University of Zurich[UZH]

BACHELOR THESIS – Communication Systems Group, Prof. Dr. Burkhard Stiller

Dataset Generation and Feature Extraction for High-Traffic Environment Personal Tracker Identification

Stefan Richard Saxer
Zürich, Switzerland
Student ID: 18-923-359

Supervisor: Katharina O.E. Müller, Weijie Niu, Prof. Dr. Burkhard Stiller
Date of Submission: 26.07.2024

University of Zurich
Department of Informatics (IFI)
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland

ifi

Bachelor Thesis
Communication Systems Group (CSG)
Department of Informatics (IFI)
University of Zurich
Binzmuehlestrasse 14, CH-8050 Zurich, Switzerland
URL: http://www.csg.uzh.ch/

Declaration of Independence

I hereby declare that I have composed this work independently and without the use of any aids other than those declared (including generative AI such as ChatGPT). I am aware that I take full responsibility for the scientific character of the submitted text myself, even if AI aids were used and declared (after written confirmation by the supervising professor). All passages taken verbatim or in sense from published or unpublished writings are identified as such. The work has not yet been submitted in the same or similar form or in excerpts as part of another examination.

Zürich, _____

Signature of student

Abstract

Persönliche Bluetooth-Tracking-Geräte, wie AirTags, sind nützlich zum Auffinden verlorener Gegenstände. Allerdings können diese Tracker auch zur Verletzung der Privatsphäre, d. h. zum Stalking, missbraucht werden. Bislang ist die Erkennung von Bluetooth-Trackern, die in die Privatsphäre eindringen, schwierig. Der derzeitige Stand der Forschung legt nahe, dass es möglich ist, Bluetooth-Tracking-Geräte anhand ihrer übertragenen Pakete zu erkennen. Andere Forschende haben gezeigt, dass maschinelle Lernmodelle zwischen Paketen von Tracking-Geräten und Nicht-Tracking-Geräten unterscheiden können. Ziel dieser Arbeit ist es, diese binäre Klassifizierung zu einer kategorialen Klassifizierung von Bluetooth-Tracking-Geräte und deren Status zu erweitern. Darüber hinaus sollte die Klassifizierung auch in stark frequentierten realen Umgebungen funktionieren, um praktisch nutzbar zu sein. Zu diesem Zweck werden Bluetooth-Pakete von einer Vielzahl von Geräten aufgezeichnet, um einen grossen Datensatz für das Training von maschinellen Lernmodellen zu generieren. Die erfassten Daten werden dann ausfürlich analysiert, um geeignete Klassen für die kategoriale Klassifizierung zu bilden. Mehrere verschiedene maschinelle Lernmodelle werden auf diesem Datensatz trainiert und evaluiert. Schliesslich wird ein maschinelles Lernmodell für die Inferenz auf reale Daten verwendet, um Bluetooth-Pakete zu klassifizieren, die am Hauptbahnhof Zürich aufgezeichnet wurden. Diese Klassifizierung wird mit einem qualitativen Ansatz evaluiert, indem die auf der Grundlage der Klassifizierung erstellten Grafiken interpretiert werden. Die Evaluation zeigt, dass die kategorische Klassifizierung von Bluetooth-Tracking-Geräten in stark frequentierten realen Umgebungen mit einem hohen Grad an Genauigkeit möglich ist.

Personal Bluetooth tracking devices, such as AirTags, are useful for locating lost items. However, these trackers can also be misused for privacy invasion, i.e., stalking. As of right now, the detection of privacy-invading Bluetooth trackers is difficult. The current state of research suggests that it is possible to detect Bluetooth tracking devices based on their transmitted packets. Other researchers have shown that machine learning models can distinguish between packets of tracking and non-tracking devices. The goal of this thesis is to extend this binary classification to a categorical classification of Bluetooth tracking devices and their states. Furthermore, the classification should also work in high-traffic real-world environments to be practically usable. For this purpose, Bluetooth packets from a variety of devices are captured to generate a large dataset for training of machine learning models. The captured data is then analyzed extensively to form appropriate classes for categorical classification. Several different machine learning models are trained and evaluated on this dataset. Finally, one machine learning model is used for inference on real-world data to classify Bluetooth packets captured at Zürich central station. This classification is evaluated with a qualitative approach by interpreting plots generated based on the classification. The evaluation shows that the categorical classification of Bluetooth tracking devices in high-traffic real-world environments is possible with a high degree of accuracy.

Acknowledgments

First and foremost, I would like to thank my supervisor, Katharina Müller, for all the support she provided during the last six months. She provided me with a great topic, gave me all the hardware I needed, and showed endless patience during hours of meetings. Her feedback was valuable and well appreciated. Without her, this thesis would not have come to fruition the way it has. It was a great opportunity to write this thesis at the Communication Systems Research Group (CSG) at the Department of Informatics of the University of Zürich (UZH). Additionally, I would like to thank my mom for carefully reviewing my lengthy thesis in the end.

Contents

Chapter 1

Introduction

Personal Bluetooth-based tracking devices like the AirTag from Apple or the SmartTag from Samsung have recently gained in popularity. These devices are helpful in finding lost items such as keys or handbags. However, similar to tracking lost items, these trackers can also be misused for the malicious purpose of stalking. It does not require a lot of imagination to come up with ideas on how this could be achieved without the victim's knowledge. As these trackers are relatively small in size, sometimes only as large as a coin, they can be hidden practically everywhere. Therefore, detecting malicious Bluetooth trackers by the naked eye can be challenging, if not impossible. For instance, a tracker could very well be mounted under a vehicle and, therefore, be nearly invisible and hard to spot.

Therefore, as high as these trackers' utility is, so is the privacy risk they impose. Additionally, this risk of privacy intrusion is certainly not beneficial to the widespread adoption of these (useful) devices. Therefore, vendors of these Bluetooth trackers, such as Apple, have created smartphone apps that can detect malicious privacy-invading trackers of their brand [1]. In other words, the tracker detection app from Apple can detect Apple's AirTags and Apple's AirTags only. This is not the most consumer-friendly approach. It would, therefore, be highly beneficial to have a universal approach for tracker detection that could detect trackers of various vendors.

1.1 Motivation and Thesis Goals

Given that the current state of detection of malicious Bluetooth trackers is not consumer-friendly and inadequate from a privacy point of view, this thesis attempts to create a machine learning model for Bluetooth tracker detection. Machine learning is known to be highly effective in classification tasks, especially when trained on large-scale training datasets.

In an ideal scenario, machine learning models generalize from the given training data and learn the underlying hidden patterns in the data to reach their target. This is especially important in the case of previously unseen data samples where adequately trained machine

learning models thrive in contrast to traditional, more rule-based approaches. Generalization is essential because collecting data from all Bluetooth devices, Bluetooth trackers, and other non-tracker Bluetooth devices would never be possible. Any approach for Bluetooth tracker detection must be able to correctly classify data samples from devices not seen during training. Therefore, the strengths of machine-learning-based approaches are promising for identifying Bluetooth trackers.

From a practical point of view, the actual identification of the devices, i.e. the inference on the machine learning models, should happen in a real-world high-traffic environment. The environment should contain typical Bluetooth devices, i.e., be "real-world", and the more devices that are seen during inference, the more there is to evaluate. Hence, the environment should be high-traffic.

Therefore, the goals of this thesis are:

- Collect Bluetooth packet data from Bluetooth devices, both trackers and non-trackers.
- Train a machine learning model on the collected data.
- Use the machine learning model to identify Bluetooth trackers in real-world high-traffic environments and evaluate the result.

1.2 Description of Work and Scope

As this thesis is rather extensive, this section shall give some guidance on the scope and the most important content of this thesis:

- There is a brief introduction to the functionality of Bluetooth tracking networks and Bluetooth Low Energy (BLE). This does not aim to provide a deep dive into Bluetooth. A look into the Bluetooth Core Specification might be advisable to gain a deeper understanding.
- A focus of this thesis is certainly the aspect of data collection. This is reasonable given that the quality of machine learning models largely depends on the data they were trained on. One out of the many results of this thesis is, therefore, an extensive dataset collected over 600 hours and containing 30 million Bluetooth packets. This dataset could potentially be used by other researchers.
- Given the large amounts of data involved, a custom solution for data processing became necessary. The result of this is the Task-Group-Framework, a high-level implementation for complex data pipelines focusing on flexibility at runtime. An entire chapter is dedicated to outlining the core aspects and functionality of this framework. A detailed documentation of the framework does not exist yet as of writing this thesis.

- Three approaches for data preprocessing and subsequent modeling are detailed in this thesis. Two of these three approaches were then applied to the training dataset using various machine learning models. These models are the main result of this thesis. All of these models were also evaluated on separate test sets.
- One of the numerous created machine learning models was ultimately selected for inference on real-world data collected in a high-traffic environment, Zürich central station (HB). The result of this inference is evaluated to the best of my abilities using insightful visualizations. These plots are the thesis's highlight.

Additionally, the Python source code, including the Task-Group Framework, my Jupyter Notebooks, all plots, and the entire dataset, can be found on the GitHub page for this thesis. The link to the repository is in the appendix.

1.3 Thesis Outline

The following chapters of this thesis cover roughly the following topics. At the end of every chapter, a brief section summarizes the key takeaways.

Chapter 2 introduces fundamental theoretical concepts that are essential for understanding what follows. Reading through this to understand the terminology and definitions used in this thesis is especially recommended.

Chapter 3 discusses related work. Furthermore, it outlines a baseline feasibility experiment that mimics and summarizes the current state of research.

Chapter 4 discusses the data collection process in detail, ranging from an overview of popular BLE trackers to a deep dive into the challenges faced during data collection, especially labeling.

Chapter 5 introduces a custom data processing framework focusing on runtime flexibility, the Task-Group-Framework. This framework was created for this thesis to tackle the challenges of processing the large amounts of data that were collected.

In Chapter 6, the collected data is extensively analyzed with visual plots in order to gain insights for feature extraction and modeling.

Chapter 7 covers the training and evaluation of various machine learning models following different approaches.

In Chapter 8, the most relevant chapter, the machine learning models from the previous chapter are used for inference on a real-world data set collected in a high-traffic environment. The result is evaluated and discussed with visual plots.

Finally, Chapter 9 analyzes the success of this thesis in the context of the goals set above and outlines potential future work.

Chapter 2

Theory

This chapter introduces numerous theoretical concepts and definitions crucial to this thesis. Therefore, it is split into three sections, which are only loosely connected with each other. Any other extensive theoretical introduction is omitted for brevity.

The first section introduces the fundamental concepts behind these devices. This includes the definition of trackers, an introduction to tracking networks, and an introduction to the operating states of trackers. Next, there is a brief introduction to the functionality of Bluetooth Low Energy (BLE) on a protocol level, focusing on aspects relevant to this thesis. The final section introduces numerous machine learning approaches for BLE device classification and discusses their respective advantages, disadvantages, and limitations.

2.1 Bluetooth Tracking Devices

2.1.1 Tracking Networks

As of writing this thesis, there are two fundamentally different types of trackers: GPS trackers and the already discussed BLE trackers. On the one hand, there are GPS trackers that use a GPS receiver to locate themselves and a cellular modem to transmit the location data to the trackers' owner via the Internet. These trackers can operate everywhere as long as GPS and cellular signals are present. However, their technical complexity, size, and battery life limit their applicability for privacy invasion.

BLE trackers, on the other hand, have neither a GPS module nor a cellular modem onboard. They contain nothing but a tiny logic board, a small battery (usually a coin battery), and a Bluetooth modem, precisely a Bluetooth Low Energy modem, to save power compared to traditional Bluetooth. Therefore, these trackers are often tiny and not larger than a coin. Instead of relying on GPS and cellular networks, these trackers rely on so-called tracker networks for locating.

BLE tracker networks enable device location using the following method, explained here using Apple's Find My network [2] as an example (Figure 2.1). Other tracker networks, such as those from Samsung or Tile, operate similarly.

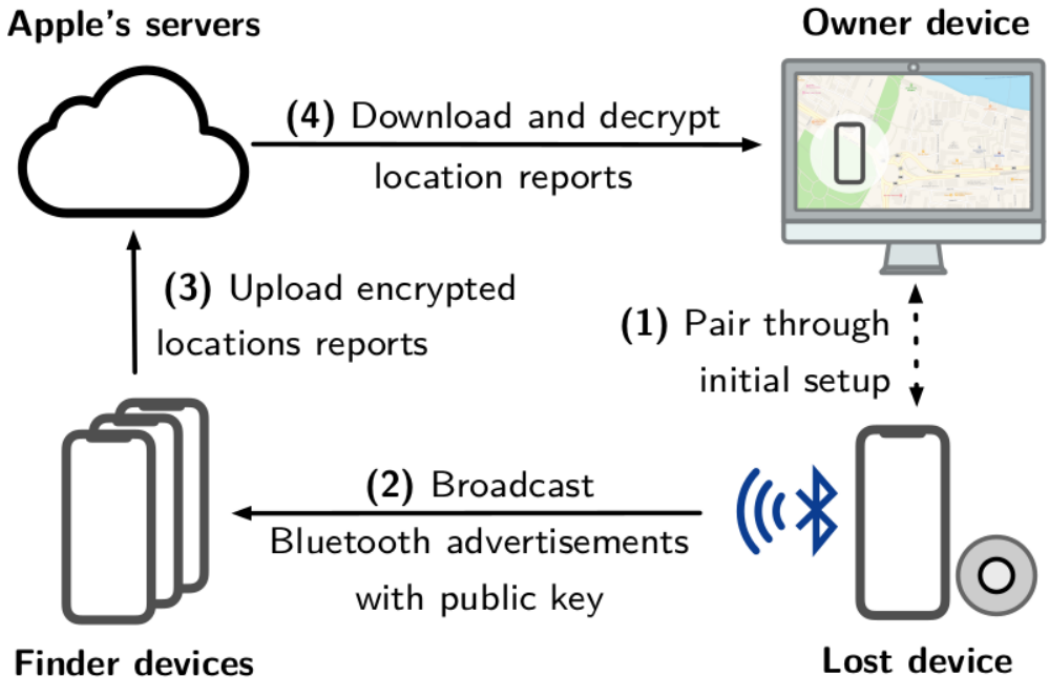

Figure 2.1: Apple Find My Tracker Network [2]

1. The tracker is linked to the tracker network via an owner device, typically a smartphone. Both a public and a private key are generated during this linking process. The public key is stored on the tracking device, and the private key is stored on the owner's device.

2. After the linking process, the BLE tracker starts broadcasting the public key via BLE advertising packets periodically.

3. Other devices registered to the same network, often smartphones, can pick up these BLE advertisements, including the public keys. These finder devices need to have a GPS module and an Internet connection. When a foreign public key is picked up via Bluetooth, the finder device locates itself via GPS, encrypts the location data using the public key, and uploads this encrypted data to the network provider's server using its Internet connection.

4. The owner's device can download the encrypted location data from the network provider's servers and decrypt the data using the previously generated private key. The result is an approximate location of the (lost) BLE tracker.

The above description of tracker networks and the technical implementation of BLE trackers lead to three important conclusions:

- Due to relying on Bluetooth Low Energy instead of GPS and cellular, BLE trackers are tiny and have extensive battery life, thus making them a potential privacy risk.

- The greatest strength of BLE trackers, i.e., relying on BLE, is also their most significant weak point. Because they must regularly broadcast BLE packets for locating, they can possibly be detected by packet sniffing.
- If it was possible to reliably classify these sniffed packets as tracker or non-tracker, for instance, with machine learning, malicious BLE trackers could easily be detected and their privacy risk somewhat mitigated.

2.1.2 Definitions of Trackers

In this thesis, BLE trackers shall be defined as devices that are trackable via the tracking networks described above. This includes the conventional BLE tracking devices such as AirTags or SmartTags and, in addition, other trackable devices such as iPhones, which are also trackable via Apple's Find My network. Therefore, this definition is significantly broader than the conventional definition of BLE trackers, which in most cases only extends to devices used for tracking purposes only, such as AirTags or SmartTags, and does not cover other devices, such as mobile phones.

This definition was chosen because, in the context of privacy invasion, it does not matter whether a device is a phone, a conventional BLE tracker, or anything else. As long as a device is potentially trackable via a Bluetooth-based tracking network, it is a potential privacy concern and can be detected via Bluetooth packets.

In addition to Bluetooth trackers, there are also non-Bluetooth trackers. These are devices that are remotely trackable only by methods other than Bluetooth, such as the cellular network and GPS. However, for the sake of this thesis, it seems to be of little gain to cover devices that are trackable only over non-Bluetooth-based networks. If a device is only trackable via such a network, it is not necessarily possible to capture any Bluetooth packets for such a device and, therefore, detect it. However, from a privacy point of view, it would make sense to attempt to detect such devices, if somehow possible.

As of April 2024, most Android phones were examples of non-Bluetooth trackers, as they were only trackable through Google's Find My Device service, which relies solely on cellular networks. Therefore, Android phones could be considered privacy-invading trackers, even though it is impossible to detect them via Bluetooth, as they do not necessarily transmit any Bluetooth packets. Bluetooth could be turned off entirely on such devices, yet they were still trackable.

At this point, there are two things to add for clarification:

- iPhones and other Apple Devices are BLE trackers under the definition of this thesis as they are trackable via Apple's Find My network, which, as described above, works over Bluetooth.
- While writing this Thesis, Google launched its Bluetooth tracking network for Android devices similar to Apple's Find My network [3]. Therefore, as of May 2024, Android phones can principally no longer be classified as non-Bluetooth trackers and would be considered trackers under the definition of this thesis. However, the launch of this network was too late to be covered in this thesis.

2.1.3 States of Trackers

Bluetooth trackers can be in various states. In this context, a state shall be defined as any externally observable behavior of the tracker depending on its environment, configuration, usage, time, or other factors. Typically, this behavior manifests in the structure of the BLE packets transmitted by the device. Different states result in different packets transmitted.

At this point, it is important to differentiate between Bluetooth-only trackers (ex: AirTag) and other non-Bluetooth-only trackers (ex: iPhone) that can also be tracked via non-Bluetooth methods, such as the cellular network. The types of states and their exact definitions vary depending on the kind of tracker. The states outlined in the following paragraphs are defined in accordance with the definitions commonly found in related literature.

Note: This differentiation between Bluetooth-only trackers and non-Bluetooth-only trackers differs from before when we separated between Bluetooth trackers and non-Bluetooth trackers. An iPhone is a tracker under the definition of this thesis (as it is trackable via Bluetooth). Still, it is not a Bluetooth-only tracker but a non-Bluetooth-only tracker (as it is also trackable via the cellular network). An Android phone (trackable via Google's Find My Device network) is neither. It is not a tracker under the definition of this thesis, as it is not trackable via Bluetooth, and it is, therefore, also neither a Bluetooth-only tracker nor a non-Bluetooth-only tracker.

The most important states for **Bluetooth-only trackers** and their definitions are:

- **powerless:** The tracker is not connected to power, i.e., the battery is not inserted or not sufficiently charged. In this state, the tracker is not trackable and cannot transmit any BLE packets. Hence, the tracker is not detectable via BLE packets. For this thesis, this state is irrelevant.
- **unpaired:** The tracker is not linked to a tracking network. In this state, the tracker is not trackable but potentially detectable via BLE packets. This state typically occurs right after the tracker is unboxed and turned on for the first time to initiate the setup and pairing process.
- **nearby:** The tracker is in Bluetooth range of its owner device, and therefore, the tracker does not have to be located via other finder devices, i.e., the public key does not need to be transmitted. In this state, the tracker is trackable and potentially detectable via BLE packets.
- **lost:** The tracker is far away from its owner device, and other finder devices in the network are used to locate the tracker. In this state, the tracker is trackable and potentially detectable via BLE packets.
- **searching:** The tracker is searching its owner-device. This state typically occurs directly after the initiation via the press of a button on the tracker itself. In this state, the tracker is trackable and potentially detectable via BLE packets.

  After some time, the tracker will automatically leave this state and return to its original state, which is often associated with some form of an acoustic signal.

It is essential to add that not all Bluetooth-only trackers necessarily need to support all of the above states. It is perfectly reasonable for some trackers not to support certain states. For instance, not all trackers might support the searching feature. Hence, not all trackers need to have a ”searching” state. Some trackers also have additional states, not covered in the list above.

Most of the above states and their definitions must be slightly adapted for non-Bluetooth-only trackers. A non-Bluetooth-only tracker does not require any pairing process to an owner device. Therefore, it has neither an ”unpaired” nor a ”nearby” nor a ”lost” nor a ”searching” state. Given that those trackers can also be tracked via the Internet and their trackability is at least partially dependent on an Internet connection, such as a cellular connection, it seems viable to differentiate between the availability/unavailability of an Internet connection for such trackers. Therefore, the states of **non-Bluetooth-only trackers** shall be defined as follows:

- **powerless:** The tracker is not connected to power, i.e., the battery is not inserted or not sufficiently charged. In this state, the tracker is not trackable and cannot transmit any BLE packets. Hence, the tracker is not detectable via BLE packets. For the scope of this thesis, this state is irrelevant.
- **unregistered:** The tracker is not registered to a tracking network. In this state, the tracker is not trackable but potentially detectable via BLE packets.
- **online:** The tracker is connected to the Internet. In this state, the tracker is trackable via the Internet and Bluetooth. Therefore, the tracker is potentially detectable via BLE packets.
- **offline:** The tracker is not connected to the Internet. In this state, the tracker is trackable via Bluetooth only and potentially detectable via BLE packets.

At this point, further discussion of the ”unregistered” state might be necessary. Non-Bluetooth-only trackers are typically automatically linked to the tracking network during the initial setup process of the device, so unregistered trackers are a rare anomaly. Furthermore, one might ask whether such trackers would be detectable in the ”unregistered” state. The tracker is not registered to the network, so it does not have to transmit any BLE packets. Unlike Bluetooth-only trackers such as AirTags, non-Bluetooth-only trackers do not need to transmit BLE packets for pairing in the unregistered state. Therefore, non-Bluetooth-only trackers are likely not detectable in the ”unregistered” state.

Having discussed the various relevant states for Bluetooth trackers, one can apply those to the context of privacy protection. From a privacy point of view, it is most relevant to distinguish between potentially privacy-invading trackers and non-privacy-invading trackers. As discussed previously, non-detectable states, such as ”powerless” or ”unregistered”, are out of the scope of this discussion.

The ”lost” state is most privacy-invading for the Bluetooth-only trackers as it allows for tracking foreigners. The ”unpaired” and ”nearby” states are significantly less privacy-invading than the ”lost” state. A tracker in the ”unpaired” state is not trackable; there is

no privacy risk. In opposition to this, the ”nearby” tracker is theoretically trackable but unsuitable for privacy invasion. A tracker in the ”nearby” state is - in all likelihood - the tracker of the person it is following/close to, as it is within proximity of its owner device. In other words, trackers do not invade privacy in the ”nearby” state as they presumably belong to the person/owner-device they are closest to.

Therefore, it seems sensible to at least differentiate between the ”lost” state and the other two states. If possible, it would be preferable to be able to distinguish all three states, as the ”nearby” tracker could - at least theoretically - still be a privacy risk.

Only two relevant states are left for the non-Bluetooth-only trackers: ”offline” and ”online”. The ”online” state is significantly more privacy-invading than the ”offline” state. In the ”online” state, the tracker can be tracked via the Internet and Bluetooth, whereas only the latter is possible in the ”offline” state. Therefore, a differentiation between the states would be highly preferable if possible.

In conclusion, trackers of all kinds can be in various states. It would be highly preferable to be able to distinguish these states as the privacy risk imposed by them varies significantly.

2.2 Bluetooth Low Energy

This section discusses some aspects of Bluetooth Low Energy (BLE) that are relevant to packet classification. However, it is outside the scope of this thesis to elaborate on the functionality, implementation, and potential upsides and downsides of Bluetooth Low Energy (BLE) to any meaningful extent. A detailed understanding of the underlying technology and implementation of Bluetooth is not relevant for classification. The sources for the content of this section are the Bluetooth Core Specification from page 2645 onwards [4] and the Core Specification Supplement 9 [5].

It is sufficient to have a high-level understanding of how BLE packets are structured and how this structure could potentially differ among packets and devices. Therefore, this section aims to explain how the structure of BLE packets can be used for feature extraction. As long as two packets differ in one feature, this is potentially already sufficient to differentiate them and classify them correctly.

2.2.1 BLE Packet Structure

Bluetooth Low Energy packets follow a certain structure. On the link layer, packets are structured as follows (Figure 2.2):

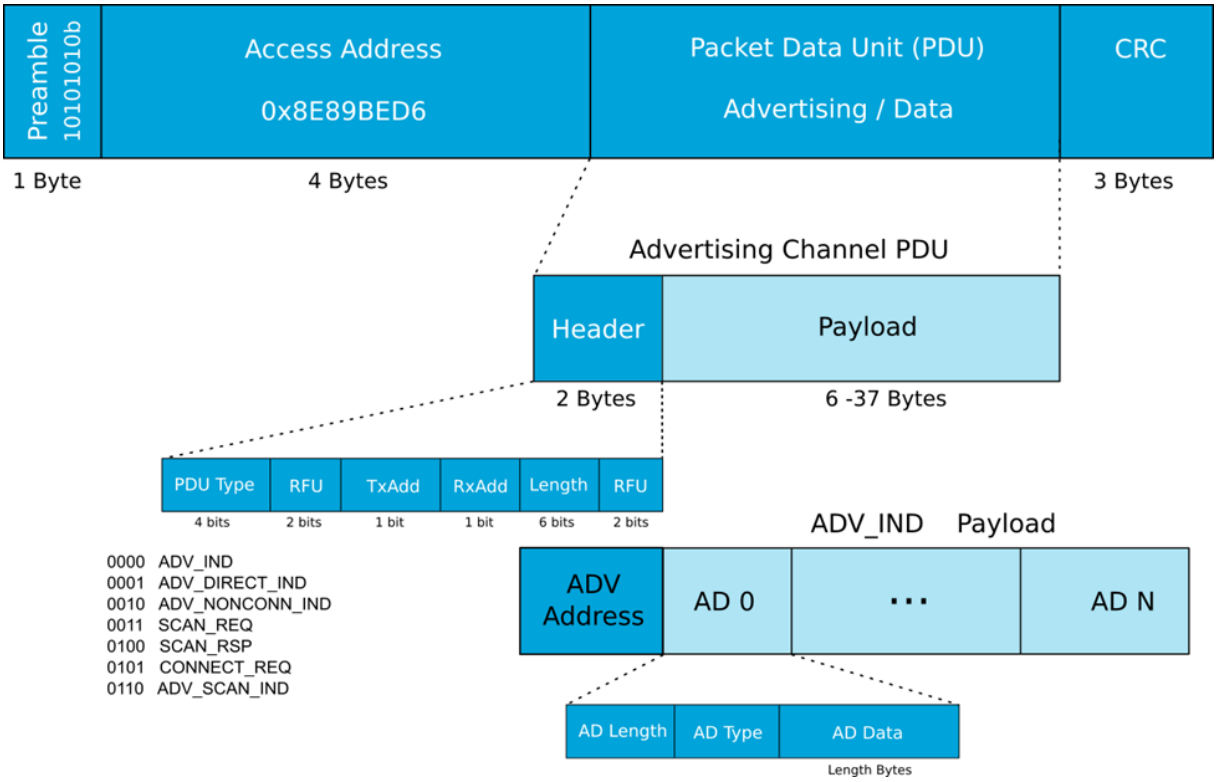

Figure 2.2: Packet Structure of a BLE Advertising Packet[6]

1. **Preamble:** This is used for various purposes, such as synchronization between devices, and is irrelevant for classifying packets.

2. **Access Address:** Most BLE packets share the same access address ”0x8E89BED6”, hence this is irrelevant for classifying packets.

3. **Protocol Data Unit (PDU):** There are two types of PDUs, the ”Advertising Physical Channel PDU” and the ”Data Physical Channel PDU”. For this thesis, only the first one is relevant. There are many types of PDUs, which are explained in more detail below.

4. **Cyclic Redundancy Check (CRC):** This is used for error detection and correction and is irrelevant for classifying packets.

From the general structure of BLE packets, only the PDU can reasonably be used for feature extraction, as the other three components are either (pseudo-)random or the same across all relevant packets. On a higher level, there are two types of PDUs: the ”Advertising Physical Channel PDU” and the ”Data Physical Channel PDU”.

The ”Data Physical Channel PDU” is only used on full connections, i.e., with a master and slave. To my knowledge and research, BLE trackers do not use full connections. Therefore, the ”Data Physical Channel PDU” is irrelevant to this thesis. The ”Advertising Physical Channel PDU” is typically used when devices do not need a full connection, for instance, in the context of master-slave detection or broadcasting; the latter is the case for any

BLE tracker. They broadcast packets that are meant to be picked up by finder devices to localize them.

The "Advertising Physical Channel PDU" itself has numerous sub-types. The most relevant ones are:

- ADV_IND
- ADV_DIRECT_IND
- ADV_NONCONN_IND
- ADV_SCAN_IND
- SCAN_REQ
- SCAN_RSP

All these PDUs above serve various purposes. However, for packet classification, a packet's purpose does not matter. It is only relevant that the PDU differs among packets.

The first four PDUs on the above list belong to a group called "Advertising PDUs", which are prominently used by trackers due to their most notable aspect, the payload. Like all "Advertising Physical Channel PDUs", they are led by a header indicating, among other things, the exact PDU type and length. A payload of variable length then follows the header.

This payload begins with the source address of the advertising device. Next, there can be an arbitrary number of advertisement data types. Each advertisement data type is a block containing some data following a predefined structure.

These blocks can be variable in length, depending on the data they contain. Some advertisement data types may also appear multiple times within the same payload of a PDU. Technically, it is also possible to have no advertisement data types within an "Advertising" PDU. Combined, all the advertisement data types may not exceed 37 Bytes in length.

Given that advertising data types can contain arbitrary data and that trackers frequently use the "Advertising" PDUs containing these advertising data types, it is relevant to investigate these data types further for classification as they possibly contain vital features for feature extraction.

2.2.2 Advertisement Data Types

There are many different advertising data types. Therefore, only the most relevant ones in the context of BLE trackers will be discussed here.

2.2.2.1 Manufacturer Specific

The manufacturer specific advertising data type consists of a company identifier from an assigned set of numbers (company ID). These numbers can be interpreted as strings, as every number is attached to a unique company name. For instance, 0x004c is the company identifier assigned to Apple. The company identifier is then followed by an arbitrary data set by the device manufacturer. The data can be of any length as long as the entire PDU does not exceed its maximum length.

