Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Dept. of Information Security and Communication Technology

Mathias Fredrik Hedberg

PacketZapper: An Automated Collection and Processing Platform for IoT Device Traffic

Master's thesis in Information Security
Supervisor: Jia-Chun Lin
Co-supervisor: Ernst Gunnar Gran, Ming-Chang Lee
June 2023

NTNU
Norwegian University of
Science and Technology

Mathias Fredrik Hedberg

PacketZapper: An Automated Collection and Processing Platform for IoT Device Traffic

Master's thesis in Information Security
Supervisor: Jia-Chun Lin
Co-supervisor: Ernst Gunnar Gran, Ming-Chang Lee
June 2023

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Dept. of Information Security and Communication Technology

NTNU
Norwegian University of
Science and Technology

Abstract

Smart devices are all around us, and have made their way into our homes, assisting us in various aspects of our daily lives. This raises some questions regarding the security implications associated with such rapid adoption of IoT devices in the home environment. Collecting and analyzing real IoT device traffic for security research purposes is often desirable, but requires knowledge on a wide variety of topics, from selecting appropriate collection hardware, to understanding how to store and process this traffic data. This presents a significant hurdle for researchers, as they must have a thorough understanding of these concepts before they can proceed with their work. Avoiding this process could lead to inconsistencies in their research experiments. In this thesis, we present PacketZapper, an automated collection and processing platform for IoT traffic. It leverages existing open-source software together with some custom components to create a scalable solution for running reproducible experiments with real IoT devices. PacketZapper relies on Apache Airflow to automate all aspects of experiments conducted on the platform, with Elasticsearch serving as the core storage component. Currently, PacketZapper supports collection of Zigbee and generic 433MHz IoT device traffic through commercial USB dongles, with code that facilitates for simple integration of additional IoT protocols in the future. Our requirements-based evaluation of the platform demonstrated that it is capable of performing basic inference experiments, and has suitable tools for analysis and exploration of the collected traffic data. Users should have an understanding of writing pipelines for Airflow before using the platform to better harness the full potential of PacketZapper.

Sammendrag

Smarte enheter blir stadig vanligere i hjemmene våre og hjelper oss med ulike aspekter av hverdagen. Dette reiser spørsmål om sikkerhetskonsekvensene knyttet til den økende bruken av IoT-enheter i hjemmet. Å samle inn og analysere ekte IoT-trafikk for sikkerhetsforskningsformål er ofte ønskelig, men krever kunnskap om mange ulike emner, fra valg av passende innsamlingsutstyr til forståelse om hvordan man lagrer og behandler trafikkdataene. Dette utgjør en betydelig utfordring for forskere, da de må ha en grundig forståelse av disse konseptene før de kan fortsette med sin forskning. Å unngå denne prosessen kan føre til komplikasjoner i forskningseksperimentene deres. I denne avhandlingen presenterer vi PacketZapper, vår automatiserte plattform for innsamling og behandling av IoT-trafikk. Vi bruker eksisterende åpen-kildekode programvare, sammen med spesiallagde komponenter for å skape en skalerbar løsning for gjennomføring av eksperimenter med ekte IoT-enheter. PacketZapper benytter seg av Apache Airflow for å automatisere alle aspekter av eksperimentene som utføres på plattformen, og Elasticsearch fungerer som kjernen for lagring av data. For øyeblikket støtter PacketZapper innsamling av Zigbee- og generisk 433MHz IoT-trafikk ved hjelp av kommersielle USB-dongler, og det er mulig å enkelt integrere ytterligere IoT-protokoller i fremtiden. Vår evaluering av plattformen viser at den er i stand til å hjelpe forskere med å svare på komplekse forskningsspørsmål som baserer seg på bruk av IoT trafikkdata, og har gode verktøy for å analysere og utforske denne dataen. Brukere bør ha kunnskap om hvordan man skriver arbeidsflyter for Airflow før de bruker plattformen, slik at de kan utnytte det fulle potensialet til PacketZapper.

Acknowledgements

I would like to thank my supervisor Jia-Chun Lin for her persistent guidance throughout my work on this thesis. Your initiative in creating a study group for sharing ideas really helped me find that additional motivation needed to complete this project. I would also like to thank my co-supervisors Ernst Gunnar Gran and Ming-Chang Lee for their countless hours dedicated to discussions over Teams.

My coworkers also deserve an acknowledgement, thank you for giving me the time and space to focus on my thesis. You have saved me from countless sleepless nights.

Last but not least, I would like to thank my family for their continued support throughout my studies.

2023-06-01, Oslo Mathias Fredrik Hedberg

Contents

Acronyms

ASK Amplitude Shift Keying. 7

AWS Amason Web Services. 9, 43

DAG Directed Acyclic Graph. 10, 11, 18, 24, 37, 38, 42–45

DSL Domain Specific Language. 9, 23

FSK Frequency Shift Keying. 7

IoT Internet of Things. 1

ISM Industrial, Scientific, and Medical. 6

LR-WPAN Low-rate Wireless Personal Area Network. 6

ML Machine Learning. 43

OOK On-Off Keying. 7

RTM Requirements Traceability Matrix. 27, 28, 30, 33, 39

SaaS Software as a Service. 9

SDN Software Defined Network. 45

SDR Software Defined radio. 7, 8, 11, 22, 41, 42

SoC System on a Chip. 21, 22

STUN Simple Traversal of UDP through NATs. 46

TPMS Tyre Pressure Monitoring System. 7, 14

ZC Zigbee Coordinator. 6, 28, 29, 31, 32, 34, 37, 38

ZED Zigbee End-Device. 6, 29, 31, 32, 37–39, 44

ZR Zigbee Router. 6, 29, 31, 32, 37, 38

Chapter 1

Introduction

Smart devices are increasingly becoming an important part of our lives, both in our homes and the cities we live in. Many of our common household amenities such as lighting and climate control have made a gradual transition over to the Internet of Things (IoT) space, helping not only to reduce our power bills, but also to increase our productivity and general health [1, 2]. With such rapid adoption of IoT, it is important that it is implemented correctly and safely, so that the devices not only function as intended, but also do not add any unnecessary risk to the user, such as stalking or harassment [3]. Researchers have been studying the security implications of IoT, and many studies make use of real-world IoT device traffic when conducting such research. This data may be sourced from existing datasets [4], or be captured by the researchers themselves in their own laboratory setups [5, 6]. Researchers often require different variables than those commonly provided by existing datasets and must therefore obtain additional knowledge on how to capture, store, and process IoT traffic that is relevant to their chosen devices. Such knowledge is often scattered and outdated, making device traffic collection for IoT research purposes more difficult than one may initially expect.

In this thesis, we present PacketZapper, an automated collection and processing platform for IoT device traffic. It was created using mostly off-the-shelf software components and methods, together with some custom-made tooling, to create a scalable platform for jump-starting IoT traffic based research projects.

1.1 Problem Description

With the accelerated adoption of IoT devices in our homes, there has been conducted a fair amount of research on better understanding the privacy and security implications associated with living alongside such technology, and how we can combat these implications [5, 7–12]. Some papers show that researchers are capable of passively understanding user interactions within the home via high-level activity inference using data from IoT sensors scattered throughout the home [8]. However, many of these projects rely on pre-existing datasets of network-traffic that have been pre-processed to allow for easier utilization of the data.

If researchers want to capture their own traffic instead of using existing datasets, then they would have to spend a significant amount of time understanding how this could be achieved and allocate time for procurement of the required software and hardware. At the current state of the art, many available research papers omit detailed descriptions of how their utilized packet data was acquired. Some papers provide only a broad overview of the collection process or limited insight into why certain datasets were selected as their data source [8, 11].

There exist some tools and platforms for conducting both collection and analysis of device traffic, however these are usually tailored to one specific type of traffic, such as Zigbee, or TCP/IP traffic. Most of the components required to put together a complete platform for analyzing data from multiple IoT communication protocols do exist; however, the organization and implementation of these components into a complete system seems to be missing, especially for use in an automated manner. For now, many researchers resort to conducting most of their data collection manually, which can easily result in mistakes, especially if the collection process involves using multiple systems/tools.

1.2 Justification, Motivation, and Benefits

The creation of an automated collection and processing platform could help alleviate some of the issues involved in conducting research based on traffic from IoT devices. By processing, we mean using the data in some shape or form, whether that be performing analysis on the data, running inference tasks, or even interfacing with external systems. Such a platform could significantly reduce the amount of time needed for researchers to go from an experiment idea, to actually acquiring results. Introducing automated pipelines to the platform could help with consistency in experiments, allowing for a higher level of reproducibility, especially in the data collection phase of research. A platform that focuses on being protocol agnostic could see expanded use in the future as it is used together with new protocols, avoiding the need to create completely new tooling from scratch.

1.3 Research Questions

We have created some research questions that capture the essence of what we want to achieve in this thesis. These questions serve as a road-map for our research process and are referenced in later chapters.

RQ1: What is a suitable architecture for a platform that performs automated collection and processing of IoT device traffic?
RQ2: Are there any existing technologies/solutions that can be implemented?
RQ3: How does the proposed solution perform?

1.4 Scope and Contributions

We scope this thesis so that it primarily focuses on targeting traffic from IoT devices and sensors that are used in a home environment. That means focusing on consumer products such as light-bulbs, motion sensors, and other low-bandwidth smart devices. Scalability of the platform is however, still an important factor that is taken into account throughout the whole thesis, highlighting any possible constraints/bottlenecks that may occur when elements of the data capture, storage, and processing components are scaled up.

The intended contributions of this thesis are twofold. Firstly, it is to give readers a better understanding of some of the caveats associated with the current state of the art when performing collection and processing of IoT device traffic. Secondly, it is to propose and test a platform on which such research can be more easily conducted. Together, these contributions could help jump-start future IoT traffic-based research projects, demystifying elements of the research that may be challenging to understand.

1.5 Ethical Considerations

The development and testing of the concepts and techniques described in this thesis involve using real devices in a lab environment in close proximity to other in-use smart-home networks. The target networks are known, and only passive data collection is conducted; however some processes, such as channel scanning involve collecting traffic on additional devices. The privacy implications of this are relatively limited, as the traffic is (in most cases) encrypted, collected over a very limited time span, and we make no attempts to infer any user activity on this excess data.

This begs the question: what would happen if a malicious actor got their hands on the platform created in this thesis? There is a similar argument that can be had regarding hacking tools such as Kali Linux or the Flipper Zero (a multi-functional handheld wireless hacking tool, dubbed the Swiss-army knife for hackers), the latter of which has been banned from import in countries such as Brazil [13].

It is our opinion that due to the passive nature of the platform, the ethics to be considered in this case are not much different from cases involving other existing programs such as Wireshark. However, we must also consider the fact that the platform may lower the entry bar for conducting forms of cyber-stalking and harassment using IoT devices as a vector. The concept of scalability is also central to the design of the platform, and a government organization could see it in their own best interest to mass deploy the platform to monitor and suppress political activists, opposition leaders, or dissidents within their own country. We highlight some of the known weaknesses with the Zigbee protocol but choose not to publish any of the leaked cryptographic keys, as many devices still rely on these secrets in their authentication/encryption functionality.

The code is published as an open-source repository, with documentation and

code examples for basic use cases. There is still some technical skill required to make use of the platform, which makes it a little less attractive for non-technical individuals to use the software. In our opinion, this reduces the chance of it being used as a go-to "one-click" solution for individuals with the sole intent of using the platform for harassment or stalking.

1.6 Thesis Outline

The rest of the thesis is organized as follows. In chapter 2, we present some of the relevant background information related to this thesis, with additional attention given to the Zigbee protocol as it is used in various examples and test cases throughout the thesis. In chapter 3 we present our IoT collection platform i.e., PacketZapper, and the process behind its development. Moving on, in chapter 4, we present how we chose to evaluate the platform, presenting the different test cases we utilized to form a better understanding of its utility in a research context. Next, we have the results in chapter 5, which demonstrate how we completed these test cases and any challenges that arose during their completion. Lastly, we present our discussion and conclusion.

Chapter 2

Background

In this chapter, we present some of the relevant technologies, concepts and existing work that can help provide a more thorough understanding of the topic and lay the foundation for the subsequent discussions. First we take a look at some of the wireless protocols relevant to the thesis, followed by a brief introduction to some of the data processing and storage solutions that we use. Finally we introduce some of the existing work we find relevant to this thesis.

2.1 Wireless Communication

Since our platform is designed to collect IoT traffic, it is important to have a baseline understanding of some of the protocols used by IoT devices for communication. While there are a plethora of protocols available, we focus specifically on the Zigbee Protocol, followed by an introduction to the 433MHz ISM band, which is commonly used by devices throughout the home environment. Increasing our knowledge base on these protocols will also help us in understanding how to capture and parse the traffic they generate.

2.1.1 Zigbee Protocol

Zigbee is a widely adopted wireless communication protocol designed for enabling low-cost, low-power, low-data-rate communication between between devices. It is a central part of many well known smart home ecosystems such as Phillips Hue and IKEA Trådfri. It provides a reliable and efficient way to connect, control and monitor devices such as light bulbs, smart-sockets and environment sensors. The standardization of the protocol means that devices from different manufacturers can interoperate, with some level of security through message encryption and authentication protocols.

Network Architecture

The operation of Zigbee networks is defined by the Zigbee Alliance which is responsible for maintaining and further developing the protocol. Zigbee builds on the foundations of the IEEE 802.15.4 Low-rate Wireless Personal Area Network (LR-WPAN) standard, which also serves as the underlying wireless technology for the Thread[1] protocol. Thread is an integral component of the recently introduced Matter framework[2] and some current Zigbee devices may support upgrading to Thread in the future [14].

There are three main types of devices on a Zigbee network; Zigbee Coordinators (ZCs), Zigbee Routers (ZRs) and Zigbee End-Devices (ZEDs). ZCs are responsible for forming the network and defining logical addresses for devices on the network. In the retail world, this device is often known as the bridge or hub (for example the Phillips Hue Bridge or IKEA Trådfri HUB). Next we have the ZRs which are responsible for routing/relaying traffic between the nodes on the network, also temporarily storing messages for battery powered devices until they come online and request new messages. Usually these are mains-powered devices such as smart-sockets or light-bulbs. Finally we have the ZEDs which do not have to continuously be online or help route traffic, allowing for sleep to conserve battery. These devices are usually battery powered sensors such as thermometers or motion sensors.

