Master Thesis, Mannheim: University of Applied Sciences Mannheim, 2020. **??** pages.

Thomas, Moritz Laurin:
Konzeption, Implementierung und Post-mortem-Analyse eines modularen Proxys zum Testen von Anwendungen im Internet der Dinge / Moritz Laurin Thomas. –
Master-Thesis, Mannheim: Hochschule Mannheim, 2020. **??** Seiten.

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Ich bin damit einverstanden, dass meine Arbeit veröffentlicht wird, d. h. dass die Arbeit elektronisch gespeichert, in andere Formate konvertiert, auf den Servern der Hochschule Mannheim öffentlich zugänglich gemacht und über das Internet verbreitet werden darf.

Mannheim, 31.05.2021

Moritz Laurin Thomas

Abstract

Conception, Implementation and Postmortem Documentation of a Modular Proxy Application for Testing Internet of Things Applications

TBD

Konzeption, Implementierung und Post-mortem-Analyse eines modularen Proxys zum Testen von Anwendungen im Internet der Dinge

TBD

Acknowledgements

TBD

Contents

Chapter 1

Introduction

This chapter will introduce the underlying motivation of this thesis. Then, it will give an overview of this thesis' purpose and structure. Lastly, this chapter will show the process that the work on this thesis went through, explaining the scientific methods applied and software engineering practices used.

1.1 Motivation

Today scientific and industrial parties work on connecting physical entities such as machines, buildings and even humans to the internet by equipping them with digital sensors and actuators, referred to as "**IoT!** (**IoT!**)". While this progression promises many positive effects, such as simplifying tasks in our personal day-to-day life ("Smart Home" applications), monitoring our personal health ("eHealth") and increasing efficiency and safety of industrial plants ("**IIoT!** (**IIoT!**)", also referred to as "Industry 4.0"), it also yields the risk of introducing new attack-vectors to parts of our environment: "smart" devices used at home or at other sensitive places may implement weak security implementations or faulty security design, resulting in private and personal data being available to parties interested in violating the privacy of one's home (e.g. vaccum robots leaking information about the interior design of homes[**wittenhorst_2019**]) or conducting industrial espionage which is an acute threat [**bartsch2018**].
The diversity of both deployed smart devices and the internet services those devices are connected to lead to the need and use of ever-increasing complex technologies used for communication, data storage and access management, further

adding to potential attack-vectors of connected devices and distributed applications [**Jaeger2016**]. This complexity and the sheer number of connected devices is actively being exploited by attackers today and the number of attacks on **IoT!** devices is increasing [**demeter_preuss_shmelev_2019**].
There are security guidelines, best practices and innovative approaches for developing secure smart applications [**Jaeger2016**][**Lesjak2016**], however testing such applications proves to be cumbersome: intercepting, dissecting, inspecting and manipulating the communication in these applications requires working on various abstraction layers. In order to evaluate the security of such applications, penetration testers often spend a considerable amount of time dissecting applications and setting up a test-environment.
The goal of this thesis is to conceptualize, implement and evaluate a modular proxy application that supports evaluation of the security implementations of **IoT!** applications.

1.2 Purpose and Structure of the Thesis

This thesis is separated into eight chapters: chapter (**??**) will give an overview of and discuss related and previous work. After that, relevant fundamentals about computer networks, **IoT!** applications and information security will be covered in chapter **??**.
The chapters **??** to **??** describe the research and development process of the **IoT!** proxy application in chronological order: the problem space of the application is shown and dissected in chapter **??**, yielding essential insights into potential challenges and technical requirements. Building upon these, the conceptual design of the **IoT!** proxy application is proposed in chapter **??**. This included the process of collecting, documenting and analysing of software requirements, describing the application's work context and designing a software architecture that complies with the aforementioned requirements. Subsequently, chapter **??** involves a prototypical implementation of the aforementioned software concept, focusing on the goals and constraints of the implementation, the tools and frameworks used and the implementation of core components of the application. The resulting implementation and the project itself are then analysed in a postmortem documentation, pointing out the reasons why and how the project ultimately failed.

The thesis ends with a summary of all results produced and conclusions drawn from the work on this thesis.

Chapter 2

Related Work

This chapter will discuss related and previous work on topics in this thesis' context. This includes network analysis in general (and **IoT!** in particular), homogenization and unification of various **IoT!** related technologies and performance of security evaluations of these technologies.

2.1 Computer Network Analysis in General

*TBD: Polymorph [***ramos_2018***]*

2.2 Homogenization of the IoT Landscape

*TBD: IoT proxy for homogenization [***wenquan2018proxy***]*

2.3 IoT Security Analysis

As part of their master's thesis, Bellemans conducted a study in 2020 that evaluated the security and privacy implementations of fifteen "*smart*" devices from a wide price range available on the market at the time. They performed automated analyses and requested data access from manufacturers [**JonahBellemans**]. The thesis showed that the devices made use of a variety of both standardized and proprietary transport and application protocols. It also found severe flaws in the devices' com-

pliance to **GDPR!** (**GDPR!**): about a third of the devices' manufacturers did not reply to **GDPR!** requests at all, however Bellemans noted that the COVID-19 pandemic may have had an impact on their data access requests. The thesis suggests that the introduction of a quality label that guarantees appropriate implementation of security and privacy aspects could prove beneficial for customers.

In 2017, Apthorpe et al. presented a three stage strategy to examine metadata of network traffic of four smart devices [**apthorpe2017smart**]. By monitoring the devices' traffic, they showed that even though the communication between the devices and their corresponding internet servers were encrypted, passive observers could deduce information about users' behaviour by identification of the destination server and analysis of the rate of traffic being sent. A noteworthy aspect of their work is that they performed this analysis from an **ISP!** (**ISP!**)'s point of view, exclusively examining metadata of the communication that took place. The strategy described in the paper consists of the following (greatly simplified) steps:

1. Identifying communication streams of individual devices (e.g. by examining the TCP packets' destination IPs).
2. Associating the streams with specific device models (e.g. by performing reverse-look ups of the aforementioned IPs).
3. Analysing traffic rates (presuming that traffic is generated upon taking measures).

TBD: Add simple process diagram

Apthorpe et al. conclude that their strategy works well on inferring behaviour from regular internet traffic of smart devices, however they assume that shaping traffic or making use of proxies (that effectively mask the destination IPs) could be effective counter-measures. It is safe to assume that regular smart home setups do not make use of proxies or traffic shaping though, thus being vulnerable to this kind of attack.