2.2.2.2 Service UUID

In the context of Bluetooth, Service UUIDs describe unique identifiers used to identify manufacturers of devices or services provided by devices. On a higher level, the Service UUID advertising data type comes in two variants: the bar-bones Service UUID containing only the Service UUID and the Service Data UUID advertising data type including both a Service UUID and Service Data of some predefined length.

The Service Data UUID then has three sub-variants, depending on its data length: 16-bit, 32-bit, and 128-bit. Furthermore, those three variants can be ”complete” or ”incomplete”.

2.2.2.3 Flags

The flags advertising data type contains four boolean flags indicating the device's capabilities and state.

2.2.2.4 TX Power Level

The TX power level advertising data type indicates the power level at which packets are transmitted. This advertising data type contains a single 8-bit integer indicating the power level at which the packet was transmitted. This can be useful for estimating distances between devices.

2.3 Approaches for Machine Learning Device Classification

So far in this thesis, the term ”classification of packets” has been used loosely to describe the process of classifying BLE packets, i.e., assigning a device label to a particular packet based on its features. However, the ultimate goal is not to classify packets but rather devices. From a privacy point of view, it is only relevant whether a device is a potentially privacy-invading tracker. The packets themselves are irrelevant. Simply assigning a label to every packet is, therefore, not sufficient.

At this point, it becomes vital to differentiate between devices and source addresses. Every BLE packet contains a source address of the sending device. However, a device, i.e., a physical BLE transmitter, can simultaneously transmit BLE packets using various source addresses. In other words, every source address belongs to precisely one device at any point in time, but multiple source addresses might belong to one device simultaneously. Therefore, two packets might have differing source addresses when they are coming from the same physical device. This poses a significant challenge for this thesis, as it makes it impossible to classify devices based on BLE packets. It is only possible to classify Bluetooth source addresses, not Bluetooth devices. Therefore, this thesis proposes three methods for classifying source addresses (and technically not devices).

On a higher level, the first step in training the machine learning models is to extract features (such as the packet length) from the packets into one-dimensional feature vectors. Additionally, the label of the sending device must be known for every packet. In the next step, the labeled packets (i.e., vectors) are transformed into labeled training samples by various methods (these resulting samples can be of arbitrary shape). The final step in training is to use these labeled samples to train a supervised machine-learning model. In other words, the source addresses are irrelevant for training.

However, the above steps need to be slightly adapted for inference, i.e., the classification of previously unseen data, as the ultimate goal is to classify source addresses and not packages/samples. Firstly, the packets are transformed into unlabeled samples using the same transformation steps. Secondly, the samples are grouped by their source address. Next, all samples are fed through the machine learning model to generate the predictions for every sample. Given that there can be multiple samples per source address, there can also be multiple different predictions per source address. For instance, one sample might be predicted as a non-tracker and another sample as a tracker, even though both samples stem from the same source address. Therefore, it is necessary to pick one of the predicted labels from the samples to assign a label to the source address. One way of doing this is to choose the label with the highest occurrence per source address.

In addition to explicit features as described above (such as the packet length), it is also possible to incorporate implicit features into the training process. An example of such an implicit feature is the packet rate, i.e., the rate at which packets are sent. Therefore, this thesis proposes two approaches for including the packet rate as an implicit feature in the training process in addition to the plain classification of packets.

2.3.1 Packet Classification

The most straightforward approach for classifying source addresses is to directly classify the individual packets, i.e., the one-dimensional feature vectors. This simple method completely ignores the aspect of the packet rate.

The main advantages of this approach are:

- A single packet is sufficient for classification. There is no need for multiple packets to obtain a result.

- Every source address can be assigned a label. There can never be too few packets for classification, as a single packet is already sufficient.
- Classification, i.e., the inference is very fast for typical machine learning models as the size of the input data is bounded by the dimensionality of the input vector, i.e., the number of features.
- The preprocessing of the data is relatively simple. There is no need for complex aggregation or high-dimensional data structures (tensors).
- Many different types of machine learning models can be used with this approach. There is no restriction to specific kinds of models, such as deep learning models.

However, this simple approach also has its limitations. The most relevant ones are:

- There can be ambiguity in the classification of source addresses. If two or more packets were captured from a single source address, the classification of the various packages may differ. This ambiguity could be solved by selecting the majority class that was predicted most often for this source address.
- The package rate as an implicit feature is ignored. However, it would theoretically be possible to incorporate the package rate to some extent by adding an additional explicit feature, such as the time interval between the current and last packet. Those extensions are, however, not covered in this thesis.

2.3.2 Aggregated Packet Classification

To overcome the main limitation of the simple packet classification, i.e., the ignorance of the packet rate, one can extend this approach by adding aggregation.

First, the data preprocessing and feature extraction are executed as before, with the difference that the packets' timestamps and source addresses must be kept as features, i.e., a column in the table containing the data. Second, all other "real" features must be strictly numerical. Categorical features must be one-hot encoded (Table 2.1).

Time	**Source Address**	**Packet Length**	**Label**
0	2E-B0-D0-63-C2-26	160	AirTag
1	2E-B0-D0-63-C2-26	80	AirTag
3	2E-B0-D0-63-C2-26	200	Tile
4	2E-B0-D0-63-C2-26	80	AirTag
6	C9-2D-8B-7F-9B-A6	120	SmartTag
12	C9-2D-8B-7F-9B-A6	120	SmartTag
13	2E-B0-D0-63-C2-26	200	Tile
14	2E-B0-D0-63-C2-26	80	SmartTag

Table 2.1: Example Packets with Time (in seconds), Source Address, Packet Length (in bits), and Label

In the third and final step, the packets' feature vectors are averaged over a fixed non-overlapping time interval, e.g., 5 seconds. This means that for every source address, the packets are aggregated into one feature vector (i.e., a sample) for each time interval (resampled). This is done by first summing all samples for every feature, and then the intermediate result is divided by the length of the time interval. This results in features that represent the average value per second. For instance, the feature "packet length" (in bits) would be converted to the average packet length per second (in bits), which conveniently translates to the gross data rate of the transmitting source address. In case there are no packets within a resampling interval, a zero sample is generated indicating the absence of packets.

As a label for the aggregated samples, the label most frequent for every source address is picked. Afterwards, the models are trained on the aggregated samples (Table 2.2). Finally, for inference, all the preprocessing steps from above (with the exception of the pick of majority labels) must be executed before feeding the sample through a model. During inference, the label of the source address can then be determined by picking the predicted majority label across all the predicted labels of the aggregated samples.

Time	**Source Address**	**Packet Length**	**Label**
0	2E-B0-D0-63-C2-26	104	AirTag
5	2E-B0-D0-63-C2-26	0	AirTag
10	2E-B0-D0-63-C2-26	56	AirTag
5	C9-2D-8B-7F-9B-A6	24	SmartTag
10	C9-2D-8B-7F-9B-A6	24	SmartTag

Table 2.2: Aggregated Example Packets with a Resampling Interval of 5 seconds

It is important to add that in a perfect world, every source address should belong to a device with a fixed label. In other words, for a given source address, every packet should have received the same label in the training and test dataset. There should never be ambiguity over the correct label for any given source address.

However, mistakes can happen, and it is very much in the realm of possibility that a packet in the training data set is incorrectly labeled; therefore, there could be ambiguity over the correct label for a source address given a set of packets. To rectify such mistakes, the majority label is picked during aggregation (as seen in Table 2.2). In the case of (mostly) correctly classified packets, the pick of the majority label can not lead to an incorrect label for the aggregated sample. However, this is possible in the case of many incorrect labels (i.e., greater than 50 percent).

Compared to the baseline packet classification, the only tangible advantage is that the packet rate is appropriately modeled. As this method is based on the simple packet classification, it also inherits some of its advantages:

- The actual inference is still very fast, as the input data size is bounded by the dimensionality of the input vector.
- The same broad range of machine learning models can be used.

Some general disadvantages of this method are:

- For some source addresses, assigning a label might not be possible. Suppose packets were not captured over a time period exceeding the time interval used during aggregation. In that case, generating even a single sample is impossible, and therefore, no classification can be done. The longer the time interval used during aggregation, the larger this problem becomes, i.e., the more source addresses can potentially not be classified.
- There is a tradeoff between incorporating the packet rate and the number of source addresses that can be classified. If the time interval is very long, the packet rate becomes more important, but it might not be possible to generate samples for many source addresses (see point above). If the time interval is very short, the packet rate is practically ignored, but most source addresses can be labeled. Therefore, picking an appropriate time interval is crucial for the success of this method.
- A complex multi-step preprocessing pipeline is required on top of any existing preprocessing.

2.3.3 Recurrent Neural Networks

A third and final method for classifying BLE source addresses is to use recurrent neural networks. Recurrent neural networks come in many variants; one prominent example is the LSTM block. The main advantage of recurrent neural networks is that the input data can be a sequence of independent vectors, i.e., a matrix. This allows the models to learn longer-term dependencies between individual samples, such as the packet rate, or other patterns, such as alternating PDU types.

Before training a recurrent neural network, the data containing the feature vectors of the individual packets must be converted into a three-dimensional tensor (with shape ”Number of Sequences” x ”Sequence Length” x ”Number of Features”). A significant problem is that sequences of BLE packets are unevenly spaced. I.e., the time interval between two consecutive packets varies and is not fixed. It would be possible to train recurrent neural networks with unevenly spaced time series data, but this is not recommended if the results should be of any meaningful quality. Therefore, before converting to tensor, the sequence of BLE packets must be converted into an evenly spaced time series.

This conversion can be done using the same method as described in the subsection above about aggregated packet classification, albeit with a short time interval, such as one second. By choosing a short time interval, most packets are aggregated only with themselves, as there are no other packets within the same time interval. However, if no packets were captured within a time interval, the corresponding time slot would be filled with a zero vector. Therefore, the shorter the time interval for aggregation, the more zero vectors there are.

After converting to an evenly spaced time series, the data must be converted to a higher-dimensional tensor. First, a fixed sequence length is picked, i.e., how many sequence

elements (aggregated samples) are bundled into one two-dimensional sample. Here, the same tradeoff as discussed with aggregated packet classification applies. A high sequence length is beneficial to the accuracy and incorporation of the packet rate but might make it impossible to assign labels to some source addresses.

Next, the samples are grouped by the source address. For every source address, the aggregated one-dimensional samples are converted into two-dimensional matrices with shapes ”Sequence Length” x ”Number of Features”, and every matrix is assigned a label based on the majority label of the aggregated samples. Depending on the number of aggregated samples per source address and the chosen sequence length, some aggregated samples may be left over in the end (zero padding might help here).

Finally, all the two-dimensional samples of every source address are stacked into one tensor of the shape ”Number of Sequences” x ”Sequence Length” x ”Number of Features” (Figure 2.3). The vector with the target labels should be one-dimensional and of length ”Number of Sequences”. The recurrent models can then be trained on these tensors. For inference, the same complex preprocessing steps must be applied to the data before feeding it into the network.

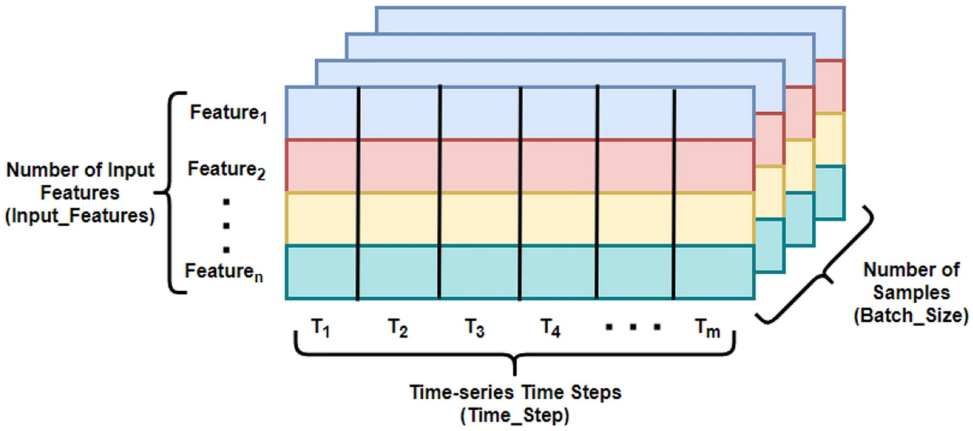

Figure 2.3: Structure of a 3D Tensor for recurrent Neural Networks [7]

The only positive aspect of this method that comes to mind is improved accuracy compared to the other two methods discussed before. In other words, this method should only be used if the performance obtained by the other two methods is insufficient. Some of the numerous disadvantages of this method are:

- It can only be used with deep learning, i.e., neural networks. Other models, such as decision trees, are not compatible.
- Very complex data preprocessing and high-dimensional data structures are required.
- As with the aggregated packet classification method, for some source addresses, it might not be possible to assign a label depending on the chosen sequence length.
- Potentially slow inference due to the nature of recurrent neural networks (the input data size is potentially massive, and every sequence element must be fed through the network individually).

2.4 Summary of Theory

The most relevant insights from this chapter are:

- Many devices can operate as BLE trackers, not only conventional trackers such as AirTags but also other devices such as iPhones.
- There are also privacy-invading tracking methods other than BLE, such as Apple Find My over the Internet.
- BLE trackers can be in various different states. Depending on the state, the tracker can be more or less privacy-invading. The ”lost” state is particularly privacy invading.
- BLE packets follow a predefined structure that can be used for feature extraction. The payload of advertising data types, such as manufacturer specific data, is especially interesting for feature extraction.
- It is impossible to classify BLE devices; therefore, source addresses are to be classified.
- For the classification of BLE source addresses, this thesis proposes three different approaches: Simple packet classification, aggregated packet rate classification, and recurrent neural networks with 3D tensors.
- These classification approaches vary significantly in complexity, limitations in application, and presumably accuracy. Each method has its advantages and disadvantages. Typically, there is a tradeoff between complexity (i.e., presumed accuracy) and practical applicability.

Chapter 3

Related Works and Initial Feasibility Experiment

3.1 Introduction

As of writing this thesis, there are only a few pieces of related scientific work. It is essential to note that the quality and thoroughness of the vast majority of the research is insufficient to classify BLE devices in the real world, let alone in high-density environments. The real world is **not** a laboratory. Often, authors conduct experiments on a small scale and then generalize inappropriately. This is highly problematic in real-world environments, as such generalizations can lead to false conclusions and will lead to sub-par classification results at best.

Therefore, this related work chapter is split into two parts. The first briefly covers other pieces of related work and discusses their key findings (and not-findings/deficits). The second part consists of an initial feasibility experiment I conducted. This experiment is also run on datasets collected on my own.

3.2 Related Works

3.2.1 IoT Device Classification

There is a lot of related work concerning the general classification/fingerprinting of IoT devices in typical Ethernet-based computer networks, such as the work in the paper "Automatic Device Classification from Network Traffic Streams of Internet of Things" [8]. Some examples of typical IoT devices are video surveillance cameras or smart TVs. These devices often must be connected to the Internet for full functionality, e.g., remote access to a video surveillance camera or streaming services on a smart television. However, compared to other online devices such as smartphones or computers, IoT devices often do not receive regular software and security updates, if they receive them at all. This

poses a significant security and privacy risk for home networks in the form of a potential weak point for network intrusion. This is especially problematic for devices equipped with cameras and/or microphones. Therefore, identifying IoT devices is relevant from a privacy point of view, similar to identifying BLE trackers.

However, as those IoT devices operate mostly over Ethernet or Wi-Fi and not Bluetooth, the results and insights are not directly transferable to the issue and research gap this thesis is trying to address. BLE packets and Ethernet frames (or IP packets) have very little in common. The only relevant takeaway from these works is the general methodology, i.e., data should be collected first, then analyzed, features extracted, and finally, machine learning models built.

The well-established CRISP-DM process model would be a more general framework for data-driven workflows. This model emphasizes an iterative procedure. Many process steps should and can be repeated at any point in time if necessary. Therefore, this thesis followed a flexible workflow inspired by the CRISP-DM model and the many papers on IoT device classification.

3.2.2 BLE Device Classification

Jie Liao's master thesis, conducted in 2023, is the work most closely related to this thesis's goal [9]. His thesis is also the first result on Google when searching for BLE device classifiers; therefore, it is presumably a highly relevant piece of literature.

In his thesis, Jie Liao collected data for various trackers and non-tracker devices. Some of the trackers covered in his thesis are the Apple AirTag, the Tile Slim, and the HuaweiTag from Huawei. He then trained various machine learning models (neural networks, decision trees, and support vector machines) to distinguish between tracker and non-tracker, i.e., binary classification. His methodology for classification follows the packet-based classification proposed in Chapter 2. The resulting classification accuracy on the test sets of well over 90% indicates that it is possible to distinguish trackers from non-tracker devices with machine learning models.

However, this thesis has numerous flaws, some of which are outlined in the following:

- The data was not collected thoroughly enough. There seems to be incorrectly labeled data.
- The process of data collection is undocumented. For instance, it is unclear how the data for trackers in the "nearby" state was collected and labeled.
- The conversion from the PCAP files from Wireshark to CSV files with extracted features is not documented.
- The feature extraction process is not documented either, and there is a significant discrepancy between the extracted features and the proposed (relevant) features based on the theoretical underpinnings of BLE. For example, the PDU type is not considered a feature.

- The states of trackers are ignored. A tracker in the unproblematic ”unpaired” state and one in the highly dangerous ”lost” state are both just classified as trackers.

Therefore, the key takeaway from this master thesis should be that it is possible to differentiate between tracker and non-tracker with machine learning models. Whether any further distinction, i.e., a more complex categorical classification, is possible is to be seen and examined by this thesis. Additionally, all the findings of this master thesis should be taken with a significant grain of salt as many highly relevant aspects of the workflow are not adequately documented.

3.2.3 Reverse Engineering the Apple AirTag

Many authors have extensively examined the ins and outs of the functionality of BLE trackers. One example would be the reverse engineering of the AirTag conducted by Adam Catley [10]. He tried to understand in detail how the AirTag operates, including a complete teardown of an AirTag. The entire work is very detailed, and the process is documented thoroughly. For instance, he used an nRF logic board similar to the one used in this thesis to capture the BLE packets. Some of his findings most relevant to this thesis are:

- AirTags have many different states of operation. His description of states includes an ”unregistered” state, a ”nearby” state, and a ”lost” state, but he also goes beyond that by defining many more states that are very specific to the AirTag.

  However, most of these additional states are irrelevant for device classification as they are often only encountered in absolute edge case scenarios and, for the most part, only last a few seconds at max.

- Depending on the state, the AirTag shows a broad behavior variance, even on the physical layer. For instance, the packet rate is dependent on the device state. This could be useful for feature extraction.

- The AirTag uses manufacturer specific advertising data in its BLE packets. The data also changes over time and is not constant.

It is important to note that I can’t entirely agree with all of his findings. My research in this thesis contradicts some of his conclusions. Finally, one should add that similar research exists for other tracking devices, such as the Samsung SmartTag.

3.2.4 Apple Continuity

Apple devices such as iPhones, Macs, and AirTags support a family of features called ”Continuity”. In short, continuity features allow users to use functionality and services across devices; an example would be the ”universal clipboard” feature that allows for copy-paste between different devices. Many of these features use Bluetooth packets for

communication between different Apple devices. One of these features is Apple's Find My network [11].

Many researchers have examined the structure of the Bluetooth packets used for continuity in detail. A particular point of interest is the manufacturer specific data. Most, if not all, of the BLE packets used for continuity carry manufacturer specific advertising data. The structure Apple uses within the manufacturer specific data on a bit-level is not public knowledge and, therefore, needs to be reverse-engineered. This is especially interesting, as a detailed understanding of this data potentially allows for feature extraction tailored towards the differentiation of states of AirTags.

On GitHub, there is a great collection and summarization of the findings of various research papers about Apple continuity and the structure of its BLE packets [12]. However, at this point, it is of the utmost importance to warn about the current state of research regarding Apple continuity and reverse engineering. The exhaustive data collection conducted during my thesis showed that the current state of research regarding this topic is lackluster at best.

Much of the current research seems incomplete, misleading, entirely wrong, or outdated. Therefore, the current state of research regarding Apple's continuity may serve as nothing more than inspiration rather than an actual source of knowledge. My thesis also showed that many types of BLE packets used for continuity are entirely undocumented to this date.

Finally, similar research exists for other tracker vendors, such as Samsung, albeit not as exhaustive as in Apple's case [13].

3.3 Initial Feasibility Experiment

In addition to the literature review, an initial feasibility experiment was conducted for this thesis. This initial experiment is heavily inspired by the work of Jie Liao's master thesis [9] and builds upon his work. For instance, it will be assumed that a binary classification "tracker" vs. "non-tracker" is principally possible, even though his results were not especially pleasing, with a classification accuracy sometimes as low as 90%.

3.3.1 Goal and Limitations

This initial feasibility experiment aims to determine a baseline for any subsequent work. In particular, I am interested in the following:

- Can the simple binary classification "tracker" vs. "non-tracker" be extended to a more complex categorical classification that also aims to identify the model of the tracker, i.e., Apple AirTag or Samsung SmartTag?
- What features need to be extracted to perform this categorical classification?

- How robust is this classification in the case of unseen devices, i.e., can the problem of open-set classification be solved effectively?
- Is machine learning necessary for this classification task, or can it also be solved with other, less complex artificial intelligence methods?

Some relevant limitations of this initial experiment are:

- The data collection process is omitted for brevity; this will be discussed later in this thesis. The datasets will be regarded as given and correctly labeled.
- Different states of trackers are not discussed. All data used in this experiment is of trackers in the "lost" state to keep complexity manageable.
- The obtained machine learning models will only be evaluated on a test set and not used for real-world inference.

3.3.2 Datasets and Devices

The data for this experiment was collected on various tracker and non-tracker devices. There is one separate dataset for every tracker device and exactly one dataset for all the non-tracker devices, as the data for these devices was collected with all devices transmitting simultaneously. All datasets used in this experiment will also be used later in this thesis. The data collection process for these datasets is discussed extensively in Chapter 4.

In this experiment, three popular tracking devices were used, these are:

- The AirTag from Apple, labeled as "**AirTag**"
- The SmartTag from Samsung, labeled as "**SmartTag**"
- Tile Mate from Tile, labeled as "**Tile**"

The following devices were used as non-tracking devices. All of them are in one dataset, and the entire dataset is labeled as "**other Device**":

- A Lenovo Yoga Laptop
- A Lenovo Tab 12 Pro tablet, with Bluetooth pencil and keyboard
- An Ultimate Ears Boom 2 speaker
- A JBL BT 510 headphone
- A Logitech K810 keyboard and an MX Anywhere 2S mouse
- A Samsung Galaxy S23 Ultra smartphone
- An Xbox One controller

3.3.3 Feature Extraction and Analysis

The feature extraction for this baseline experiment was kept as simple as possible. Therefore, only one feature, which seemed very promising for classification because it is presumably unique to this exact device, was picked for every tracker device.

Upon visual inspection of the three datasets of the tracker devices, a few aspects are noticeable:

- For every device, all the packets it transmits are very similar, if not identical, in structure and payload. There is little, if any, difference between individual packets of the same tracker.
- All three trackers transmit packets regularly on all three available advertising channels, typically every few seconds.
- All three trackers only transmit advertisements of the PDU type ADV_IND.
- All three trackers use a very similar structure for their packages. They either use manufacturer specific data or Service Data to transmit the public key.
- The complexity of the packet structure between the three trackers varies widely. For instance, the number of advertising data types ranges from one (AirTag) to three (Tile).

Based on these observations, the following two features were picked to differentiate the three tracking devices. For the other devices, i.e., non-tracker devices, no features were picked:

- The **company ID** in the manufacturer specific advertising data. The Apple AirTag uses manufacturer specific data and advertises Apple's company ID.
- The **UUID** in the Service Data. Both the trackers from Tile and Samsung use Service Data. However, their manufacturer's UUIDs differ (Tile and Samsung, respectively).

Following the above feature extraction, the company ID and UUID were one-hot encoded for modeling. However, only the company ID "Apple" and the UUID "Tile" and "Samsung" were kept as columns. The one-hot encoded columns of other company IDs and UUIDs were dropped. Additionally, a label column was added to label all packets with their corresponding class label. The table below visualizes the resulting rows (Table 3.1). For each class, one representative row is shown.

Company ID Apple	UUID Samsung	UUID Tile	Label
1	0	0	AirTag
0	1	0	SmartTag
0	0	1	Tile
0	0	0	other Device

Table 3.1: Examples of Rows after One-Hot Encoding

It should be clear that these feature columns make the four classes separable. However, to ensure this is the case among all packets and not just representative ones, the distribution of these feature columns, i.e., manufacturers, can be visualized in a plot.

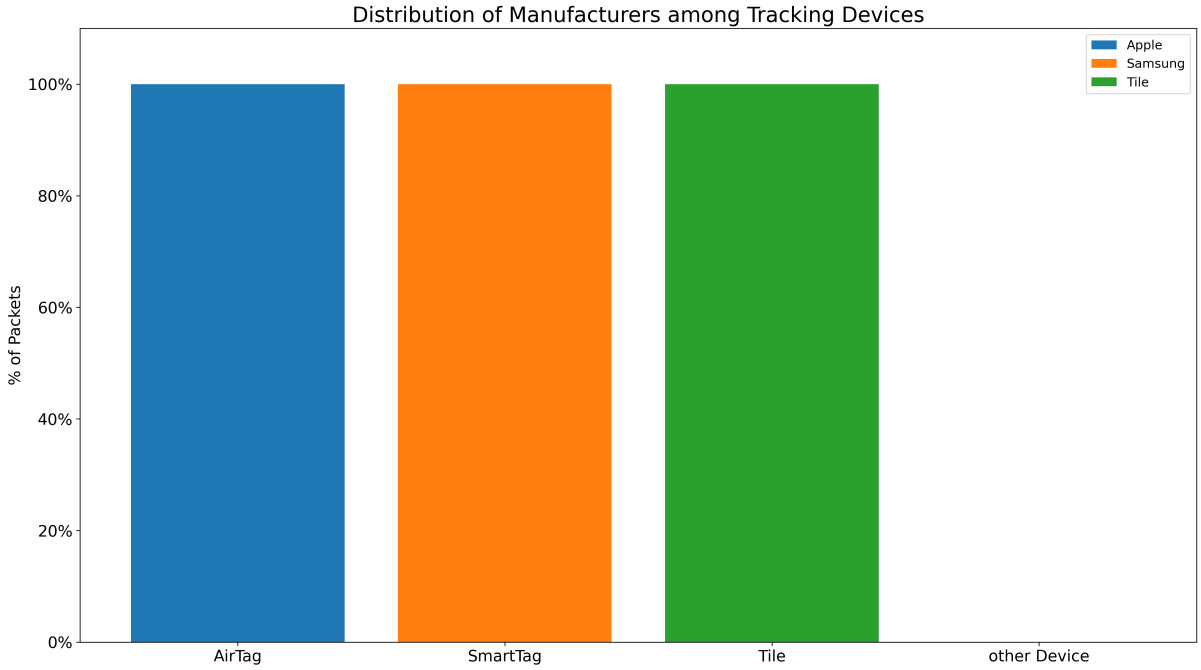

Figure 3.1: Distribution of Manufacturers among Tracking Devices

From the plot above, it becomes obvious that these one-hot feature columns are sufficient to distinguish all the classes for 100% of the packets (Figure 3.1). The trackers all have their unique company ID or UUID, and the ”other devices” never transmitted a package containing one of the company IDs or UUIDs used by the three trackers. This is especially interesting in the case of the Samsung SmartTag, as one of the other devices is a Samsung Galaxy S23 smartphone, a device made by the same manufacturer that could potentially use the same UUID (which is not the case).

In conclusion, the two extracted features (company ID and UUID) are likely sufficient to achieve a high classification accuracy. Therefore, the next step is to pick and train a machine learning model.

3.3.4 Modeling

The final step before the actual training of the model is the preparation of the dataset for training. After the above-described feature extraction and one-hot encoding, the following steps were taken:

- The dataset was balanced by under-sampling. In other words, sufficiently many labels were picked randomly and removed for every class so that the dataset contains

the same number of samples/rows for all classes. No samples were removed from the class with the fewest samples. After balancing, there were roughly 32'000 samples per class, i.e., 128'000 samples in total. This was done to ensure the model learns equally from all classes and generalizes appropriately.

- The balanced dataset was split into a training and a test set with a 75/25 split. The training set contains roughly 96'000 samples, and the test set roughly 32'000.

At this point, the dataset is ready for modeling. A simple model was picked for this initial experiment: a decision tree. Decision trees have advantages in that they are easy to understand/interpret and quick to train. As a library, sci-kit learn (i.e., sklearn) was used. The decision tree was instantiated with default parameters, i.e., no hyper-parameter tuning was applied to get a baseline performance. After training, the model was evaluated on a test set using a confusion matrix. The confusion matrix was normalized across the true label axis (Figure 3.2).

It is clear at first sight that the classification result is flawless. All classes are predicted with 100% accuracy in all cases. This shows that it is perfectly possible to classify devices (in this example, strictly speaking packets) based on BLE packets. Plotting this tree using the sklearn library makes it possible to understand the model's inner workings better (Figure 3.3).

The decision tree's visualization shows that the model learned as intended. The decision tree uses the provided features appropriately to separate the classes. For each tracker device, a unique feature column was used for classification, and if no feature was present, the model correctly predicted "other Device". However, to obtain this kind of result, it is not necessary to use a machine learning model as this can also be achieved with other less complex methods of artificial intelligence, e.g., an if-else statement (Listing 3.1).

```
def predictLabel(row):
    if bool(row['UUID Tile']):
        return 'Tile'
    elif bool(row['UUID Samsung']):
        return 'SmartTag'
    elif bool(row['Company ID Apple']):
        return 'AirTag'
    else:
        return 'other Device'
```

Listing 3.1: Manual Labeling

This if-else statement works similarly to the decision tree. It checks for the presence of a particular feature and predicts the corresponding class, e.g., "AirTag", in case the company ID Apple is present. The statement predicts "other Device" with the else block if all features are absent. This if-else statement can also be applied to the same test set as before, and its performance can be evaluated with a confusion matrix (Figure 3.4).