Zigbee networks can operate using either distributed or centralized architectures. Distributed networks aim for ease of use and don't have a ZC, while centralized networks aim for higher security, managing encryption through the ZC which acts as a trust coordinator. Much of the current Zigbee security research [15–17] is done on centralized networks.

The Zigbee protocol supports 128-bit AES encryption for message authentication and payload confidentiality, with devices having some encryption keys preinstalled from the factory. While AES in itself is a sturdy encryption algorithm, it's actual implementation in the Zigbee protocol has not been ideal [16]. For example, the default link-key used for device authentication was publicly leaked, introducing a high risk to secrecy of the network [17]. Researchers have also demonstrated that there is a significant amount of information leakage from this encrypted traffic [11, 18].

Wirelessly, Zigbee operates on the 2.4GHz Industrial, Scientific, and Medical (ISM) band, which is the same as 802.11b/g/n WiFi. A Zigbee network will operate on a single channel, from 11 to 26. Many ZCs allow switching network channels to avoid interference from other Zigbee or WiFi networks.

Capturing Zigbee Traffic

Zigbee packets can be sniffed using cheap USB dongles designed for interaction with Zigbee networks. One of more popular dongles is the TI CC2531, which can

[1] Thread Protocol - `https://www.threadgroup.org/What-is-Thread/Thread-Benefits`

[2] Matter Framework - `https://csa-iot.org/all-solutions/matter/`

be had for less than 5 USD from websites such as Ali-Express. It is also possible to decode the packets in software using a Software Defined radio (SDR) paired together with GNU-Radio[3] modules designed for decoding Zigbee [15]. This latter approach requires more expensive hardware, and experience with setting up and using GNU-Radio, a piece of software that has a significant learning curve associated with it.

One important thing to note when capturing Zigbee traffic, is that we do not need to be authenticated on the network to sniff traffic. The full packet structure can be captured, as only the data payload is encrypted. Programs like Wireshark can decrypt payloads from Zigbee packets given that the encryption keys are provided [15].

2.1.2 433MHz ISM Band

The 433MHz ISM (Industrial, Scientific, and Medical) band is a frequency range allocated for unlicensed use in many countries worldwide. It is commonly used for remote control and sensor systems, such as garage-openers, Tyre Pressure Monitoring System (TPMS), weather sensors, and wireless energy monitoring systems. Many of the devices that use this frequency band, make use of simple modulation techniques, such as Amplitude Shift Keying (ASK), Frequency Shift Keying (FSK), or On-Off Keying (OOK). Given the simplicity of these modulation techniques, means the cost and complexity of having such a wireless implementation in a smart device can be significantly reduced.

There are many smart home ecosystems that make use of this frequency band. Some of the more popular systems in Europe include Telldus[4] and Nexa[5]. These companies supply devices such as weather stations, smoke-detectors, light bulbs and thermometers. It is important to note that devices in these ecosystems run proprietary communication protocols, so interoperability between brands is not commonplace. Some research has been done on analyzing the security of smart home ecosystems that utilize these systems [19, 20].

Capturing 433MHz Traffic

Most of the common types of modulation used for 433MHz communication can easily be decoded with handheld devices such as the previously mentioned Flipper Zero, or via a SDR paired with some simple demodulating software. A popular piece of software for decoding such signals is `rtl_433`[6].

`rtl_433` is able to recognize different types of messages being transmitted, and can display that to the user. For example, if `rtl_433` detects a signal coming from a TPMS device, then it will decode the message, and display the actual tire pressure on screen. It is also great for reading and decoding messages from

[3] GNU Radio - `https://www.gnuradio.org/`
[4] Telldus - `https://telldus.com/`
[5] Nexa - `https://nexa.se/`
[6] RTL_433 - `https://github.com/merbanan/rtl_433`

weather-stations, displaying all attributes that the device transmits, such as wind speed or humidity. This piece of software supports many of the most popular SDR devices, including the cheaper variants that are based on the Realtek RTL2832U chipset (which was originally designed for use as a DVB-T tuner). These can be had for around 20 USD from sites like Ali-Express.

2.2 Data Storage and Processing

We now take a look at some of the data processing and storage systems/techniques relevant to this thesis. Having a systematic approach to data management allows for efficient organization, retrieval, and processing of data, saving time and effort in searching for or analyzing data. We start off with an introduction the Elasticsearch search engine, followed by some details on the Apache Airflow workflow orchestration system.

2.2.1 Elasticsearch

Elasticsearch is a highly scalable, distributed search and analytics engine, with fast and efficient analysis capabilities, designed to handle large volumes of data. Elasticsearch is part of a well-known technology suite called the ELK stack[7], which is comprised of Elasticsearch, Logstash, and Kibana. It is built on top of Apache Lucene, which is a high-performance search engine.

Storage Architecture

Data added to Elasticsearch is stored in indexes, which in turn are divided up into shards. The indexes can be compared to SQL tables, while the shards are a way to organize the data across multiple systems. It is important to note, however, that Elasticsearch is a schema-less database. Documents are the basic unit of data in Elasticsearch, which are stored in these indexes. Each document has a unique identifier, which is used to update or retrieve the record. When a new document is added to Elasticsearch (usually in JSON format), each data field is automatically assigned an appropriate data mapping in the index. This data mapping defines the data type of the field (such as text, date, or numeric), helping to optimize the performance of search queries. Elasticsearch infers the data types and properties by analyzing the contents of the document, however, it can make mistakes. It is possible to make these fixes later, but for many use cases, it is best to create an explicit definition of the data types. This definition can be done through an index template, which is a JSON file that defines the data fields for documents added to the index. This can be combined with dynamic data mapping in Elasticsearch to only define a subset of the fields and allow the rest to be auto-mapped by the system.

[7] ELK Stack - `https://www.elastic.co/elastic-stack/`

Scalability

A powerful aspect of Elasticsearch is that it allows horizontal scalability. In general, horizontal scalability is a way to add capacity to a system by adding more computing resources, as opposed to having more powerful machines (known as vertical scaling). This is a key feature of Elasticsearch, allowing it to run as both a single node instance (for example, on a Raspberry Pi) or on a cluster of servers. Also, more compute and storage resources can be added as the amount of data increases [21].

When Elasticsearch is configured as a cluster, data is distributed across the nodes by splitting the index into shards, which are each allocated to a specific node in the cluster. These shards are self-contained indexes, allowing for searches on the index to be completed independently on each shard, before being merged together for the final result.

Open Distro Alternative

Elasticsearch is actively maintained and supported by its founding company Elastic, which offers users and customers the possibility to use its products as a part of their Software as a Service (SaaS) offerings. Elastic has recently closed-sourced their code in an attempt to compete with Amason Web Services (AWS), who also serve the ELK stack as a paid service on their platform [22]. This resulted in Amazon forking the project to create their own open-source offering known as Open Distro[8], which now includes additional features compared to the Elastic version such as security controls and alerts.

Elasticsearch DSL and SQL API

The Elasticsearch Domain Specific Language (DSL) is an extensive syntax for creating powerful Elasticsearch queries. It uses JSON to define these queries, however the queries can also be formed using high-level client libraries such as Python DSL[9], which are often more simple to use. Listing 2.1 shows a functioning DSL query in Python. This query searches for a JSON document in the `data_space` index that has the title of `demo`, which can then be found in the `results` object. For those that prefer the SQL syntax, Elasticsearch provides an SQL translation layer API, allowing users to use the search functionality without having to learn Elasticsearch DSL. It must be noted, however, that the SQL API, while being easy to use, does have some limitations compared to the Elasticsearch DSL.

```
from elasticsearch import Elasticsearch
from elasticsearch_dsl import Search
# Connect to Elasticsearch
client = Elasticsearch('localhost:9200')
# Create a Search object
search = Search(using=client, index='data_space')
```

[8] Open Distro Elasticsearch - `https://opendistro.github.io/for-elasticsearch/`

[9] Elasticsearch DSL (Python) - `https://elasticsearch-dsl.readthedocs.io/en/latest/`

```
# Add query criteria
search = search.query('match', title='demo')
# Execute the search and retrieve the results
response = search.execute()
```

Code listing 2.1: Example DSL Query in Python

2.2.2 Apache Airflow

Apache Airflow[10] is an open-source platform for the management of batch-oriented workflows. It is built on an extensible Python framework, allowing developers to build workflows tailored to their specific technology needs. It is designed to handle large scale data processing and execution of complex workflows.

Core Functionality

Workflows in Apache Airflow are defined as Directed Acyclic Graphs (DAGs). DAGs allow the user to define workflows and data pipelines that are comprised of smaller tasks. The DAGs tailor for specifying the task order, triggers, dependencies, and possible areas of parallelization. Tasks in a DAG represent individual units of work or actions that need to be performed. This could be anything from retrieving data from a database, to invoking an API call.

The DAGs are defined using python code located in the `dags/` folder on the Airflow host. Airflow will automatically load any python files in this directory, and add the DAGs it finds to the system. Listing 2.2 shows an example DAG that is comprised of two functions (`task1` and `task2`) that will run in sequence once a day. Notice that the tasks call `function_1` and `function_2` from an external library, however this could be any python callable.

```
from airflow import DAG
from airflow.operators.python_operator import PythonOperator
from datetime import datetime
from yourcode import function_1, function_2

# Define default arguments for the DAG
default_args = {'owner': 'my name', 'start_date': datetime(2023, 1, 1)}
# Create a DAG object
dag = DAG('demo_dag', default_args=default_args, schedule_interval='@daily')
# Create task instances using the defined functions
task1 = PythonOperator(task_id='task1', python_callable=function_1, dag=dag)
task2 = PythonOperator(task_id='task2', python_callable=function_2, dag=dag)
# Define dependencies between tasks
task1 >> task2  # task2 depends on task1
```

Code listing 2.2: Example Airflow DAG written in Python

Although the DAGs files are defined as code, Airflow has a web interface for managing workflows, checking task status, viewing logs, and other troubleshooting functionality. The interface is a great tool for visualizing the DAG files in a

[10] Apache Airflow - `https://airflow.apache.org/`

graph view, expanding the tasks as nodes, and dependencies between them as edges. It is important to note that the web interface does not cater to the creation and modification of DAG files, so this must be handled externally.

Scalability

Much like Elasticsearch, Airflow can make use of a distributed architecture for horizontal scaling. New nodes can be added to the cluster to handle additional computational requirements. In scenarios where the inter-component message load is extra high, the message brokering component can be outsourced to other services such as Apache Kafka, which is known for its high performance as a message broker. If there is a Kubernetes cluster available, then it is possible to utilize the Kubernetes Executor[11] to run each Airflow task in its own Kubernetes pod.

2.3 Related Work

There are many existing projects that rely on real IoT traffic to perform different types of inference and analysis tasks. However from our understanding, there are none that propose a platform for collection and processing of IoT traffic in a way that is protocol agnostic, meaning that they instead focus on specific forms of IoT traffic such as Zigbee, Bluetooth, or plain TCP/IP.

The authors of "Information Exposure From Consumer IoT Devices" [5] set up multiple IoT device labs, and created a platform for automating the testing of devices. This test infrastructure allows them to remotely control Android devices connected to the network. Their collection is limited to TCP/IP traffic passing through the router, and all traffic analysis is done manually using tools like tcpdump and Wireshark. The paper presents no method for analyzing the wireless traffic directly.

The authors of "IoTSpy" [9] have a different approach to collection, and use dongle-based sniffers to collect wireless traffic from Zigbee devices. They make use of the CC2531 Zigbee dongle to eavesdrop on the Zigbee devices in an attempt to infer user activities. They do mention Z-wave as a potential attack vector, but did not perform any experiments with this type of traffic data. All collection and analysis of this traffic was done manually.

The authors of the "Zigator" paper [15] propose a platform for analysis of Zigbee traffic. The code for their platform is published as an open-source project on Github[12]. They make use of an SDR device to not only passively sniff traffic data, but also perform active attacks on Zigbee networks. Their code has been utilized in additional works analyzing the security of such networks [23, 24]. Zigator is limited to the analysis of Zigbee networks, and requires the user to have a thorough understanding of complex SDR control tools such as GNU-Radio. The

[11] Airflow Kubernetes Executor - `https://airflow.apache.org/docs/apache-airflow/stable/core-concepts/executor/kubernetes.html`

[12] Zigator on Github - `https://github.com/akestoridis/zigator`

authors of the "Zleaks" paper [16] built on the knowledge gained from analysis of Zigbee networks in the Zigator paper to find methods of inferring user activity through passive collection. They also do not require the use of an SDR, employing a CC2531 USB dongle instead.

The "Peek-a-Boo" paper [11] is possibly the closest to what we want our platform to look like in terms of collecting traffic data from multiple types of traffic sources. They collect and analyze traffic from Zigbee, WiFi and Bluetooth. While their contributions in regard to analyzing this traffic data are significant, they omit all details regarding the capture process, and seem to manually process the collected data.

While there have been significant contributions to the space of analyzing IoT traffic for security research problems, it seems like this progress still relies on a heavy amount of manual labor. Also, the platforms that automate some aspects of this collection and analysis, are tailored for specific IoT protocols.

Chapter 3

Design of PacketZapper

PacketZapper is the name given to our automated collection and processing platform for IoT traffic. It is capable of collecting, parsing, storing, and processing various types of IoT traffic in near real-time, simplifying the process of systematically acquiring traffic for research purposes. PacketZapper is built using existing and well-known open-source solutions, together with some custom components, creating a scalable and customizable platform that can be tailored for use in a variety of IoT traffic-based research applications. It can run on a wide range of hardware, from a single Raspberry Pi to a cluster of high-powered servers. The source-code is published online on Github[1] for anyone to use. In this chapter, we detail the development process, including requirements for the platform and the resulting technical implementation.