*TBD: NVISO Labs: Théo Rigas, IOXY [***rigas_ioxy***]*

Chapter 3

Theoretical Background

This chapter provides an overview of the technologies and concepts referred to in subsequent chapters. Starting with section **??**, essential concepts of computer communication in networks will be presented and examined, covering the concept of network layers, intercepting of communication between two parties and analysis of transferred data. Building upon these fundamentals, section **??** introduces the fields of use of **IoT!** applications, common architectures used today to implement them and popular protocols they make use of. Lastly, it will discuss security considerations important to **IoT!** applications. After that, section **??** will provide insights into relevant concepts and the practices used and applied in information security. It covers key concepts and legal considerations, integration of information security in software development and common practices and methods involved.

3.1 Computer Networks

3.1.1 Network Layers

TBD **TCP!** (**TCP!**)

3.1.2 Proxying Network Traffic

TBD; planned:

1. Definition; Working Principle
2. Use Cases

3. Abuse Cases

3.1.3 Deep Packet Inspection

TBD

3.2 (Industrial) Internet of Things

3.2.1 Fields of Use

3.2.2 Application Architectures

3.2.3 Common Protocols

Building up on pre-existing network infrastructure and in order to meet requirements specific to individual fields of use and use-case scenarios, the landscape of **IoT!** attends with a great variety of *communication protocols* (further used to refer to both transport and application protocols). This section will provide a brief overview of the working principles, use cases and history of some protocols commonly used in **IoT!** and **IIoT!** applications today.

HTTP! (HTTP!) *TBD*

WS! (WS!) *TBD*

MQTT! (MQTT!) *TBD* **AWS! (AWS!) IoT!**

Modbus TCP! *TBD*

Profibus/Profinet *TBD*

OPC U/A! (OPC U/A!) *TBD*

3.2.4 Security Considerations

3.3 Information Security

TBD

3.3.1 Key Concepts

3.3.2 Legal Background

Compliance

Data Protection

3.3.3 Integration in Software Development

Traditional Approaches

Modern Approaches

3.3.4 Methodology

Risk Management

Incident Response

Reverse Engineering

(Physical) Penetration Testing

Source Code Audits

Application Configuration

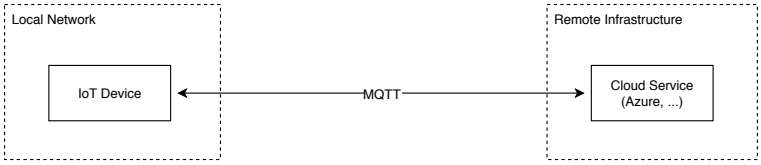
Chapter 4

Understanding the Problem Space

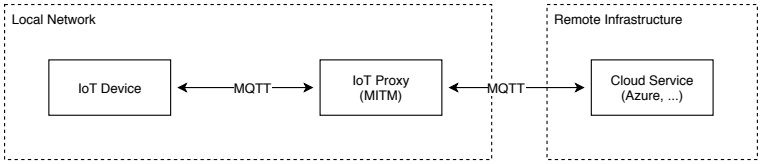
In order to provide a satisfying solution to the problem at hand, the problem itself and the environment it occurs in must be researched. This chapter aims to explore and examine the problem space, resulting in a set of artefacts (namely a domain model and a set of requirements) that aid in understanding the context and designing an appropriate solution. First, a prototypical network proxy is designed and implemented in section **??** to get an understanding of the problems and challenges involved in designing, implementing and using such software. Based on these experiences, interviews with experts in penetration testing are conducted and evaluated in section **??** to get a proper understanding of their everyday work and resulting problems. Lastly, existing software that aims to intercept communication for various scenarios and technologies is examined in section **??**, compared to each other and their usefulness for the problem-specific scenarios is assessed.

4.1 Prototypical Implementation

The prototype was designed to be used in two realistic scenarios; one in an **ICS!** (**ICS!**) context and a more complex one in an **IoT!** cloud context. The goal of this section was to implement a prototype that could be used as a proxy to intercept communication between an **IoT!** device and its cloud service as shown in figure **??**. It was developed incrementally so individual components could be derived from requirements, designed, implemented and evaluated in fixed sprints.



(a) Regular communication between an **IoT!** device and a cloud service.



(b) Communication intercepted by a **MITM!** (**MITM!**) proxy.

Figure 4.1: Installing a **MITM!** proxy to intercept network communication for penetration testing.

4.1.1 Example Scenarios

The following scenarios describe realistic configurations of **IoT!/IIoT!** devices that should be tested with the prototype:

Scenario #1: Legacy ICS! Application In this **IIoT!** scenario, a **HMI!** (**HMI!**) (*Siemens KTP400 Basic*) sends commands to and receives data from a **PLC!** (**PLC!**) (*Siemens S7-1200*) using Modbus **TCP!**. The **PLC!** continually counts up a value up to 100 and begins anew at zero while the **HMI!** displays the current value and provides a button that, upon being pressed by a user, resets the current value to zero. In this scenario, attackers could perform a variety of attacks on the system by intercepting and manipulating network traffic, for example:

- By dropping messages sent from the **PLC!** to the **HMI!**, the application may appear unresponsive as new data is not displayed on the **HMI!**. In production environments, this could lead to dangerous situations as sensor readings that indicate harmful environmental conditions would not be presented to supervising personnel.
- When dropping messages sent from the **HMI!** to the **PLC!**, control commands can be suppressed. This attack can result in catastrophic situations when emergency shutdowns issued by supervising personnel are not registered by the affected machines.

Due to the rather simple nature of the Modbus **TCP!** protocol, intercepting and manipulating communication is expected to be trivial.

Scenario #2: IoT! Cloud Application This **IoT!** smart home scenario utilizes two local **IoT!** devices that are integrated into a cloud environment such as the **AWS! IoT!** platform: a thermometer and an **A/C!** (**A/C!**) unit. Both devices connect to the cloud platform, authorize themselves at a **REST!** (**REST!**) interface via **HTTP!** and upgrade their **HTTP!**-connection to **WS!** streams upon successful authorization. They eventually communicate to a remote **MQTT!** broker by tunnelling **MQTT!** packets over the **WS!** stream. At this stage, the thermometer publishes temperature readings to an **MQTT!** topic while the **A/C!** unit subscribes to the same topic and adjusts its operation depending on the incoming temperature readings.
This distributed communication setup introduces a set of possible attacks that could be performed when attackers *impersonated* client-devices or the remote server:

- Impersonating the thermometer, attackers could send incorrect temperature data and effectively control the **A/C!** unit. When sending low temperature readings while the environment temperature is high, the **A/C!** unit would stop running. Conversely, when high temperature readings are sent while the environment temperature is low, the **A/C!** unit would run, and thus further cool down the environment.
- Attackers that impersonate the remote server could drop or manipulate incoming publish packets, thus altering whether and/or what information is relayed other connected devices. For example, temperature readings that indicate a high environment temperature that would lead to the **A/C!** unit to be powered up could be rewritten in such a way that the transmitted temperature value is considered to indicate a low environment temperature, thus preventing the **A/C!** unit from running automatically.