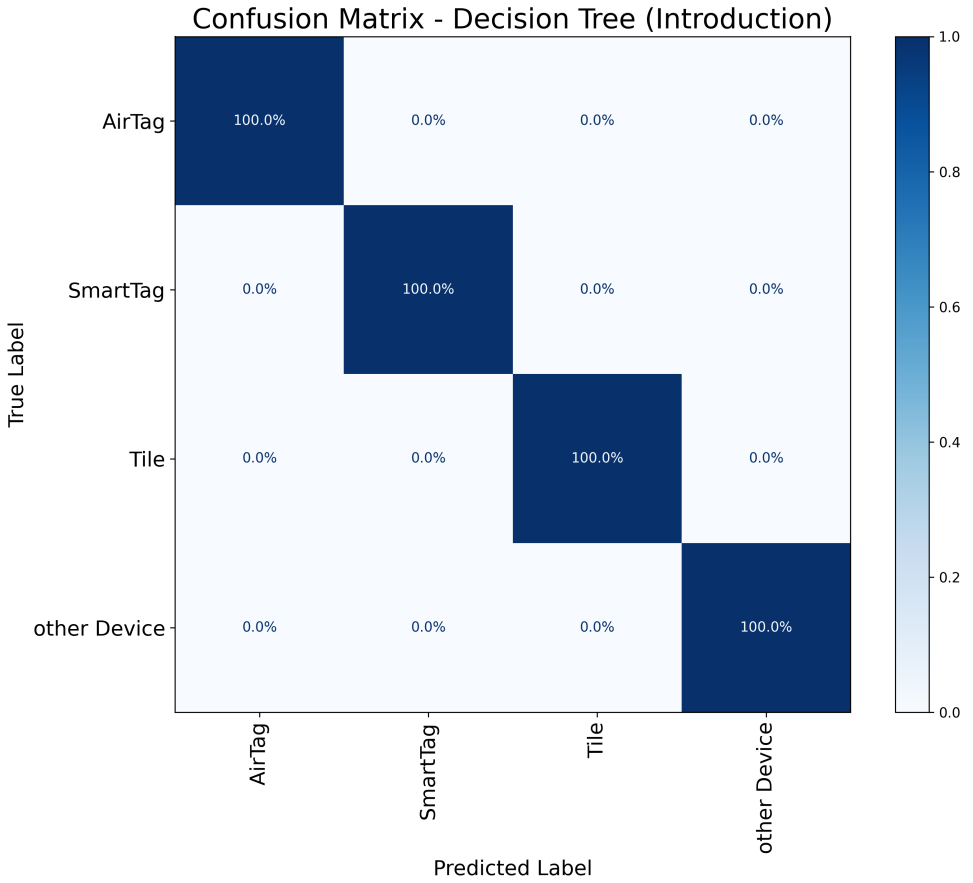

Figure 3.2: Confusion Matrix - Decision Tree (Introduction)

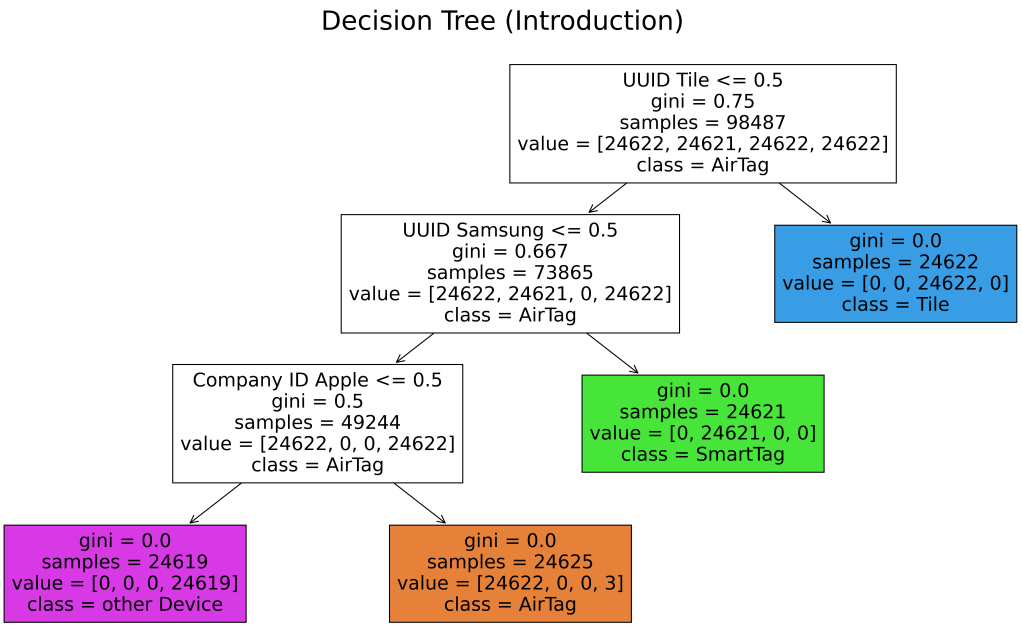

Figure 3.3: Decision Tree (Introduction)

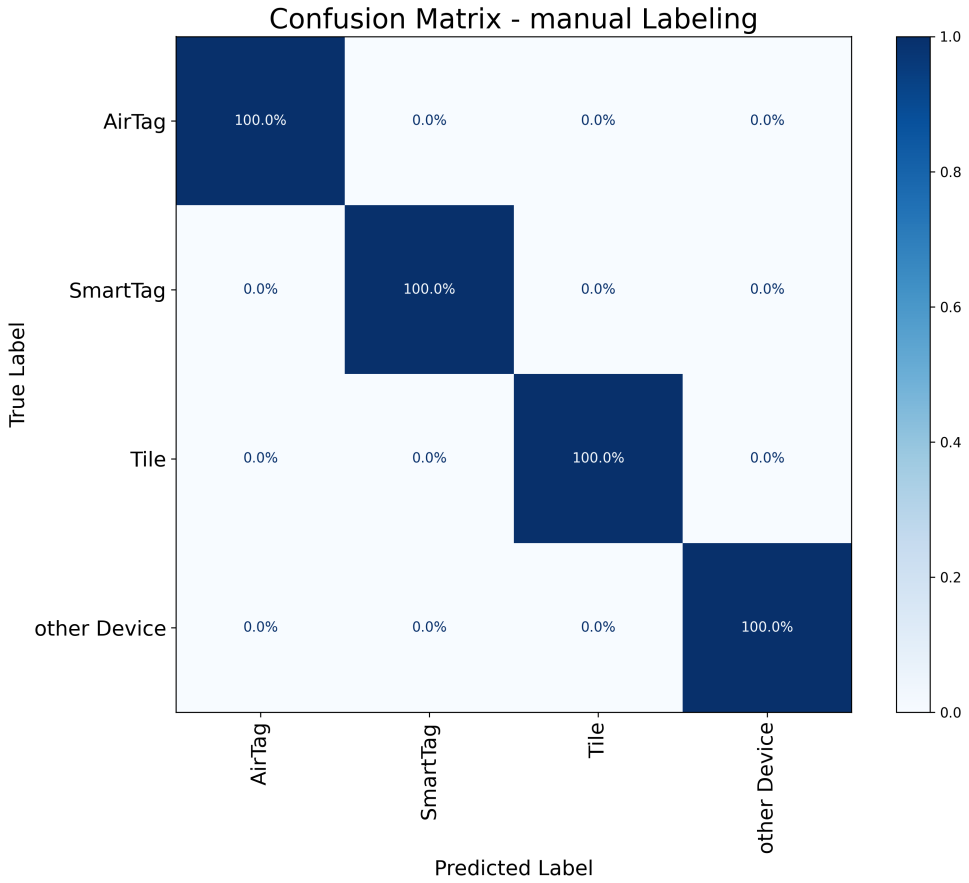

Figure 3.4: Confusion Matrix - manual Labeling

Unsurprisingly, the performance of the if-else statement is equal to the decision tree's performance. This is expected, given that the if-else statement perfectly mimics the decision tree. Therefore, it is not necessary to use machine learning for BLE device classification. However, given more features, it might still be convenient to use machine learning, as manual programming of if-else statements might become challenging and impractical.

3.3.5 Discussion of Results

So far, the scope of this experiment might have been small, but the flawless results are nonetheless interesting and certainly exceed expectations. As discussed before, the scope of this experiment is limited. For instance, states of trackers are ignored. However, as outstanding as the result might be, it also has its (significant) limitations, which shall be shown in the following subsection. What happens if new devices are added to the garbage class, i.e., the class labeled "other Device"?

To examine this, a second and third dataset labeled "other Device" were added to the existing pool of four datasets. These additional datasets contain BLE packets captured from an iPhone and a MacBook. As an iPhone and a MacBook are neither an AirTag nor

a SmartTag nor a Tile tracker, they were labeled as ”other Device”, even though they are trackers under the definition of this thesis. The goal is to understand whether a decision tree would still be able to distinguish between the three trackers and all the other devices.

All six datasets were subject to the same feature extraction and test/train split as before. Given that the datasets are balanced, the size of the training and test set stayed the same. Next, a new decision tree was trained on this new training set (including data on the iPhone and MacBook labeled as ”other Device”). When the resulting model is evaluated on the test set with a confusion matrix (Figure 3.5), the results are mostly the same as before. However, for the majority of samples (57%), the other devices are misclassified as ”AirTag”. In other words, the model cannot distinguish between the AirTag and the other devices.

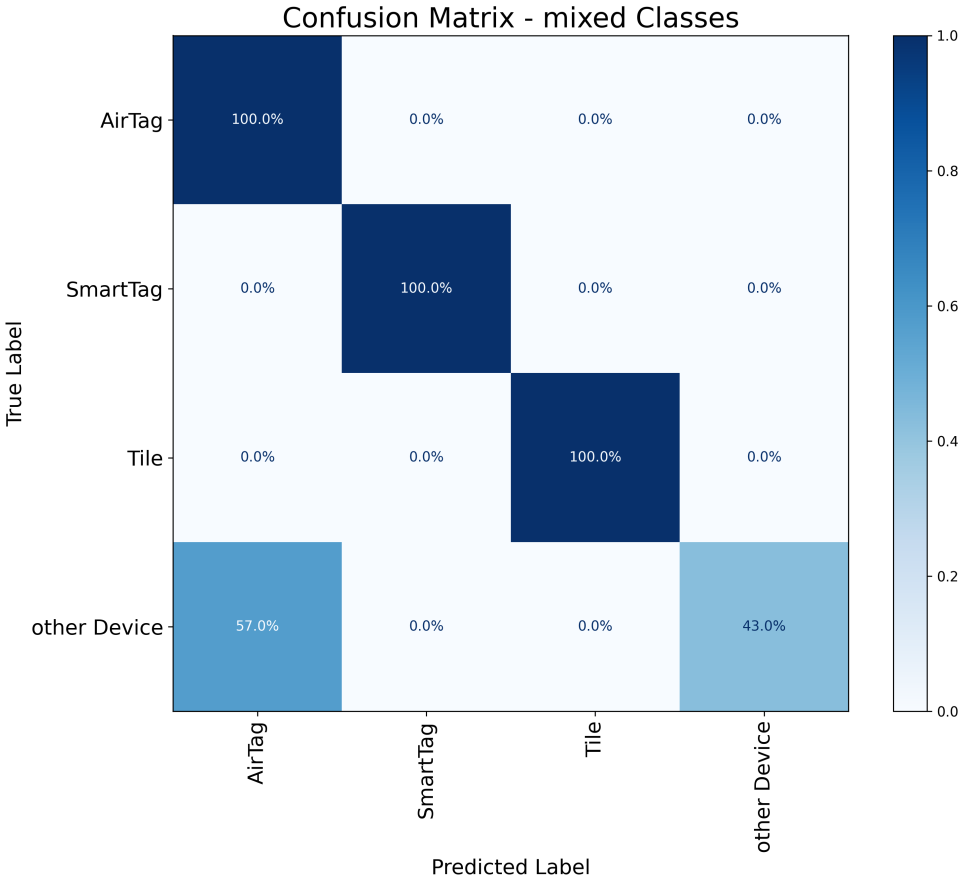

Figure 3.5: Confusion Matrix - mixed Classes

However, the distribution of the manufacturers (i.e., the one-hot encoded feature columns) provides a perfectly reasonable explanation for this mediocre performance (Figure 3.6). Many packets labeled as ”other Device” also show the company ID Apple feature, i.e., the packets contain the Apple company ID.

This can be explained given that iPhones, MacBooks, and AirTags are all made by the same company, Apple. Because Apple uses a very similar packet structure across all their

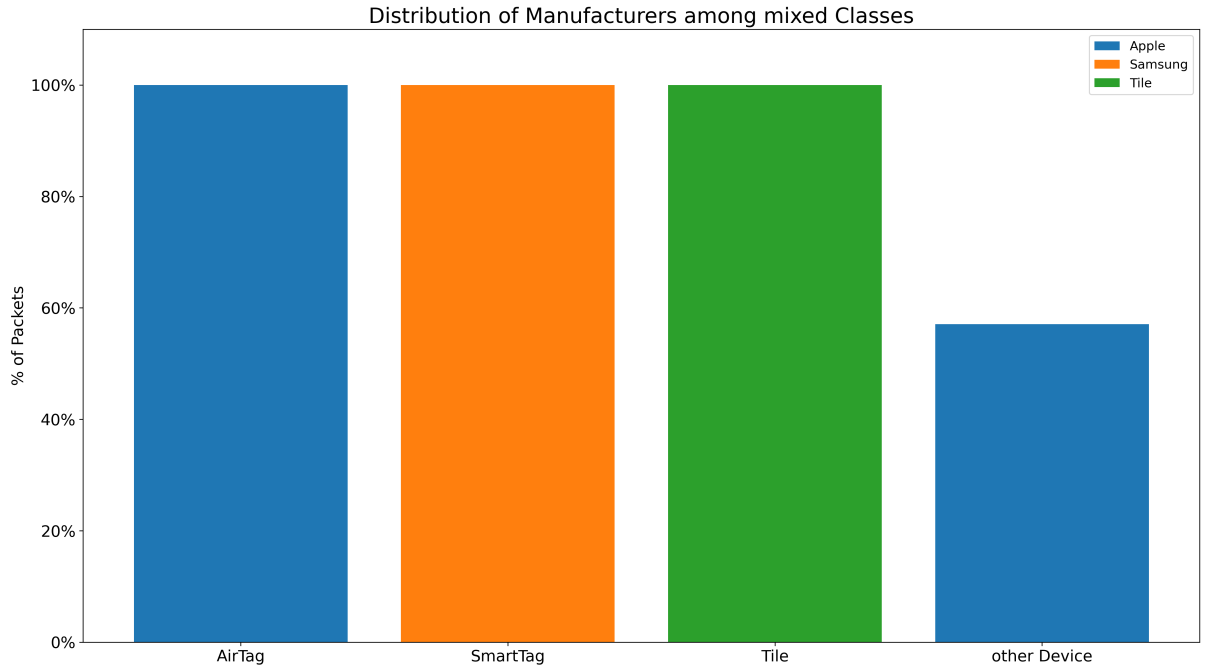

Figure 3.6: Distribution of Manufacturers among mixed Classes

devices and BLE packets, their company ID is not exclusively used by the AirTag. Hence, a decision tree cannot separate these devices.

This false positive detection of other devices, such as AirTags, is highly problematic from a practical point of view. Let's assume that in real-world environments, the ratio between the number of AirTag packets and other devices is 1:100, i.e., 1 of 101 BLE packets are coming from AirTags, and the other 100 are from other devices. This is reasonable as there are many more other devices out there than AirTags.

If 57% of packets of these other devices are falsely classified as "AirTag" and 100% of the AirTag packets are correctly classified as "AirTag," then 57 out of 58 detections (98.3%) of AirTags would be false positives. In other words, in 57 out of 58 cases (98.3%), a device detected and classified as AirTag would not actually be an AirTag but an other device. This would render the classification model useless in a practical application.

3.3.6 Conclusion

This initial feasibility experiment outlined several important flaws of current work and how to improve upon this work to obtain a machine learning model for the real-world detection of tracking devices. Some of the most relevant conclusions for this thesis are:

- It is not sufficient to collect data from a few devices. Data should be collected from as many devices as possible and feasible.
- The collected data should include similar devices, i.e., devices made by the same vendor. Similar devices are likely to share features such as the packet structure or

payloads with identical company IDs and have a potential for false-positive classifications.

- The feature extraction should be targeted toward the uniqueness of features for specific devices, as this worked well for some devices, such as the Tile tracker. In other words, features that are promising for classification should be picked. Features that are the same across many devices should be avoided, as they are not helpful for classification.
- The extracted features should be analyzed before modeling. If, and only if, the features allow for the unique identification of all classes, the classification will be successful. Otherwise, there is a risk of high false-positive rates.
- Machine learning models are technically not required for this task, but given the complexity, they are highly advisable.

3.4 Summary of Related Works and Initial Feasibility Experiment

The most relevant insights from this chapter are:

- The current state of research does not cover categorical tracking device classification. However, plenty of work exists on general IoT device classification. This is valuable guidance for my thesis.
- Existing research is often not thorough enough to tackle the problem of real-world classification. Data collection and general documentation, in particular, seem to be a problem.
- There is related work on the reverse engineering of tracking devices and their implementation at a packet/protocol level. This could be useful for feature extraction.
- The initial feasibility experiment showed that categorical classification of tracking and non-tracking devices with high accuracy is possible, albeit within a minimal scope.
- Categorical device classification is somewhat tricky, even within a limited scope. A lot of data and sophisticated feature extraction are required for acceptable performance.
- The more advanced classification of device states might be more challenging due to presumed similarities in BLE packets. The current state of research does not cover this.
- Whether or not categorical device classification in the real world is possible, and if so, to what extent and quality is to be seen and shall be answered by this thesis.

Chapter 4

Dataset Generation

The first step to creating machine learning models is to collect training data. There are two fundamentally different approaches to data collection.

1. Collect existing data from foreign sources. Examples of such sources are typically platforms like Kaggle or GitHub.
2. Collect data yourself without relying on foreign sources.

As established in the previous chapter, the dataset for this thesis must meet the following requirements cumulatively:

- For all the various device states, data is required. If the model should be capable of distinguishing device states, it must have been trained on data of all relevant states. It is not sufficient to have data from some states and none of others.
- Data for both the conventional BLE trackers, such as AirTags, and other BLE tracking capable devices, such as iPhones, should be included. As mentioned before, a model can only recognize devices whose data it was trained on.
- Many other relevant devices besides the trackers should be covered, as the more devices the models can be trained on, the better the inference results will be. Here, relevance refers to their potential to be seen in a real-world environment.

  Given that Zürich central station was selected for the real-world environment, devices typical for that environment should be present in the datasets. Such devices include smartphones, tablets, headphones, or laptops (list not exhaustive). An atypical device would be, for instance, a smart fridge, a television, or a car. Such devices are generally not seen at train stations and, therefore, only have a negligible chance of showing up during inference.
- For each device and state, many samples, i.e., packets, should be captured. The more samples a model is trained on, the better.

This is especially true for deep neural networks. Deep neural networks have many trainable parameters; hence, much data is required. This is even more true for recurrent neural networks. Firstly, recurrent neural networks have more parameters than comparable non-recurrent networks (due to the additional weight matrices for the recurrence). Secondly, every training sample is a matrix consisting of individual sequence elements, i.e., every packet can be a sequence element. Therefore, the number of training samples (matrices) can be as low as the number of packets divided by the sequence length.

Additionally, any aggregation will reduce the number of training samples. For instance, if a device's packet rate is 10 packets per second and every 15 seconds of packets are aggregated into one sample to incorporate the packet rate (i.e., apply the aggregated modeling approach as seen in Chapter 2), then the number of training samples reduces from 600 samples per minute down to 4 samples per minute (one for every 15 seconds). This is a reduction by a factor of 150.

As of writing this thesis, I am unaware of any existing dataset that would suffice its needs. For instance, a search on the data science platform Kaggle yielded no results. Therefore, I was forced to collect an extensive dataset. The following sections of this chapter will go into great detail, explaining and discussing this extensive process. The entire data collection took place roughly from February to April 2024. Approximately 30 Million packets were captured over a time frame of more than 600 hours.

However, not all data collected is directly used in this thesis. The dataset provided on GitHub does not contain all the data I collected. It only contains the data used to create the written thesis, i.e., what was required in the Jupyter Notebooks found on GitHub.

Roughly half of the collected data is not on GitHub. This can have several reasons:

- The data was not collected in a format usable for the final result, for instance, over a too short time period.
- The data contains errors, such as a device not being in the desired state.
- The data was (no longer) needed.
- The data served a purpose other than training or analysis, such as validation of results.

4.1 Device Overview

The devices needed for this thesis can be roughly put into three categories:

- **Conventional BLE trackers**, such as the AirTag, the SmartTag, or the Tile.

- **Other BLE tracking capable devices.** This category includes mainly Apple devices ranging from iPhones to MacBooks to accessories such as AirPods headphones. All devices capable of being tracked via Bluetooth tracking networks that are not in the category above are in this group. Often, these devices are also trackable via the Internet, i.e., they are non-Bluetooth-only trackers.

- **Other BLE devices** that belong into neither of the categories above. There are countless examples of devices in this category. From a practical point of view, almost all Bluetooth devices are in this category. In machine learning, this category is called a garbage category or garbage class. Sometimes, these devices are trackable via the Internet (such as Android phones).

The following subsections detail the selection of specific devices and the reasoning behind the respective choices.

4.1.1 BLE Trackers

The BLE trackers were selected based on popularity in the Swiss tracker market. The more popular a BLE tracker is, the more likely it appears during inference. The popularity was evaluated by visiting Switzerland's most popular online marketplace, Digitec Galaxus. All BLE trackers available on the said marketplace when writing this thesis were ordered and are part of the dataset of this thesis. These trackers are:

- The AirTag from Apple operating on Apple's Find My network. This tracker is labeled with the class label "AirTag".

- The SkyTag from 4Smarts operating on Apple's Find My network. This tracker is labeled with the class label "SkyTag".

- The One from Chipolo operating on Apple's Find My network. This tracker is labeled with the class label "Chipolo".

- The SmartTag from Samsung operating on Samsung's Galaxy Find network. This tracker is labeled with the class label "SmartTag".

- The Mate from Tile operating on Tile's network. This tracker is labeled with the class label "Tile".

Some vendors, such as Tile, sell multiple very similar tracker models. For practical reasons, only one of each vendor's trackers was selected, given that they presumably use an identical approach for tracking, i.e., the same packets or similar packet rates.

As a side note, other BLE trackers are available from popular vendors like HUAWEI. However, not all trackers are available on the Swiss market, let alone popular. As a point of reference, Apple and Samsung have a cumulative market share of over 70% in the local smartphone market [14].

4.1.2 Other BLE Tracking capable Devices

In addition to the conventional BLE trackers, some non-conventional trackers were considered. These devices are also trackable through Bluetooth tracking networks. Some relevant examples are:

- Most Apple devices are trackable via their Find My network. This includes iPhones, iPads, MacBooks, AirPods, and other devices. Given the widespread adoption of Apple devices, Apple's Find My network is Switzerland's most relevant tracking network.
- Some devices support tracking via Tile's tracking network. Examples of these devices are laptops from HP and Lenovo.
- As of May 2024, Google launched its own Find My Device network in Switzerland, making many Android Phones capable of BLE tracking [3].

Given the large number of devices available in this category, a selection had to be made to limit the scope of this thesis. It would be best to capture all devices exhaustively; however, this is not feasible for a Bachelor thesis. Therefore, devices were chosen based on relevance and availability.

Given that Google's tracking network only launched at the very end of this thesis, no devices from this tracking network are part of this thesis. No data was collected for any of these devices. Additionally, the dataset captured at Zürich central station was captured before the launch of Google's Find My Device network. Therefore, this inherent training data bias does not skew the final results.

The Tile network only supports a handful of these devices. As none of them were available to me, I could not collect data for them except for the Tile tracker itself. The resulting data bias is negligible, if not irrelevant, given that none of the supported devices seem relevant to me from an adoption point of view compared to other devices, such as Apple or Samsung products.

This leaves the Apple Find My network as the final one to consider. Given the widespread adoption of Apple devices in Switzerland, as many devices as possible should be covered in this thesis. It is safe to assume that any Apple device is highly relevant from an inference point of view. Omitting a device from the dataset must result in relevant data bias. Therefore, I collected data on all the Apple devices I could access. The Apple devices included in the dataset are:

- An iPhone 11. This device is labeled with the class label "iPhone".
- A 3rd generation iPad Pro 12.9". This device is labeled with the class label "iPad".
- A 2019 MacBook Pro 15". This device is labeled with the class label "MacBook".
- 3rd generation AirPods. This device is labeled with the class label "AirPod".

Another important Apple device that should have been considered is the Apple Watch. However, I did not have access to one for this thesis.

4.1.3 Other BLE Devices

Data on other BLE devices, in addition to tracker devices, needed to be collected. It would be best to collect data exhaustively from as many devices as possible. The more devices in the training set, the smaller the data bias. However, given the availability of devices, the following devices were chosen to be included in the dataset. All of these devices are labeled with the class label "other Device":

- A Lenovo Yoga Laptop
- A Lenovo Tab 12 Pro tablet, with Bluetooth pencil and keyboard
- An Ultimate Ears Boom 2 speaker
- A JBL BT 510 headphone
- A Logitech K810 keyboard and an MX Anywhere 2S mouse
- A Samsung Galaxy S23 Ultra smartphone
- An Xbox One controller

This thesis's single most significant limitation is that the garbage class of other devices only contains data from these few devices. This must hurt the model's capability to generalize sufficiently and will impact the final results during inference (negatively). Presumably, the only way to tackle this limitation would be to use synthetic data as an exhaustive collection of data from all available BLE devices seems infeasible. Popular approaches for data synthesis include rule-based approaches or the usage of deep learning in the form of variational autoencoders (VAEs) or general adversarial networks (GANs). However, this is outside the scope of the written thesis and was covered in the final presentation on the 15th of July instead.

4.2 BLE Capture

After selecting the devices for the training dataset, the data needed to be collected, i.e., BLE packets had to be captured. There are two fundamentally separate challenges to tackle when it comes to dataset generation for categorical classification:

- The data collection process, i.e., how the BLE packets are captured.
- Labeling the collected data. Only labeled data is useful for the training of supervised machine learning models/categorical classification.

This section will cover the first of these two challenges: physical data collection. The following section covers the significantly more challenging labeling of the collected data. However, it isn't easy to separate these two aspects. Therefore, this section on capturing BLE packets will already discuss certain aspects relevant to labeling.

4.2.1 Process of BLE Capture

In general terms, capturing BLE packets means sniffing packets from transmitting devices with a receiver. However, raw bit data for BLE packets is not necessarily useful. Therefore, the packets must be dissected in the next step, i.e., interpreted on a raw bit level. These dissected packets then need to be stored on disk and converted to a data format usable for machine learning models, preferably in a tabular form. The following subsections will detail the process of sniffing, dissection, and storage of BLE packets.

4.2.2 Packet Sniffing

An nRF 52840 DK logic board from Nordic Semiconductor was used for packet sniffing. This logic board is typically used for Bluetooth device development and is capable of Bluetooth 5.4, the latest version of Bluetooth, which includes support for Bluetooth Low Energy.

The nRF 52840 DK can sniff all three primary advertising channels, 37, 38, and 39, quasi-simultaneously. The manual indicates that the sniffer hops through the channels one after another, i.e., it starts at 37, proceeds through 38 to 39, and then begins at 37 again ([15]). Principally speaking, this is not simultaneous. However, this hopping happens at a rate exceeding 1000 Hz based on my measurements (and it might be much higher). Therefore, this can be considered quasi-simultaneous.

The nRF 52840 DK is connected to a computer via a USB Micro B cable. All the captured BLE data is sent via the USB cable to the computer where any further packet dissection takes place. For the packet sniffing, default settings were used on the nRF 52840 DK logic board.

However, while sniffing packets from wireless devices, one will eventually encounter an inevitable obstacle: traffic from other devices. From a dataset generation point of view, it would be highly preferable to capture individual devices in radio isolation without sniffing packets from other unwanted devices. If the sniffer only captures packets from one device, it is straightforward to label them because there is only one device. If the sniffer also captures packets of other unwanted devices, the labeling process would become very challenging, especially if source address randomization comes into play. Therefore, the nRF 52840 DK and the device from which BLE packets must be captured should be isolated from any unwanted background traffic.

Because the nRF 52840 DK is capable of real-time packet filtering by RSSI values, perfect radio isolation is unnecessary. Background devices with low RSSI values below -70 dBm were filtered out in real-time. However, (almost) perfect radio isolation is necessary for some devices. All trackers except for the Tile tracker can sometimes switch states very quickly, i.e., if the owner device comes near, the tracker will almost immediately switch from the ”lost” state to the ”nearby” state. Therefore, trackers need to be perfectly isolated from any owner devices.

Figure 4.1: The Faraday Cage (metal Box)

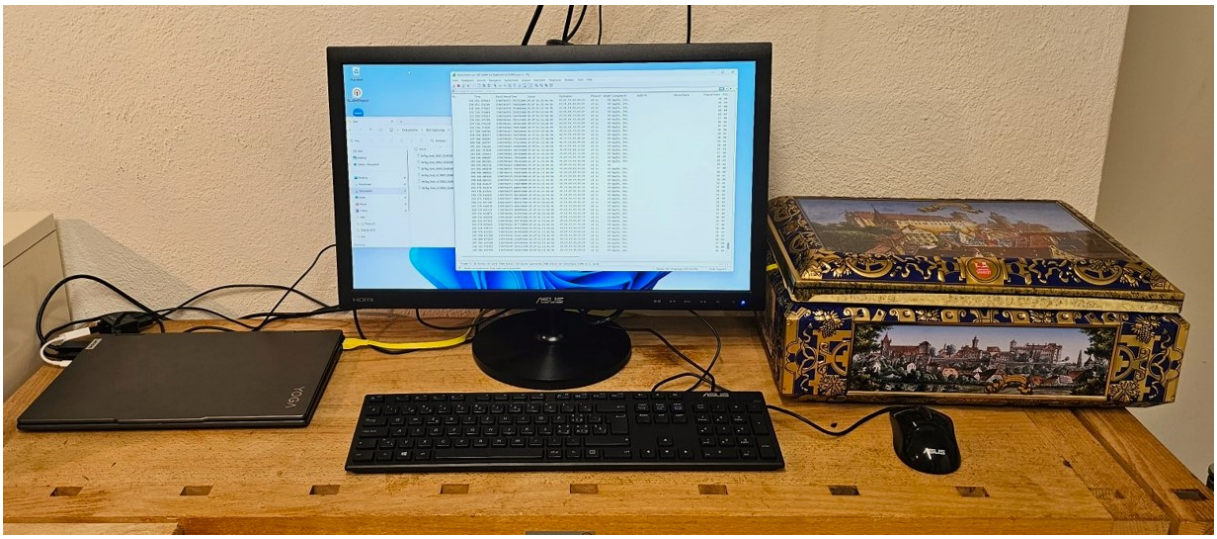

Figure 4.2: The Desk Setup used for BLE Capturing

To achieve BLE packet sniffing in radio isolation, I crafted a metal box lined with multiple layers of aluminum foil on the inside (Figure 4.1). The device to capture and the sniffer were placed inside this box, and the lid was closed. Additionally, the entire setup for capturing was placed on a workbench in my basement, where no other Bluetooth devices were within the same room (Figure 4.2). At this point, the metal box and the basement walls act as a Faraday cage and isolate the sniffer and the devices inside from the outside world. A flat wire was used to connect the sniffer to the computer outside the box, and Bluetooth was turned off on this computer.