3.1 Development Process

The development of PacketZapper was structured using the reuse-based software development methodology [25]. The methodology focuses on identifying where and how existing resources can be reused, while also making enhancements during development to ensure that the end product functions as intended.

Our first step in this development process was to ensure that we had a proper understanding of the problem, which we then used to create a solution structure. This structure could then be broken down further to create a more detailed understanding of how each of the components would function with one another. We could then develop a reuse plan to identify which problems could be solved with existing software, and which parts needed more specific code contributions.

This formalized approach to the development of the platform helped ensure that the resulting product sufficiently targeted the goals of our platform, as highlighted by our research questions.

[1] PacketZapper source code - `https://github.com/PacketZapper/PacketZapper`

3.1.1 Solution Structure

Our solution structure was based on our understanding of the problem, which we discussed in Section 1.1. This also ties well into the first research question (RC1: What is a suitable architecture for a platform that performs automated collection and processing of IoT device traffic?), which captures the essence of what we wanted the platform to do. The earlier scoping of the project was an important factor when performing this structure modelling. We decided to limit the platform to target low bandwidth home IoT devices, meaning that we could avoid having to scale every component to deal with the possibility of multi-gigabit data-streams. Horizontal scaling was however still an important factor, as we did not want to directly limit the *amount* of devices the platform could handle at once.

An existing survey [26] on traffic analysis concepts, together with our problem understanding and scope, served as a baseline for breaking down the problem into smaller components. The result of this was a four-stage solution structure. The four stages are identified as **collect**, **parse**, **store**, and **process**. Figure 3.1 shows this four-stage solution structure. Together they make up what is required to perform automated collection and processing of IoT device traffic. The sequential arrows from stage 1 to stage 4 show the general flow of data, while the arrow back to the first stage from the process stage signifies a form of feedback loop, illustrating the automation aspect of the platform.

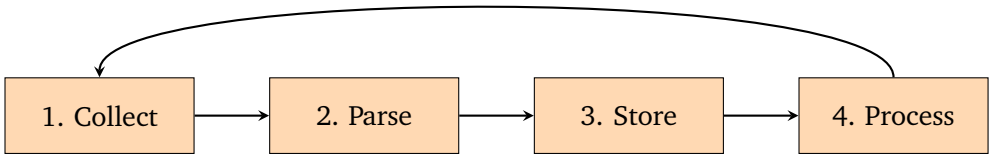


Figure 3.1: The four-stage solution structure of PacketZapper.

Moving forward we break down each of these stages, describing their intended functionality, which we later used to create the system requirements for the platform.

Collect

The role of the first stage is to collect the data that will be processed. This could be anything from wireless Zigbee traffic, to the signals coming from a TPMS sensor. This component handles how the platform is able to interface with the real world. In essence, it bears the role of demodulating signal data so that the actual information being transmitted/modulated can be extracted. Looking at the OSI model, this stage can be seen as filling the role of layer 1 in the model, receiving the raw data that is sent over a physical medium.

Parse

The second stage involves parsing the collected data and extracting as many useful data points from the collected bit-streams as possible. Parsing the raw bit streams into a structured format enables the data to be indexed and queried based on it's different fields and attributes, especially if saved in a database. Ideally, this stage would result in an OSI-like data structure, outputting information on any protocols that the collected data is comprised of.

Store

The store stage involves committing the parsed data to some storage solution so that it can be processed at a later time. The store stage makes use of a database, allowing a subset of the data to be retrieved based on the requirements of the querier, such as the collection time, or some other field in the data structure. The store component needs to be able to scale based on both the amount of data being committed and also the complexity of the queries being performed.

Process

The process stage is where the users of the platform are able to experiment with the collected data. Ideally, this means creating an environment for collaboration between users and enabling the creation and execution of reproducible experiments. This stage should also provide an interface for manual exploration and analysis of the collected data.

3.1.2 System Requirements

We created some functional and non-functional system requirements that were derived from our solution structure. We followed a black-box style for requirement specification as outlined in [27].

Functional Requirements

In general, functional requirements refer to the specific features and capabilities that a software system must have to perform its intended tasks. In the case of PacketZapper, these requirements define the functionality of the four stages of our solution structure, and define some of the interfaces between them. Table 3.1 lists all of these functional requirements for PacketZapper. Note that the identifier value for each item starts with FR, indicating a functional requirement.

These requirements are quite general and do not specify exactly which technologies or tools the platform should use. We are however specific on having the platform evolve around a JSON based data structure.

Table 3.1: Functional requirements for PacketZapper.

Identifier	Requirement description
FR001	Implementation conforms to the four-stage solution structure (collect, parse, store and process)
FR002	Can handle collection of multiple data sources simultaneously.
FR003	Supports collection of at least 2 different wireless IoT communication protocols
FR004	Implements an API for remotely controlling the collection of data
FR005	Collection of data can run at physical location separate from other platform components
FR006	Parsing component outputs timestamped JSON formatted data
FR007	Parsing component supports basic filtration of traffic
FR008	Data storage solution provides graphical interface for verifying data collection/availability
FR009	Data storage solution can handle time-sequence based JSON formatted data
FR010	Data storage solution provides interface for querying data based on the data attributes
FR011	Multiple users can collaborate on creating data pipelines
FR012	Pipelines can interact with the collection API (as mentioned in FR004)
FR013	Complete platform is capable of executing basic inference attacks

Non-functional requirements

While functional requirements define the features and capabilities of the platform, the non-functional requirements specify how well it should do it. Measuring non-functional requirements can be a challenging task as they often involve qualitative aspects that are difficult to quantify objectively. Despite this, integrating non-functional requirements into the design process could help ensure that the platform meets the necessary quality attributes and constraints to achieve its intended purpose. Table 3.2 lists all non-functional requirements for PacketZapper.

Table 3.2: Non-functional requirements for PacketZapper.

Identifier	Requirement description
NFR001	The platform should be easy to install, including initial configuration.
NFR002	The platform should have a suitable approach to enabling automation of workflows.
NFR003	The platform should be scalable and capable of running on a variety of hardware.

Note that the non-functional requirements are identified with the "NFR" prefix. These non-functional requirements guided the selection of appropriate design patterns, architectures, and technologies during the platform implementation process.

3.2 Implementation of PacketZapper

In this section, we present the details on the actual implementation of our automated collection and processing platform for IoT device traffic, i.e., PacketZapper. We start with an overview of PacketZapper and then break it down into more detail on the different components it is comprised of.

3.2.1 Implementation Overview

PacketZapper implements the four-stage solution structure described in Section 4.1. The platform has three core components that handle all four stages of this solution structure. In brief, we have the Collection Agent, which is custom software for handling the collection and parsing of IoT traffic data (i.e., stages 1. collect and 2. parse). The next component is Elasticsearch for storing and searching the data (i.e., stage 3. store). The last component is Airflow for processing the data (i.e., stage 4. process). These components were chosen based on their perceived ability to satisfy the requirements of the platform, such as the requirement of having scalable components. Figure 3.2 shows a brief outline of how these core components fit into the four-stage solution structure introduced in Section 3.1.1.

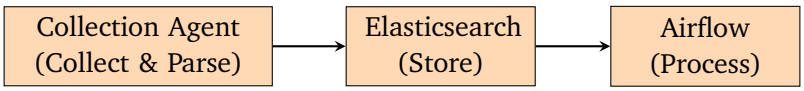


Figure 3.2: The implementation architecture of PacketZapper.

In addition to these three core components, we have Jupyter Lab for prototyping, management, and general administration of the platform, and Kibana for easy visualization of the data stored in Elasticsearch. Together, all these components make up the PacketZapper platform.

Platform Architecture

Figure 3.3 shows a logical overview of how PacketZapper is designed to be deployed. We have one or many remote nodes located in the target IoT environments, then the backend, which runs many of the core components on a server, and finally the user environment, where the platform is administered from.

The Collection Agent is located in the target environment and collects a given type of IoT traffic using one or many USB dongles specific to the type of data being collected (for example, collecting Zigbee data using a CC2531 USB dongle). It performs some basic filtering and parsing of the data before sending it to the

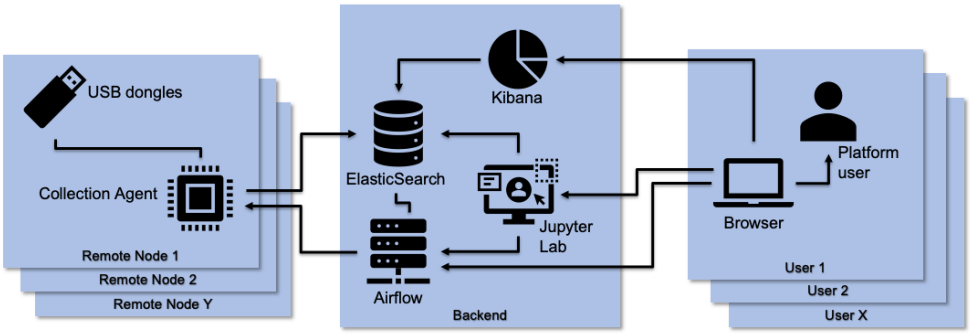


Figure 3.3: The logical deployment overview of PacketZapper.

backend where Elasticsearch takes care of storing the data. The Kibana interface can be used to view the data that is stored in Elasticsearch, create visualizations or find new trends in the data. Jupyter allows the user to test out Python code snippets, and make modifications to the DAGs that are run by Airflow.

The Airflow server is, in essence, the core engine of any research experiment run on PacketZapper. It pulls the levers, controlling the sequence of operations in the experiment. It has full control over the Collection Agents and can start/stop/-tune the collection based on the results it sees in the database. It can also call on external systems or services that an experiment may require. This all happens through DAG files which allow the user to specify in detail the complete sequence of events for their chosen experiment/pipeline.

Example Usage

Figure 3.4 shows an example of the sequence of events involved in a simple Zigbee packet capture-based experiment that is automated through an Ariflow DAG. The user triggers the pre-configured DAG which initiates the Collection Agent to start capturing Zigbee packets. The agent periodically posts its collected data to Elasticsearch as packets are being captured. Meanwhile, Airflow is checking the Elasticsearch database for a specified amount of data to arrive before running an inference task on the data. Airflow then posts these results to Elasticsearch and requests the Collection Agent to stop sniffing traffic. This could then be expanded to for example trigger a new inference task, or start collection of a different network. All this can be specified by the user in the DAG file.

Core Infrastructure

PacketZapper is in its default configuration packaged to run inside a Docker runtime environment. This simplifies much of the setup procedure as the individual Dockerfiles handle installation of dependencies and so on, while the docker-compose files handle networking, storage, and basic configuration of the system. This allows for quick and easy installation of the platform while also giving users the

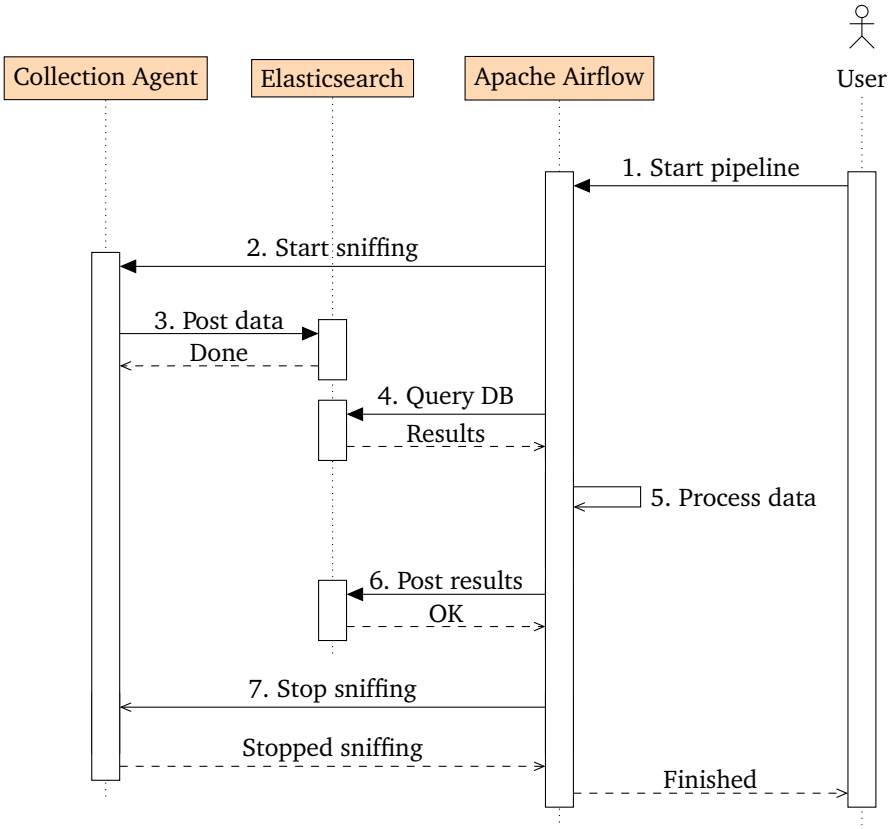


Figure 3.4: PacketZapper example use case for sniffing and analyzing Zigbee traffic.

flexibility to customize their installation as desired, whether that be scaling up individual components or adding new components.

Being built on Docker also reveals all the details of how PacketZapper can be installed on bare-metal (non-virtualized) environments. Users with dev-ops experience could adapt the code to run on a variety of hardware, even outsourcing parts such as the Elasticsearch component to a third-party cloud provider.

3.2.2 Collection Agent

The Collection Agent is responsible for collecting and parsing IoT traffic. The agent relies on physical hardware such as USB dongles to collect traffic, and then uses software to parse the data coming from these dongles. This traffic data is then formatted as JSON strings before being sent to the Elasticsearch database. The Agent runs a lightweight REST API built using the Python FastAPI[2] framework. This allows full remote control of the Collection Agent from an external system.

The agent was designed to function separately from the rest of the platform

[2]FastAPI framework - `https://fastapi.tiangolo.com/`

and can be scaled out as multiple instances. The current version supports sniffing typical Zigbee traffic (via whsniff and a CC2531 USB dongle) and also generic 433MHz data (via rtl_433 and an RTL_SDR dongle). Future dongle/protocol support is planned.