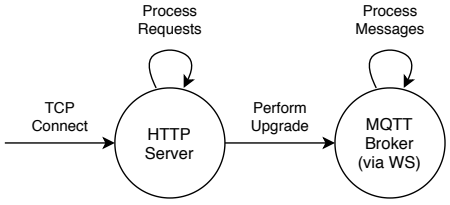


Figure 4.2: State machine of **AWS! IoT!** communication

This scenario makes use of three communication protocols, uses these protocols dependent on the state of authentication and even tunnels one protocol through another one. Therefore the proxy application has to implement a state-machine (as seen in **??**) and testing communication in this scenario is expected to be more complex than the first one.

Scenario #3: Water Treatment Plant Similar to scenario #2, this scenario makes use of **MQTT!** for transporting messages. However, the scenario takes place in an **ICS!** context of critical infrastructure.

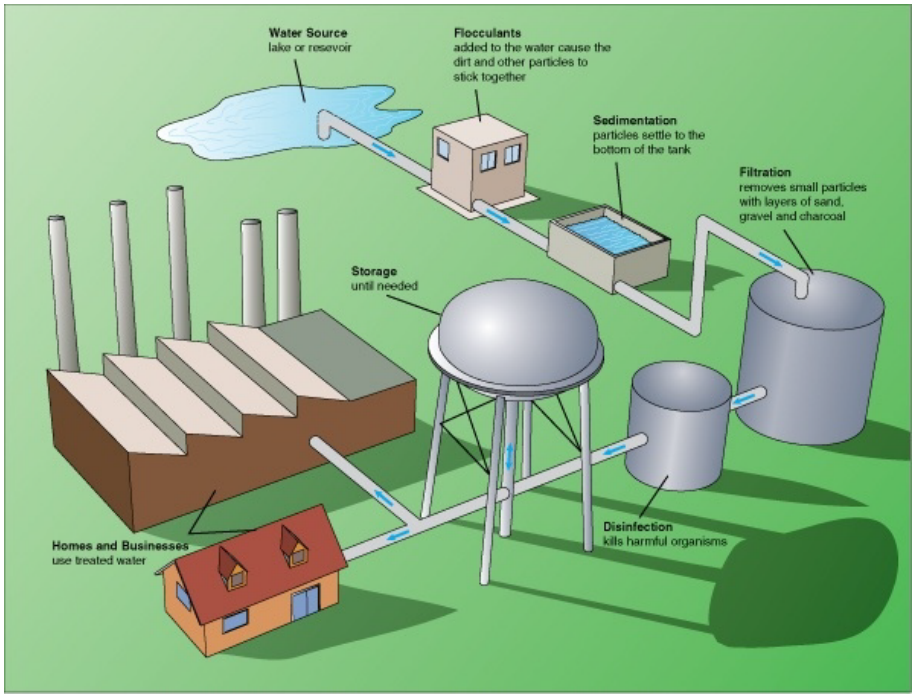


Figure 4.3: "Illustration of a typical drinking water treatment process." (by the CK-12 Foundation)[1]

As shown in **??**, there are multiple steps involved in treating water for drinking. The scenario represents these steps as separate stations ("source", "flocculants", "sedimentation", "filtration" and "storage") that act as **MQTT!** clients. Each station receives water into an input tank, processes water from its input tank in a specified rate and flushes processed water into an output tank. Similar to how threads can suffer from starvation in a multithreading environment, these stations can either "run dry" when their input tank is empty or overflow when either tank is filled beyond their capacity. In this scenario, stations will only process water from their input tanks if their output tank provides sufficient available capacity.

4.1.2 Requirements

To be able to operate in both of the aforementioned scenarios, the prototype had to implement a set of functional requirements:

F1 Protocols: The software must implement parsing/crafting messages/packets of the following communication protocols: **HTTP!**, **WS!**, **MQTT!** and Modbus **TCP!**.
Fit criterion: TBD

F2 Network Stacks: The software must be able to parse protocols that are tunnelled through other protocols ("*stacked*"). It must provide an interface to the user where they can specify which communication protocols are used and whether and how they are stacked (further referred to as *network stack*).
Fit criterion: The software processes a configuration file that lets users specify which protocols to be used and whether/how they are stacked.

F3 State-Machines: The software must be able to switch network stacks and scripts for processing dependent on configurable *states* and *transitions* between them. It must provide an interface for the user to specify when to switch to using another network stack, represented using **FSM!**s (**FSM!**s) and rule sets for transmission between states.
Fit criterion: The software processes a configuration file that lets users specify when to switch between network stacks.

F4 Integration: The software shall provide interfaces for integration of third-party software.
Fit criterion: The software implements interfaces that allow sending packets to other applications such as "Burp Suite".

F5 Scripting: The software shall provide scripting capabilities for automated manipulation of messages/packets.
Fit criterion: Users can define script-snippets to be executed on messages/-packets.

The following non-functional requirements were defined:

N1 Extensibility: To allow for future implementation of further communication protocols the software shall be implemented in a modular fashion.

N2 Platform Compatibility: In order to support a broad spectrum of target platforms, the software shall be implemented platform-independently.

N3 Reusability: The software shall be reusable so it can be used in future tests that may feature new configurations of network stacks.

N4 Open Source: The software shall be available as open source software so programmers and members of the IT community may contribute to improving it.

Due to this implementation serving as a prototype and being of an academic nature, no specific constraints were defined. It was to be developed strictly ignoring aspects of usability and stability as it should not be used in production environments but in laboratories exclusively.

4.1.3 Design

The prototype was designed to be fit for use in the second scenario as, regarding network communication, it was more complex than the first one. Specifically, the second scenario demanded the implementation of a network stack and a state machine to switch between states. Parsing protocols that were tunnelled through other protocols appeared to be a potentially challenging requirement. In order to tackle it, a variation of the *"pipeline"* (sometimes referred to *"pipes and filters"*) design pattern was used (as shown in **??**). It was designed to be used as follows:

"Messages" originate from a listener, for example messages with raw byte payloads are received from a **TCP!** socket. These messages are sent to an initial "pipe" to be processed *down*. Pipes are bi-directional routers that perform the following actions on messages:

1. Use optional "encoders" to disassemble/de-serialize messages when processing them *down* the pipeline and re-assemble/serialize them when they process messages *up* the pipeline.
2. Use optional "filters" to perform operations on messages such as replacing header values or manipulating payloads.
3. Forward messages to the next pipe in its pipeline when processing messages down or to the previous pipe when processing messages back up.