4.2.3 Packet Dissection and Storage

Wireshark, a popular networking application, executed the packet dissection. Nordic Semiconductor, the manufacturer of the nRF 52840 DK, provides a plugin for the automatic dissection of the packets in Wireshark. The content of the packets is then presented in a human-readable form. Finally, the captured packets are stored in PCAP files generated by Wireshark.

4.2.4 Conversion to tabular Form

The PCAP file generated by Wireshark is not directly usable for machine learning as it lacks any tabular or tensor-like structure. Therefore, all the PCAP files were converted to CSV files. Wireshark provides direct support for exporting from PCAP to CSV.

Every row in the exported CSV files represents one captured packet, and every column represents one extracted feature. Therefore, the extracted features must be chosen before conversion to CSV. Features were extracted using Wiresharks' profile feature. Profiles in Wireshark allow the definition of columns with custom content, such as the packet length, the source address, a timestamp, the manufacturer specific data, and many more. These custom columns are defined for the display of PCAP files within the Wireshark application. When exporting a PCAP file to CSV, all columns specified in the profile are exported as columns in the CSV file with the same column name and order.

The exported CSV files can then be imported with Python libraries such as pandas. From this point onward, all work, including the labeling process, can be done in Python.

4.3 Labeling

Labeling describes the process of assigning a class label to a data sample. In this case, the samples to be labeled are the BLE packets. It would principally be possible to label source addresses instead of packets. However, this would only complicate the preprocessing, as labeled packets are required anyway for all three machine learning approaches proposed in Chapter 2 for training.

There are three fundamentally different approaches for labeling packets depending on how many devices were captured at a time:

- A single device was captured at a time, i.e., there was exactly one device within the metal box (Faraday Cage) next to the BLE sniffer.
- Multiple similar devices were captured simultaneously, i.e., there were multiple devices within the metal box (Faraday Cage) next to the BLE sniffer, and these devices share the same class label.
- Multiple different devices were captured simultaneously, i.e., there were multiple devices within the metal box (Faraday Cage) next to the BLE sniffer, and these devices do not share the same class label.

The following subsections discuss the labeling process for these three scenarios in detail.

4.3.1 Labeling of one Device

Labeling the packets is straightforward if only one device is captured. Assuming the capturing occurred in radio isolation from background Bluetooth traffic, all packets within the resulting CSV file stem from this one device. All packets in the file are labeled with the class label of the captured device.

This scenario applied to most of the data collected, i.e., most often packets of only one device at a time were captured. Additionally, source address randomization is also irrelevant. Even if the source address changes, it can still be assigned to one device, as only one device was captured.

4.3.2 Labeling of multiple Devices with same Class Label

If multiple devices are captured simultaneously and share the same class label, all packets in the CSV file are labeled with the corresponding label. This scenario applied only to the ”other devices”, as all other devices were captured simultaneously into one PCAP file.

4.3.3 Labeling of multiple Devices with different Class Labels

If multiple devices are captured simultaneously and do not share the same class label, labeling packets suddenly becomes incredibly challenging. The approach described above is no longer applicable. Labeling multiple devices with different class labels was one of the hardest problems to solve for this thesis. Therefore, first, the scenarios in which this is applicable are discussed, and then a solution to this problem is outlined in detail.

4.3.3.1 Applicable Scenarios

The scenarios for this type of labeling are rather limited. In principle, there are two scenarios for which this is applicable:

- Trackers in the ”nearby” state. A tracker in the ”nearby” state must be in close proximity to its owner device. Therefore, the owner device and the tracker have to be captured simultaneously, and obviously, the tracker and the owner device never share a class label.
- The AirPod when it is connected to another device. The same logic as above applies.

4.3.3.2 Naive Approach based on Source Addresses

A naive approach to labeling multiple devices with different class labels is to use the source address as a device identifier. In other words, if the source addresses of all the captured devices are known, the packets can be labeled based on these known source addresses.

Due to source address randomization, this naive approach is only applicable to datasets captured over a very short time frame (maybe a few minutes). Most devices, including many trackers, use randomized source addresses, i.e., the address changes every few minutes. Therefore, this approach is unsuitable for data collection on a larger scale over hours on end. However, as discussed before, such extensive data collection is required to train machine learning models, especially deep neural networks. Therefore, naive labeling based on source addresses is unsuitable for this thesis.

4.3.3.3 Machine Learning Approach for Labeling multiple Devices with Different Class Labels

Labeling multiple devices with different class labels is not easy. Therefore, this thesis proposes a rather general machine-learning-based approach that is transferable across various devices and scenarios as long as only **two** devices are involved. This approach does not cover the case of three or more devices. The general idea proposed in this thesis is to train a semi-supervised machine learning model that can automatically label datasets captured on the same devices. This offers flexibility and scalability as, theoretically, datasets of arbitrary length can be labeled with such a machine learning model.

The high-level idea is to exploit the naive approach described above to generate a small labeled training dataset with a few thousand samples. Next, a machine learning model, in this case, a neural network, is trained on this small labeled dataset (i.e., supervised learning). Afterward, the already-trained base model is trained again on a much larger unlabeled training dataset using a self-training classifier to increase the robustness of the model. This learning is unsupervised, as the data is not labeled. Finally, the improved model can be evaluated on a small test set labeled with the above-described naive approach (Figure 4.3).

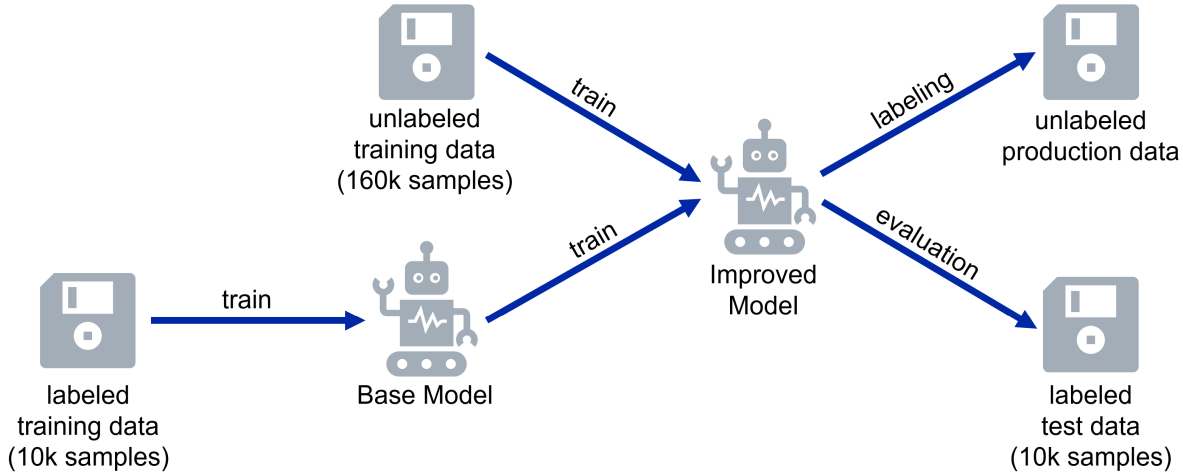

Figure 4.3: Semi-Supervised learning

This improved model can then be used to label any unlabeled production data containing the devices used during training in their respective states. I.e., a model trained on a dataset from an AirTag in its ”nearby” state in proximity to an iPhone can label any dataset containing packets from an AirTag in its ”nearby” state and an iPhone. This approach will often be referred to as ”automatic labeling” in this thesis.

On a lower level, the first issue is implementing naive labeling, especially given that devices can use more than one source address at a time and that these addresses might initially be unknown. Naive labeling for smaller datasets with unknown source addresses and two devices can be implemented as follows:

1. One out of the two devices is placed inside the metal box for capturing, and the capturing process is started. The other device remains switched off. Typically, the device put into the box first is the tracker, which commonly uses only one source address at a time.
2. A few packets of the first device are captured. This reveals the source addresses of the first device (Table 4.1).
3. The second device, previously switched off, is turned on, i.e., Bluetooth activated.
4. The tracker (i.e., the first device) will now switch into its ”nearby” state, and both devices are captured simultaneously. All source addresses not belonging to the first device must belong to the second device if packets are captured in radio isolation from any other devices (Table 4.2).
5. Once the source addresses of the first device are no longer visible, the capturing is stopped. At this point, address randomization has kicked in, and the addresses of the first device are no longer known (Table 4.3).
6. The captured dataset is clipped. The first packet in the clipped dataset is the first packet captured from the second device. The last packet in the clipped dataset is the last packet captured by a known source address of the first device (Table 4.4).

A dataset generated as above can be labeled using the naive labeling method. All packets belonging to the known source addresses of the first device are labeled with the respective class label of the first device. All other packets are labeled with the class label of the second device. This process is repeated at least once to obtain one labeled dataset for training and one for testing the machine learning model. This entire naive labeling process can, of course, be automated in Python.

Packet Nr.	**Source Address**	**Label**
1	2E-B0-D0-63-C2-26	AirTag
2	2E-B0-D0-63-C2-26	AirTag

Table 4.1: Packets of an AirTag only

Packet Nr.	**Source Address**	**Label**
1	2E-B0-D0-63-C2-26	AirTag
2	2E-B0-D0-63-C2-26	AirTag
3	00-5F-67-D3-1D-35	iPhone
4	C9-2D-8B-7F-9B-A6	iPhone
5	2E-B0-D0-63-C2-26	AirTag
6	00-5F-67-D3-1D-35	iPhone

Table 4.2: Packets of an AirTag in the ”nearby” State and an iPhone

Packet Nr.	**Source Address**	**Label**
1	2E-B0-D0-63-C2-26	AirTag
2	2E-B0-D0-63-C2-26	AirTag
3	00-5F-67-D3-1D-35	iPhone
4	C9-2D-8B-7F-9B-A6	iPhone
5	2E-B0-D0-63-C2-26	AirTag
6	00-5F-67-D3-1D-35	iPhone
...	...	...
10272	2E-B0-D0-63-C2-26	AirTag
10273	5D-1B-44-11-3A-B7	?

Table 4.3: Packets of an AirTag in the ”nearby” State and an iPhone with Source Address Randomization

Packet Nr.	**Source Address**	**Label**
3	00-5F-67-D3-1D-35	iPhone
4	C9-2D-8B-7F-9B-A6	iPhone
5	2E-B0-D0-63-C2-26	AirTag
6	00-5F-67-D3-1D-35	iPhone
...	...	...
10272	2E-B0-D0-63-C2-26	AirTag

Table 4.4: Clipped and Labeled Dataset of an AirTag in the ”nearby” State and an iPhone

Additionally, features must be extracted for all three datasets involved, both labeled datasets and the unlabeled dataset, for semi-supervised learning, as the training of machine learning models requires features. In the case of the two labeled datasets, the feature extraction can occur before or after labeling. In this thesis, feature extraction always occurred before labeling.

However, all unlabeled production data that will later be labeled using a machine learning model must undergo the exact same feature extraction as the datasets used for training and evaluating the machine learning model. Therefore, it is advisable to apply the feature extraction from the production data, i.e., the data to be labeled by the machine learning model, to the training data of the machine learning models for labeling and not the other way around. As unintuitive as this might be at first, it dramatically simplifies the labeling of the production data in the end, as only one feature extraction will be necessary; otherwise, the production data would have to undergo two separate feature extraction processes, one feature extraction for labeling and then one feature extraction for the training of the machine learning model used for device classification.

For this thesis, there are two different feature extractions: one for analyzing the captured data and one for modeling. Because the feature extractions differ, the machine learning models for labeling had to be trained twice, once for every feature extraction. Therefore, there are two machine learning models for every combination of devices.

However, once the features are extracted and the training dataset and the evaluation dataset are labeled using naive labeling, the machine learning model can begin training. For this thesis, the chosen base model is the default neural network from the scikit-learn library. The model has one hidden layer containing 100 hidden neurons and a RELU activation function. Next, this model is trained using the labeled training dataset. Then, the model is trained using the much larger 3-hour-long unlabeled training dataset. This training occurs as self-training, a simple semi-supervised learning method. Again, the corresponding implementation from scikit-learn was used for this thesis.

The purpose of the semi-supervised self-training is to improve the models' robustness and ability to react to unseen samples. However, the self-training does not necessarily have to improve the model's performance on the test set. When the performance worsens, the self-training can be omitted, and solely the base model can be used for labeling. Principally speaking, training the model on only a few labeled samples is not the best idea. However, given that this binary classification task is relatively simple, few samples may be sufficient to obtain a decent model.

In the final step, the model's performance can be evaluated using the labeled test set. In most cases, the models for this thesis achieved an excellent accuracy of over 99% (Figure 4.4).

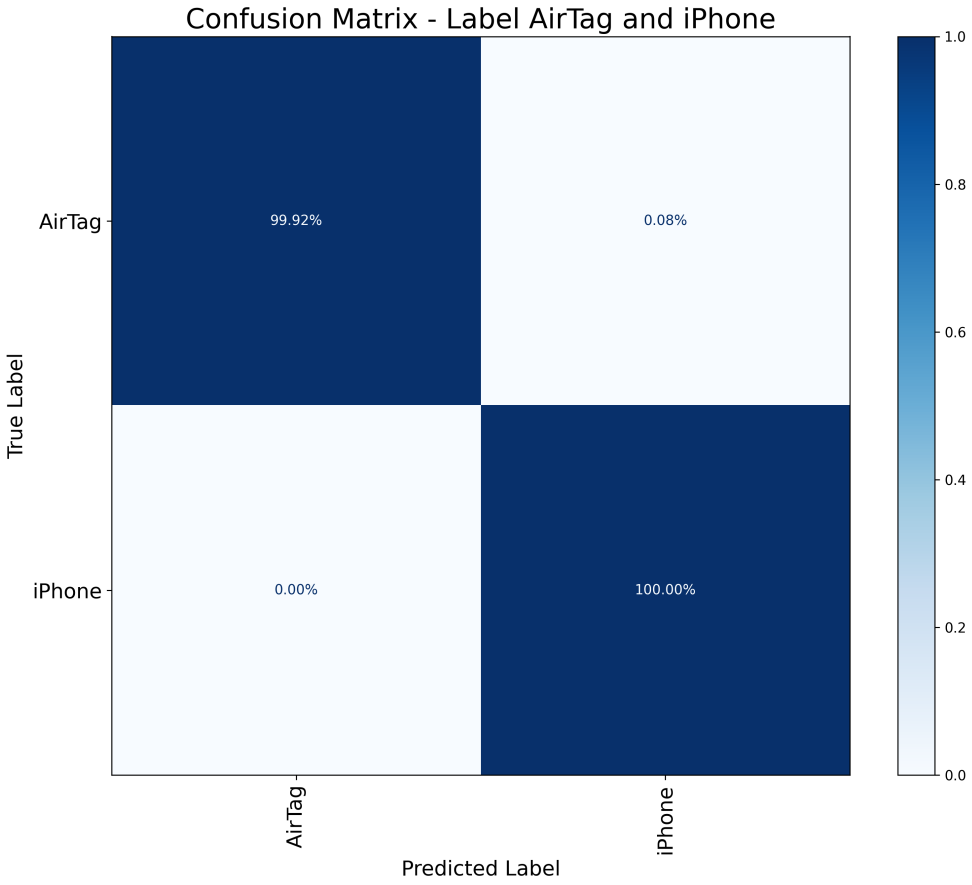

Figure 4.4: Evaluation of Machine Learning based Labeling

4.4 Overview of generated Datasets

The generated datasets vary from device to device based on their individual characteristics. The following subsections will provide a brief overview of the datasets for the respective devices/device categories.

Some general remarks about the process and considerations for the generated data sets:

- All individual datasets were captured non-stop continuously. In other words, every single file stems from one continuous capture. No files of different non-continuous captures were ever merged.
- The production datasets used for analysis and modeling are 12 hours long, if possible. As described extensively in previous chapters, both the aggregated and the recurrent neural network approach require vast amounts of data. If the time interval for aggregation/sequence length is set to 10 seconds, 12 hours (43'200 seconds) of data will result in 4'320 samples. If split into a 75% training set and a 25% test set, the number of samples in the test set is 1'080, resulting in a granularity of plus-minus 0.1% for the evaluation accuracy.

- The ”unpaired” state of trackers was always captured with the tracker starting up and broadcasting for the very first time. This means that original factory-sealed trackers were started for the very first time by removing the factory-installed seals (i.e., they were unboxed) while the Bluetooth sniffer was ruining and capturing.
- Every precaution possible was taken to capture the devices in radio isolation from background traffic. However, this isolation is not perfect. Therefore, there can occasionally be unwanted packets in the datasets.
- As discussed in the chapter, not all device states are relevant to this thesis. These states were not captured.
- All files are labeled with the captured device and state.
- The files labeled with ”labeled_training” and ”labeled_evaluation” can be labeled with naive labeling and are meant for supervised training and evaluation of machine learning models for automatic labeling. The files labeled with ”3h” are meant for the semi-supervised training of these models.
- For every file, there is a PCAP and a CSV version. Both share the same name.
- The Wireshark profile used to export the CSV files is on GitHub.
- All files (PCAP and CSV) can be found on GitHub/Kaggle (see Appendix).

4.4.1 Find My Trackers

For each of the trackers operating on Apple’s Find My network, i.e., the Apple AirTag, the 4Smarts SkyTag, and the Chipolo One, the following datasets are available. Here, ”Tracker” is the placeholder for the corresponding class label/device:

- **Tracker_(lost):** A 12 hours long dataset of the tracker in its ”lost” state.
- **Tracker_(nearby):** A 12 hours long dataset of the tracker in its ”nearby” state and an iPhone in its ”online” state.
- **Tracker_(nearby)_3h:** A 3 hours long dataset of the tracker in its ”nearby” state and an iPhone in its ”online” state.
- **Tracker_(nearby)_labeled_training:** A ca. 10 minutes long dataset of the tracker in its ”nearby” state and an iPhone in its ”online” state. This dataset can be labeled using the naive labeling approach for multiple devices.
- **Tracker_(nearby)_labeled_evaluation:** A ca. 10 minutes long dataset of the tracker in its ”nearby” state and an iPhone in its ”online” state. This dataset can be labeled using the naive labeling approach for multiple devices.
- **Tracker_(unpaired):** A dataset of the tracker in its ”unpaired” state. For the AirTag, the length is 12 hours; for the Chipolo One, the length is 60 seconds; and for the SkyTag, the length is roughly 4 minutes.

4.4.2 Samsung SmartTag

For the Samsung SmartTag, the following datasets are available:

- **SmartTag_(lost):** A 12 hours long dataset of the SmartTag in its ”lost” state.
- **SmartTag_(nearby):** A 12 hours long dataset of the SmartTag in its ”nearby” state and an other Device (Samsung Galaxy S23 Ultra).
- **SmartTag_(nearby)_3h:** A 3 hours long dataset of the SmartTag in its ”nearby” state and an other device (Samsung Galaxy S23 Ultra).
- **SmartTag_(nearby)_labeled_training:** A short dataset of the SmartTag in its ”nearby” state and an other Device (Samsung Galaxy S23 Ultra). This dataset can be labeled using the naive labeling approach for multiple devices.
- **SmartTag_(nearby)_labeled_evaluation:** A short dataset of the SmartTag in its ”nearby” state and an other device (Samsung Galaxy S23 Ultra). This dataset can be labeled using the naive labeling approach for multiple devices.
- **SmartTag_(unpaired):** A 5 minutes long dataset of the SmartTag in its ”unpaired” state.
- **SmartTag_(searching):** A 12 seconds long dataset of the SmartTag in its ”searching” state.

Note: Due to the relatively quick source address randomization interval of the Samsung SmartTag, there are multiple labeled datasets of each type for the tracker in its ”nearby” state in an additional subfolder.

4.4.3 Tile Mate

For the Tile Mate, the following datasets are available:

- **Tile_(lost):** A 12 hours long dataset of the Tile in its ”lost” state.
- **Tile_(nearby):** A 12 hours long dataset of the Tile in its ”nearby” state and an iPhone in its ”online” state.
- **Tile_(nearby)_3h:** A 3 hours long dataset of the Tile in its ”nearby” state and an iPhone in its ”online” state.
- **Tile_(nearby)_labeled_training:** A ca. 13 minutes long dataset of the Tile in its ”nearby” state and an iPhone in its ”online” state. This dataset can be labeled using the naive labeling approach for multiple devices.

- **Tile_(nearby)_labeled_evaluation:** A ca. 23 minutes long dataset of the Tile in its "nearby" state and an iPhone in its "online" state. This dataset can be labeled using the naive labeling approach for multiple devices.
- **Tile_(unpaired):** A 60 seconds long dataset of the Tile in its "unpaired" state.
- **Tile_(searching):** A 10 seconds long dataset of the Tile in its "searching" state.

4.4.4 AirPod

For the Apple AirPod, the following datasets are available:

- **AirPod_(lost):** A 12 hours long dataset of the AirPod in its "lost" state.
- **AirPod_(nearby):** A 12 hours long dataset of the AirPod in its "nearby" state and an iPhone in its "online" state.
- **AirPod_(nearby)_3h:** A 3 hours long dataset of the AirPod in its "nearby" state and an iPhone in its "online" state.
- **AirPod_(nearby)_labeled_training:** A ca. 5 minutes long dataset of the AirPod in its "nearby" state and an iPhone in its "online" state. This dataset can be labeled using the naive labeling approach for multiple devices
- **AirPod_(nearby)_labeled_evaluation:** A ca. 5 minutes long dataset of the AirPod in its "nearby" state and an iPhone in its "online" state. This dataset can be labeled using the naive labeling approach for multiple devices

Note:

- The BLE packets were always captured for one single AirPod. The other AirPod and the battery charging case were out of range for the Bluetooth sniffer.
- The AirPod is not a conventional BLE tracker and should, therefore, in principle, not have a "nearby" or "lost" state but rather an "online" and "offline" state similar to the iPhone or all other Apple devices (iDevices). However, as this thesis will reveal, the AirPod is, in fact, a conventional BLE tracker similar to the AirTag in terms of its implementation. Therefore, it also shares the states of these conventional BLE trackers.

4.4.5 Other Apple Devices (iDevices)

For each of the other Apple devices, i.e., the iPhone, the MacBook, and the iPad, the following datasets are available. Here, "iDevice" is a placeholder for the class label of the corresponding device:

- **iDevice_(online):** A 12 hours long dataset of the iDevice in its "online" state.
- **iDevice_(offline):** A 12 hours long dataset of the iDevice in its "offline" state.

4.4.6 Other Devices

All the other devices forming the garbage class were captured simultaneously. Therefore, there is one 2.5 hours long dataset for all of these devices named ”other Device”.

4.4.7 Inference

For inference, there are two datasets. Both were captured at Zürich central station on the 25th of April 2024 during the afternoon rush hour in a high-density environment. One dataset is 10 minutes long (”Bahnhof_V1”), and the other is 35 minutes long (”Bahnhof_V2”). Both datasets were captured at the exact same location under the same conditions.

4.5 Summary of Dataset Generation

The most relevant insights from this chapter are:

- As of writing this thesis, no suitable dataset is available for categorical classification of BLE devices with machine learning.
- Over 30 Million packets over 600 hours were captured for this thesis to generate a large dataset.
- The generated dataset covers many devices, including conventional BLE trackers, various Apple devices, and other non-tracking devices.
- For every device, all relevant states were captured.
- The BLE packets were captured with an nRF 52840 DK Bluetooth sniffer and dissected (i.e., interpreted) by Wireshark. The devices and the sniffer were put into a Faraday cage for capturing to isolate them from unwanted background traffic.
- Labeling of packets is simple in the case of one device and much more difficult for multiple devices not sharing the same class label.
- Machine learning can be used to label datasets from multiple devices with different class labels. This is especially relevant for the trackers in the ”nearby” state where two devices (the tracker and the owner device) must be captured simultaneously.
- The entire generated dataset is available on GitHub. There are files, both PCAP and CSV, available for every device and state captured.

Chapter 5

Task-Group-Framework

The following chapter covers the Task-Group-Framework, a custom-designed framework for implementing data pipelines. All data processing in this thesis is performed exclusively with the Task-Group-Framework. Additionally, the plotting in this thesis is also powered by the Task-Group-Framework. The entire implementation of the Task-Group-Framework, including many unit and integration tests, can be found on GitHub.

The following sections describe the motivation and goal of the framework, introduce the most important components, and explain the application in this thesis.

5.1 Motivation and Goal

So far in this thesis, the data processing was only discussed on a higher level. On the lower level, the preprocessing can be split into individual steps (or groups of steps). The image shows a high-level visualization of this pipeline (Figure 5.1):

1. **Feature Extraction:** In the first step, features must be extracted from the input CSV files. For instance, the categorical features must be one-hot encoded, or NULL values might have to be filled.

2. **Labeling:** Next, each row (i.e., packet) must be labeled with a device label. This can happen either by giving every row the same label or via automatic machine-learning-based labeling in the case of "nearby" trackers.

3. **States:** Finally, the class labels must be extended with a state, if necessary. For example, the class label "AirTag" might become "AirTag (lost)" in the case of an AirTag in the "lost" state.

Based on this lower-level description of the pipeline, it becomes clear that some pipeline steps, such as feature extraction, can be shared across datasets, and other parts, such as labeling and state assignment, cannot. From a practical point of view, it would be highly

convenient to share the common steps across pipelines. This would significantly reduce code duplication.

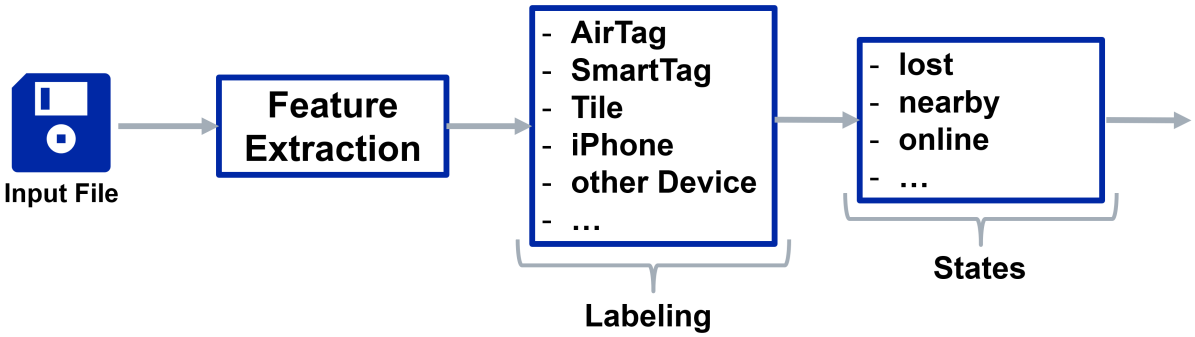

Figure 5.1: High-level Depiction of Data Pipeline

There are two ways to approach this. The first would be to build separate pipelines for each dataset where the common steps are shared and only implemented once. However, this would still result in one pipeline for every combination of device and state, presumably unmanageable and certainly not scalable. This approach would have resulted in roughly 100 data pipelines for this thesis.

The second approach would be to design one pipeline and use parameters to change its behavior at runtime. In other words, a pipeline would be designed where individual steps can be activated or deactivated for every execution of the pipeline.

Current popular implementations of data pipelines only support the first approach, which I considered infeasible for this thesis. The number of pipelines required would exceed practical limits. Therefore, I designed the Task-Group-Framework, which follows the second approach of designing one pipeline only and changing its behavior at runtime via parameters.

5.2 Structural Elements

The following subsections will detail the most relevant components of the Task-Group-Framework, beginning with the lowest-level component and following a bottom-up approach to the highest-level component. This is, by no intents and purposes, a documentation of the framework and shall only give a superficial grasp of the implementation of data processing in this thesis.

This introduction of the most high-level components follows the example of applying linear-affine functions to pandas DataFrames. Specifically, the goal is to apply the function $f(x) = 3x + 1$ elementwise to 2D arrays of numbers in the form of pandas DataFrames. Additionally, the implementation should allow for flexibility at runtime; it should be possible to apply only parts of the function if necessary.

The function $f(x) = 3x + 1$ can also be understood as the concatenation of the functions $g(x) = x + 1$ and $h(x) = 3x$ because $g \circ h = g(h(x)) = 3x + 1$. The concatenation of functions is conceptually equivalent to the concatenation of processing steps in a data pipeline. The ultimate goal of this example is to be able to apply either only $h(x)$ or only $g(x)$ or $f(x)$, the concatenation of both, elementwise to a pandas DataFrame.

The first step is to define a DataFrame. The 3x3 identity matrix will serve as mock data for this example (Listing 5.1).

```
matrix = pd.DataFrame(np.identity(3, dtype=int), columns=[0, 1, 2])
```

Listing 5.1: Definition of a 3x3 Identity Matrix

When the function $f(x) = 3x + 1$ is applied elementwise to this identity matrix, the expected result is as follows:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \xrightarrow{f(x)} \begin{pmatrix} 4 & 1 & 1 \\ 1 & 4 & 1 \\ 1 & 1 & 4 \end{pmatrix}$$

When only $g(x) = x + 1$ is applied, the expected result is as follows:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \xrightarrow{g(x)} \begin{pmatrix} 2 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 2 \end{pmatrix}$$

And when only $h(x) = 3x$ is applied, the expected result is as follows:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \xrightarrow{h(x)} \begin{pmatrix} 3 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 3 \end{pmatrix}$$

Finally, it is very important to note that the order of execution of the functions $g(x)$ and $h(x)$ matters. The concatenation of these functions is not commutative. The same applies to general data processing steps, there, the order of execution also matters.