Core Infrastructure

The Collection Agent was written in Python (version 3.9) and is packaged to run as a module from the command line. We wrapped the code to run inside a Docker environment by default, but it can easily be moved out to run directly on the host. The included Dockerfile defines how the package is built, and details all the external dependencies it relies on. Since the code runs in Docker, it can be installed on various types of machines, from am ARM based Raspberry Pi, to a powerful x86 based server. We used docker-compose to create an environment for the code to run in, defining variables such as the Elasticsearch server endpoint, the API listen port and so on. It is possible to list all configuration options by calling the `-h/--help` flag in the command line.

Having the code written in Python should make it simple to add additional dongle/protocol support. The code is also laid out in a manner that should make it easy to understand this process. Our main limitation here is the use of UNIX pipes, so adding Windows-only parsing software is not supported.

USB Device Passthrough

Running the Collection Agent in Docker means that it is mostly isolated from the host system, USB devices included. While there are methods to pass through USB devices to running Docker containers, it is unfortunately not possible on all host operating systems. For example, macOS runs Docker inside a virtual machine, making device passthrough even more complex. In the docker-compose file that is included with the Collection Agent, we run the container in privileged mode and pass through the whole USB bus so that any USB devices attached to the host system can be detected and used by the agent. From testing, this works fine in Linux-based operating systems. Users of macOS would have to run the agent directly on the host.

Elasticsearch Connection

When running the agent, the user must provide the Elasticsearch endpoint and also ensure that the agent is reachable by the Airflow service. In the default runtime configuration via Docker, this connectivity functions out of the box. However, custom setups such as running the agent in a remote location would require some modification to the network layer, such as creating an encrypted network tunnel between the locations (see Section 6.1.2 for more details).

The agent has a built-in index template that is posted to the Elasticsearch server at startup. It applies rules for any data uploaded to the "packetzapper"

index, which is the index used by the Collection Agent for all data. This file can be customized to statically define special variable types if Elasticsearch is unable to guess the type automatically. In its current configuration, this template will ensure that numeric values formatted as strings, are interpreted as numeric values, and not strings. This ensures that aggregation queries can be done using these values (as functions such as `sum()` work as intended). The file is included in the Appendix at A.3.

Data is sent in batches to Elasticsearch in groups of 10-50 items at a time. This is necessary to avoid sending a deluge of HTTP requests to the instance when large amounts of data are being collected. Currently, the batch size is hard-coded for each protocol being collected. However, it would be more optimal for the program to dynamically adjust the size based on the variations in the influx of data being collected. This is especially important in scenarios where only a couple of packets are being collected per minute, as the whole buffer must be filled before the batch is sent.

Control API

The API runs on port 8000 of the host and exposes endpoints for starting, stopping, and checking the status of the sniffing processes running on the agent. The API uses HTTP POST methods for any actions that make state changes to the system (for example, start/stop collection) and HTTP GET methods for basic information calls (such as checking the health of the service).

The endpoints for starting the collection of data use the POST data payload to specify options that would be relevant for that type of collection. For example, when starting Zigbee sniffing, the user can specify which channel to sniff, and if none is specified, then it will fall back to a default channel value.

There is also an endpoint at `/docs` which exposes the auto-generated documentation, describing in detail the API schemes for all the available endpoints. This is done using OpenAPI (formerly Swagger) and allows easy debugging of the endpoints through POST and GET requests, which can be run directly from the web browser.

The source code is written in a way that should make it easy for a developer that has some experience with Bash and Python to easily add their own endpoints for additional services they want to expose over the API.

We also created a basic Python client library for interacting with this API. This allows for easier integration of the Collection Agent with Python projects, without having to study the documentation for the different endpoints.

Zigbee Support

Zigbee support was built and tested using USB dongles based on the TI CC2531 System on a Chip (SoC). This somewhat outdated chip is still prevalent and can be purchased online in a USB-dongle form factor for under 5 USD. The chip must be flashed with an alternate sniffing firmware, which can be downloaded directly

from TI[3]. Unfortunately, flashing the SoC must be done via external hardware, such as a CC-debugger, which is a special hardware tool for debugging and programming SoCs. Many alternate ZigBee projects use the CC2531 and have detailed descriptions of how to flash the chip. The Zigbee2MQTT[4] project, for example, has up-to-date and detailed instructions on this flashing process for most operating systems.

The open-source Zigbee sniffing software whsniff[5] is used together with the CC2531 dongle (flashed with the alternative firmware image) to sniff and decode Zigbee traffic into PCAP format. By piping its output to tshark, the packets can then be exported in JSON format for ingestion into the database. As the original data from whsniff is in PCAP format, the whole OSI structure for the packets is preserved, even when converted to JSON format. We make no attempts to decipher the encrypted Zigbee payload; however, all the headers for the different layers of the packet structure are still readable.

433Mhz Support

Generic 433Mhz message decoding was implemented using a Nooelec NESDR SMART[6] dongle together with the open-source `rtl_433` software package. `rtl_433` leverages generic SDR receivers, such as the Realtek RTL2832U chipset, to decode a variety of common wireless protocols from devices such as weather stations, temperature sensors, garage door openers, and remote controls.

In the lab environment, running `rtl_433` revealed many neighboring devices. The dongle discovered a handful of sensors that published environment data, such as temperature and humidity, via the Nexus temperature and humidity sensor protocol. Often, these are simple weather stations, but verifying their brand and model were not possible as they were not a part of the lab.

`rtl_433` can export live JSON data, which we can ingest into the Elasticsearch database on the fly. This is done by adding the `-F json` flag to the argument list of `rtl_433`. The software will output a JSON structure to STDOUT every time a 433MHz signal is detected and decoded. `rtl_433` supports a large amount of command line arguments to for example, limit collection to a specific device type, or signal strength. It is therefore possible to add extra arguments to `rtl_433` when starting it over the Collection Agent API by specifying the `EXTRA_ARGS` variable when sending the POST request to start sniffing.

3.2.3 Elasticsearch

We run an instance of Elasticsearch to store the traffic data coming from the Collection Agents. Elasticsearch was chosen as the main storage system mainly due to its ability to handle a variety of data structures through its dynamic data mapping

[3]TI PACKET-SNIFFER - `https://www.ti.com/tool/PACKET-SNIFFER`
[4]Zigbee2MQTT - `https://www.zigbee2mqtt.io`
[5]homewsn/whsniff - `https://github.com/homewsn/whsniff`
[6]Nooelec NESDR - `https://www.nooelec.com/store/sdr/sdr-receivers`

functionality. Also, its advanced DSL query functionality opens up the possibility for different forms of search aggregation. For example, it is possible to define queries that find the average packet size being transmitted over Zigbee in groups of 10-minute intervals, or take this a step further by grouping this data based on the source/destination address.

Index Structure

We do not restrict incoming data to conform to any specific JSON based data structure as we did not want to limit the types of data that could be added to the system. If a user decides to add a new protocol for collection to the system, then they are free to structure the data into a format they feel suitable (given that Elasticsearch is capable of accepting the data). As mentioned earlier, we make use of an index-template to control how Elasticsearch reads the data we feed it. In our current configuration, all data is added to the same `packetzapper` index, irregardless of the protocol being collected. When looking at the data in Elasticsearch, the user can differentiate between the different protocols by filtering on the `sniffer` key in Elasticsearch. For example, Zigbee traffic is given the `sniffer` value `whsniff`.

Kibana Visualizations

We include an instance of Kibana in PacketZapper to allow for easier exploration and visualization of the data stored in Elasticsearch. Kibana provides a user friendly interface to create powerful graphical representations of the data, with additional aggregation and filtering functionality. These visualizations can then be combined together in dashboards, assisting the user in identifying trends and getting more insight into the data. There is also search functionality built into Kibana, aiding the user in creating intricate search queries that use functionality such as fuzzy-matching and range queries.

If the user creates a filter or visualization that is relevant to some task that they would like to automate, then it is possible to export these queries as DSL, and run them directly against the Elasticsearch instance through a Python script or similar.

We do not include any default Kibana visualizations as we feel this is often very specific to the task being performed, however we may include some basic example visualizations in the future to help show the user what is possible.

3.2.4 Apache Airflow

We use Apache Airflow as the main workspace for running pipelines of tasks, everything from initiating the collection of data on the Collection Agent, to calculating and presenting inference results. This is done through DAG files that are run by Airflow. These define the sequence of events to be performed as part of a pipeline, representing the automation aspect of PacketZapper.

Instance Architecture

PacketZapper runs an instance of Apache Airflow through a Dockerfile based on the official Airflow Docker image. It is through this Dockerfile that users can define additional dependencies needed for their DAG tasks. For example, we add the Python Elasticsearch packages through this file. We make use of the official docker-compose.yml file to initiate a standard working environment. An internal bridge network is also present between Airflow and all the other components (Jupyter, Collection Agent, and Elasticsearch) so that they can communicate freely with each other. The docker-compose includes internal services for Airflow such as a Postgres database instance and a Redis server. The DAGs folder (where the DAG files are stored) is bind-mounted to the host system so that they can be modified both on the host operating system and also via the Jupyter instance.

PacketZapper Integration

We provide an example DAG to showcase some of the interactions that are possible using Airflow in PacketZapper. The DAG is located in the `/dags/packetzapper` directory and serves as a great baseline for potential collection experiments on the platform. It is here that we demonstrate the functionality of the Collection Agent Python client library implementation, which we introduced earlier in Subsection 3.2.2.

The library can be used to control one or more Collection Agents. An instance of the `PacketZapperClient` class is initiated with a list of Collection Agent URLs, which can then be controlled using this object. It exposes methods for various functions, such as starting collection with specific parameter, checking that an agent is online, and stopping collection. Listing 3.1 shows a basic example of how this library is used for starting Zigbee traffic collection on Zigbee channel 12:

```
from packetzapper.packetzapper import PacketZapperClient
# Initiate Collection Agent client using a list of hosts
pz = PacketZapperClient(['http://192.168.1.100:8000'])
# start Zigbee sniffing on channel 12 if Collection Agent hosts are online
if pz.ping_hosts:
    pz.start_whsniff(channel=12)
```

Code listing 3.1: Using the Collection Agent client library to start Zigbee traffic collection

We also show some basic Elasticsearch functionality in the DAG, such as searching in the Elasticsearch database, but there are far more possibilities than this. For example, we can use Elasticsearch as an Airflow Sensor task to wait for X amount of data to show up in an Elasticsearch index before continuing with the next task. This is especially useful for checking that a required amount of traffic data has been collected before continuing on with an inference task.

Being that Airflow can run standard Python code as an operator task means that other external services can be called as a part of the pipeline. We were for

example successful in switching the power of an IoT-enabled relay switch (like an AC smart socket) on and off by calling the device's built-in API.

3.2.5 JupyterLab

A JupyterLab instance is included in PacketZapper to allow for remote management of the system through a web browser. By default, it has access to all the configured services and is a great platform for prototyping elements of a collection pipeline before creating DAGs for Airflow. Jupyter exposes a command line for easy installation of new Python packages or other system dependencies. Collaboration is also possible by having multiple users connected to the Jupyter instance.

The `/dags` folder from Airflow is mounted to the Docker image and visible in the directory structure view shown in the sidebar. If creating or modifying the DAG files, it may be a good idea to install the "apache-airflow" package from pip to make development easier (IDE code completion, etc).

Although JupyterLab is a great tool for prototyping, complete management of the system must still be done through a host shell. For example, starting/stopping/building the Docker containers should be done directly on the host system. It is possible on Linux systems to pass through the Docker socket to the JupyterLab container and run the commands from there. However, this is something we did not implement in an attempt to make the platform as host-platform-agnostic as possible.

Chapter 4

Evaluation Methodology

In this chapter, we detail the evaluation methodology used to assess the performance of PacketZapper, which was guided by the functional and non-functional system requirements. We designed and ran a series of test cases targeting these requirements, which resulted in a pass/fail grading. This evaluation process included setting up a small lab environment comprised of IoT devices and basic network infrastructure. This allowed the test cases to be performed using real-world devices.

4.1 Platform Evaluation

Assessing the performance of PacketZapper was crucial in determining whether PacketZapper's intended objectives had been accomplished. The evaluation process consisted of a requirements-based assessment of PacketZapper, including a real-world case study to assess its suitability in a specific research context.

The requirements shown in Subsection 3.1.2 served as the baseline for creating test cases for PacketZapper. Each individual test case was designed to tailor to one or more of these requirements, bringing us closer to a slightly more quantitative assessment of the platform. These test cases were placed in two Requirements Traceability Matrices (RTMs). Each row in these tables represent a unique test case with an identifier and description, along with the requirements that the test case targets. The matrices provided the necessary structure needed to complete such an evaluation process.

4.1.1 Functional Requirements Evaluation

Table 4.1 shows the RTM with test cases targeted specifically at the functional requirements of PacketZapper. More details on each of these test cases is provided below.

Table 4.1: RTM used for evaluation of the functional requirements for PacketZapper.

Identifier	Test case summary	Reqs served
TC001	**Design review:** Review design documents to assess if the platform conforms to the four-stage solution structure	FR001
TC002	**Parallel collection:** Demonstrate filtered collection of Zigbee and 433Mhz IoT traffic simultaneously via the collection API that is running on a separate system. Verify data availability in database.	FR002-008
TC003	**Query capabilities:** Demonstrate that a single JSON document containing a Zigbee packet can be retrieved from the database based on a time and attribute query	FR006, FR009-0010
TC004	**Case study:** Run a case study to demonstrate a basic inference attack on a Zigbee network	FR011, FR012, FR013

TC001: Design Review

The intention of this test case is to assess if PacketZapper conforms to the four-stage solution structure described in Subsection 3.1.1 and review if the application of this structure has enabled the structure's expected benefits (reuse of existing software, module replacement, scalability, etc). This is a qualitative assessment that results in a pass/fail score.