There are extensions to basic pipes such as:

- "EndPipes" are appended to the end of a pipeline and reverse the message processing direction so messages that were processed down are sent back up the pipeline to be processed up.

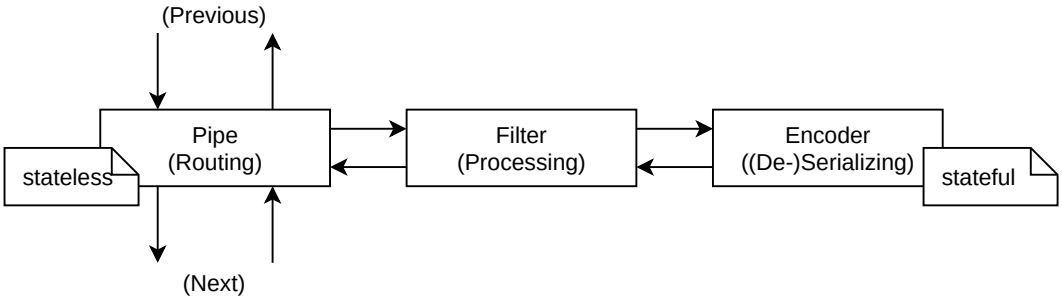


Figure 4.4: The variation of the *"pipes and filters"*/*"pipeline"* design pattern used in the prototype.

- "ProcessingPipes" mandate encoders and filters to be used. These pipes are used to indicate that messages are not not only routed but also processed and encoded or decoded.
- "IntegrationPipes" allow integration of other software into the pipeline. For example, penetration testing software such as Burp Suite could be integrated.

TBD:

- *Designed during sprints so only pipes are designed, state-machine only rough concept (States, Transitions, Rules)!*
- *Show diagrams of messages?*
- *Explain Figure* **??** *and* **??**

4.1.4 Testing

To test the prototype, a simple testbed (shown in **??**) was designed and implemented. It consisted of two Debian 10 machines that acted as a **MQTT!** broker and client and a Kali Linux machine that ran the prototype. All machines were connected to the "IoT Network" and were assigned static **IP!** (**IP!**) addresses. While this setup allowed for more sophisticated proxy mechanisms such as **ARP!** (**ARP!**) spoofing, the *client* machine directly connected to the *kali* machine, which in turn directly connected to the *broker* machine. The *client* machine spawned two **MQTT!** client instances; one that subscribed to a */counter* topic and printed incoming values to console and one that constantly published an incrementing value to the */counter* topic.

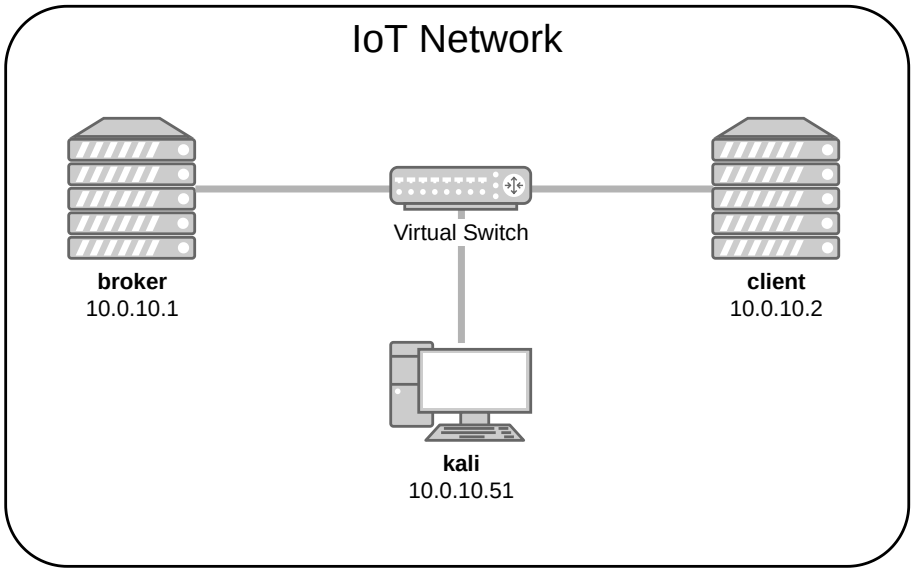


Figure 4.5: A network diagram of the testbed that was used for testing the prototype.

4.1.5 Implementation

TBD:

- *technology: typescript*
- *sprints: two sprints, started with communication (http + ws + mqtt)*
- *failed: high-level API, callback-hell (debugging/tracing), missing typescript typings, very tight coupling*

4.1.6 Insights Gained

The following insights were gained through the prototypical implementation. Some resulted in questions relevant for the expert interviews that were to be held:

- Due to the **MTU!** (**MTU!**), large messages are broken into chunks that are transferred sequentially. This requires the proxy to work on streams of incoming data and reassemble messages from said chunks.
- Support of multiple clients is non-trivial as communication between clients and servers is not necessarily connection-oriented (e.g. **HTTP!**).
  Q: Do penetration testers need to test multiple devices at the same time?
- Increasing the size of the payload of a messages can result in the payload being split upon multiple messages (e.g. **WS!**).

Q: Do penetration testers require exact control over the implementation of protocols?

- Manipulating messages, on the fly via scripting or by hand using third-party integrations (e.g. to *Burp Suite*), can introduce latency to the communication. *Q: Are there strict timing requirements during penetration tests?*
- Many libraries offer high-level functions to the programmer while avoiding exposure of low-level functionalities like crafting or parsing messages.

4.2 Interviewing Experts for Insights

Interviews may be an efficient way to get an expert's opinion on something they are proficient in. Thus, expert interviews were conducted to let security researchers give insight into their everyday work and the challenges they face when working with **IoT!** and **IIoT!** applications. The information and insights gathered in these interviews were then used to model a persona, various work scenarios and use-cases that as a whole aim to represent their work.