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \xrightarrow{h(g(x))} \begin{pmatrix} 6 & 3 & 3 \\ 3 & 6 & 3 \\ 3 & 3 & 6 \end{pmatrix}$$

5.2.1 Executors

The most low-level component of the Task-Group-Framework is the executor. An executor is essentially one step in the pipeline that performs data processing of some kind. Such a processing step can be of arbitrary complexity, ranging from the simplest executor possible, the one that returns the data as is and does nothing to it, to the most sophisticated

data manipulation steps that can be hundreds of lines of code long (an example of this would be the executors used for plotting in this thesis).

On a programmatic level, an executor is simply a class implementing the ExecutorInterface. The ExecutorInterface has exactly one method: the execute() method, which has one parameter, a pandas DataFrame, and one return value, also a pandas DataFrame. Additionally, it is possible to customize executors further, for instance, by overriding the constructor.

For the running example of implementing the linear-affine function $f(x) = 3x + 1$, two executors are required, one executor that implements the slope of the function, i.e. the function $h(x) = 3x$, and one that defines the constant, i.e. $g(x) = x + 1$ (Listing 5.2). The executor defining the constant has its constructor overridden to allow for customization of the constant. If desired, this would allow for changing the constant without defining a new executor class.

```
class Slope(ExecutorInterface):
    def execute(self, dataToProcess: pd.DataFrame) -> pd.DataFrame:
        return  3 * dataToProcess

class Constant(ExecutorInterface):
    def __init__(self, constant: int):
        self.constant = constant

    def execute(self, dataToProcess: pd.DataFrame) -> pd.DataFrame:
        return self.constant + dataToProcess
```

Listing 5.2: Definition of Executors

It is possible to use these executors as is by passing the initial matrix to one executor and then passing the intermediate result to the next executor (Listing 5.3). This will produce the correct output matrix.

```
result = Slope().execute(matrix)
result = Constant(1).execute(result)
```

Listing 5.3: Execution of Executors

However, this is not very scalable and needs to be expanded to be useful. Additionally, so far, any flexibility at runtime is absent. Therefore, additional framework components are needed.

5.2.2 Tasks

The Task component serves as a wrapper for the executors and provides additional functionality on top of an executor. The conceptual idea is to use the executor for dependency injection to change the behavior of a Task, i.e., implement the command design pattern where the Task is the command wrapper and the executor the command. This implementation makes Tasks open for modification and expansion without any changes to existing executors, i.e., modification.

A Task implements the ExecutorInterface and the abstract base class AbstractTask. The AbstractTask class provides base functionality to both Tasks and TaskGroups. The ExecutorInterface ensures that Tasks can be used just like any executor if necessary.

A Task can be instantiated with the following arguments:

- **name:** Defines the name of the Task and does not have a default value.
- **priority:** The execution order priority of the Task as an integer value. This is only relevant in the context of TaskGroups. The default value is None.
- **flags:** A list of Flags or a single Flag. The default value is the BaseFlag. For more information on Flags, see the section on the Flag component.
- **inplace:** A bool indicating whether the underlying executor is executed on the data passed to its execute() method or on a deep copy. If inplace is set to false, the original data passed to the executor is not mutated and returned as is. If inplace is set to false, the executor is executed on the original data, and the data is mutated. The default value is true. Setting to false can be useful for plot pipelines where executors are used for plotting.
- **executor:** An executor implementing ExecutorInterface that determines the behavior of the Task when execute() or process() are called. The default value is the SimpleExecutor, which returns the data without mutating it. Passing a custom executor as an argument to the constructor is sensible in most cases. Otherwise, the Task does not do anything useful.

Instantiating two Tasks for the linear-affine function is done as follows. All optional constructor arguments are set to their default values, except for the executors, to add useful behavior to the Tasks. The constant executor has a 1 passed as an argument to its constructor to set the constant to the desired value of 1 (Listing 5.4).

```
task_slope = Task(name = "Slope", executor=Slope())
task_constant = Task(name = "Constant", executor=Constant(1))
```

Listing 5.4: Definition of Tasks with default Arguments

Tasks have several different class methods, some of which are outlined in the following:

- **execute():** The execute() method inherited from the ExecutorInterface. Calling execute() on a Task with a DataFrame as an argument will result in a call of the execute() method on the dependency-injected executor of the Task. The call of this method is equivalent to calling execute() directly on the underlying executor.
- **process():** The process() method inherited from the TaskInterface. Calling process() on a Task with a DataFrame and a Flag as an argument will result in a call of the execute() method on the dependency-injected executor of the Task if and only if the Flag passed an argument is a child of at least one of the Flags passed to the constructor method of the Task. If the Flag passed as an argument is not a child of

any of the Flags of the Task, execute() is not called, and the DataFrame is returned as is. For more information on Flags, see the section on the Flag component. Important: Every Flag is a child of itself.

- **print():** The print() method prints the name of the Task to the console. If requested by the arguments passed to the print() method, it is also possible to print the priority and Flags of the Task.

As done previously with executors, execute() can be called on the first Task, and the intermediate result is passed to the second Task to obtain the desired final result. This is the most simple application of Tasks (Listing 5.5).

```
result = task_slope.execute(matrix)
result = task_constant.execute(result)
```

Listing 5.5: Execution of Tasks

However, this is still not a helpful data pipeline, as it is not scalable. In other words, Tasks are not necessarily useful on their own. So far, adding a Task wrapper around an executor has little to no benefit. This will change with the introduction of TaskGroups and Flags.

5.2.3 TaskGroups

The TaskGroup component is very similar to the Task component as it also implements the abstract base class AbstractTask. TaskGroups are essentially a set of Tasks that are aggregated together into a group, and the TaskGroup is the owner of the Tasks. The Tasks are stored in a custom implementation of a priority queue. The order in the priority queue is determined by the priority set during construction of the Tasks. The lower the priority of a Task, the further upfront it is in the queue. Tasks with default priority None are always at the very back of the queue. In the case of the same priority, the tie-breaker is the order of addition to the TaskGroup. The earlier a Task was added to the TaskGroup, the further up front it is in the queue.

Since Tasks and TaskGroups both implement AbstractTask, the constructor method also uses the same argument with the same default values. TaskGroups have one additional argument: idempotency. The idempotency argument is a bool with the default value set to false. Setting idempotency to true will result in an idempotent behavior when adding Tasks to the TaskGroup. In other words, if the same Tasks is added again and again to a TaskGroup, it is only added once and exchanged every time it is added again. This is mainly useful in Jupyter Notebooks, where the same cell might be accidentally executed multiple times.

The TaskGroups implement a variety of methods that are different in functionality and behavior from the implementation for Tasks.

- **add():** Adds one object of type AbstractTask to the TaskGroup. This object can be either a Task or a TaskGroup. The insertion happens priority-based.

- **addAll():** Adds a list of objects of type AbstractTask to the TaskGroup. These objects can be either a Task or a TaskGroup. The insertion happens priority-based.

- **__getitem__():** Returns the object in the priority queue corresponding to the provided subscript. The subscript can be either a slice or the string name of the AbstractTask object. The slicing is performed based on the position of objects in the TaskGroup (and not based on the priority), which is equivalent to slicing a list.

- **__delitem__():** Deletes the object in the priority queue corresponding to the provided subscript. The subscript can be either a slice or the string name of the AbstractTask object. The slicing is performed based on the position of objects in the TaskGroup (and not based on the priority), which is equivalent to slicing a list.

- **copy():** Returns a deep copy of the TaskGroup.

- **execute():** Calls the execute() methods on all the objects in the priority queue. This method is inherited from the ExecutorInterface. The output of the execute() method of one object in the queue is the input argument for the call of the execute() method of the next object in the queue. After the call of execute() on the final object in the queue, the processed DataFrame is returned.

- **process():** Calls the process() methods on all the objects in the priority queue with the Flag that was passed as an argument if and only if the Flag is a child of one of the Flags set for the TaskGroup. If so, the output of the process() method of one object in the queue is the input argument for the call of the process() method of the next object in the queue. After the call of process() on the final object in the queue, the processed DataFrame is returned. If the Flag passed as an argument is not a child of any of the Flags of the TaskGroup, the DataFrame is immediately returned as is. Important: Every Flag is a child of itself.

- **print():** The print() method prints the name of the TaskGroup to the console. Afterward, the print() method is called on all objects in the priority queue. If requested by the arguments passed to the print() method, it is also possible to print the priority and Flags of the TaskGroup and all its objects in the queue to the console.

The construction of a TaskGroup for the running example of the function $f(x) = 3x + 1$ can be done as follows (Listing 5.6). Note that the Tasks also need to be redefined because of the Task priority to ensure the correct execution order. As explained above, the order of the functions $g(x)$ and $h(x)$, here represented by the two Tasks, matters.

```
task_slope = Task(name = "Slope", executor=Slope(), priority=1)
task_constant = Task(name = "Constant", executor=Constant(1), priority
   =2)

task_group = TaskGroup("Linear Function")
task_group.addAll([task_constant, task_slope])
```

Listing 5.6: Definition of TaskGroup

At this point, it is important to emphasize that TaskGroups support nesting. One can add not only Tasks to a TaskGroup but also any object implementing the abstract class AbstractTask, most notably TaskGroups. This means that TaskGroups can be added to other TaskGroups, i.e., nested within other TaskGroups. This is especially useful for defining complex data pipelines such as the ones used in this thesis.

Next, it is possible to call the execute() method on the entire TaskGroup (Listing 5.7). This will result in a sequential execution of the slope Task and then the constant Task. The output is the elementwise application of the function $f(x)$ to the defined matrix.

```
result = task_group.execute(matrix)
```

Listing 5.7: Execution of TaskGroup

At this point, the TaskGroup behaves like an actual data pipeline that executes Tasks sequentially in a predefined order. The ultimate step is to add flexibility at runtime via the implementation of Flags. Flags ultimately allow the user to activate and deactivate Tasks and TaskGroups at runtime.

5.2.4 Flags

The Flag component allows for flexibility at runtime. So far, it has only been possible to execute all tasks within a TaskGroup at once. But what if only parts of the TaskGroup shall be executed? In the running example of the linear-affine function, this would be equivalent to calling only $g(x)$ or $h(x)$. Flags enable the partial execution of processing steps in a data pipeline.

First, the constructor of the Flag takes the following two arguments:

- **name:** The name of the Flag as a string value.
- **parents:** A list of Flags or a single Flag. The constructed Flag is a child of all its parent Flags (and their parents, respectively) and "inherits" its properties for the execution of AbstractTasks from all parents (including parents of parents). Important: A Flag is also always automatically a child of itself.

Additionally, Flags have multiple methods, some of which are:

- **getParents():** A getter method that returns all the parent Flags in a list. If the verbose argument is set to true, the list contains the string names of the Flags rather than the objects.
- **getAllParents():** A getter method that returns all the parent Flags, including all the parent Flags of the parents in a list. If the verbose argument is set to true, the list contains the string names of the Flags rather than the objects.

- **contains():** The contains() method takes another Flag as an argument and returns whether this Flag is a parent (or parent of a parent) of the Flag the method was called on. If so, it returns true; otherwise false. Note: Every Flag is also always a parent/child of itself.

When Flags are instantiated, the Task-Group-Framework builds a Flag tree in the background. The root of this tree is the so-called BaseFlag. The BaseFlag component is a singleton object that is automatically instantiated in the background. For a user, it is never necessary to instantiate a BaseFlag. Whenever another Flag is instantiated, it is always a child node of this BaseFlag in the Flag tree. Additionally, the newly instantiated Flag is also a child node of all Flags that were set as parents during construction. It is, therefore, possible, unlike in a binary tree, for a child node, i.e., a Flag, to have multiple parent nodes, i.e., parent Flags. Even if BaseFlag is not explicitly passed as a parent Flag, the newly created flag will always be an (in)direct child of BaseFlag, and BaseFlag will be an (in)direct parent. BaseFlag is the direct/indirect parent of all Flags. Finally, a Flag is also always a child/parent of itself.

First, three Flags are created to visualize the Flag tree, ”Flag_Linear_Function” has the two other Flags as direct parents (Listing 5.8).

```
flag_slope = Flag(name = "Flag_Slope")
flag_constant = Flag(name = "Flag_Constant")
flag_linear_function = Flag(name="Flag_Linear_Function", parents=[
    flag_slope, flag_constant])
```

Listing 5.8: Definition of Flags

The resulting Flag tree looks like the following (Figure 5.2). As seen in the visualization, the bottom Flag ”Flag_Linear_Function” is a child of the two other Flags and indirectly inherits from the BaseFlag because it is the indirect parent.

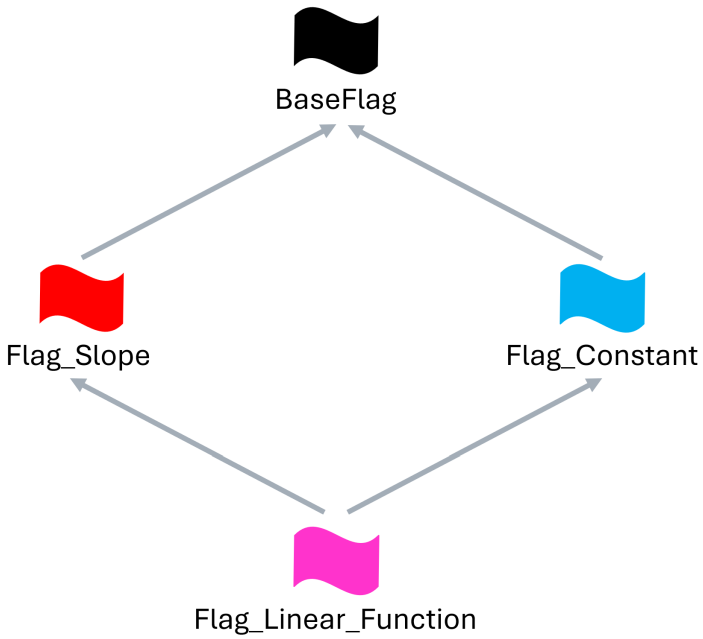

Figure 5.2: Flag Tree

In the next step, Flags can be placed on Tasks and TaskGroups. For this purpose, the above Tasks and TaskGroups are again redefined. As seen in the code snippet (Listing 5.9), the TaskGroup itself does not receive a Flag as an argument. In this case, the Flag for the TaskGroup is set automatically to its default value, BaseFlag.

```
task_slope = Task(name = "Slope", executor=Slope(), priority=1, flags=
    flag_slope)
task_constant = Task(name = "Constant", executor=Constant(1), priority
    =2, flags=flag_constant)

task_group = TaskGroup(name = "Linear Function")
task_group.addAll([task_constant, task_slope])
```

Listing 5.9: Definition of Tasks and TaskGroup with Flags

Finally, a Flag can be passed as an argument to the process() method of an AbastractTask object (i.e., a Task or a TaskGroup). The behavior of this process() method varies between Tasks and TaskGorups.

In the case of a Task, the execute() method of the executor is called if the Flag passed as an argument to the process() method is a child (directly or indirectly of itself) of the Flag from the Task. Remember: A Flag is also always a child of itself. If the Flag is not a child of any kind, the DataFrame is returned as is.

In the case of a TaskGroup, the behavior of the process() method is more complex. First, if the Flag passed as an argument to the process() method is not a child of any of the Flags from the TaskGroup (directly or indirectly or of itself), the entire TaskGroup is skipped, and the DataFrame is returned as is. None of the Tasks within the TaskGroup are executed in this case, even if there were Tasks within the TaskGroup that would be executable based on the Flag passed as an argument to the process() method.

However, if the Flag passed as an argument to the process() method is a child of at least one of the Flags from the TaskGroup, the process() method of each AbastractTask object within the TaskGroup is called sequentially in order of priority. In each of these calls of the process() method, the Flags are rechecked, and if, and only if, the passed Flag is a child of one of the Flags from the Task the method is called on, an execution takes place.

This mechanism of Flag inheritance allows for the desired flexibility at runtime. With Flags, any Tasks can be activated or deactivated at will at any time. With the inheritance mechanism (i.e., the parent-child relation) between Flags, it is possible to create complex variants for the execution of data pipelines. This can be illustrated with the example of the linear-affine function.

First, the process() method of the TaskGroup is called with the identity matrix and the ”Flag_Slope” as an argument (Listing 5.10).

```
result = task_group.process(matrix, flag_slope)
```

Listing 5.10: Call of process() method with Slope Flag

The resulting output matrix is the identity matrix multiplied by 3, similar to applying only $h(x) = 3x$ to the matrix:

$$\begin{pmatrix} 3 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 3 \end{pmatrix}$$

Note that the Task that adds the constant 1 to the matrix is skipped, and only the Task for the slope is executed. What happens in the background is relatively straightforward. First, the TaskGroup is entered because its Flag is BaseFlag, and the "Flag_Slope" is a child of BaseFlag. Next, the Task for the slope is executed because the "Flag_Slope" is a child of itself. Finally, the Task for the constant is not executed because the Flag associated with this Task is not a parent of the "Flag_Slope", as seen in the Flag tree.

A more complex example would be passing the "Flag_Linear_Function" as an argument to process() alongside the identity matrix (Listing 5.11).

```
result = task_group.process(matrix, flag_linear_function)
```

Listing 5.11: Call of process() method with Linear Function Flag

The resulting output matrix is equivalent to applying $f(x) = 3x + 1$ to the matrix:

$$\begin{pmatrix} 4 & 1 & 1 \\ 1 & 4 & 1 \\ 1 & 1 & 4 \end{pmatrix}$$

Because the "Flag_Linear_Function" is a child of both "Flag_Slope" and "Flag_Constant", both Tasks in the TaskGroup are executed. In this case calling process() with this Flag is equivalent to calling execute() on the TaskGroup.

5.3 Application in this Thesis

The Task-Group-Framework is used throughout this thesis for various purposes. All data processing is powered by the Task-Group-Framework. Additionally, plotting is also facilitated by this framework. All plots are individual executors that create the various plots. All plots are concatenated into a TaskGroup to form a plot pipeline.

All data processing for modeling and analysis is performed with this framework. The following image shows the console output of calling print() on the pipeline for dataset analysis (Figure 5.3). Every row in the console print represents one Task or TaskGroup. On the highest level, the "Analysis Pipeline" consists of five nested TaskGroups: "Pre Processing", "Dummy Processing", "Labeling", "States" and "Modeling". Each of these TaskGroups contains further Tasks or TaskGroups. Every tab indentation symbolizes a layer of nesting, and the numbers represent the execution priorities within each (nested) TaskGroup. The general execution order of the pipeline is top-down. Sometimes, there is a ":" after a Task or TaskGroup. The string after the ":" is the name of the Flag associated with the corresponding Task or TaskGroup (in most cases, the name of the

Flag and the Task/TaskGroup are identical). No ":" after a Task or TaskGroup indicates that the associated Flag is BaseFlag.

And finally, the framework was designed around the requirements for this thesis. Therefore, many features could still be added, such as parallelization, where applicable, to speed up computing. It is by no means complete at this point.

5.4 Summary of Task-Group-Framework

The most relevant insights from this chapter are:

- A custom implementation for data processing was necessary, as existing implementations of data pipelines did not cut it.
- The Task-Group-Framework allows for implementing data pipelines with flexibility at runtime. Any pipeline step can be activated or deactivated for every execution of the pipeline at runtime.
- The Task-Group-Framework has four important structural elements: Executors, Tasks, TaskGroups, and Flags. The Flag component allows for flexibility at runtime.
- The data processing pipelines in this thesis are implemented exclusively with the Task-Group-Framework.

```
Analysis Pipeline
        10 Pre Processing: Pre Processing
                10 Select and order columns
                15 Company ID and UUID
                        10 Fill Company ID with None
                        20 Fill UUID with None
                        30 Replace Company IDs
                        40 Replace Company UUIDs
                20 MS Data Processing
                        10 Fill MS Data with empty String
                        20 Length of MS Data
                        30 Continuity Type
                        40 Drop MS Data Column
                25 Service Data Processing
                        10 Fill Service Data with empty String
                        20 Length of Service Data
                        30 Samsung Type
                        40 Drop Service Data Column
                30 Fill Numeric NA with 0
                40 Fill String NA with None
                50 Broadcast
                60 Datetime conversion
                70 Malformed Packet
                80 Order DataFrame
                90 Convert object type to string
        20 Dummy Processing: Dummy Processing
                20 Dummies Protocol
                30 Dummies Channel
                40 Dummies AD Type
                50 Dummies Company
                55 Dummies UUID
                70 Dummies PDU Type
                80 Dummies Continuity Type
                90 Dummies SmartTag Type
        30 Labeling: Labeling
                10 Labeling auto: Labeling auto
                        Label AirTag and iPhone: Label AirTag and iPhone
                        Label Chipolo and iPhone: Label Chipolo and iPhone
                        Label SkyTag and iPhone: Label SkyTag and iPhone
                        Label Tile and iPhone: Label Tile and iPhone
                        Label AirPod and iPhone: Label AirPod and iPhone
                        Label other Device and SmartTag: Label SmartTag and other Device
                20 Labeling manual: Labeling manual
                        Label AirTag: Label AirTag
                        Label Chipolo: Label Chipolo
                        Label SkyTag: Label SkyTag
                        Label iPhone: Label iPhone
                        Label iPad: Label iPad
                        Label MacBook: Label MacBook
                        Label AirPod: Label AirPod
                        Label SmartTag: Label SmartTag
                        Label Tile: Label Tile
                        Label other Device: Label other Device
        40 States: States
                10 State Column
                20 States iDevices: States iDevices
                        Continuity
                        online: State online
                        offline: State offline
                30 States Trackers: States Tracker
                        lost: State lost
                        nearby: State nearby
                        unpaired: State unpaired
                        searching: State searching
                Collapse State Column
        80 Modeling: Modeling
                0 Drop Column Continuity
                10 Drop Columns
```

Figure 5.3: Analysis Pipeline

Chapter 6

Analysis

Before modeling, it is crucial to gain a deep understanding of the training data. This is important to extract the right features and form appropriate classes for categorical classification. The following sections will detail the entire process, from the datasets' preprocessing through their analysis to the insights gained for modeling.

6.1 Data Preprocessing

The data preprocessing describes the entire workflow from the export to CSV in Wireshark to the final processed dataset ready for analysis. The data preprocessing is implemented with a data pipeline using the Task-Group-Framework (Figure 5.3). This data pipeline has four relevant sections:

1. **General Data Preprocessing:** This includes many simpler preprocessing steps, such as filling NULL values, and advanced feature extraction, such as the continuity type.
2. **Conversion to Dummy Variables:**. For some datasets, automatic machine-learning-based labeling is used. Therefore, all non-numerical features, e.g., categorical features, must be converted to some form of one-hot encoding. This step is irrelevant to the analysis, i.e., the creation of the plots.
3. **Labeling:** All packets need to be labeled. This can happen either manually, where every packet is assigned the same label, or automatically using the machine learning approach.
4. **States and Continuity Type:**: The class labels must be extended to class state labels. For instance, in the case of an AirTag in its "lost" state, the previously assigned class label "AirTag" should be extended to "AirTag (lost)".

The following subsections detail the individual processing steps in this pipeline. The feature selection, i.e., the reasoning behind the detected features, is also briefly touched on.

6.1.1 Feature Selection, Extraction, and General Data Preprocessing

In the first step, features were selected in Wireshark as columns for the CSV files before export. The selected features are the same across all datasets. Next, the raw, unedited CSV files passed through the first section of the preprocessing pipeline. For every feature selected in Wireshark, the individual preprocessing steps are explained.

This explanation is split into two parts. The first section covers simple feature extraction for the features selected in Wireshark (i.e., the columns of the CSV file) and their preprocessing steps. The second section covers more advanced feature extraction, i.e., features that can be extracted from the simpler features with advanced knowledge.

6.1.1.1 Simple Feature Extraction

6.1.1.1.1 Time The time column contains the UTC time in milliseconds from when it was captured in Wireshark. This column is unusable as a feature. However, it is crucial for modeling packet rates. For the more advanced machine learning approaches outlined in Chapter 2, the time column is used to aggregate the packets within time intervals.

6.1.1.1.2 Source The source column contains the source address of the transmitting Bluetooth device. This column is unusable as a feature. However, it is also crucial for modeling the packet rate, as the packet rate is calculated per source address. Finally, the ultimate goal during inference will be to assign labels to the source addresses. Therefore, this is a highly relevant column, albeit not as a feature.

6.1.1.1.3 Destination The destination column contains the device's address to which the packet was sent. This can be used indirectly as a feature, based on whether the packet is broadcasted, i.e., sent to the broadcasting address FF:FF:FF:FF:FF:FF. Therefore, this column was converted to a one-hot encoded column where 1 denotes broadcast and 0 any other destination address. This is presumably a highly relevant feature, as BLE trackers use broadcasted advertisements.

6.1.1.1.4 Protocol Type The protocol type column indicates the protocol used for transmission. This column requires no further preprocessing.

6.1.1.1.5 Channel The channel column indicates the advertising channel used for transmission. This column does not require any further preprocessing.

6.1.1.1.6 Packet Length The packet length column indicates the packet length in bytes on the layer used by the nRF 58420 DK for transmission to the host device, i.e., the computer running Wireshark. **The packet length on this layer has nothing to do with the packet length on the Bluetooth Low Energy link layer.** The packet length is always exactly 26 bytes longer than the header length . This is the terminology used by Nordic Semiconductor, the maker of the nRF 58420 DK, and not mine. Finally, this column does not require any further preprocessing.

6.1.1.1.7 Header Length The header length column indicates the packet length in bytes on the Bluetooth Low Energy link layer. As described above, this value is always 26 bytes shorter than the packet length on the nRF 58420 DK layer. This column does not require any further preprocessing.

6.1.1.1.8 Advertisement Data Type The AD type column contains the advertising data types used in the packet, separated by commas. This column requires no further preprocessing.

6.1.1.1.9 Company ID The company ID column contains the company ID found in the manufacturer specific data. This field is Null if no manufacturer specific data is present in the packet. All Null values were filled with the string ”None”. Additionally, the most frequent and relevant company IDs were replaced with simpler versions. For example, the company ID ”Apple, Inc.” was replaced with ”Apple”.

6.1.1.1.10 Manufacturer Specific Data The MS Data column contains the raw data payload from the manufacturer specific data as a string, if present. Otherwise, this field is Null. The Null values were filled with an empty string. From this column, the length of the raw data in bits was extracted as a feature.

6.1.1.1.11 UUID The UUID column contains all UUIDs present in the packet, separated by commas. Multiple UUIDs can be in one packet. Null values were filled with ”None”. Additionally, the most frequent and relevant UUIDs were replaced with simpler versions. For example, the UUID ”Tile, Inc.” was replaced with ”Tile”. This will become relevant when creating the one-hot encoding for this column.

6.1.1.1.12 Service Data The Service Data column contains the raw data payload from the Service Data as a string, if present. Otherwise, this field is Null. The Null values were filled with an empty string. From this column, the length of the raw Service Data in bits was extracted as a feature.

6.1.1.1.13 PDU The PDU column indicates the package’s PDU type. This column does not require any further preprocessing.

6.1.1.2 Advanced Feature Extraction

6.1.1.2.1 Continuity Type As already discussed in the related works chapter, the structure of packets used by Apple's continuity services, such as the Find My network, is well-researched. Essentially, Apple uses the raw data within the manufacturer specific data to encode information following a well-defined structure (Figure 6.1). There are many different encodings depending on the type of packet, i.e., the type of continuity service the packet is used for. The encoding used for the Find My network packets is noticeably different from the package used for any other service. However, all continuity packets, no matter what service they are used for, indicate their type with the first leading byte of the raw manufacturer specific data.

Typically, this leading byte is provided as a hexadecimal number. I.e., the leading byte 00010010 becomes 0x12. This leading byte is also referred to as the continuity type. The continuity type 0x12 indicates a Find My network packet [12].

This leading byte can be extracted as a feature. This feature should be extracted for packages where the company ID stems from Apple. In all other cases, this feature cannot be extracted, and the corresponding feature column must be filled with a filling value, in this case, "None".

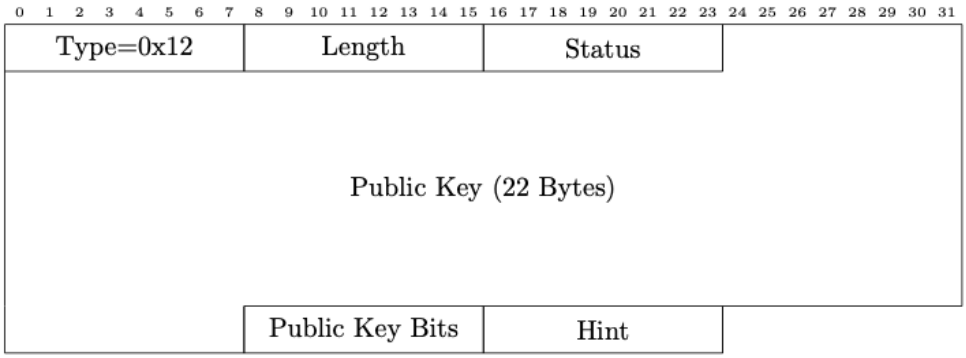

Figure 6.1: The Encoding of the manufacturer specific data of a Find My Packet [12]

6.1.1.2.2 SmartTag Type Like Apple, Samsung also uses a well-researched encoding of its Service Data [13]. Some bits indicate the state of the tracker (Figure 6.2). In this thesis, this shall be called the SmartTag type. Concretely, bits 5 to 7 of the Service Data indicate the state of the SmartTag. This is highly relevant for the categorical classification of states and, therefore, must be extracted as a feature.

This feature should be extracted for packages where the UUID of the corresponding Service Data comes from Samsung. In all other cases, it cannot be extracted, and the corresponding feature column must be filled with a filling value, in this case, "None".