TC002: Parallel Collection

In this test, we demonstrate that parallel collection of IoT traffic is possible, and that the collection can be started/stopped from a machine other than the one conducting collection. To evaluate all the requirements targeted for this test, the following must be demonstrated:

1. Start the collection of Zigbee traffic via API, filtering out packets sent from the ZC.
2. Start the collection of 433Mhz traffic via API, while Zigbee traffic is still being collected.
3. Stop the collection of both traffic sources via API.
4. Provide a graphical time-series representation of data from both traffic sources.
5. Confirm that no packets originating from the ZC are present in the data.

The test is passed if all items are demonstrated.

TC003: Query Capabilities

In this test, we demonstrate some of the query capabilities of the storage solution. For this test, we assume that Zigbee traffic data is in the database from the previous test case (TC002). To evaluate all the requirements for this test case, the following must be demonstrated:

1. Query the database for the last collected Zigbee packet, and verify that the database has retained the structure of the packet.
2. Query the database for the sum of bytes of Zigbee packets transmitted over the network in 10-minute intervals, grouped by the transmitting Zigbee device.

TC004: Case Study

This case study is designed to assess if PacketZapper is capable of implementing some of the passive inference attacks described in the Zleaks [16] paper. This assists in grounding the evaluation of PacketZapper to concepts and methods that are currently in use by researchers today. PacketZapper by design, is agnostic to how the components are used, and it is up to the user to implement their own custom workflows and solutions. The purpose of this case study is to assess if the chosen approach to collecting, storing, and analyzing data is suitable for real-world scientific research on IoT device traffic. The Zleaks paper has significant contributions with regard to inferring user activity; however, we limit this case study to only complete some of the more basic inference tasks described in the paper.

In this case study, PacketZapper is used to:

- Identify the total number of Zigbee devices on the network
- Identify the number of active ZR on the network
- Identify the number of active ZED on the network
- Identify the network ZC

Using PacketZapper to complete these tasks involves creating custom airflow pipelines, which will require some programming knowledge. The test case is passed if the platform is capable of running these inference tasks in an automated manner.

4.1.2 Non-functional Requirements Evaluation

We have not designed any quantitative test cases to directly assess the non-functional requirements; however, qualitative assessments can be done to target them individually. Therefore, we have three test cases as shown in Table 4.2.

Table 4.2: RTM used for evaluation of the non-functional requirements for PacketZapper.

Identifier	Test case summary	Reqs served
TC005	**Easy installation:** Review installation and configuration procedure	NFR001
TC006	**Automation:** Review automation functionality of PacketZapper in the context of typical use cases	NFR002
TC007	**Scalability:** Assess scalability availability and limitations of PacketZapper	NFR003

TC005: Easy Installation

In this test case, we review the installation procedure and evaluate if it serves the requirement NFR001: "The platform should be easy to install, including initial configuration". Extra attention is given to the default configuration and what further modifications must take place to be up and running.

TC006: Automation

In this test case we review the usefulness of the automation functionality provided by PacketZapper, and we evaluate how well it serves the requirement NFR002: "The platform should have a suitable approach to enabling automation of workflows". Additional focus should also be given to the ability to debug and monitor any automation attempted on PacketZapper.

TC007: Scalability

In this test case we assesses the scalability of PacketZapper, including any hardware limitations. This assessment should better highlight if PacketZapper serves the requirement NFR003: "The platform should be scalable and capable of running on a variety of hardware". Additional focus should be given to identifying potential indicators of bottlenecks that may arise as the platform load increases, and how these can potentially be mitigated.

4.2 Lab Environment

A lab environment was set up in a residential apartment housing students from NTNU. The apartment and lab environment were not RF-isolated, so wireless signals from devices in neighboring apartments were still present in the lab. This included signals from smart infrastructure such as power meters, weather stations and other Zigbee networks.

The lab was designed to replicate a typical smart home installation while also allowing for additional insight and modifications where needed. Figure 4.1 shows

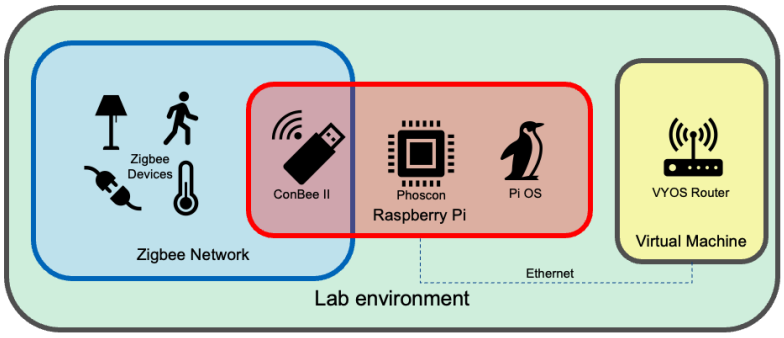


Figure 4.1: The lab environment for evaluating PacketZapper.

an overview of this setup. A virtual machine running the VyOS[1] router operating system was connected to the internet on a local isolated subnet. Basic services such as NAT, DHCP, and DNS were configured in addition to a basic firewall configuration. A Raspberry Pi 3B running Raspberry Pi OS Lite was connected to this local subnet via Ethernet. A ConBee II[2] Zigbee dongle was connected to the Raspberry Pi, together with installations of the Phoscon[3] and deCONZ [4] apps, allowing the Pi to act as its own Zigbee gateway. As for the Zigbee network nodes, there were a selection of devices connected to the network, both ZRs and ZEDs. This included one smart-socket, two environment sensors, and one smart light. These devices are shown in table 4.3. The devices were connected using the instructions provided through the Phoscon web app and it's discovery process. No further customizations were made to the devices once they were on the network.

Table 4.3: List of Zigbee devices connected to the network, their function, device type, and MAC address.

Device	Function	Type	MAC Address
ConBee II	Zigbee gateway node	ZC	`00:21:2e:ff:ff:06:0b:e4`
Hue Bloom	Desk lamp	ZR	`00:17:88:01:0c:67:e6:27`
Hue Smart Plug	Smart socket relay switch	ZR	`00:17:88:01:0b:e1:19:53`
Hue Motion sensor	Environment sensor	ZED	`00:17:88:01:0b:d0:aa:cf`
Aquara temperature	Environment sensor	ZED	`00:15:8d:00:05:44:ee:f6`

The Phoscon web interface provided a way to control the Zigbee devices, such as dimming lights, switching power to smart plugs, and also reading sensor values. deCONZ on the other hand, provided detailed insight into the Zigbee network, allowing low level inspection of the network devices and their connectivity (such as device mac addresses, device types, and link quality).

We used deCONZ to find out more information about the devices on the net-

[1] VyOS open source router and firewall platform - `https://vyos.io/`
[2] ConBee II Zigbee dongle - `https://www.phoscon.de/en/conbee2`
[3] Phoscon App - `https://phoscon.de/en/app/doc`
[4] deCONZ - `https://github.com/deconz-community/deconz-docker`

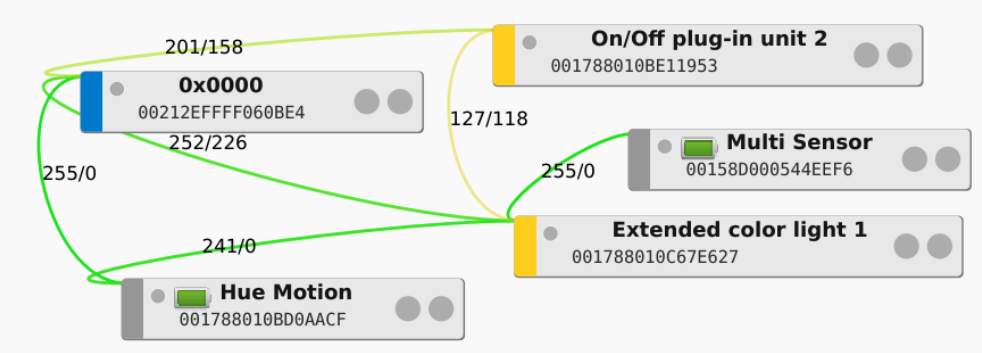


Figure 4.2: deCONZ graph view of the Zigbee network.

work, such as the device types and mac addresses. Figure 4.2 shows a graph view of the network topology which was provided by deCONZ. In this graph view, the yellow nodes indicate ZRs, the grey ZEDs and blue ZCs. This information was used to fill in the device type and mac address seen in Table 4.3.

4.2.1 Attack Vector

The intention of this lab was to simulate a home environment, so in this case, a potential threat actor does not have physical access to the devices or equipment located within the lab (as these would be inside the home). The threat actor must therefore rely on the use of wireless signals to perform their attacks.

Chapter 5

Evaluation Results

In this Chapter, we present the results from the requirements-based evaluation. The Requirements Traceability Matrices (RTMs) have been updated to include the results from testing the functional and non-functional requirements.

5.1 Functional Requirements

As seen in Table 5.1, all the test cases aimed at assessing the functional requirements were passed, including the case study. We now detail the results from each of these test cases, detailing any additional steps that were taken to achieve these results.

Table 5.1: Evaluation results for the functional requirements of PacketZapper.

Identifier	Test case	Reqs served	Results
TC001	**Design review**	FR001	Pass
TC002	**Parallel collection**	FR002-008	Pass
TC003	**Query capabilities**	FR006, FR009-0010	Pass
TC004	**Case study**	FR011, FR012, FR013	Pass

5.1.1 TC001: Design Review

The design review was based on the source code available to check out from Git. As described in the PacketZapper implementation overview (Section 3.2.1), the implementation merges stages 1 and 2 of the four-stage solution structure into a single piece of software, i.e., the Collection Agent.

Despite this, we believe that the design falls well within the structure of the four-stage design. The Collection Agent should be viewed as a facilitator for integrating various existing software components with the rest of the platform. It enables existing collection hardware such as the CC2531 Zigbee sniffer to be paired

with its protocol-specific parser (in this case, whsniff). Figure 5.1 shows this four-stage solution structure with more granularity with regard to the Collection Agent, demonstrating that the intended structure is in fact retained.

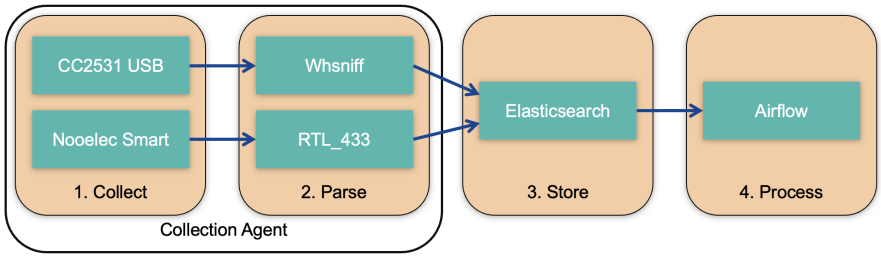


Figure 5.1: PacketZapper four-stage analysis.

Adding an additional protocol to the platform, therefore, simply involves finding new collect-parse pairings for that specific protocol, which, after some testing, can then be added as a part of the Collection Agent. If a USB dongle does not work as intended, or a different parsing software is to be used, then the structure of the platform allows for such modifications with relative ease. If the user for example, wants to swap out Elasticsearch with an Open Distro variant, then this is also possible with some minor modifications, and does not require a complete rewrite of the system architecture. This is in part thanks to the platform following this four-stage solution structure.

Based on this code review and subsequent assessment, we evaluated the test as PASSED.

5.1.2 TC002: Parallel Collection

For this test, the Collection Agent was installed on a Raspberry Pi 3B placed within wireless proximity to the Zigbee network of the lab environment. The rest of the platform was installed via the included Docker configuration on a MacBook Air. The MacBook and Raspberry Pi were reachable over the same WiFi network. Configuration of the platform involved modifying the Elasticsearch address in the Collection Agent configuration file to point to the external instance.

The Collection Agent has a range of API endpoints that can be called to control collection. Since the code is built on top of FastAPI, we can leverage the auto-generated swagger interface at `/docs` to manually interact with these endpoints from our web browser. We started Zigbee capturing by posting to `/whsniff/start` with the payload shown in Listing 5.1. In the payload, we find the display filter used to filter out traffic coming from the Zigbee Coordinator (ZC). This filter is based on the findings from [16] where the Zigbee coordinator is often found to use `0x0000` as its source address when sending packets.

```
{
  "batch_size": 5,
  "sniffer": "whsniff",
  "channel": 25,
  "display_filter": "zbee_nwk.src != 0x0000"
}
```

Code listing 5.1: JSON payload posted to /whsniff/start.

After 4 minutes of Zigbee capturing, we started capturing 433MHz traffic by sending a POST request to the `/rtl_433/start` endpoint. These then ran for an additional 3 minutes before shutting them both down via the `/whsniff/stop` and `/rtl_433/stop` API endpoints.

Using the visualization functionality of Kibana, we created a heat-map view of the records stored in Elasticsearch, grouped by the capture program (whsniff and rtl_433). As shown in Figure 5.2, we can visually confirm the start/stop order of the Zigbee and 433MHz capture tasks. We also observe that far more Zigbee packets were captured than 433MHz packets.

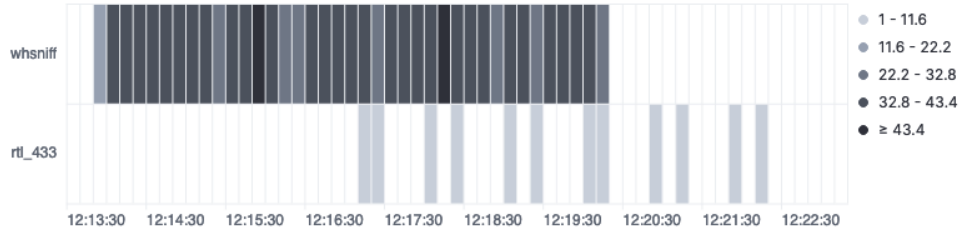


Figure 5.2: Kibana heat-map visualization of the number of Zigbee and 433MHz packets captured to the specifications of TC002. X-axis (time) is grouped in 10 second intervals.

To complete the test, we used Kibana to visualize all the values of `zbee_nwk.src` to confirm that there were no instances of the value 0x0000 present in the graph, indicating that the filter worked as intended. Figure 5.3 confirms this result. We then started whsniff again without the filter to confirm that traffic originating from `0x0000` was in fact present on the network. Based on the results shown here, we evaluated TC002 as PASSED.