4.2.1 Interview Guideline

An interview guideline (shown in *TBD*) was created to keep focus on key points during interviews so that interviewees would not stray too far from the relevant points. The guideline also served as a checklist so the interviewer could make sure that all questions and points that should be covered initially, were in fact covered by the end of the interviews. It was composed of three sections:

1. Experiences with IoT The answers to these questions would give insights into what kind of applications the security researchers had worked on in the past. Answers to question *1.1.* were of particular interest as they might represent what technologies were being examined by security researchers and may be popular in today's applications.

2. Processes in Everyday Life This section aimed to cover questions about the processes and tasks security researchers perform during penetration tests of IoT

applications in their everyday life. Ideally, answers to those questions would show the approaches taken and challenges faced during their work, uncovering potential needs and underlying motivation.

3. The Future of IoT This section had security researchers assess what the future of IoT may be like from their point of view. This required the interviewees to make a critical assessment of the status quo.

4.2.2 Conducting Interviews

Interviews were conducted with six *NVISO* employees that all had worked on security assignments on **IoT!** or **IIoT!** applications in the past. There is considerable variety in

- the experience they had in working on security assignments in general: all interviewees had a strong background in cyber security that reached back multiple years except one who was a working student at *NVISO Labs*.
- and the experience they had in working on **IoT!/IIoT!** applications: two interviewees worked on assessing **IoT!/IIoT!** applications only occasionally, one was part of a car manufacturer's automotive security team in the past and three were part of *NVISO Labs* and worked with smart devices on a regular basis.

The duration of the interviews varied from 45 minutes to two hours depending on the amount and level of detail of information provided by the interviewees and the number of times that the interviewer had to ask further questions.

TBD:

- *Summary of the interviews*
- *conclusions drawn*
- *personas and user stories -> new requirements!*

4.3 Analysis of Existing Software

Wireshark more than 3,400,000 lines of C code[2] *TBD*

MITMf *TBD*

Ettercap *TBD*

bettercap *TBD*

mitmproxy *TBD*

mProxy *TBD*

IOXY *TBD*

scapy *TBD*

TBD; planned: paragraph about each program including a general description, uses, capabilities and usefulness

Name	*Latest Release*	*Implemented in*	*Supported Protocols*	*R*	*W*	*D*
Wireshark	2020-07-01	C	Various	F	N	N
MITMf	2015-08-28	Python	Various	?	F	F
Ettercap	2019-07-01	C	Various	F	F	F
bettercap	2020-03-13	Go	Various	F	F	F
mitmproxy	2020-03-13	Python	HTTP/S, WS	P	P	P
mProxy	Pre-Releases only	Go	MQTT	?	F	-
IOXY	Source only	Go	MQTT	F	F	F

Table 4.1: Comparison of existing software where R, W and D describe read, write and deletion capabilities, respectively. F, N and P indicate full, no or partial functionality, respectively.

[2] This number was returned by the *cloc* utility run on commit *3a8111e1c2adcdc0603993c6ed5d20a40f162125* from Aug. 4th 2020 of Wireshark's Github mirror.

Chapter 5

Conceptual Design

This chapter will detail the process of conceptualizing the design of the modular proxy application based on the results of the preceding chapter. First, the requirements are analysed for their potential design implications in section **??**. Afterwards the user interactions and domain entities identified in chapter **??** are examined and broken down into communication flows between actors and systems in section **??** and individual software components that complete the design are discussed in section **??**. Lastly, an overview of the complete design concept is given in section **??**, discussing potential advantages and constraints.

Note: sections **??** *and* **??** *should probably be merged as they overlap a lot*

5.1 Requirements: Design Implications

TBD

- *Stream-based: treat communication as streams. message-based systems are simpler and supported by design*
- *Server-client: proxy is server, client can interface to control + monitor, communication via REST + WS*

5.2 User Interactions: Designing the Intended Workflow

TBD

- *Passive: Logging*
- *Passive: Scripting*
- *Passive: Fuzzing*
- *(Inter-)Active: REST+WS or Burp Suite integration*

5.3 Inferring Software Components

TBD

- *State-machine: active network stack/pipeline dependent on state of the connection*
- *NetStacks: series of pipelines, bound to states*
- *Pipes: basic pipes, loose routing, injectable, specialized, generic processors + specialized encoders*
- *Factory: parse state-machine and netstack configuration and instantiate + configure instances*

5.4 Summary: An Abstract Design Concept

TBD (maybe obsolete as this is covered in preceding sections)

- *Component view?*

Chapter 6

Implementing the Modular Proxy Application

This chapter covers an exemplaric implementation of the concept that was worked out in chapter **??**, starting with formally describing the goals and constraints of this implementation in section **??**. Afterwards, an overview and comparison of available and suitable tools for the task is performed in section **??**. The chapter concludes with details about the implementation of individual components in section **??**, describing how specific challenges were overcome and what design patterns were used.

6.1 Goals and Constraints

TBD

- *Focus on complex scenario #2: HTTP, WS, MQTT*
- *No interactive mode*
- *Fully implement factories, state-machines and netstacks as POC*

6.2 Tool Selection

6.2.1 Requirements to the Tools

TBD

- *Scriptable!*
- *Low-level access to APIs*
- *Rich set of low-level libraries for protocol implementations*
- *Accessible and easily extendable*

6.2.2 Comparison of Programming Languages, Frameworks and Libraries

TBD: Discuss how the candidates match the above-mentioned criteria and point out why python was chosen

- *Native C: Win32 API / Linux ABI*
- *.NET Visual C# & NuGet*
- *JavaScript/TypeScript & npm*
- *Python & pip*

6.3 Individual Components

6.3.1 Network Stack

Gateways

Pipes, Encoders and Processors

Scripting

Pipelines

6.3.2 Finite State Machine

States

Transitions

Nested FSM!s

6.3.3 Configuration Parsing and Building

Factories, Builders and Templates

Chapter 7

Postmortem Documentation

This chapter attempts to identify and spell out the causes of the project failure. The project timeline will allow a quantitative overview of the project progression and show what parts of the project slowed down progress. Then, an overview of the qualitative aspects of the deliverables will discuss the maturity of the implementation and which parts reached a satisfactory level.

7.1 Quantitative Overview: Time Management

Comparing the planned thesis schedule to the actual course it has taken, this section discusses how the intended plan was implemented and changed at certain places. Also, it will examine the causes of the delays during development.

7.1.1 Project Timeline

Table **??** shows the initially planned thesis schedule divided into four phases, laying out the course of the thesis over a span of 24 weeks.