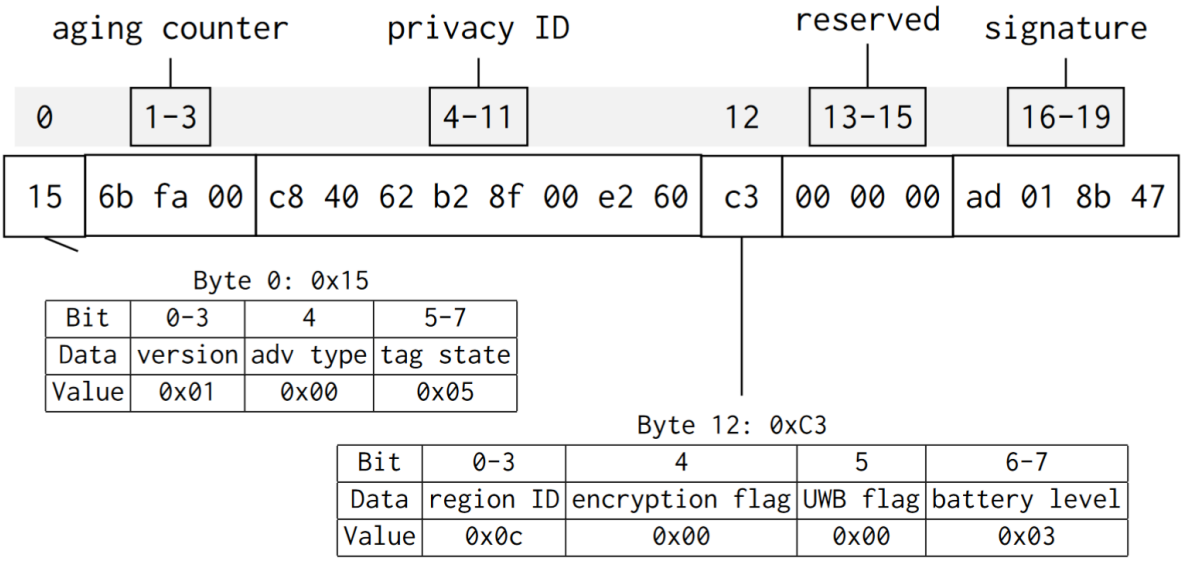

Figure 6.2: The Structure of the Service Data used by the Samsung SmartTag [13]

6.1.1.2.3 Malformed Packet Sometimes, a packet is malformed, i.e., the packet captured does not adhere to the Bluetooth specification, such as a failing cyclic redundancy check. Wireshark can detect this, indicating a malformed packet by extending the PDU. For instance, an ADV_IND PDU would become ADV_IND[Malformed Packet] if malformed. This can be extracted as a feature similar to the broadcast feature. This malformed packet feature is 1 if the packet is malformed and 0 otherwise.

6.1.2 Conversion to Dummy Variables

The conversion to dummy variables in the analysis section is only relevant for automatic machine-learning-based labeling. The machine learning models require numeric input data, so the categorical features have to be converted to dummy variables, i.e., one-hot encoded.

For this thesis, the one-hot encoding of the categorical features must meet certain requirements. These are:

- The dummy viable columns must be deterministic. In other words, for any given input data, the generated dummy variable columns and their order must be the same, i.e., independent of the occurrences of categorical values in the input data. This is relevant as the machine learning models for automatic labeling require a deterministic input.

- Categorical features can have multiple values at once, separated by commas. Every value should be one-hot encoded individually.

- Categorical features can have multiple occurrences of the same value. For example, the same UUID can be contained multiple times in one packet. In such cases, the one-hot encoded value should represent the number of occurrences in the packet.
- Every categorical feature can be absent, i.e., NULL. This is especially likely in the context of malformed packets that do not adhere to the Bluetooth specification. This absence should be represented with zeros, i.e., all dummy variable columns should be set to zero.

To tackle the above-described requirements, a solution was implemented using the Task-Group-Framework, i.e., a custom executor for one-hot encoding was built. Some of the strategies implemented are:

- Deterministic selection of dummy variable columns: The dummy variable columns are selected prior to the actual one-hot encoding. The order of columns can be specified, too. These selected columns cannot include the absence of a value, i.e., NULL or NONE.
- In addition to conventional dummy variable columns, there are two optional additional columns, one for other values and one for NULL values. The column for other values is used as a garbage category in case a value falls into none of the selected columns and is also not absent, i.e., NULL or NONE. The optional column for NULL values is used only for the NULL and NONE values. These values can be customized, i.e., the string "None" (as used above in the data cleaning) can be defined as a NULL value.
- In case of multiple values separated by commas, the one-hot encoding is done individually for each value. For this purpose, the comma-separated values are first split by commas and then one-hot encoded.
- Multiple occurrences of the same value can be one-hot encoded by counting. If a categorical value occurs multiple times in one sample, the one-hot encoded value reflects the number of occurrences, i.e., the count. The count is always a non-negative integer.

The strategies described above were implemented individually for all categorical features. The implementation varies from feature to feature. Some general remarks about the implementation that apply to all categorical features are the following:

- An empty packet (or absence of categorical features) results in all dummy variables set to zero. This is important, as the packet-rate approaches for machine learning can generate zero rows in the absence of packets. When the time-series data is resampled, and no sample is found within one sampling interval, a zero sample is generated. This zero sample must reflect the absence of a packet.
- All dummy variable columns were selected after visual inspection of the packets in Wireshark. Only dummy variable columns of seemingly relevant values were picked.

- The optional column for NULL values is never used. Otherwise, a zero row would not represent the absence of a packet.
- All categorical features have the "other column" except for the channel feature (there are precisely three channels, 37, 38, and 39, and no other channels). Given that the selected list of dummy variable columns is, in most cases, not exhaustive, this is necessary for proper one-hot encoding.
- For company IDs and UUIDs, the splitting of the comma-separated values is problematic. In case the UUID contains a comma, such as "Tile, Inc.", the splitting would separate "Tile" and "Inc.", resulting in two distinct values. Therefore, for all UUIDs and company IDs for which dummy variable columns were created, the complex variants were replaced with simpler versions without a comma. For example, "Tile, Inc." was replaced with "Tile".

  This ensures that the one-hot encoding results in a 1 in the respective column. This process of data cleaning was already described in the previous section. In all other cases, the incorrect splitting for values containing a comma will simply result in a 2 instead of a 1 for the one-hot encoded value in the column for other values, which is not a relevant issue.

The table below (Table 6.1) depicts the one-hot encoding for the fictional feature "Manufacturer". The values in this fictional column are separated by commas. The selected dummy variable columns are "Apple" and "Tile". Furthermore, there is an "Other" column for other values but no "NULL" column. The NULL value is the string "None". This fictional example is representative of the one-hot encoding of the "real" features.

Packet Nr.	Manufacturer	Dummy Apple	Dummy Tile	Dummy Other
1	Apple	1	0	0
2	Tile,Tile	0	2	0
3	Samsung	0	0	1
4	None	0	0	0
5	Apple, Inc.	1	0	1

Table 6.1: Example of One-Hot Encoding for the fictional Feature "Manufacturer"

6.1.3 Labeling

The packets' labeling varies depending on the type of labeling required. If the labeling can be done manually, i.e., no automatic machine-learning-based labeling is needed, all packets are assigned the same label. This was done using the Task-Group-Framework and Flags. In the other case, namely, in the case of "nearby" trackers, machine-learning-based labeling had to be used. This labeling was also implemented with the Task-Group-Framework and Flags. At this point, it is of utmost importance to note that machine-learning-based labeling is not perfect. However, in all cases, the test accuracy exceeded 99%. Therefore, the results are perfectly usable.

For the analysis part, all devices received a label. These labels are:

- Apple AirTag: AirTag
- 4Smarts SkyTag: SkyTag
- Chipolo One: Chipolo
- Samsung SmartTag: SmartTag
- Tile Mate: Tile
- Apple iPhone: iPhone
- Apple iPad: iPad
- Apple MacBook: MacBook
- Apple AirPod: AirPod
- other Devices: other Device

6.1.4 States and Continuity Type

In the final step, the class labels assigned to the packets were extended with a state. This means the state was added to the back of the class label using string concatenation. Every combination of devices and states forms its own respective category for analysis. The goal is to analyze each device in each of its states separately to gain as many insights as possible for modeling. The states were appended to the class labels using the Task-Group-Framework and Flags. The states are always in brackets after the class label separated by a space, i.e., the class label ”AirTag” in its ”lost” state becomes ”AirTag (lost)”.

The states to be analyzed vary from device to device and not all trackers can be in all states. For the trackers, the states are the following:

- All trackers: ”lost”
- All trackers: ”nearby”
- All trackers: ”unpaired”
- SmartTag and Tile: ”searching”

For the iDevices, i.e., iPhone, iPad, MacBook, the states are ”online” and ”offline”. For the AirPod, the states are ”nearby” and ”lost”. All the other devices do not have states.

However, there is one more thing to consider for the other Apple devices other than AirTags: the continuity packet types. First and foremost, it is reasonable to assume, after visual inspection, that continuity packets of the same type (i.e., packets used for the same service) are the same across devices. In other words, it is perfectly reasonable to

assume that a packet of continuity type 0x12 used for Find My is structured the same across all Apple devices. The same applies to other continuity packets of other types, such as 0x10 or 0x16. Second, later on, during inference, it will be highly relevant to distinguish between ”trackable” packets, i.e., packets of continuity type 0x12 (Find My packets), and any other continuity packets. Therefore, there is no point in aggregating packets of varying continuity types into one class label. The insights gained during the analysis must be minimal.

Subsequently, it is necessary to separate the varying continuity packets with different class labels and analyze each type for each device individually. Therefore, similarly to the device states, the Task-Group-Framework was used to append the continuity type of a packet to the class label. The continuity type was added with a space and the prefix ”CT” to the existing class label proceeding the state. For example, ”iPhone (offline)” would become ”iPhone CT 10 (offline)”. The continuity types are always in hexadecimal writing. No addition of a continuity type to a packet indicates that the packet is not part of the continuity protocol stack.

In the case of a packet of continuity type 0x12 (Find My), ”CT 12” was replaced with ”FindMy (online)” or ”FindMy (offline)” depending on the type of packet. The continuity type 0x12 packets vary in the length of the manufacturer specific data based on the type of packet used. The longer variant contains the full public key, whereas the shorter variant does not. The longer one indicates the device is offline and can only be tracked via Bluetooth (hence the public key). The shorter one indicates that the device is online and trackable via the internet, hence online. At this point, one might ask why it is necessary to encode the device state twice, once in the continuity type packet and once in the device state with brackets in the end, i.e., why is it necessary to label a 0x12 continuity type packet of an iPhone in its ”online” state as ”iPhone FindMy online (online)”?

The answer is scientific integrity. It is reasonable to assume that an Apple device will use the 0x12 packets correctly, i.e., online packets in its online state and offline packets in its offline state. But just because it is reasonable does not mean that this must be the case. It is, in fact, possible to induce an ”iPhone FindMy online (offline)” packet. This is not common, but possible. If the internet connection via WiFi and cellular is not correctly switched off, Apple devices will turn both back on after some grace period. At this point, the iPhone switches from transmitting ”Find My offline” packets to transmitting ”Find My online packets”. However, because the iPhone was initially switched off, the state label for the captured file is ”offline”; hence, an ”iPhone FindMy online (offline)” packet is created.

6.2 Analysis

This section covers the analysis of the captured data of all the devices. The analysis is performed using the same plots for every device. The first subsection provides details about these plots, and the following subsections analyze the captured data for every device and discuss the findings. For simplicity, some devices were grouped into one plot/analysis, and not all available plots are shown for every device. The goal was to show the most

interesting plots and not bombard the reader with visualization over visualization. All plots can be found on GitHub.

The analysis aims to identify features that potentially allow for the categorical classification of devices and to identify potential categories of devices, i.e., find devices that should be put into the same class for categorization.

6.2.1 Plots

For this analysis, plots relevant to the extracted features (and subsequent modeling) were created using Matplotlib and the Task-Group-Framework. Essentially, a pipeline of executors for plotting was created to streamline the plotting process. The following plots were created for analysis:

- **Number of Packets**: A bar chart of the number of packets for each class label. For the training of machine learning models, the amount of available data is highly relevant.
- **BoxPlot of BLE Address Interval:** A box plot of the time interval in minutes between source address changes. This is especially relevant for calculating the packet rate. The time interval over which the packet rate is calculated must be significantly below the source address change interval.
- **Average BLE Address Interval:** A bar chart of the average time interval in minutes between source address changes.
- **Percentage of Broadcast Packets:** A bar chart of the relative share of broadcast packets.
- **Protocol:** A stacked bar chart of the relative share of the transmission protocols.
- **Channel:** A stacked bar chart of the relative share of the advertising channels.
- **BoxPlot of Length of Header:** A box plot of the length of the packet header in bytes (i.e., the length of the packet on the BLE link layer).
- **Average Length of Header:** A bar chart of the average length of the packet header in bytes (i.e., the length of the packet on the BLE link layer).
- **BoxPlot of Length of Packet:** A box plot of the length of the packet in bytes (i.e., the length of the packet on the nRF 58240 DK layer).
- **Average Length of Packet:** A bar chart of the average length of the packet in bytes (i.e., the length of the packet on the nRF 58240 DK layer).
- **BoxPlot Length of Manufacturer specific Data:** A box plot of the length of the manufacturer specific data in bits.

- **Average of Length of Manufacturer specific Data:** A bar chart of the average length of the manufacturer specific data in bits. This average is only calculated over the packets where the length is greater than zero.
- **BoxPlot of Length of Service Data:** A box plot of the length of the Service Data in bits.
- **Average Length of Service Data:** A bar chart of the average length of the Service Data in bits. This average is only calculated over the packets where the length is greater than zero.
- **Company ID:** A stacked bar chart of the relative share of company IDs.
- **UUIDs:** A multiple bar chart showing the average count of a UUID per packet.
- **Continuity Type:** A stacked bar chart of the relative share of continuity types of packets.
- **SmartTag Type:** A stacked bar chart of the relative share of SmartTag types of packets (i.e., the status bits used by the Samsung SmartTag tracker).
- **Advertisement Data Type:** A multiple bar chart of the usage of advertisement data types among packets.
- **PDU Type:** A stacked bar chart of the relative share of PDU types.
- **Malformed Packet:** A bar chart of the relative share of malformed packets.
- **BoxPlot of Packet Rate:** A box plot of the packet rate per channel and class label. The packet rate is calculated over a 15-second non-overlapping window.
- **Average Packet Rate:** A multiple bar chart of the average packet rate per channel and class label. The packet rate is calculated over a 15-second non-overlapping window.
- **Graph of Packet Rate:** A line plot showing the packet rate per channel and class label over time for every file. The packet rate is calculated over a 15-second non-overlapping window.

At this point, it is important to note that these plots do not depict the raw captured data in the most truthful way possible. The plots should provide a representative picture of the devices and not represent the data exactly as captured. For instance, some class labels were omitted for brevity if only a handful of packets were captured for said label. Another example would be the plot of the graph of the packet rate. The packet rate is not shown for the first and last interval. These two intervals must not necessarily be 15 seconds long. Hence, the packet rate can vary widely for these two intervals.

6.2.2 Find My Trackers

The following subsection contains almost all of the available plots for the three Find My trackers from Apple, Chipolo, and Tile; some of the less interesting packet rate plots were omitted for brevity.

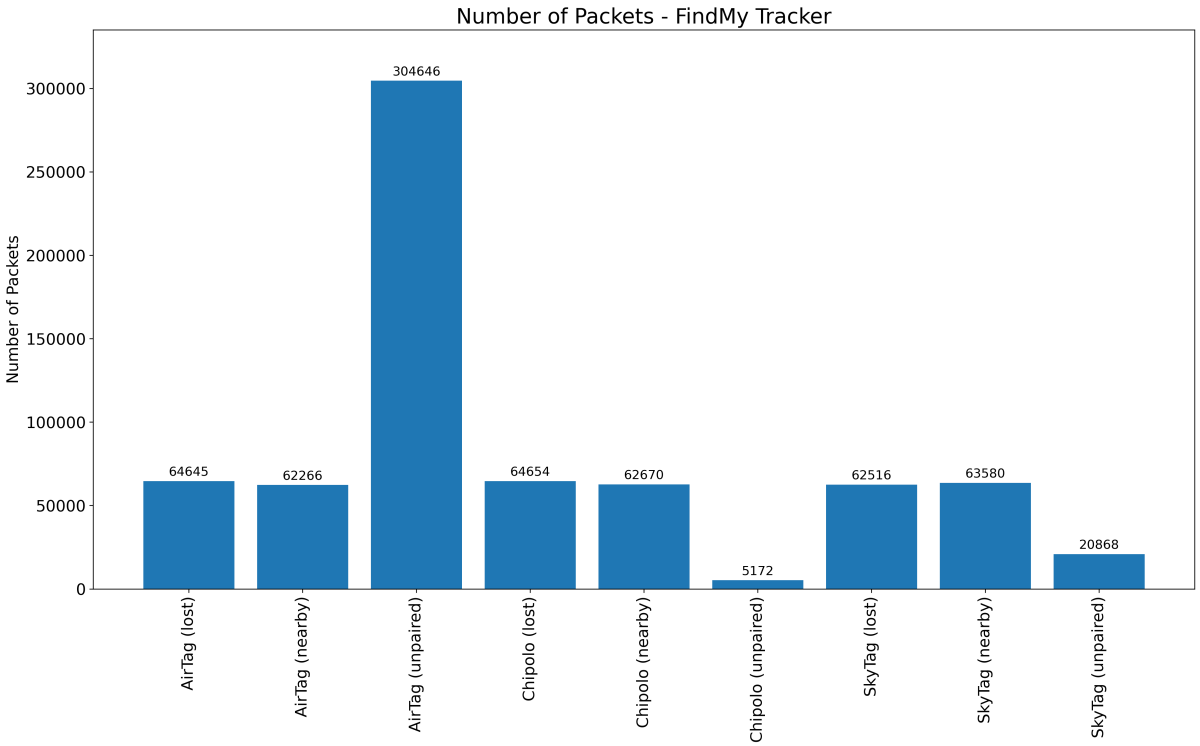

Figure 6.3: Number of Packets - FindMy Tracker

The number of packets captured is almost the same for all three trackers in the ”nearby” and ”lost” states, indicating the same packet rate too. The number of packets captured in the ”unpaired” states varies widely, given the time frames over which the packets were captured (see Chapter 4 on the captured files). It is questionable whether the Chipolo tracker and the SkyTag in their ”unpaired” state can be used for modeling, given the few packets captured.

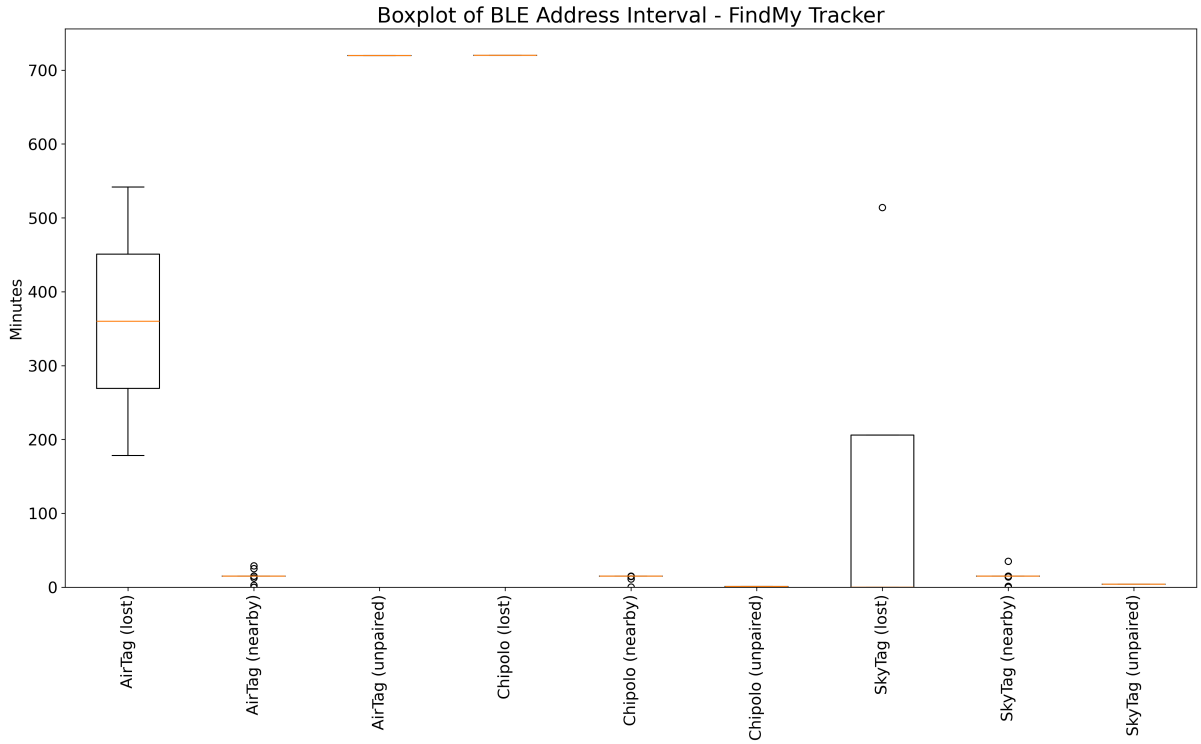

Figure 6.4: Boxplot of BLE Address Interval - FindMy Tracker

The address of the Find My trackers in the ”lost” state changes exactly once per day at 04:00 a.m., hence the strange-looking box plot. However, this is not visible in the case of the Chipolo, as the file was not captured overnight. The variation of the address intervals is otherwise negligible in all states other than the ”lost” state.

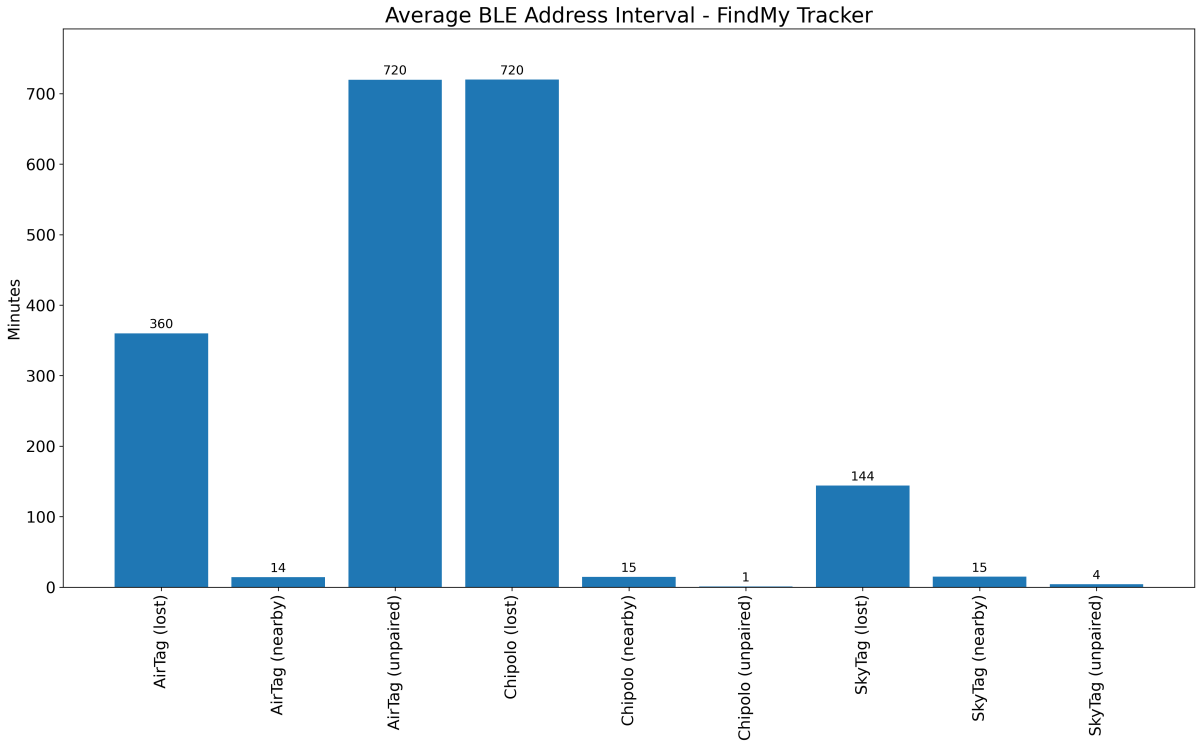

Figure 6.5: Average BLE Address Interval - FindMy Tracker

The average address interval in the ”nearby” state is relatively short, with about 15 minutes on average, but still large enough for packet rate modeling. In the ”unpaired” state of the AirTag, the address is fixed and never changes. The address interval of the Chipolo and the SkyTag in their ”lost” state is equal to the time length of the captured data because the capture time was very short. For this state and these two trackers, the BLE address interval is probably too short for modeling the packet rate. All other combinations of trackers and states are suitable for modeling the packet rate.

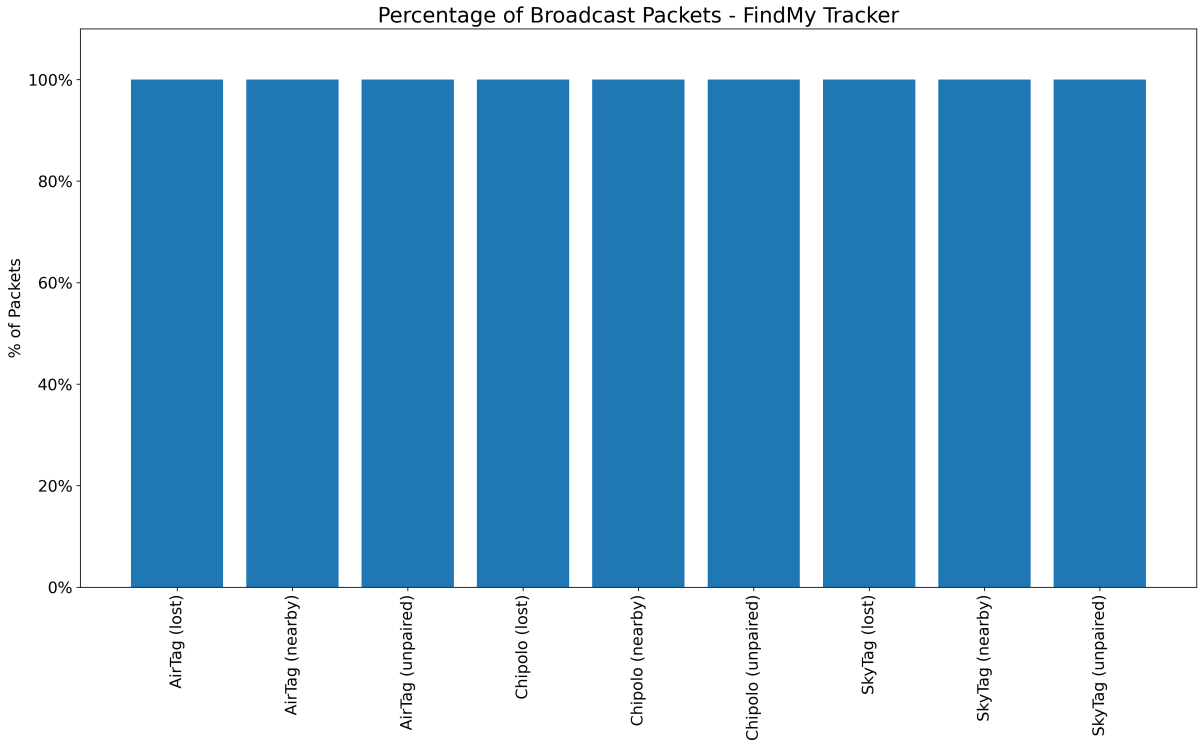

Figure 6.6: Percentage of Broadcast Packets - FindMy Tracker

The percentage of broadcast packets is about 100% for all trackers and states. These trackers never use non-broadcast packets.

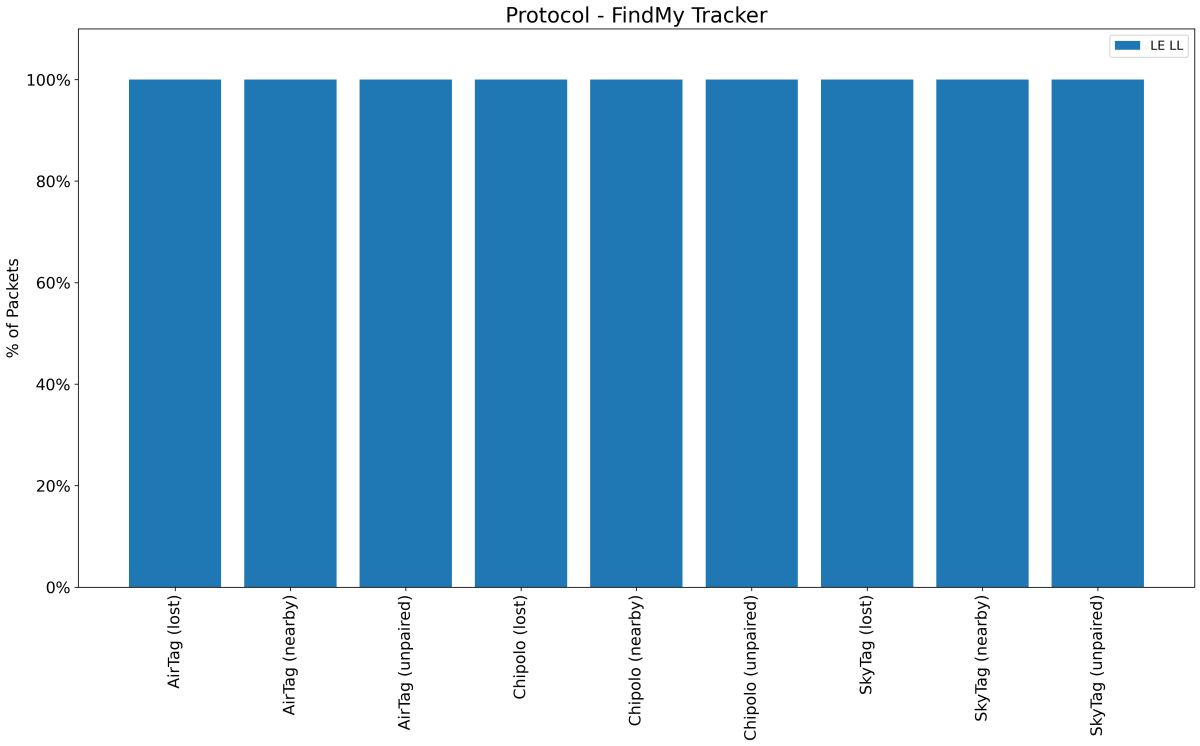

Figure 6.7: Protocol - FindMy Tracker

The protocol does not vary. All combinations of trackers and states always use the same protocol.