5.1.3 TC003: Query Capabilities

This test made use of the data collected in TC002. To perform these queries, we used a combination of Kibana for prototyping and Jupyter for the final implementation.

First, we demonstrated querying for the last collected Zigbee packet and verified the data structure, as instructed by the test. To interact with Elasticsearch, we used the Python Elasticsearch library. Listing 5.2 shows the code that was run in a Jupyter notebook to perform this part of the test. Notice that the `es.search()`

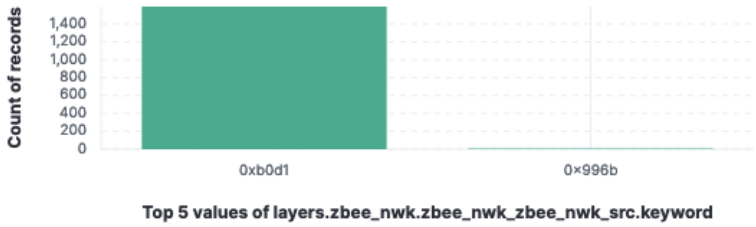


Figure 5.3: Kibana visualization confirming that src address 0x0000 is not present in the data, indicating that the filter functioned as intended.

function queries for a document where the sniffer type is "whsniff", it only retrieves a single document, ordered by the timestamp field.

```
from elasticsearch import Elasticsearch
es = Elasticsearch(['http://elastic:changeme@elasticsearch:9200'])
res = es.search(index="packetzapper",
    query={"match_phrase": {"packetzapper.request.sniffer": "whsniff"}},
    size=1, sort={ "timestamp": "desc"})
for hit in res['hits']['hits']:
    print(hit["_source"])
```

Code listing 5.2: Jupyter code to query for last Zigbee packet.

The full JSON packet structure retrieved from this command is attached in the appendix at A.1. It clearly shows that information from all layers of the Zigbee packet is retained, including a hex representation of the encrypted data payload.

Elasticsearch supports aggregation in queries. We created a query to give us the sum of transmitted data per Zigbee device in 10-minute intervals. The full query is found in the appendix at A.2. The results from this query are shown below in listing 5.3. This answers part two of the test.

```
Interval: 2023-05-08T14:00:00.000Z, Device: 0xb0d1, Bytes Sum: 70684.0
Interval: 2023-05-08T14:00:00.000Z, Device: 0x996b, Bytes Sum: 248.0
Interval: 2023-05-08T14:10:00.000Z, Device: 0xb0d1, Bytes Sum: 128183.0
Interval: 2023-05-08T14:10:00.000Z, Device: 0x996b, Bytes Sum: 944.0
Interval: 2023-05-08T14:20:00.000Z, Device: 0xb0d1, Bytes Sum: 85529.0
Interval: 2023-05-08T14:20:00.000Z, Device: 0x996b, Bytes Sum: 391.0
Interval: 2023-05-08T14:30:00.000Z, Device: 0xb0d1, Bytes Sum: 47175.0
Interval: 2023-05-08T14:30:00.000Z, Device: 0x996b, Bytes Sum: 196.0
```

Code listing 5.3: Result from aggregation query.

Creating this aggregation was not as simple as initially expected, resulting in a very long query. Despite this, the Elasticsearch database was proven capable of performing all the tasks defined in the test criteria. We therefore evaluated TC003 as PASSED.

5.1.4 TC004: Case Study

For this case study, we made heavy use of the included Apache Airflow installation for automation of all tasks. The Jupyter installation was used for the creation of

the Airflow Directed Acyclic Graphs (DAGs) and debugging of code to be used in the DAGs. We created a simple DAG structure to complete the inference tasks. Figure 5.4 shows a screenshot from the Airflow web interface graph view of the DAG.

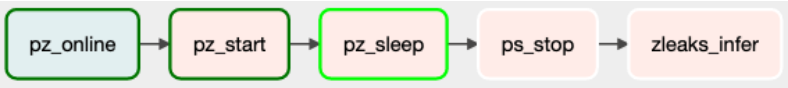


Figure 5.4: Screenshot of the DAG graph view taken from the Airflow web interface while the pz_sleep task was running.

From left to right, we first have `pz_online` which checks to see that the Collection Agent is online and responding to API calls. Next, we have `pz_start` which starts Zigbee sniffing. `pz_sleep` makes the pipeline wait for 5 minutes, `pz_stop` stops the Collection Agent capturing Zigbee traffic, and finally `zleaks_infer` contains the inference tasks. This is a simple yet automatic pipeline that could be expanded to do other tasks. In our specific case, we have set it to only run when manually triggered. We make use of the Collection Agent Python client library that was briefly introduced in section 3.2.4 to interact with the API in a Pythonic manner. The `zleaks_infer` task contains all the logic for our inference tasks. In this case, it is to infer the number of devices, Zigbee Routers (ZRs), and Zigbee End-Devices (ZEDs) on the network, in addition to identifying the ZC. The task runs these inference tasks and posts the results back to Elasticsearch where the user can view them.

To count the total number of devices on the network, we can simply do a count of unique Zigbee packet destination addresses. To find the number of ZRs on the network, we used the formula described in [16] on broadcast packets with a Zigbee radius value of 1:

$$\text{ZRs} = \frac{\text{packet payload length} - 2}{3}$$

To implement this in Airflow, we first confirmed its validity in Kibana by filtering for packets with the destination logical address of `0xffff`, and network radius of 1. Then we observed the field indicating the Zigbee payload length and applied it to the formula. Once the formula was confirmed to be working, we wrapped it together into a Python implementation and added it to our `zleaks_infer` task in Airflow.

Next, implementing functionality to identify the number of active ZEDs on the network proved difficult. Both papers detailing the possible passive attack vector [15, 16] were quite vague in the methodology of such an attack. We reverted to creating a Kibana dashboard to explore the data and find if there were any methods that we could implement in Airflow. A screenshot from this workflow is shown in Figure 5.5. In this dashboard we see graphical representations of the traffic grouped by the different source addresses and message types.

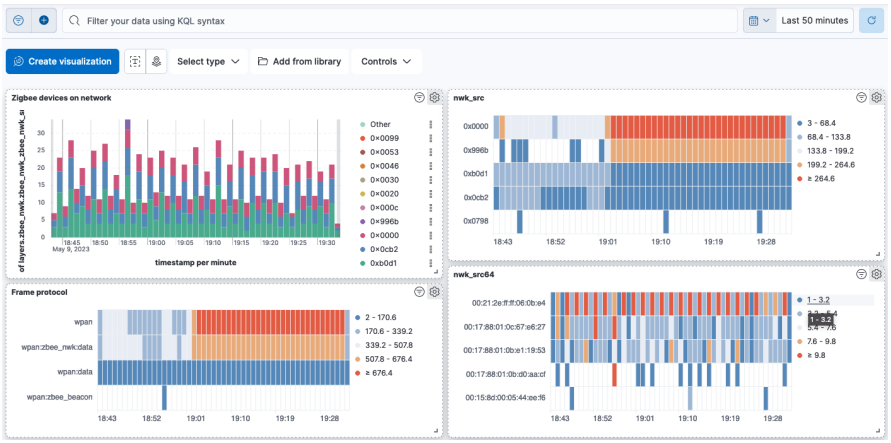


Figure 5.5: Kibana web interface view used when debugging the ZED count methodology.

Through this process of analyzing the data in Kibana and digging through the Zigbee and LR-WPAN specifications [28, 29], we discovered that it was possible to identify ZEDs that do not perform routing by searching for data requests that included the WPAN long address, not the short address as instructed by [15, 16]. We also discovered that while Hue Motion Sensor is a battery-powered device, it did not act like a typical ZED as it was relaying messages like a ZR. Therefore, it did not show up as a ZED using this new technique we developed (as it was routing messages). All in all, we could only identify the Aquara temperature sensor as a ZED. This process showed that the technique for identifying ZED as described in [16] may be inaccurate, however we were able to come close to finding the true value.

Finally, to identify the ZC by MAC, we filtered for packets where the Zigbee source address was `0x0000` and retrieved the last value for the 64-bit Zigbee source MAC address. We implemented all this functionality into our `zleaks_infer` task and triggered the DAG using the Airflow interface. We monitored the progress and viewed the results in Kibana. Table 5.2 shows the inference results. Details on how we discovered the true values are detailed in Section 4.2.

Table 5.2: Results from Airflow Zigbee inference pipeline posted to Elasticsearch.

Inferred item	True value	Inference Result
Device count	5	5
ZRs	2	2
ZEDs	2	**1**
ZC MAC	00:21:2e:ff:ff:06:0b:e4	00:21:2e:ff:ff:06:0b:e4

As we can see in the table, we were able to infer many attributes from the network using only passive methods. However, we were unable to infer the correct

number of ZEDs using the technique described in [16], or by developing our own methods. The discrepancy was however not the fault of the platform, but rather the inference techniques used.

Reviewing the overall test case, we assess the platform as a suitable choice for conducting inference experiments. This case study showcased the functionality of the platform for both automating tasks and developing and debugging methods. We assess this test case as PASSED.

5.2 Non-functional Requirements

We now present the results from the test results targeting the non-functional requirements of our platform, which were obtained through test cases TC005-007. These results provided insight regarding the installation, automation, and scalability of the platform. Table 5.3 presents an updated RTM that includes the test results.

Table 5.3: RTM evaluation results for the non-functional requirements of PacketZapper.

Identifier	Test case	Reqs served	Results
TC005	**Easy installation**	NFR001	Pass
TC006	**Automation**	NFR002	Pass
TC007	**Scalability**	NFR003	Pass

5.2.1 TC005: Easy Installation

In this test, we reviewed the installation procedure to evaluate if it meets the requirement of being easy to install (see NFR001). As presented in our implementation overview of PacketZapper in section 3.2.1, the platform is in its default configuration built on top of the Docker ecosystem. With such a configuration, it is possible to include a default configuration that is seemingly identical across deployments. Docker is supported on Windows, Mac, and Linux, achieving a level of cross-platform compatibility for the PacketZapper.

Installation becomes slightly more difficult when moving the Collection Agent out from the Docker context. This is required on hosts where Docker does not support device passthrough (such as MacOS, and likely Windows). In this case, the user must handle the installation of dependencies themselves, which while being possible, is not recommended. We also make use of bash pipe functionality, which is not supported on Windows hosts. It may be beneficial to include host requirements (such as running on a Linux machine) in the installation instructions. Based on the fact that the platform functions out-of-the-box (given that the user can install Docker), we assess that the test case is PASSED.

5.2.2 TC006: Automation

In this test, we assessed the implementation of automation in the platform and it's suitability with regard to the types of tasks that may be run on the platform. Automation in the platform is administered from Apache Airflow. Airflow can interact with all components of the platform, including controlling the collection of data on the Collection Agent. Being that Airflow DAG tasks are implemented in Python, we assess Airflow as a suitable building block for building out automation tasks. While Airflow may be slightly difficult to get started with, it is a popular technology that has proven itself in many companies over the last couple of years. The automation functionality of the platform worked as intended when performing the case study in TC004. We assess that this test case is PASSED.

5.2.3 TC007: Scalability

In this test, we assessed the scalability of PacketZapper. Scaling is handled in all stages of collection in PacketZapper. The Collection-Agent can be scaled horizontally, meaning that multiple instances of this component can run simultaneously. As discussed in section 2.2.1, Elasticsearch is built for scaling, implementing horizontal scalability through the use of node clustering. Airflow is also in a similar situation as Elasticsearch, being built with scale in mind.

Although the core technologies of PacketZapper support horizontal scaling, the supporting infrastructure in its current configuration does not cater to such modifications of Airflow and Elasticsearch in an easy manner. Some work would have to be done by the user to move the deployment away from docker-compose to a technology suitable for their use case (for example Kubernetes). Despite these limitations, we do not believe that such massive horizontal scaling of these components would be required when collecting IoT traffic.

We do however, believe that having a single Collection Agent could be a potential bottleneck for the platform. Fortunately, installation of the Collection Agent on, for example, a Raspberry Pi proved relatively simple. Such an installation could be duplicated across multiple Raspberry Pis and added to the platform with relative ease as this does not require changes to the rest of the platform.

While there are some challenges involved in achieving massive scaling of the platform, it is still possible as there are no fundamental issues with the technologies used. We assess that scaling is taken into account, and the test case is therefore PASSED.

Chapter 6

Discussion

We were successful in designing, building and testing a platform to collect, parse, store and process IoT device traffic. We followed a reuse-based software development methodology, which prompted the creation of a solution structure and requirements for the system. After the development of PacketZapper was completed, we performed a series of test cases to assess if the platform conformed to the system requirements. In this chapter, we discuss this development process, along with an in-depth assessment of the resulting platform as a whole.

6.1 Design of PacketZapper

PacketZapper was designed and developed using our four-stage solution structure (collect, parse, store, process). We created a custom software component (i.e., the Collection Agent) which handled stages 1 and 2, while the remaining stages were handled by existing open source projects; Elasticsearch and Apache Airflow respectively. We start of by discussing our choice of components, followed by a discussion on alternate configurations that could be implemented with said components.

6.1.1 Choice of Components

As noted in the results from TC001 (Design review) in Subsection 5.1.1, the Collection Agent is a facilitator for connecting collection hardware to its appropriate parsing software, while also handling transport of any resulting data to Elasticsearch. In its current state, PacketZapper supports Zigbee and 433MHz collection. Zigbee collection relies on a TI CC2531 USB dongle paired with the `whsniff` software, while 433MHz on a Nooelec dongle paired with the `rtl_433` software. The intention of the Agent is to allow for simple dongle-software pairings based on the requirements of the user. However, an alternative approach to this could be the use of a Software Defined radio (SDR) to handle all wireless signals, and then decode them in GNU Radio running the appropriate decoding modules. This was

how the authors of the Zigator paper [15] approached performing sniffing, spoofing and jamming attacks targeting Zigbee networks. Although such an approach allows for more fine grain control over the collection process, it requires a whole new skill-set to use complex software such as GNU Radio, not to mention the cost of SDRs often being in the thousands of dollars. We believe that the current implementation of collection and parsing is suitable, balancing cost, ease of use and performance. It is worth noting however, that SDR and GNU Radio pairings are still at a theoretical level still possible to add within the structure of the Collection Agent. Implementing the software used in Zigator into the Collection Agent may be an idea for future work.