1. Preparation The initial phase covered preparation tasks for further work on the thesis. Literature research on the topics covered and touched in this thesis was carried out. Related work on **IoT!** and **ICS!** security analysis (as discussed in chapter **??**) was of special interest as those showed what approaches had been taken to assess security implementations. Also, a testbed (shown in **??** for running the proxy

Phase / Task	Duration
1. Preparation	4 weeks (16, 66%)
Literature Research	1 week
Expert Interviews	1 week
Testbed Configuration	2 weeks
2. Prototype	7 weeks (29, 16%)
Prototype Conception	2 weeks
Prototype Implementation	4 weeks
Expert Feedback	1 week
3. Release Candidate	7 weeks (29, 16%)
RC Conception	2 weeks
RC Implementation	4 weeks
Expert Feedback	1 week
4. Finalization	6 weeks (25%)
MQTT Case Study	2 weeks
Thesis Finalization	4 weeks
Total	24 *weeks*

Table 7.1: Initially planned schedule for the thesis

application was built. A decision was made against conducting expert interviews before implementing a first prototype on the assumption that practical experience with the subject matter would benefit the expert interviews. The fact that a number of important questions arose from work on the first prototype later proved this decision to be correct. Performing the literature research and building a testbed was completed within the intended schedule of three weeks.

2. Prototype In the second phase, the prototype discussed in section **??** was designed and implemented in weekly sprints. Preceding these sprints, a rough design of the prototype's architecture and runtime behaviour was worked out in one week that would serve as a base for further design refinement and implementation in the sprints. These sprints ran for eight weeks in total: the initial design turned out to be too oversimplified so that sprints aiming to design and implement specific components were conducted rather isolated from other components that still needed to be worked on. As a result, both the integration of individual components and their interaction would fail and require redesigns and time-consuming adjustments to their implementation. Also, neither was the prototype mature enough to be used as a proxy application, nor was the resulting design and implementation clean enough to suggest putting further effort into working on them. After these eight sprints, work on this prototype was stopped and the expert interviews discussed in section **??** were prepared and conducted.

3. Release Candidate The third phase was intended to yield a fully functional proxy application. This was initiated by switching the technology stack from TypeScript to Python and re-designing and re-implementing large parts of the first prototype. In order to avoid the same mistake of refining a vague design concept and spending time adjusting the design and implementation to make them work, two weeks were spent on a new design concept shown in section **??**. This concept did not only define single components (discussed in section **??**) but also interfaces that specified how those components interacted with each other, aiming for clear separation of components and high flexibility in implementation. Components of the prototype that were independent of the communication protocols used at runtime, such as NetStacks and **FSM!**s, were implemented first over the span of four weeks. Then, implementations for supporting the **HTTP!**, **WS!** and **MQTT!** protocols followed over a span of another six weeks. Work on this prototype was stopped after

Phase / Task	Duration
1. Preparation	3 weeks (12%)
Literature Research	1 week
Testbed Configuration	2 weeks
2. TypeScript Prototype	10 weeks (40%)
Prototype Conception	1 week
Prototype Implementation	8 weeks
Expert Interviews	1 week
3. Python Candidate	12 weeks (48%)
RC Conception	2 weeks
RC Implementation	10 weeks
Total	*25 weeks*

Table 7.2: Actual schedule of the project

those ten weeks as the technical difficulties discussed in section **??** made estimations over the remaining time needed to finish the prototype both hard to make and rather unreliable.

4. Finalization The final phase was intended to conduct a case study on how the proxy application would perform on scenario # 2 from section **??**. Tests were made to run the proxy application in the testbed shown in section **??** which featured the same communication protocols that were used in scenario # 2. However, the proxy application failed to reliably transmit or encode the messages sent between the **MQTT!** client and broker, thus resulting in a broken communication channel. The complex runtime behaviour and very time-consuming debugging of the proxy application (further elaborated on in section **??**) lead to the decision to stop the project.

Table **??** shows the actual schedule of the thesis. As can be seen, 88% (22 weeks) of the time working on the thesis was spent designing and implementing the prototypes compared to a planned portion of roughly 60% (14 weeks).

7.1.2 Development Challenges

There was a series of development challenges that slowed down implementation of both prototypes considerably:

Complex runtime behaviour The combination of nested **FSM!**s and pipelines lead to several problems during development. Even comparatively simple scenarios to use the proxy application in required a complete configuration file made of a global state machine and at least one netstack. This lead to a dynamic and long chain of references at runtime that made tracing back calls and attributing them to specific instances difficult.
Some problems such as a timing problem in the implementation of **FSM!**s were very time consuming to debug: an **FSM!** would change its state when any of its rules was evaluated successfully and indicated a state change. By design, all **FSM!**s of an active netstack would evaluate their rules when a message entered or left any netstack. When a higher-level **FSM!** (e.g. the global state-machine) changed its state *while* a message was still being processed in a lower-level **FSM!**, the higher-level state-machine would change to another netstack, thus disconnect the lower-level state-machine. Eventually, the message would be processed back up and run into a pipe that had no upstream connection anymore, raising an exception and terminating the program. This particular error was discovered during the implementation and testing of the **MQTT!** encoder, in a runtime setup that involved a global default state-machine with a default **TCP!** netstack and a state-machine that handled **HTTP!** to **WS!** upgrades and processed **MQTT!** messages utilizing network stacks for **HTTP!** and **WS!/MQTT!**.
Other problems uncovered design flaws and required prompt changes to the software design or, in some cases, introduced new constraints to the project. One such example was discovered while testing the **HTTP!** encoder implementation using Mozilla FireFox as an **HTTP!** client. When browsing websites, the browser would open multiple connections to the target host to acquire multiple files at the same time[1]. This required the proxy application to instantiate a new pipeline per incoming connection rather than reside on using a single pipeline. Also, this broke the design intention of pipes being connected to at most one preceding and one succeeding pipe as at some point, the pipelines needed to connect back to the global

[1] For testing single **HTTP!** connections, the key *network.http.max-connections-per-server* could be set to 1 in the *about:config* page.

state-machine. However, when multiple pipelines connected back to a single **FSM!** and the only objects pipes would connect to were other pipes, a multiplexing pipe needed to be implemented. This specific case required to make a decision for the proxy application to support multiple simultaneous connections or enforce the use of only one single connection. For a lab environment, enforcing the use of a single connection might work, however in real scenarios this constraint could potentially lead to the proxy application breaking applications at runtime. The decision was made to change the software design in a way that would allow the proxy to handle multiple connections, however the prototype would only support a single connection at a time.