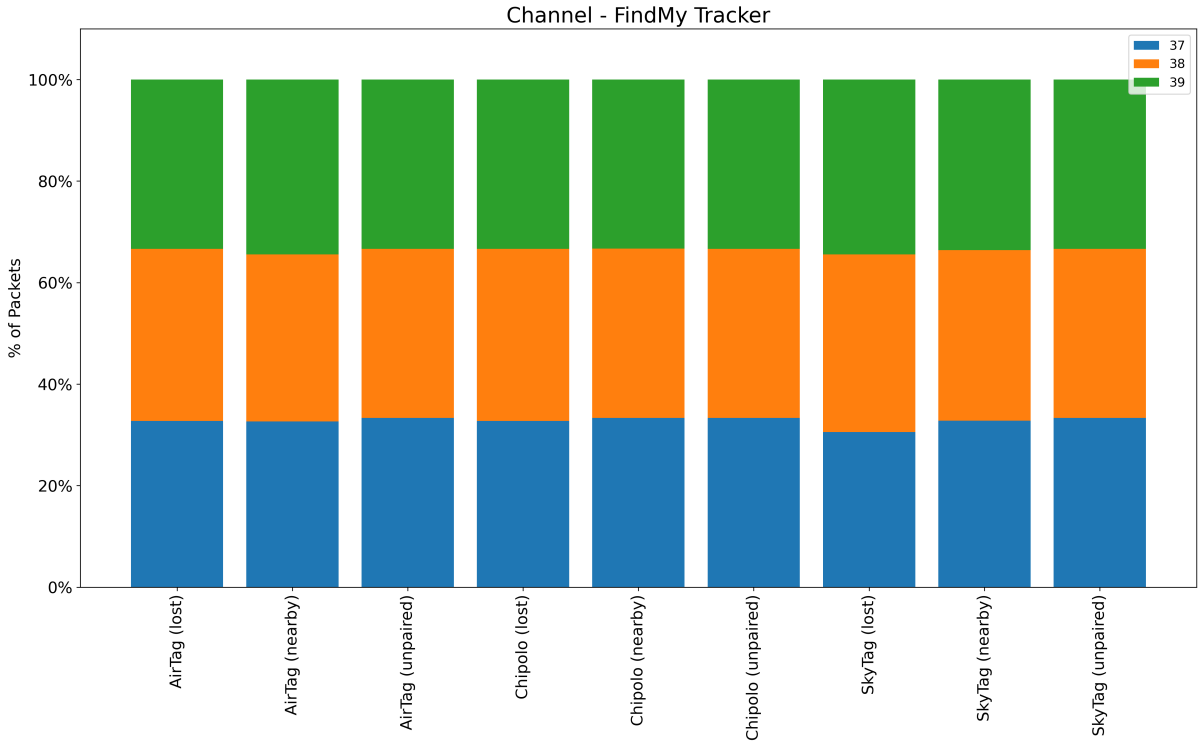

Figure 6.8: Channel - FindMy Tracker

All trackers in every state use each channel equally, i.e., every channel is used for roughly one-third of packets.

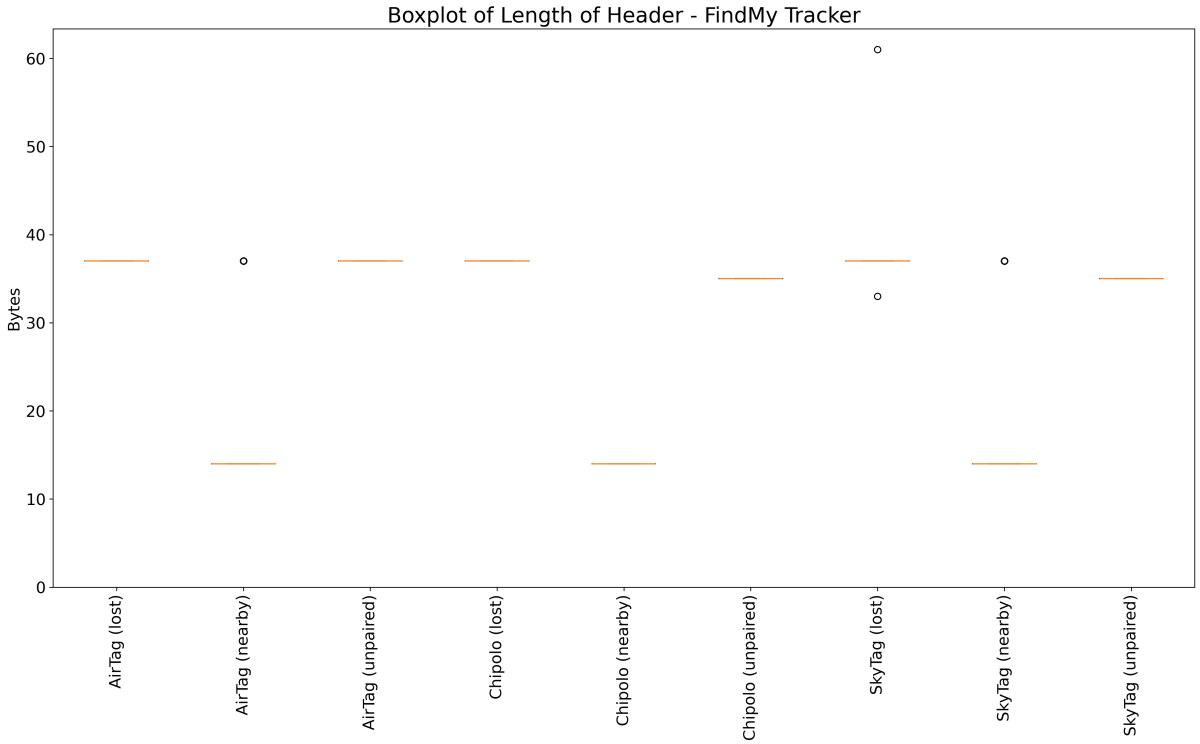

Figure 6.9: Boxplot of Length of Header - FindMy Tracker

The length of the header shows little to no variance, except for some outliers. Interestingly, in the case of the AirTag and the SkyTag in their ”nearby” state, the outlier value is equal to the median value of the header length in the ”lost” state. Additionally, the SkyTag in the ”lost” state shows outliers to values uncorrelated with other states, potentially a capture of unrelated background traffic.

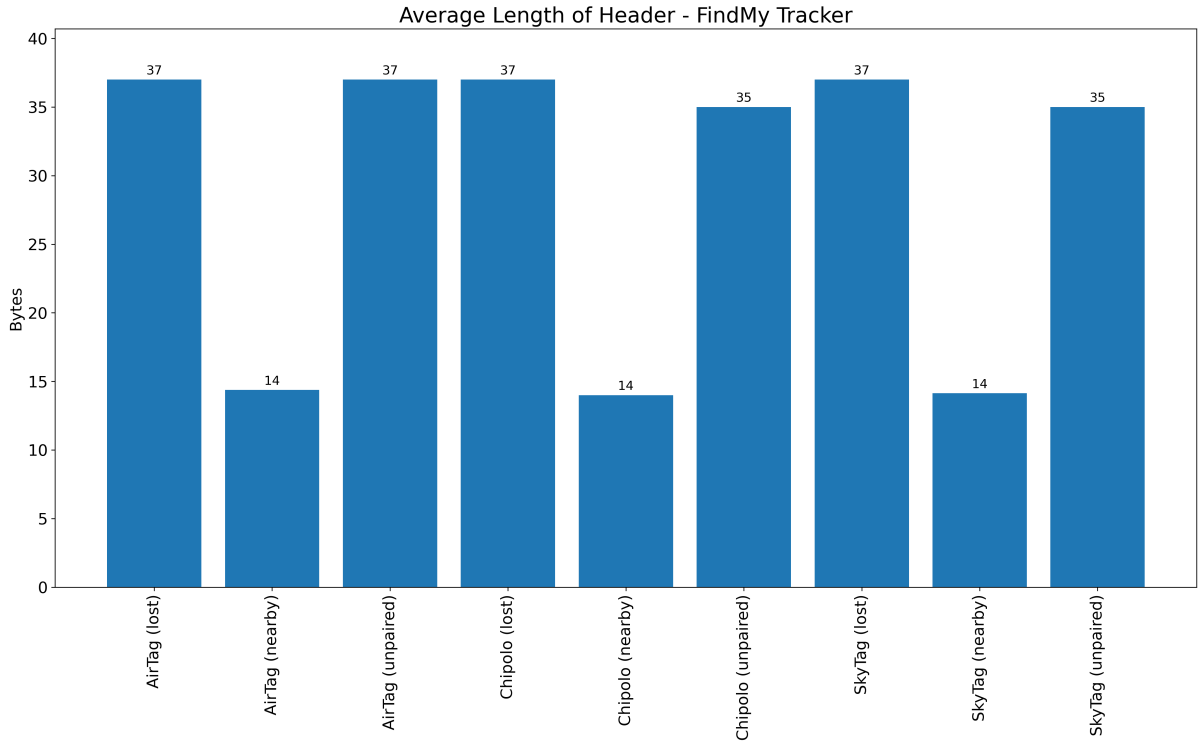

Figure 6.10: Average Length of Header - FindMy Tracker

The average values for the header length reveal that the packet length on the link layer differs between the ”unpaired” state of the Chiplo/SkyTag and the AirTag. This indicates a different implementation of the ”unpaired” state between Find My trackers from Apple and Find My trackers from other companies. The significant difference between the length in the ”lost” and ”nearby” states indicates a very different packet structure. This is most likely the case because of the inclusion/exclusion of the public key, which depends on the device state.

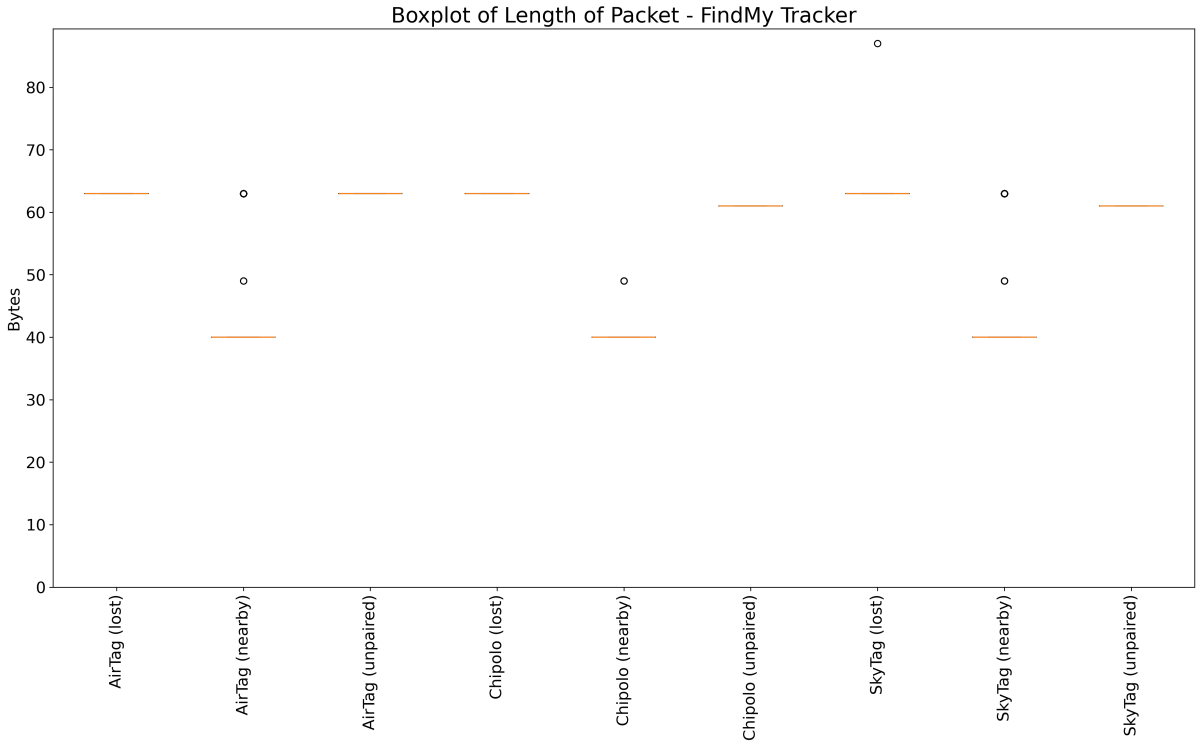

Figure 6.11: Boxplot of Length of Packet - FindMy Tracker

Given that the packet length, i.e., the length of the packets sent by the nRF 58240 DK to Wireshark, is dependent on the length of the header, the box plot should be similar to the box plot for the header length. As can be seen above, this is very much the case. The variance is equally small, and the outliers indicate that sometimes the packets in the ”nearby” state are similar to those in the ”lost” state.

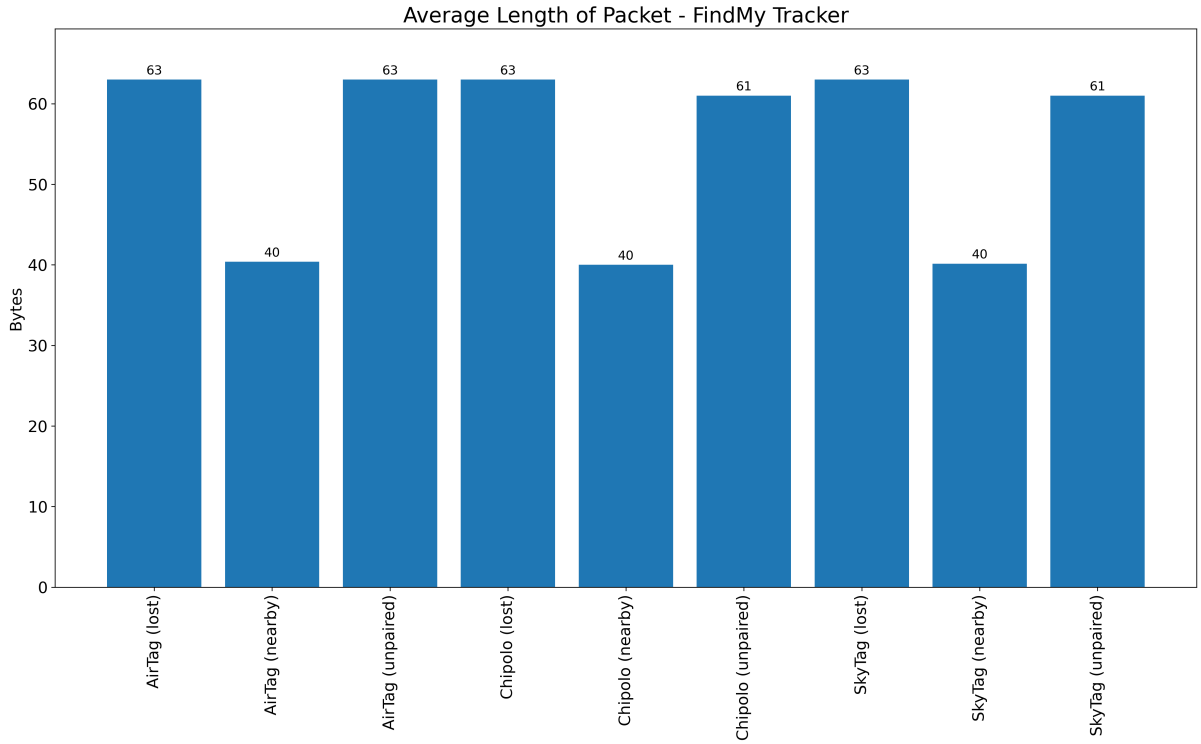

Figure 6.12: Average Length of Packet - FindMy Tracker

The average length of the packets again indicates that the implementation of the ”unpaired” state is different for Find My trackers from Apple and other companies. This kind of result allows for the verification of assumptions.

Figure 6.13: Boxplot of Length of Manufacturer specific Data - FindMy Tracker

There are a number of interesting aspects of the box plot of the manufacturer specific data:

- The length differs between the ”nearby” state and the ”lost” state due to the inclusion/exclusion of the public key.
- The difference in length aligns with the difference in header and packet length.
- The Chipolo and SkyTag do not use manufacturer specific data for their ”unpaired” state. This aligns with the difference in header length and implies a technical implementation that is very different from that of the Airtag.
- Some of the outliers in the ”nearby” state are particularly interesting. Some of the outliers can be attributed to errors during automatic labeling. However, it appears as if the trackers in the ”nearby” state would sometimes switch to the ”lost” state. Sometimes, the outlier is precisely equal to the median length in the ”lost” state.

  Visual inspection of the packets in Wireshark revealed that, in fact, the trackers sometimes leave the ”nearby” state for a brief period of time and enter the ”lost” state. I do not have any explanation for this behavior. However, it is abundantly clear that it is not a fluke but a reproducible behavior that was captured many times.

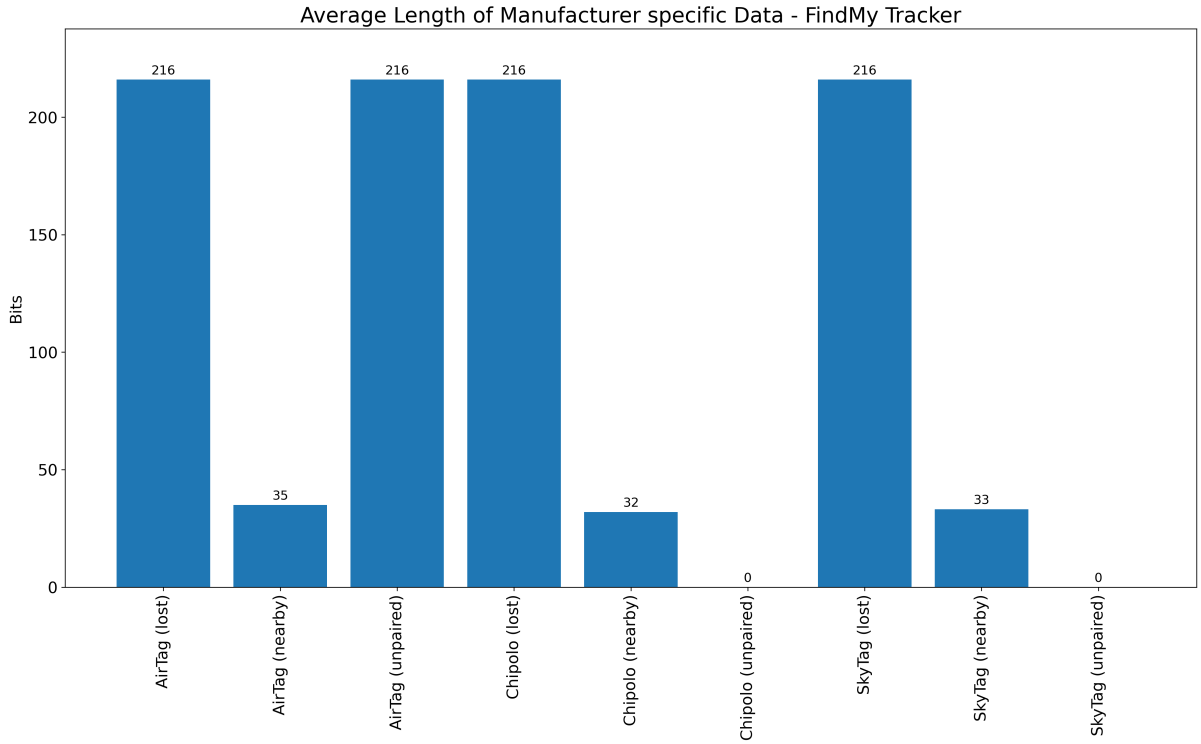

Figure 6.14: Average Length of Manufacturer specific Data - FindMy Tracker

The plot of the average length of the manufacturer specific data aligns with the previous box plot. Most interesting is, that the lengths in the ”nearby” state are slightly above the expected value found by visual inspection (32 bits). This is due to the switching from the ”nearby” state to the ”lost” state. The length in the ”lost” state is longer than in the ”nearby” state, hence a higher overall average.

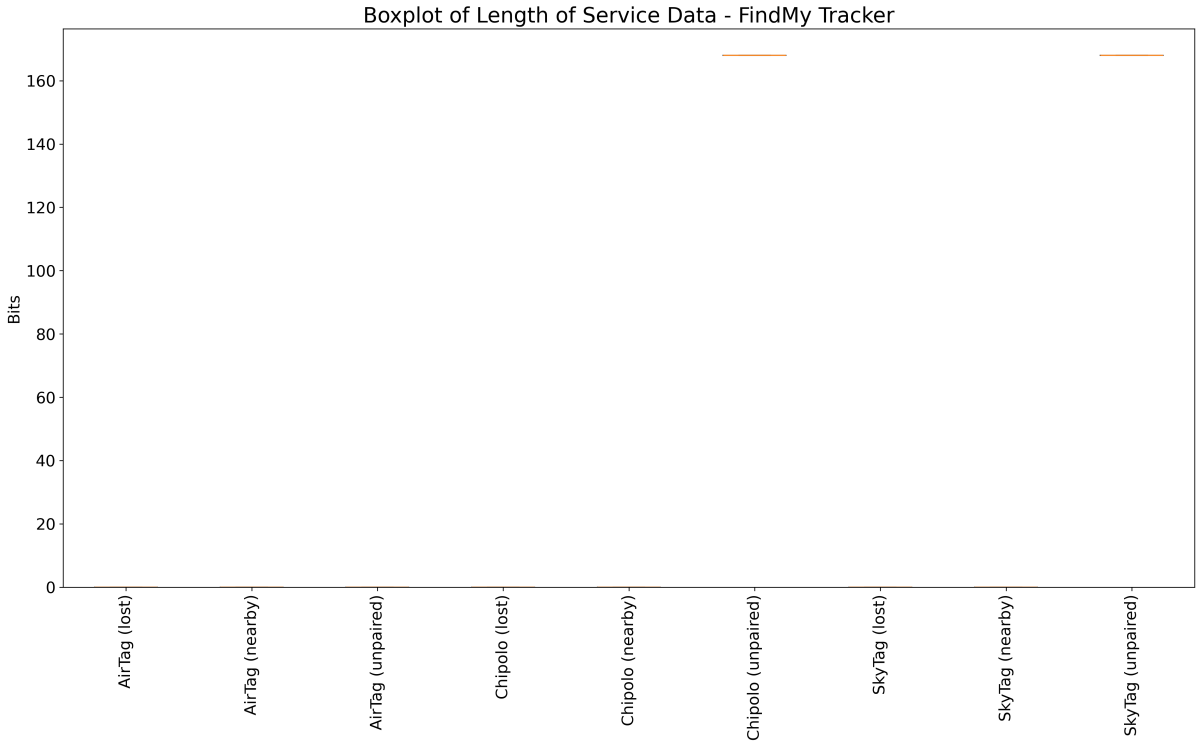

Figure 6.15: Boxplot of Length of Service Data - FindMy Tracker

The box plot for the Service Data reveals that the Chipolo and the SkyTag use Service Data in their ”unpaired” state. This aligns with previous findings indicating a very different technical implementation. The variation is again minimal, and there are no outliers.

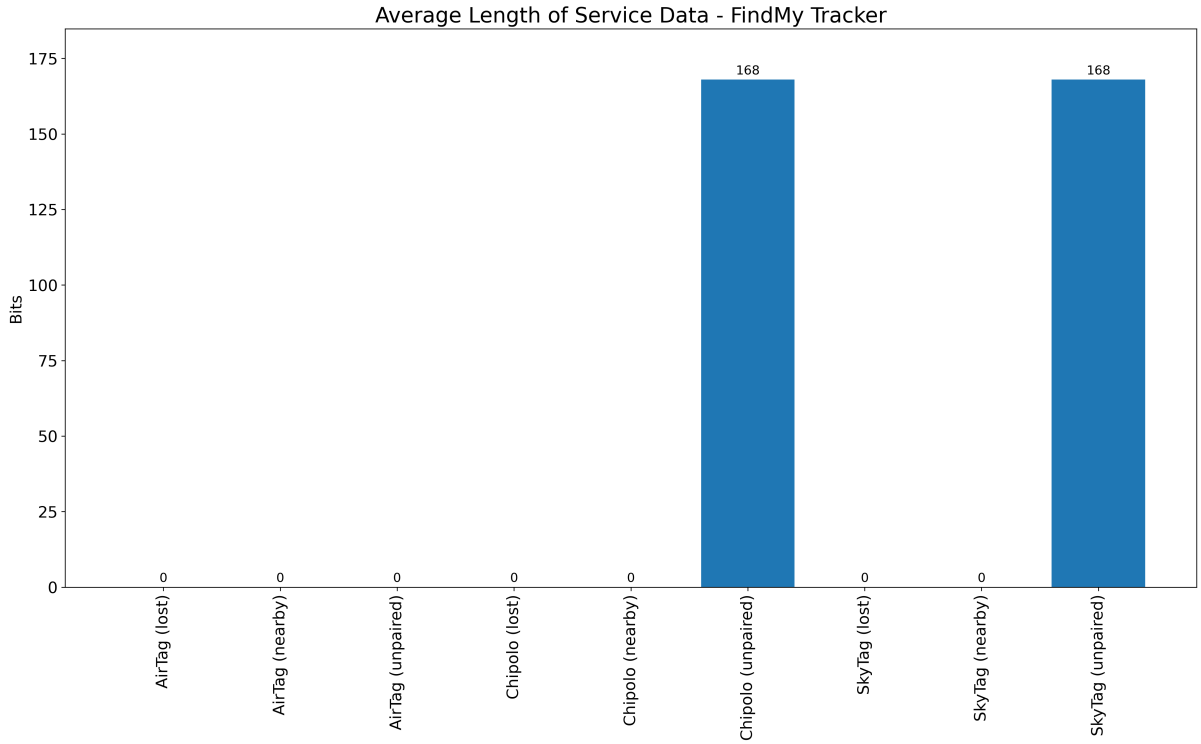

Figure 6.16: Average Length of Service Data - FindMy Tracker

As expected, the average length of the Service Data is zero except for the Chipolo and SkyTag in their ”unpaired” state. However, in both cases, the average length is identical, indicating a very similar technical implementation.

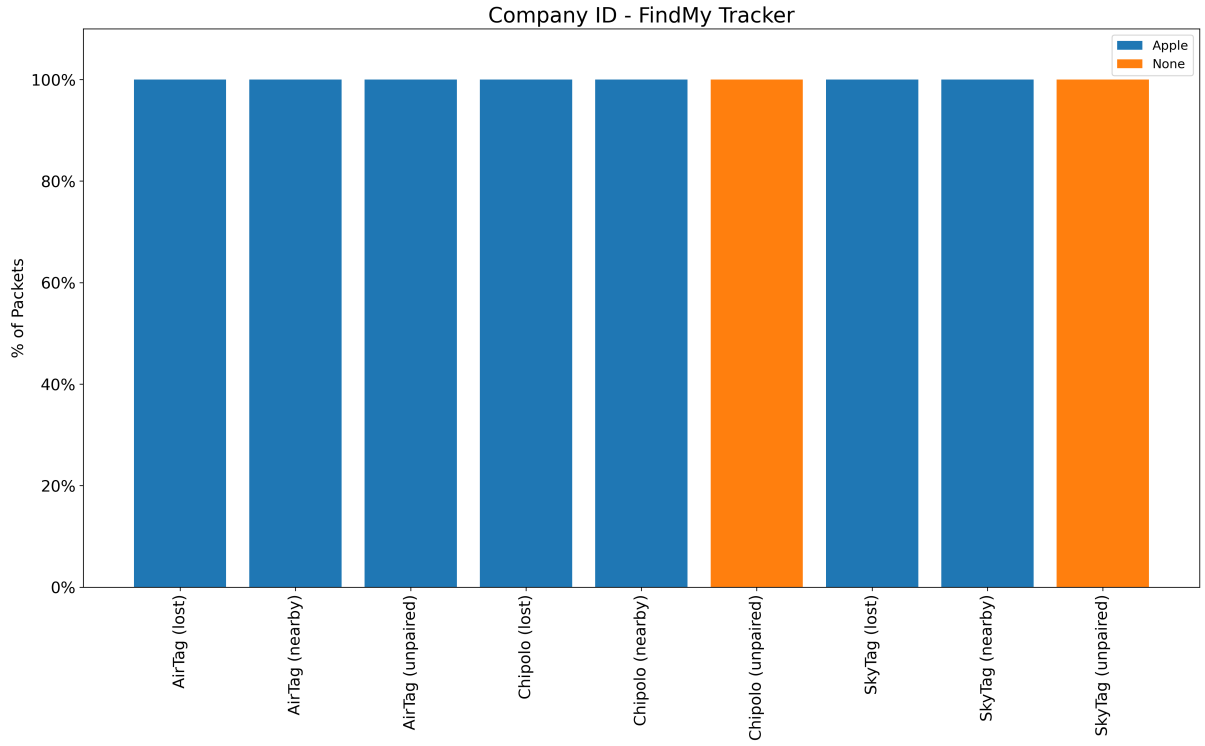

Figure 6.17: Company ID - FindMy Tracker

The distribution of the company IDs reveals that the ID is always from Apple except when devices do not use manufacturer specific data (Chipolo and SkyTag in their ”unpaired” state).