We chose Elasticsearch for the storage stage. This choice boiled down to it's heavy adoption in the data analytics industry, along with it's powerful query features and scaling functionality. We did initially look at some other well known, lightweight data storage platforms such as InfluxDB[1] and MongoDB[2]. InfluxDB was a strong competitor to Elasticsearch as they both function using a schemaless design for ingest of data. InfluxDB however does not support nested JSON structures. This means that new data sources would have to have be flattened and converted into InfluxDB's own ingest format known as line protocol before ingest into the database. This increases the amount of work needed to implement a new collection protocol into the Collection Agent. MongoDB was a stronger candidate, supporting complex nested JSON structures, with a powerful query engine that supports various types of data aggregation queries. This final choice boiled down to the data exploration functionality that can be achieved in Elasticsearch via Kibana, which is part of the Elastic Stack. TC004 highlighted the importance of being able to do data exploration, prototyping new queries and discovering new insights in the data. MongoDB does have the MongoDB charts tool for data visualization, however this was not something we tested as the Elasticsearch-Kibana combination served all our requirements as is, while having an established industry adoption.

We chose Apache Airflow as the solution for the process stage. Airflow is a relatively new piece of software, which made us cautious when deciding if it was suitable for implementation in PacketZapper. The Apache Software Foundation, who are the maintainers of Airflow, have done some surveying of Airflow users which can help shine light on this question. The 2022 survey [30] is the latest of 3 surveys completed since 2019. It shows that the platform has had a consistent growth in use in large companies, with also a steady increase in the willingness to recommend the technology to new users. Based on this it could seem like Airflow is a relatively established technology, and we may see even more widespread adoption of Airflow in the future, to the benefit of PacketZapper. In it's current state, and as seen through the testing of PacketZapper via the case study (TC004), it becomes clearer that Airflow is not as simple and user friendly to use as one may initially expect and desire. The use of DAGs are for many people a new concept

[1] InfluxDB - `https://www.influxdata.com/`

[2] MongoDB - `https://www.mongodb.com/`

that users must get an understanding of from a conceptual level, before they can even start coding their pipelines using this technology. Example DAGs could definitely help kick-start new types of collection on PacketZapper, so having some examples bundled with the platform is almost a must. There are some alternative to Airflow, that could function as an almost drop-in replacement. For example, there is Kubeflow[3] and Kedro[4], which are tailored more towards Machine Learning (ML) tasks. If the user is familiar with these technologies, then this could be an option. We are however of the opinion that Airflow is generic enough to tailor to a variety of tasks that may be run on the platform.

6.1.2 Alternate Configurations

As described throughout Chapter 3, the PacketZapper platform comes bundled together via docker-compose files. Everything from the Collection Agent to the Jupyter Lab instance runs on the same host. But thanks to the modularity of the components and open-source nature of the code, it is possible to make alternate configurations of the platform based on variations in the user's requirements.

Firstly, it is likely that some scenarios could require the Collection Agent to run at a remote location away from the rest of the infrastructure. Cloud hosting is a great example of this, where the user can deploy most of the platform on powerful infrastructure at Google Cloud or Amazon Amason Web Services (AWS), but have the Collection Agent installed locally. Users could also decide to collect data from multiple locations at the same time using a single instance of the core components of PacketZapper. While we did not directly test any of these configurations ourselves, they still had an influence during the design and implementation of the platform. In its current state, the platform handles multiple Collection Agent instances, but it does not take into account inter-service communication over unsecured networks such as the internet (as would be required in the cloud scenario). It is difficult to recommend a one-size-fits-all solution to this problem as it depends on a variety of variables such as firewalls and internal company/university policies, especially if some form of port forwarding has to be enabled on the network. We discuss possible solutions to this in Section 6.2.3.

6.1.3 PacketZapper Evaluation

We had a two part evaluation, grouped by the test cases targeting the functional and non-functional system requirements. The test cases targeted at the functional requirements enabled us to demonstrate the capabilities of the platform. Having a case study as a part of this testing was a good method for showcasing that the platform could be used in real world inference tasks. Optimally, there should have been more than one case study, possibly even targeting the 433MHz traffic data.

[3]Kubeflow - `https://www.kubeflow.org`

[4]Kedro - `https://kedro.org/`

Many of the test cases involved making queries to run on the Elasticsearch database. The Kibana interface was crucial in helping create these queries, as it has tools for creating queries using graphical drag-and-drop tools. The resulting queries were quite large, and complex to understand at a glance. It might have been beneficial to instead highlight some of the SQL functionality that is possible in Elasticsearch, especially since SQL is such a universally understood query language among IT professionals.

The case study in TC004 had issues counting the number of ZED on the network. Despite this, we still evaluated the test as a success, as this discrepancy was not the fault of PacketZapper. We could even argue that this served in the interest of PacketZapper, because it allowed us to demonstrate the data exploration functionality that the platform provides. Although the resulting query was not capable of counting the true number of ZEDs, it was capable of counting ZEDs that did not also implement routing functionality (which is uncommon for ZEDs).

The tests targeting the non-functional requirements were mostly reviews of code and design documents. It would have been beneficial to also have some form of user survey to better understand the usefulness of PacketZapper from the perspective of a potential user. We gave all the test cases a passing score, however a potential user of the platform may view the need to code DAGs as regressive in the lens of automation functionality.

Circling back to the problem description and research questions from Chapter 1, we believe that the PacketZapper platform is more than sufficient for researchers who want to run reproducible experiments that make use of real IoT device traffic. It provides the building blocks for creating automated collection workflows, but does require some Python coding experience from the user. PacketZapper makes use of industry established software such as Elasticsearch and Apache Airflow to help achieve its goals. Requirements testing demonstrated that the platform is capable of performing basic inference tasks.

6.2 Future Work

We now detail some items of interest that should be considered for future work. These items could help elevate the functionality of the platform, enabling the platform to be used in additional environments and use cases.

6.2.1 Additional Protocol Support

Currently, PacketZapper only supports two different wireless protocols. It should be a trivial task to add more low-bandwidth IoT protocols such as Bluetooth or Z-Wave to the platform. This process would involve finding USB dongles capable of sniffing, together with some software component that is capable of parsing this data, and outputting it in a JSON format.

From an initial glance, Bluetooth support could be implemented with a fairly cheap USB dongle from either Nordic Semiconductor (using the PCA10031 or

PCA10059 USB dongle) or Texas Instruments (using the CC2540EMK USB dongle). These cost in the range of 10-50 USD, and can sniff the Bluetooth advertising packets with ease. We found a github repository called Blesniffer[5] that seems to function in the same manner as `whsniff` does for Zigbee sniffing. This could be a potential candidate for implementing Bluetooth sniffing.

Z-Wave support could also be interesting for users of PacketZapper. From what we gather, there are open-source projects capable of sniffing and parsing Z-wave data using a USB dongle such as the UZB1 USB stick[6] running modified firmware. We have not tested any of these software solutions ourselves, but it seems like it should be possible to implement. This is definitely a topic of interest for future work.

6.2.2 Instrumentation of Devices

The lab environment we used had a series of devices that reacted to interactions from users in the operating environment. For example, the motion sensor would register motion if a person walked in front of the sensor. It would however be interesting to interact with these devices automatically, something known as instrumentation. For example, one may be interested in observing the authentication handshake of a device when it powers on, and may want to run this sequence of events hundreds of times. The Airflow instance could potentially interact with a smart socket connected to the device, powering it on or off based on the requirements of the task. One approach to this could be to deploy instrumentation devices running the ESPHome[7] custom firmware around the operating environment. Devices running ESPHome can be controlled over a built in API, so these devices could be commanded to for example interact with a motion sensor, or toggle the power to a device, all from an Airflow DAG. This should be possible to implement in PacketZapper with the current infrastructure it provides.

6.2.3 Enhanced Networking

As mentioned earlier, there are many scenarios that may benefit from having some of the components located in different locations/networks from the rest of the platform. There is at the time of writing, no functionality to handle the configuration of such complex network setups. A possible solution to this could be running a Software Defined Network (SDN) such as ZeroTier[8] to connect the different components together. ZeroTier is a service for creating peer-to-peer encrypted virtual networks without having to expose any firewall ports. More traditional forms of tunneling could be used such as WireGuard[9], OpenVPN[10] or even IPSec, how-

[5] ura14h/Blesniffer - `https://github.com/ura14h/Blesniffer`
[6] UZB 1 - `https://z-wave.me/products/uzb/`
[7] ESPHome - `https://esphome.io/`
[8] ZeroTier - `https://www.zerotier.com/`
[9] WireGuard - `https://www.wireguard.com/`
[10] OpenVPN - `https://openvpn.net/`

ever one should note that these would require some form of firewall configuration, external VPN gateway, or similar to run. ZeroTier avoids the port-forwarding complications as it makes use of UDP hole punching to traverse NAT configured networks, establishing true end-to-end connections. This does, however, require that the nodes can establish a connection to the ZeroTier Simple Traversal of UDP through NATs (STUN) servers, so offline use could be more difficult.

Chapter 7

Conclusion

PacketZapper is our automated collection and processing platform for IoT device traffic. It was created to assist researchers in performing IoT traffic based research, handling everything from capturing the traffic via a USB dongle, to running inference tasks on said traffic data.

PacketZapper was designed and created using a reuse-based software development methodology. It's design was based on a four-stage solution structure, serving as the framework for the reuse of existing software, while also enabling future expansion of the platform's functionality and protocol support. In it's current state, PacketZapper supports collection of Zigbee and 433Mhz traffic via USB dongles, which is coordinated through our horizontally scaleable Collection Agent software. PacketZapper relies on Elasticsearch for data storage, and Apache Airflow for enabling users to automate their traffic collection and processing workflows.

We demonstrated that PacketZapper is capable of handling a variety of tasks, conforming to the system requirements defined for it during its design process. Through the use of a case study, we demonstrated the application of a basic inference experiment running in an automated manner, and could also highlight the platform's data-exploration capabilities.

PacketZapper's complexity necessitates overcoming a learning curve before users can fully realize and harness its potential. Proficiency in Python, Elasticsearch, and Apache Airflow would significantly enhance users' ability to navigate and leverage the platform's advanced features, particularly in terms of automation and data exploration. More work should be done on enabling collection of additional protocols such as Bluetooth and Z-wave to further PacketZapper's relevance in IoT-traffic based research activities.

Bibliography

[1] R. F. Karlicek, 'Smart lighting-beyond simple illumination,' in *2012 IEEE Photonics Society Summer Topical Meeting Series*, IEEE, 2012, pp. 147–148.

[2] J. Hye Oh, S. Ji Yang and Y. Rag Do, 'Healthy, natural, efficient and tunable lighting: Four-package white leds for optimizing the circadian effect, color quality and vision performance,' *Light: Science & Applications*, vol. 3, no. 2, e141–e141, 2014.

[3] I. Lopez-Neira, T. Patel, S. Parkin, G. Danezis and L. Tanczer, "internet of things': How abuse is getting smarter,' *Safe–The Domestic Abuse Quarterly*, vol. 63, pp. 22–26, 2019.

[4] D. Cook, M. Schmitter-Edgecombe, A. Crandall, C. Sanders and B. Thomas, 'Collecting and disseminating smart home sensor data in the casas project,' in *Proceedings of the CHI workshop on developing shared home behavior datasets to advance HCI and ubiquitous computing research*, 2009, pp. 1–7.

[5] J. Ren, D. J. Dubois, D. Choffnes, A. M. Mandalari, R. Kolcun and H. Haddadi, 'Information exposure from consumer iot devices: A multidimensional, network-informed measurement approach,' in *Proceedings of the Internet Measurement Conference*, 2019, pp. 267–279.

[6] I. Hafeez, M. Antikainen, A. Y. Ding and S. Tarkoma, 'Iot-keeper: Detecting malicious iot network activity using online traffic analysis at the edge,' *IEEE Transactions on Network and Service Management*, vol. 17, no. 1, pp. 45–59, 2020.

[7] X. He, Y. Yang, W. Zhou, W. Wang, P. Liu and Y. Zhang, 'Fingerprinting mainstream iot platforms using traffic analysis,' *IEEE Internet of Things Journal*, vol. 9, no. 3, pp. 2083–2093, 2021.

[8] M.-C. Lee, J.-C. Lin and O. Owe, 'Pds: Deduce elder privacy from smart homes,' *Internet of Things*, vol. 7, p. 100 072, 2019.

[9] T. Gu, Z. Fang, A. Abhishek and P. Mohapatra, 'Iotspy: Uncovering human privacy leakage in iot networks via mining wireless context,' in *2020 IEEE 31st Annual International Symposium on Personal, Indoor and Mobile Radio Communications*, IEEE, 2020, pp. 1–7.

[10] N. Apthorpe, D. Y. Huang, D. Reisman, A. Narayanan and N. Feamster, 'Keeping the smart home private with smart (er) iot traffic shaping,' *arXiv preprint arXiv:1812.00955*, 2018.

[11] A. Acar, H. Fereidooni, T. Abera, A. K. Sikder, M. Miettinen, H. Aksu, M. Conti, A.-R. Sadeghi and S. Uluagac, 'Peek-a-boo: I see your smart home activities, even encrypted!' In *Proceedings of the 13th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, 2020, pp. 207–218.

[12] N. Apthorpe, D. Reisman, S. Sundaresan, A. Narayanan and N. Feamster, 'Spying on the smart home: Privacy attacks and defenses on encrypted iot traffic,' *arXiv preprint arXiv:1708.05044*, 2017.

[13] B. Toulas, 'Brazil seizing Flipper Zero shipments to prevent use in crime,' *BleepingComputer*, Mar. 2023.