Open source libraries Both prototypes made use of open source libraries that implemented various protocols and included serialization and de-serialization routines for handling protocol specific packets. However, such libraries appeared to be intended to be used for developing applications that used those protocols as a means for transporting data rather than directly parsing packets.
Usually, these libraries would offer an API that allowed to instantiate and operate clients and servers and bind callbacks to events. The implementations of packet serialization and de-serialization were often times hidden through encapsulation, missing typings or poorly documented. For instance, the JavaScript library "ws" provided methods for serialization and de-serialization but lacked typings[2]. Typings for this library were made available by the project "DefinitelyTyped", however those did not include the classes relevant for serialization and de-serialization ("Sender" and "Receiver")[3]. At the time of implementing the Python prototype, it used the library "websockets" that offered only an async de-serialization method[4] ("framing.Frame.read"), requiring the use of asyncio which was circumvented by implementing a wrapper around it.
The Python prototype also used the "hbmqtt" library to (de-)serialize **MQTT!** messages. The library used an object-oriented implementation for (de-)serializing **MQTT!** messages where a class for each **MQTT!** message type (e.g. *CONNECT*, *CONNACK*...) inherited from an abstract "MQTTPacket" superclass that defined a "to_bytes" method for serialization and an async "from_stream" method for de-serialization.

[2] The version used for the TypeScript prototype was version 7.0.0, source code is available at https://github.com/websockets/ws.
[3] As can be seen here: https://github.com/DefinitelyTyped/DefinitelyTyped/blob/4bf23527293b2943c7fc12585c21473905a564d7/types/ws
[4] The python prototype used the library at commit 6b5cbaf41cdbc9a2074e357ccc613ef25517dd32: https://github.com/aaugustin/websockets/blob/6b5cbaf41cdbc9a2074e357ccc613ef25517dd32/src/websockets/framing.py

Since the library did not implement a generic method that parsed a byte-buffer and returned the appropriate **MQTT!** message object, this logic had to be implemented as part of the work in the prototype, requiring investigation of the (largely uncommented) source code of the library as its documentation did not cover these internal (de-)serialization methods but focused on high-level use of the API it implemented. From a software engineering point of view, omitting public interfaces to internal (de-)serialization methods and forcing specific programming patterns (such as async programming) are perfectly valid decisions in the context of single, individual modules. However, for those reasons, making use of the "heavy lifting" those libraries performed, was not trivial and came with workarounds and investigating the libraries' source code which in turn took up time during the implementation phases.

Then there were also instances of incomplete documentation: the Python library "websockets" implemented (de-)serialization of **WS!** packets and also implemented the **PMCE!** (**PMCE!**)[5] of the **WS!** protocol. Calling the (de-)serialization methods of the "websockets" library and specifying the use of **PMCE!**, the first incoming and outgoing messages would be compressed correctly, however following messages would be compressed incorrectly. This rendered the prototype useless as **WS!** may use **PMCE!** by default to reduce bandwidth. The library failed to raise exceptions or return error codes so from the prototype's runtime point of view it appeared to work just fine. After investigating the library's source code it was found that the instances implementing the extension were stateful. When supplying newly created instances of said extension implementation to the (de-)serialization methods, they worked as intended, compressing and decompressing any amount of **WS!** packets. This could be due to a multitude of reasons including improper use of the **PMCE!** instances or improper calling of the (de-)serialization methods. No documentation could be found about specifics on those specific topics, though.

For Python libraries, one reason why documentation was in some cases sparse, only documented high-level features and largely omitted in-code documentation (such as comments) might be the "pythonic" approach to writing Python code. "Pythonic" code values readability higher than performance and encourages writing self-explanatory code. While this way of programming may help to understand individual methods or even algorithms that use multiple methods, it does not by itself aid in documentation of high-level concepts or complex interaction. Another reason for sparse documentation in open source libraries might be the developers'

[5] Available at https://github.com/aaugustin/websockets/blob/6b5cbaf41cdbc9a2074e357ccc613ef25517dd32/src/websockets/extensions/pe

focus on implementing more features or improving the code-base instead of aiming for more complete documentation. Contrary to commercial products, there usually are no monetary incentives for developers of open source software to write documentation.

7.2 Qualitative: Deliverables

TBD: Which fit-criteria were met? What is the implementation currently capable of? Which requirements were not full-filled?

Chapter 8

Summary

This chapter provides a summary of the concept shown in chapter **??** and the implementation thereof in chapter **??**.

8.1 Requirements Engineering

8.2 Concept

8.3 Implementation

TBD:

- *Discuss technical debt!*
- *Large config files: bad usability, also confusing*

Chapter 9

Conclusion

TBD

9.1 Outlook

TBD: Discuss specific steps that can be taken to fully implement the concept.

- *UI based config editor*

List of Abbreviations

A/C air conditioner
ARP Address Resolution Protocol
AWS Amazon Web Services
FSM finite-state machine
GDPR General Data Protection Regulation
HMI human-machine interface
HTTP Hypertext Transfer Protocol
ICS Industrial Control System
IIoT Industrial Internet of Things
IoT Internet of Things
IP Internet Protocol
ISP Internet Service Provider
MITM man-in-the-middle
MQTT Message Queuing Telemetry Transport
MTU maximum transmission unit
PLC programmable logic controller
PMCE Per-Message Compression Extension
REST Representational State Transfer
TCP Transmission Control Protocol
OPC U/A OPC Unified Architecture
WS WebSockets