[14] I. Unwala, Z. Taqvi and J. Lu, 'Thread: An iot protocol,' in *2018 IEEE Green Technologies Conference (GreenTech)*, IEEE, 2018, pp. 161–167.

[15] D.-G. Akestoridis, M. Harishankar, M. Weber and P. Tague, 'Zigator: Analyzing the security of zigbee-enabled smart homes,' in *Proceedings of the 13th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, 2020, pp. 77–88.

[16] N. Shafqat, D. J. Dubois, D. Choffnes, A. Schulman, D. Bharadia and A. Ranganathan, 'Zleaks: Passive inference attacks on zigbee based smart homes,' in *Applied Cryptography and Network Security: 20th International Conference, ACNS 2022, Rome, Italy, June 20–23, 2022, Proceedings*, Springer, 2022, pp. 105–125.

[17] T. Zillner and S. Strobl, 'Zigbee exploited: The good, the bad and the ugly,' *Black Hat 2015*, 2015.

[18] T. Gu, Z. Fang, A. Abhishek, H. Fu, P. Hu and P. Mohapatra, 'Iotgaze: Iot security enforcement via wireless context analysis,' in *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*, IEEE, 2020, pp. 884–893.

[19] A. Hariri, N. Giannelos and B. Arief, 'Selective forwarding attack on iot home security kits,' in *Computer Security: ESORICS 2019 International Workshops, CyberICPS, SECPRE, SPOSE, and ADIoT, Luxembourg City, Luxembourg, September 26–27, 2019 Revised Selected Papers 5*, Springer, 2020, pp. 360–373.

[20] A. Iqbal, J. Olegård, R. Ghimire, S. Jamshir and A. Shalaginov, 'Smart home forensics: An exploratory study on smart plug forensic analysis,' in *2020 IEEE International Conference on Big Data (Big Data)*, IEEE, 2020, pp. 2283–2290.

[21] M. Bajer, 'Building an iot data hub with elasticsearch, logstash and kibana,' in *2017 5th International Conference on Future Internet of Things and Cloud Workshops (FiCloudW)*, 2017, pp. 63–68. DOI: 10.1109/FiCloudW.2017.101.

[22] C. Hull, 'Not So Open Any More: Elasticsearch Relicensing and Implications for Open Source Search,' *Reworked*, Jan. 2021. [Online]. Available: https://www.reworked.co/knowledge-findability/not-so-open-any-more-elasticsearch-relicensing-and-implications-for-open-source-search.

[23] D.-G. Akestoridis, V. Sekar and P. Tague, 'On the security of thread networks: Experimentation with openthread-enabled devices,' in *Proceedings of the 15th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, 2022, pp. 233–244.

[24] D.-G. Akestoridis and P. Tague, 'Hiveguard: A network security monitoring architecture for zigbee networks,' in *2021 IEEE Conference on Communications and Network Security (CNS)*, IEEE, 2021, pp. 209–217.

[25] K. C. Kang, S. Cohen, R. Holibaugh, J. Perry and A. S. Peterson, 'A reuse-based software development methodology,' CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST, Tech. Rep., 1992.

[26] A. D'Alconzo, I. Drago, A. Morichetta, M. Mellia and P. Casas, 'A survey on big data for network traffic monitoring and analysis,' *IEEE Transactions on Network and Service Management*, vol. 16, no. 3, pp. 800–813, 2019.

[27] N. Leveson, M. Heimdahl, H. Hildreth and J. Reese, 'Requirements specification for process-control systems,' *IEEE Transactions on Software Engineering*, vol. 20, no. 9, pp. 684–707, 1994. DOI: 10.1109/32.317428.

[28] 'Zigbee specification,' *ZigBee Alliance - ZigBee Document 05-3474-21*, 2015.

[29] 'Ieee standard for local and metropolitan area networks–part 15.4: Low-rate wireless personal area networks (lr-wpans),' *IEEE Std 802.15.4-2011 (Revision of IEEE Std 802.15.4-2006)*, pp. 1–314, 2011. DOI: 10.1109/IEEESTD.2011.6012487.

[30] *Airflow Survey 2022*, [Online; accessed 23. Apr. 2023], Apr. 2023. [Online]. Available: https://airflow.apache.org/blog/airflow-survey-2022.

Appendix A

Additional Material

A.1 Zigbee packet structure

Zigbee packet structure retained by the elasticsearch database.


```
{
  "timestamp": "2023-05-08T12:45:45.125000",
  "layers": {
    "frame": {
      "frame_frame_section_number": "1",
      "frame_frame_interface_id": "0",
      "frame_frame_interface_name": "-",
      "frame_frame_encap_type": "104",
      "frame_frame_time": "2023-05-08T12:45:45.125203000Z",
      "frame_frame_offset_shift": "0.000000000",
      "frame_frame_time_epoch": "1683549945.125203000",
      "frame_frame_time_delta": "0.006337000",
      "frame_frame_time_delta_displayed": "1.100469000",
      "frame_frame_time_relative": "5250.125203000",
      "frame_frame_number": "63240",
      "frame_frame_len": "53",
      "frame_frame_cap_len": "53",
      "frame_frame_marked": false,
      "frame_frame_ignored": false,
      "frame_frame_protocols": "wpan:zbee_nwk:data"
    },
    "wpan": {
      "wpan_wpan_frame_length": "51",
      "wpan_wpan_fcf": "0x8861",
      "wpan_wpan_frame_type": "0x0001",
      "wpan_wpan_security": false,
      "wpan_wpan_pending": false,
      "wpan_wpan_ack_request": true,
      "wpan_wpan_pan_id_compression": true,
      "wpan_wpan_fcf_reserved": false,
      "wpan_wpan_seqno_suppression": false,
      "wpan_wpan_ie_present": false,
      "wpan_wpan_dst_addr_mode": "0x0002",
      "wpan_wpan_version": "0",
      "wpan_wpan_src_addr_mode": "0x0002",
      "wpan_wpan_seq_no": "214",
      "wpan_wpan_dst_pan": "0x539c",
```

```
      "wpan_wpan_dst16": "0x0000",
      "wpan_wpan_addr16": [
        "0x0000",
        "0xb0d1"
      ],
      "wpan_wpan_src16": "0xb0d1",
      "wpan_wpan_fcs": "0x910c",
      "wpan_wpan_fcs_ok": true
    },
    "zbee_nwk": {
      "zbee_nwk_zbee_nwk_fcf": "0x0248",
      "zbee_nwk_zbee_nwk_frame_type": "0x0000",
      "zbee_nwk_zbee_nwk_proto_version": "2",
      "zbee_nwk_zbee_nwk_discovery": "0x0001",
      "zbee_nwk_zbee_nwk_multicast": false,
      "zbee_nwk_zbee_nwk_security": true,
      "zbee_nwk_zbee_nwk_src_route": false,
      "zbee_nwk_zbee_nwk_ext_dst": false,
      "zbee_nwk_zbee_nwk_ext_src": false,
      "zbee_nwk_zbee_nwk_end_device_initiator": false,
      "zbee_nwk_zbee_nwk_dst": "0x0000",
      "zbee_nwk_zbee_nwk_addr": [
        "0x0000",
        "0xb0d1"
      ],
      "zbee_nwk_zbee_nwk_src": "0xb0d1",
      "zbee_nwk_zbee_nwk_radius": "30",
      "zbee_nwk_zbee_nwk_seqno": "255",
      "text": "ZigBee Security Header",
      "zbee_nwk_zbee_sec_field": "0x28",
      "zbee_nwk_zbee_sec_key_id": "0x01",
      "zbee_nwk_zbee_sec_ext_nonce": true,
      "zbee_nwk_zbee_sec_counter": "14740956",
      "zbee_nwk_zbee_sec_src64": "00:17:88:01:0c:67:e6:27",
      "zbee_nwk_zbee_sec_key_seqno": "0",
      "zbee_nwk_zbee_sec_mic": "4d:bf:fe:75",
      "_ws_expert": {
        "zbee_nwk_zbee_sec_encrypted_payload": null,
        "_ws_expert__ws_expert_message": "Encrypted Payload",
        "_ws_expert__ws_expert_severity": "6291456",
        "_ws_expert__ws_expert_group": "83886080"
      },
      "data": {
        "data_data_data": "8d:2f:fb:ed:a4:7b:b2:bf:8b:ee:a7:b3:e8:77:5d:fb",
        "data_data_len": "16"
      }
    }
  },
  "packetzapper": {
    "request": {
      "batch_size": 5,
      "sniffer": "whsniff",
      "channel": 15,
      "display_filter": "zbee_nwk.src != 0x0000"
    }
  }
}
```

A.2 Elasticsearch aggregation query

Aggregation query used in TC003.

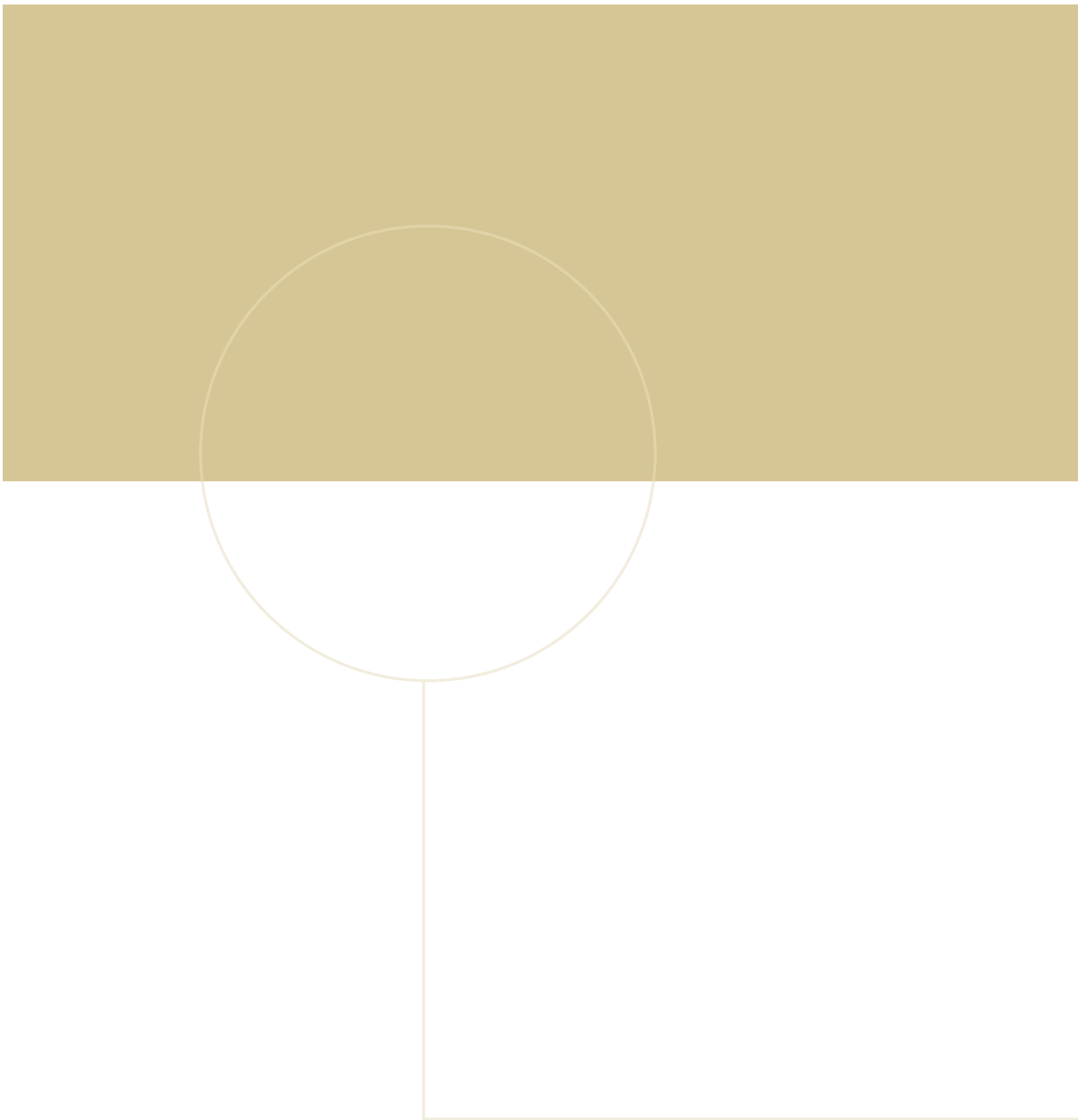
```
query = {
    "query": {
        "range": {
            "timestamp": {"gte": datetime.now() - timedelta(minutes=30),
            "lt": datetime.now()}
        }
    },
    "aggs": {
        "by_interval": {
            "date_histogram": {
                "field": "timestamp",
                "fixed_interval": "10m"
            },
            "aggs": {
                "by_device": {
                    "terms":
                        {"field": "layers.zbee_nwk.zbee_nwk_zbee_nwk_src.keyword"},
                    "aggs": {
                        "bytes_sum":
                            {"sum": {"field": "layers.frame.frame_frame_len"}}
                    }
                }
            }
        }
    }
}
response = client.search(index="packetzapper", body=query)
for interval in response["aggregations"]["by_interval"]["buckets"]:
    interval_time = interval["key_as_string"]
    for device in interval["by_device"]["buckets"]:
        device_name = device["key"]
        bytes_sum = device["bytes_sum"]["value"]
        print(f"Interval: {interval_time}, Device: {device_name}, Bytes: {bytes_sum}")
```

A.3 Elasticsearch index template

This is the index template used for PacketZapper.

```
PUT _index_template/packetzapper
{
  "template": {
    "mappings": {
      "dynamic": true,
      "numeric_detection": true,
      "date_detection": true,
      "dynamic_date_formats": [
        "strict_date_optional_time",
        "yyyy/MM/dd HH:mm:ss Z||yyyy/MM/dd Z"
      ],
      "_source": {
        "enabled": true,
        "includes": [],
        "excludes": []
```

```
      },
      "_routing": {
        "required": false
      },
      "dynamic_templates": []
    }
  },
  "index_patterns": [
    "packetzapper"
  ]
}
```



NTNU

Norwegian University of
Science and Technology