List of Tables

List of Figures

Listings

Appendix A

Diagrams

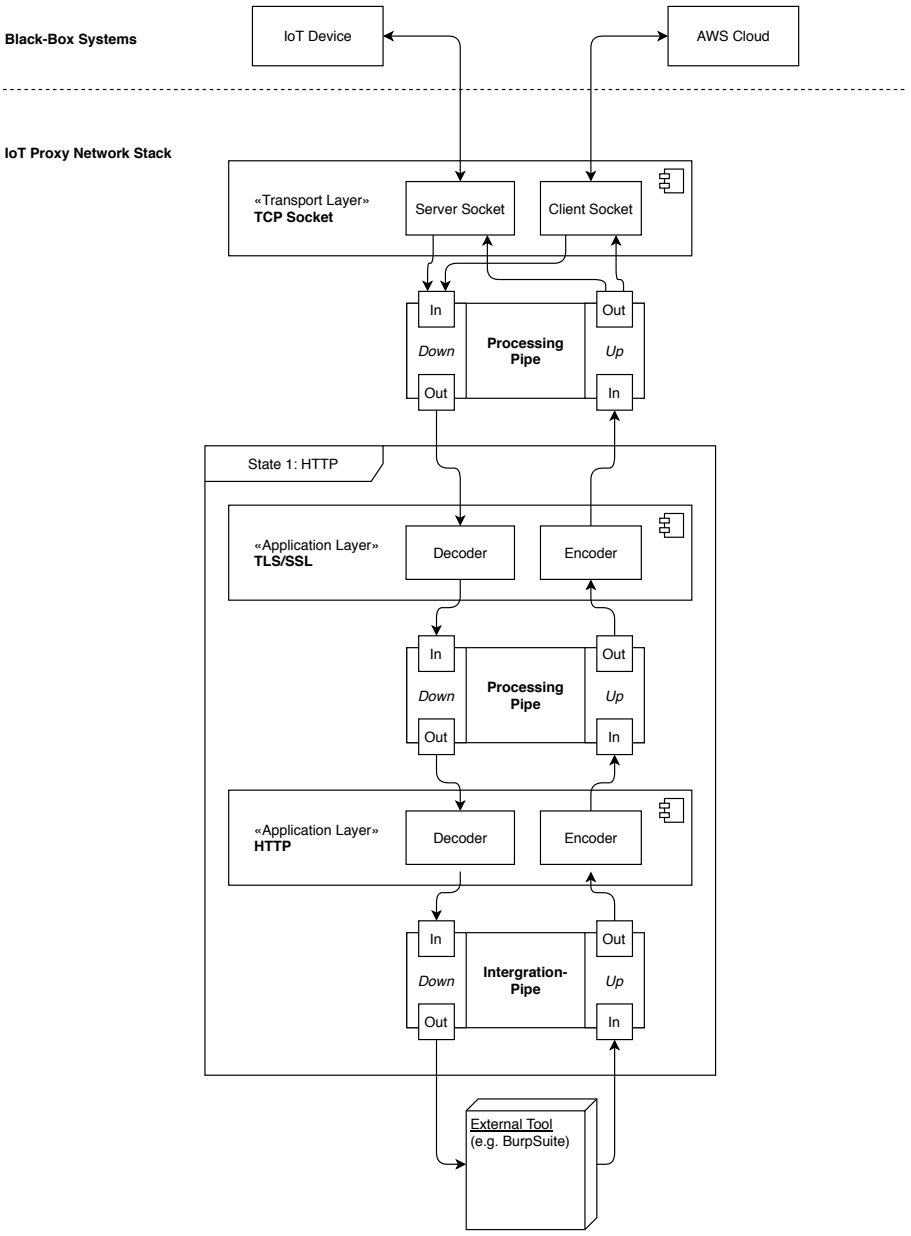


Figure A.1: AWS! IoT! Scenario - State 1: **HTTP!** Server

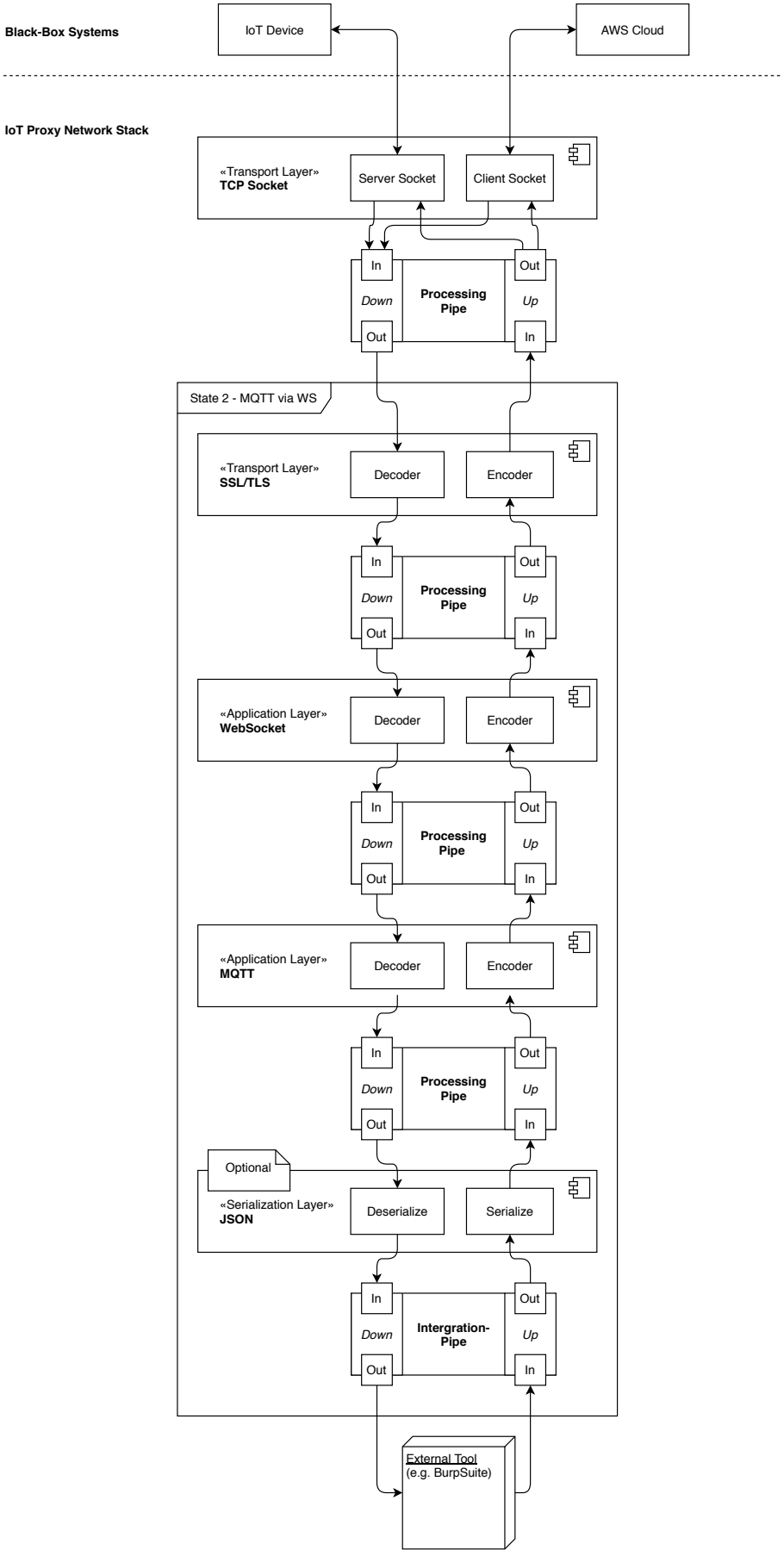


Figure A.2: AWS! IoT! Scenario - State 2: **MQTT!** via **WS!**