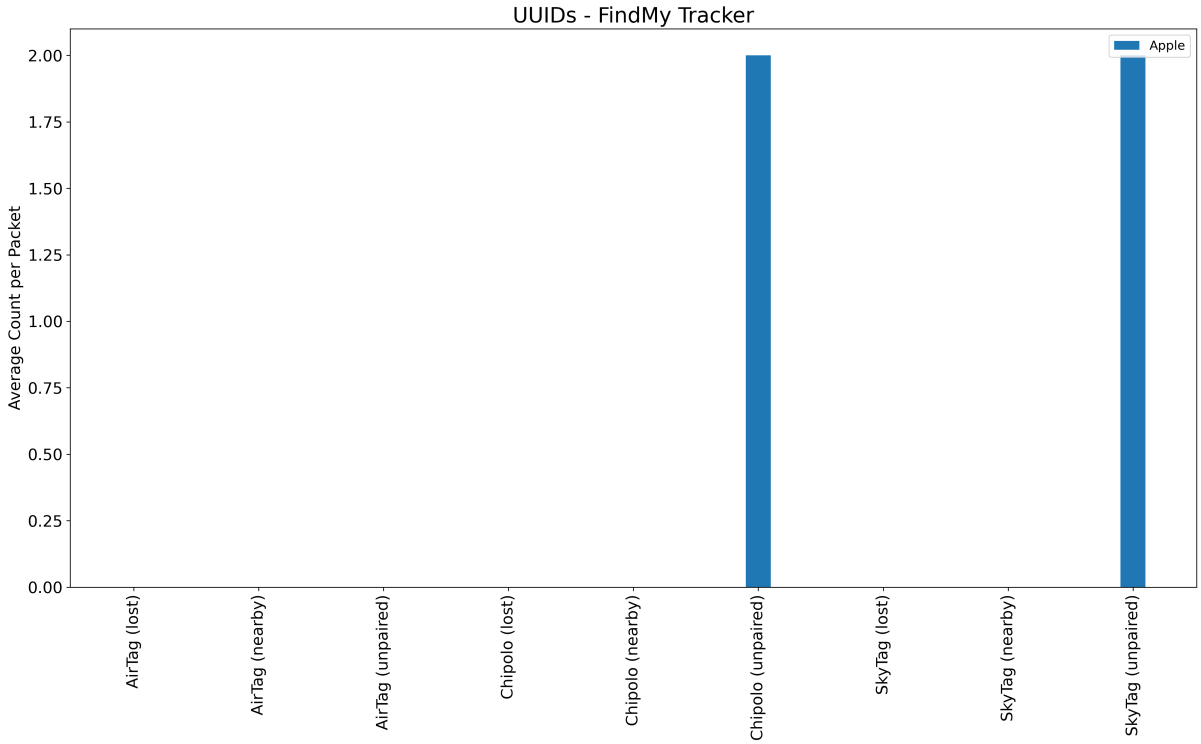

Figure 6.18: UUIDs - FindMy Tracker

Conversely, to the previous plot, only the Chipolo and SkyTag in their ”unpaired” state use UUIDs. Both use, on average, precisely two UUIDs, both from Apple. In any other case, no UUIDs are used. This further emphasizes these two trackers’ very different technical implementations of the ”unpaired” state.

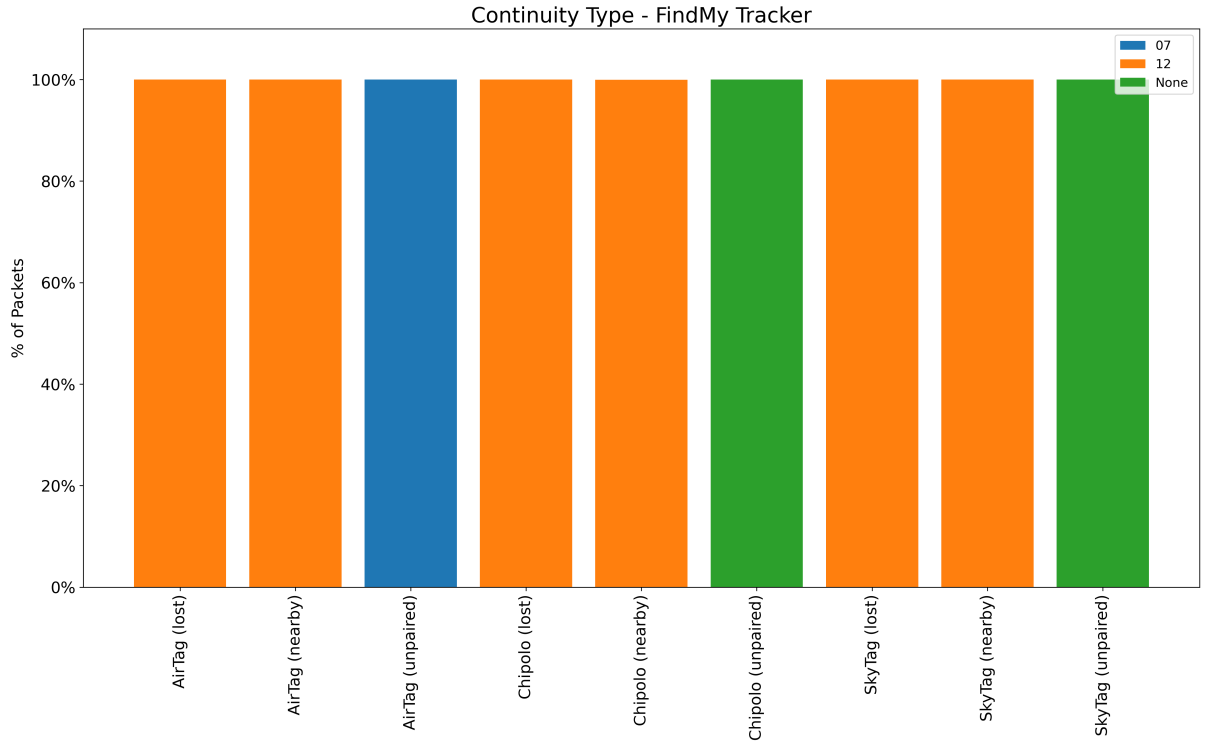

Figure 6.19: Continuity Type - FindMy Tracker

Unsurprisingly, the continuity type is mostly 0x12, except for the ”unpaired” states. The Chiplo and the SkyTag do not use manufacturer specific data; hence, the continuity type is always none. For the AirTag in its ”lost” state, the continuity type 0x07 indicates a packet used for ”proximity paring” [12]. That is a reasonable finding, as an AirTag in its ”unpaired” state needs to be paired to a device in close proximity.

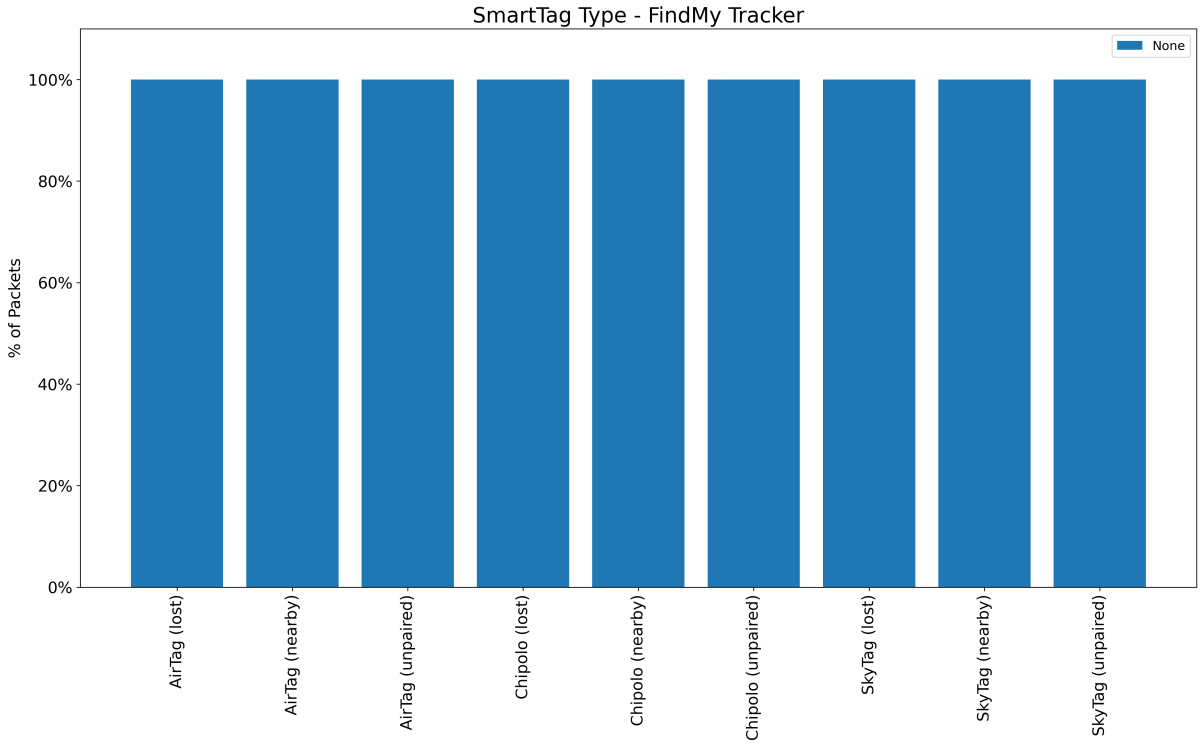

Figure 6.20: SmartTag Type - FindMy Tracker

The SmartTag type, i.e., the leading bits used by the SmartTag tracker, is none in all cases. Anything else would have been very surprising for Find My trackers.

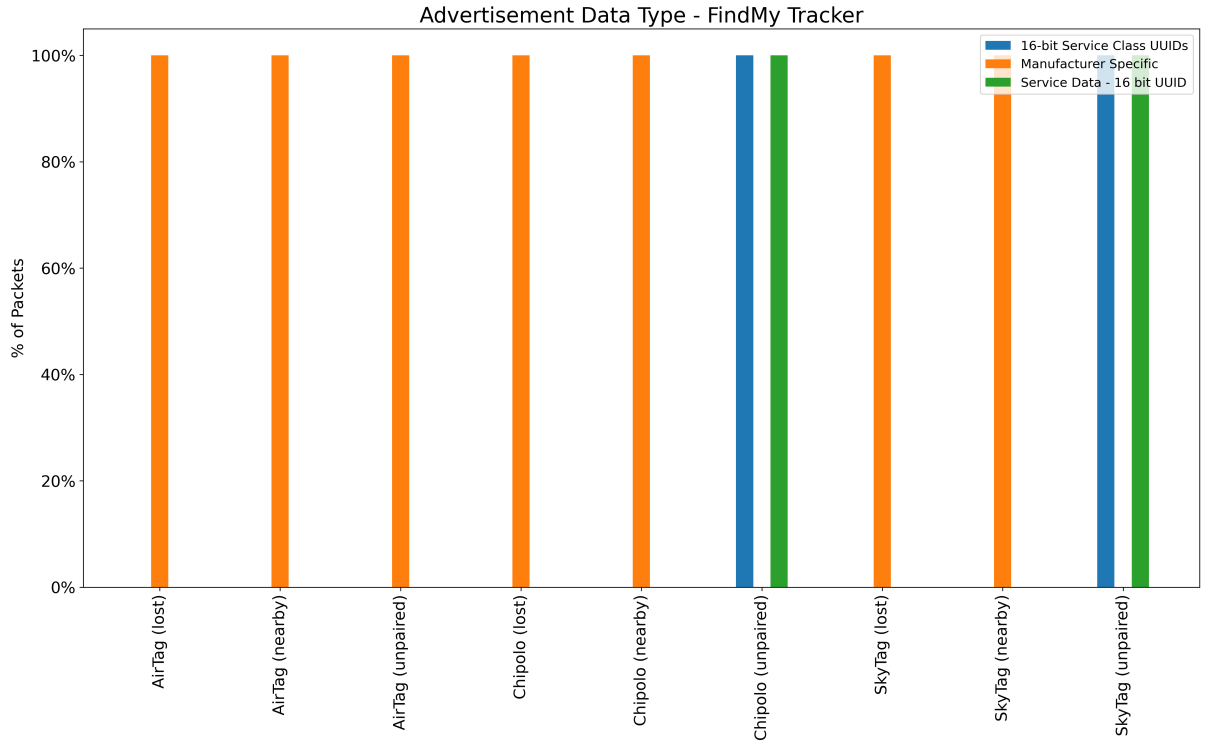

Figure 6.21: Advertisement Data Type - FindMy Tracker

Correlating with previous findings, almost all trackers and states use manufacturer specific data as their only PDU payload. Only the Chiplo and the SkyTag in their ”unpaired” state use Service Data and Service Class UUIDs. This also explains why the average number of UUIDs is 2 and not 1 per packet.

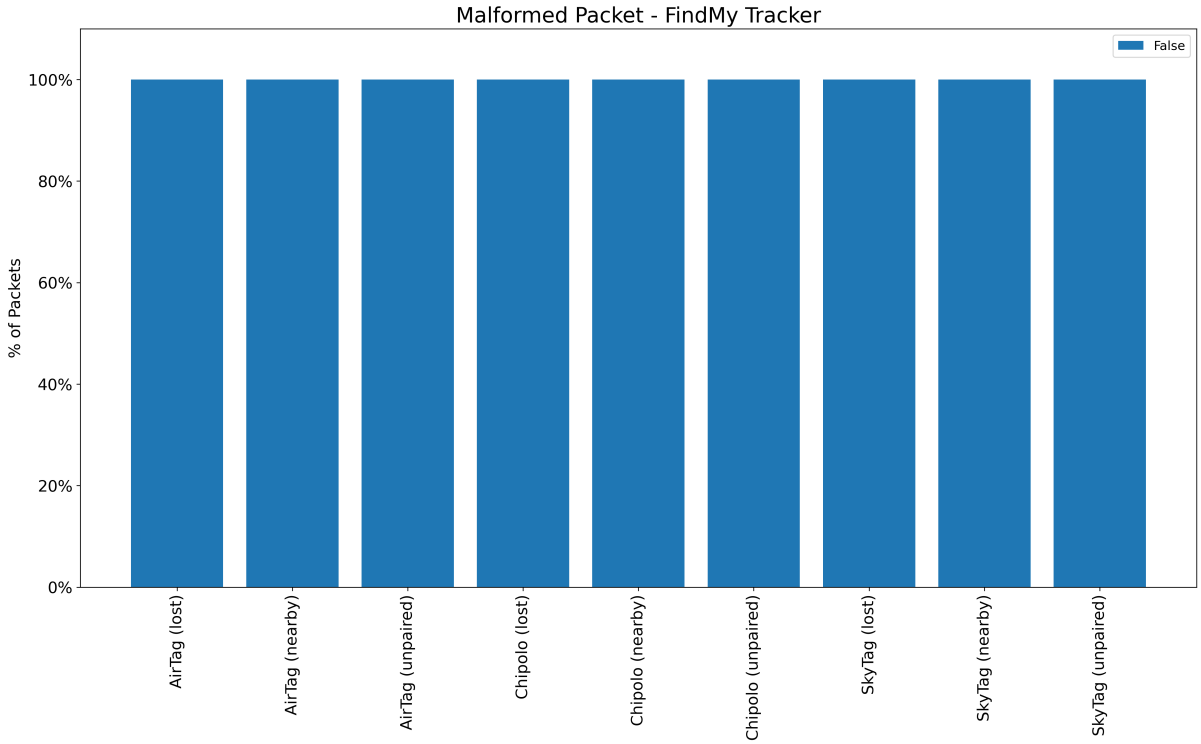

Figure 6.22: Malformed Packet - FindMy Tracker

Find My trackers never transmit malformed packets (at least not in laboratory-like conditions).

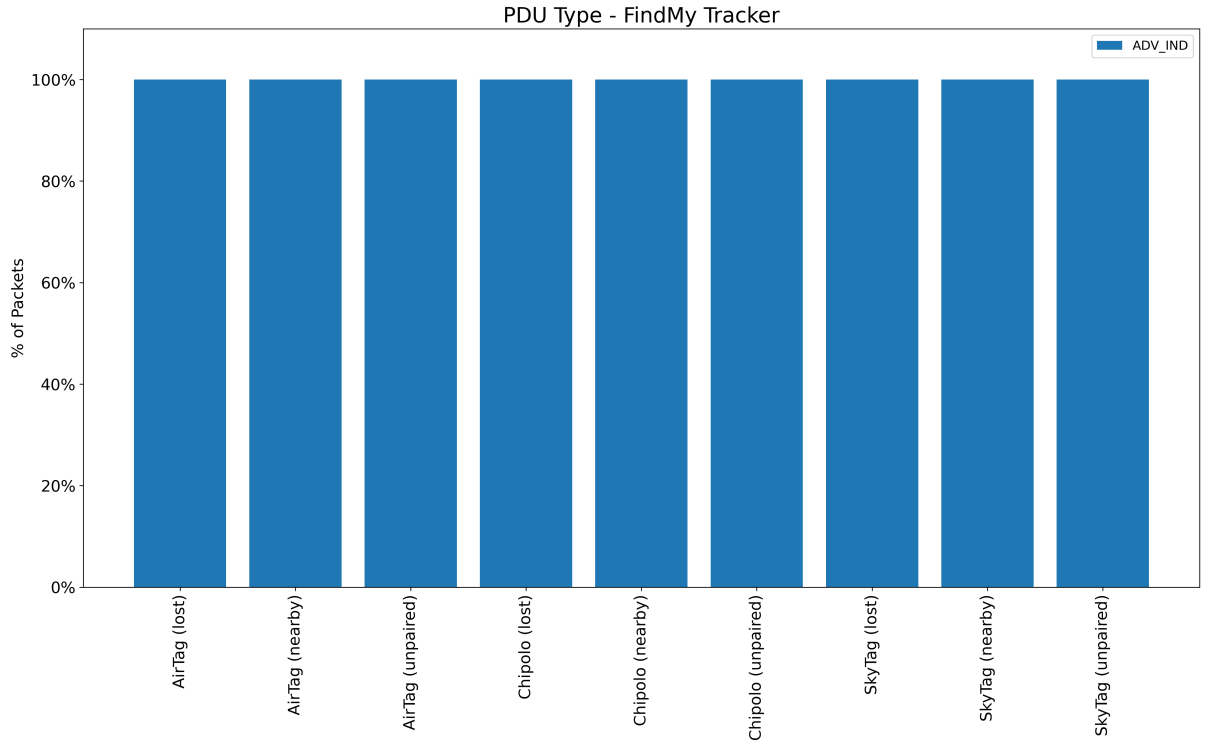

Figure 6.23: PDU Type - FindMy Tracker

The PDU type used by Find My trackers is always ADV_IND. These trackers also do not use SCAN_REQ or SCAN_RSP PDUs, but that is most likely only due to the laboratory-like conditions. This might look very different in real-world environments.

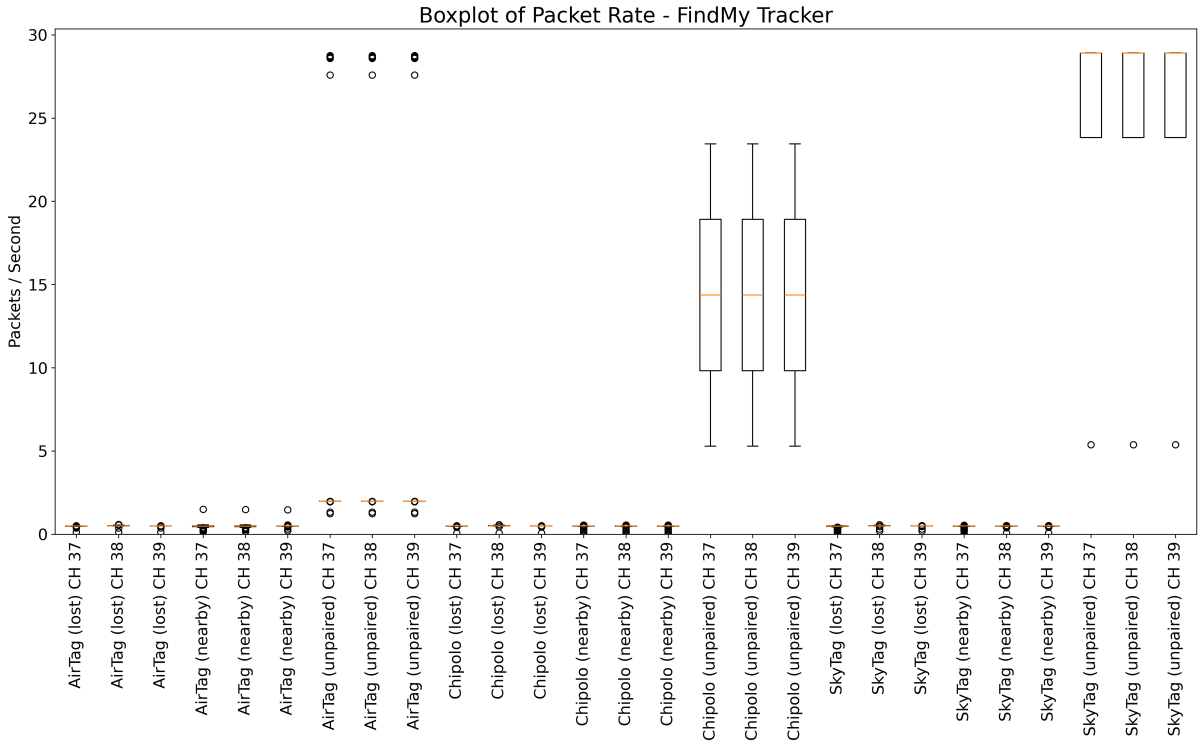

Figure 6.24: Boxplot of Packet Rate - FindMy Tracker

The box plot of the packet rate reveals three notable things. First, there is little variation in the packet rate, except for the Chipolo and the SkyTag in their ”unpaired” state. This is due to the short capture time for these two cases. Second, there are quite a few outliers for the AirTag in its ”unpaired” state. These outliers are about an order of magnitude above the median value. Finally, there are no significant differences between the three channels.

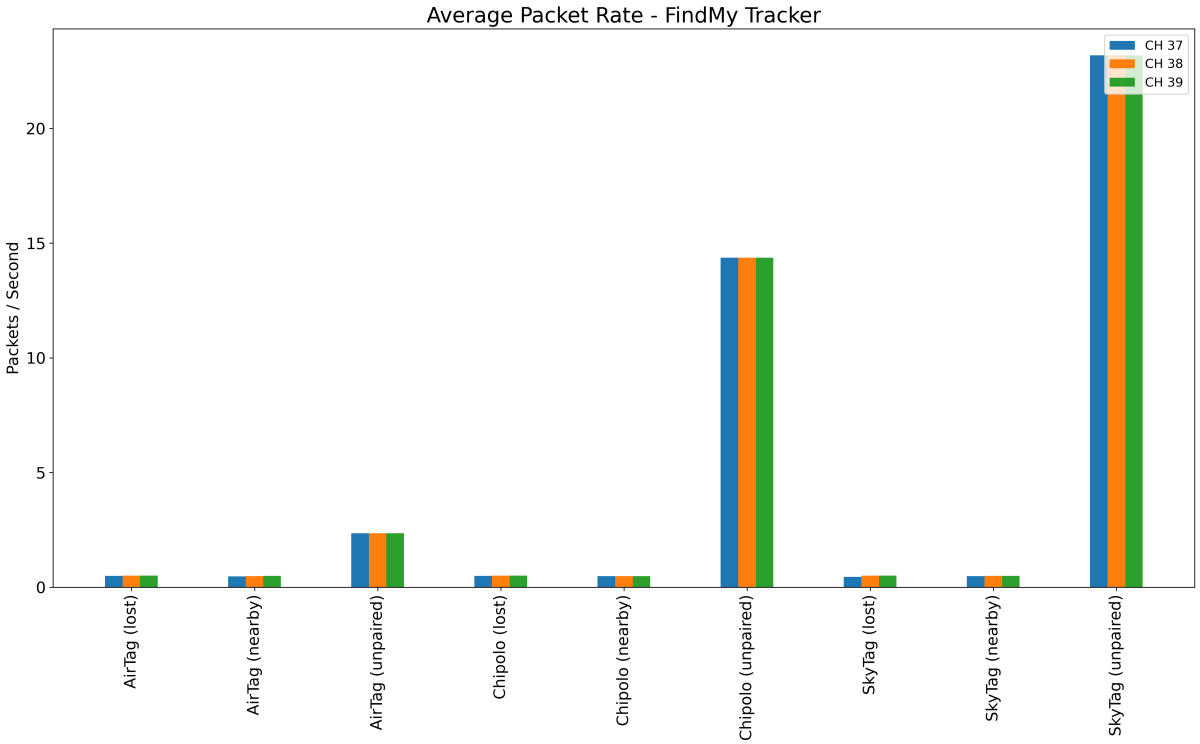

Figure 6.25: Average Packet Rate - FindMy Tracker

The average packet rate plot aligns with the corresponding box plot. Most interesting is the average for the AirTag in its ”unpaired” state. The average is roughly at the median value, which indicates that there are only a few of these extreme outliers.

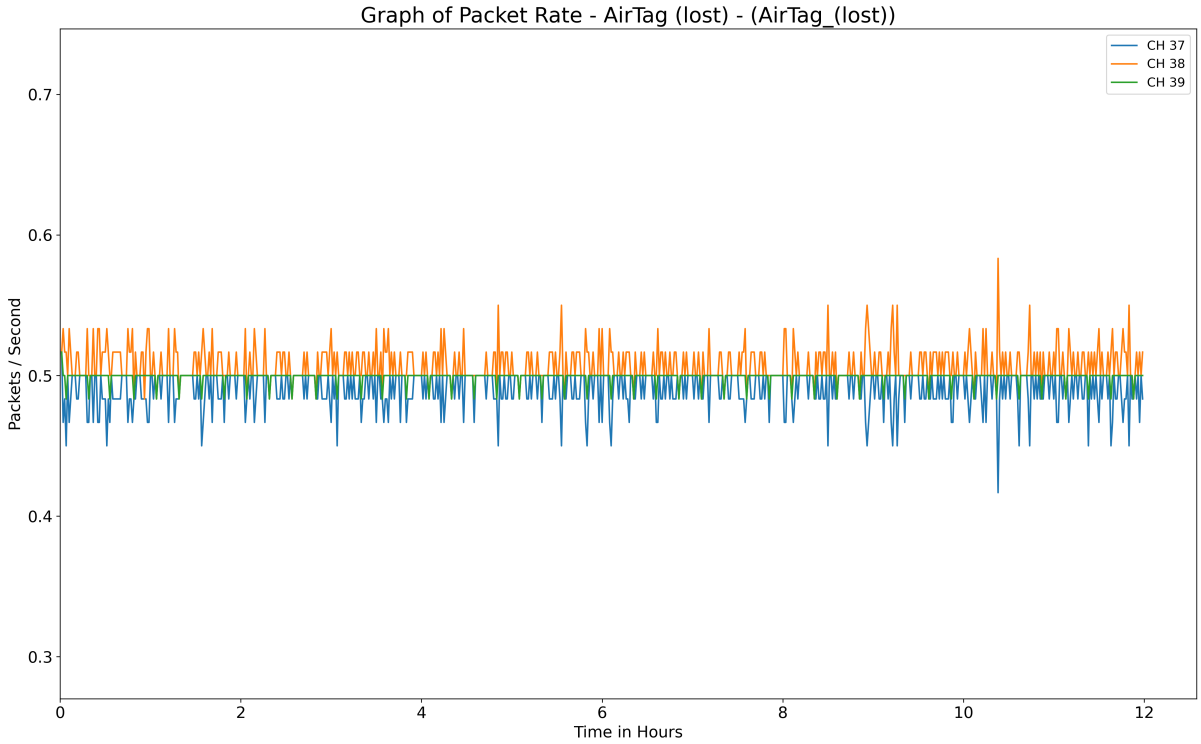

Figure 6.26: Graph of Packet Rate - AirTag (lost)

The graph of the packet rate for the AirTag in its ”lost” state perfectly aligns with the previous plots. There are no significant deviations from the mean, little variance over time, and no differences between channels.

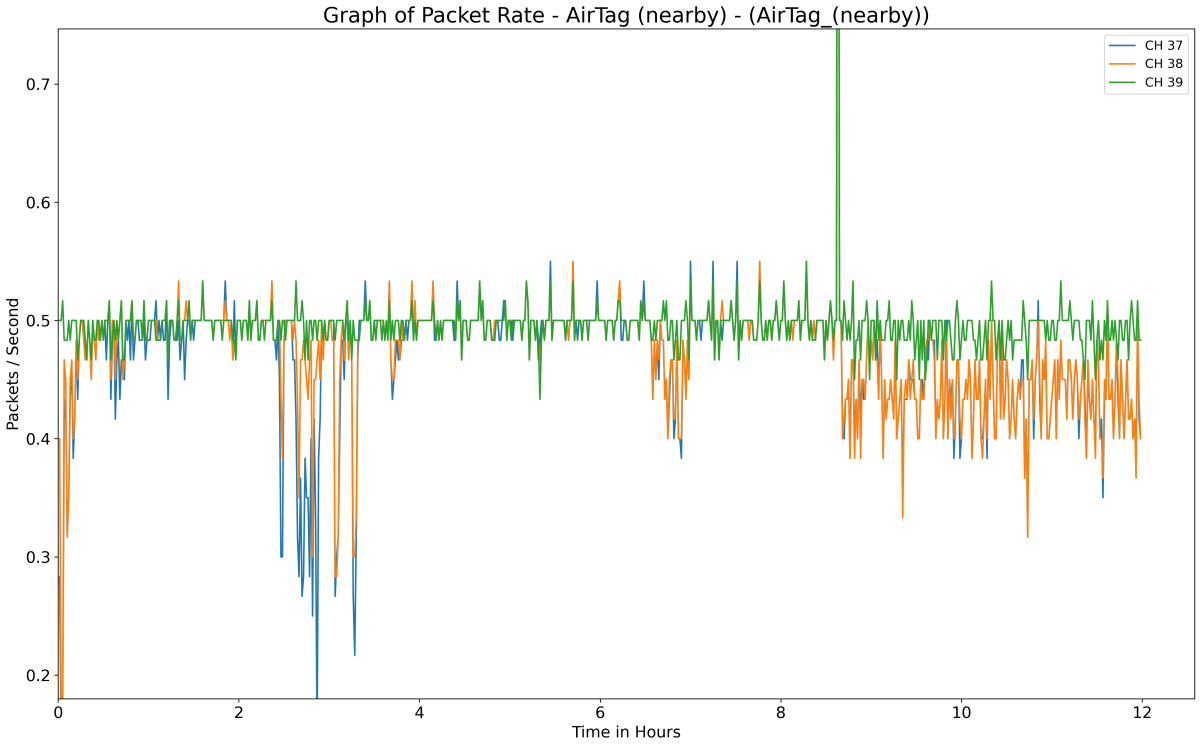

Figure 6.27: Graph of Packet Rate - AirTag (nearby)

The packet rate in the ”nearby” state of the AirTag shows much more variance than in the ”lost” state and also peaks beyond the scale of the plot. This most likely comes from inaccuracies during the automatic labeling process.

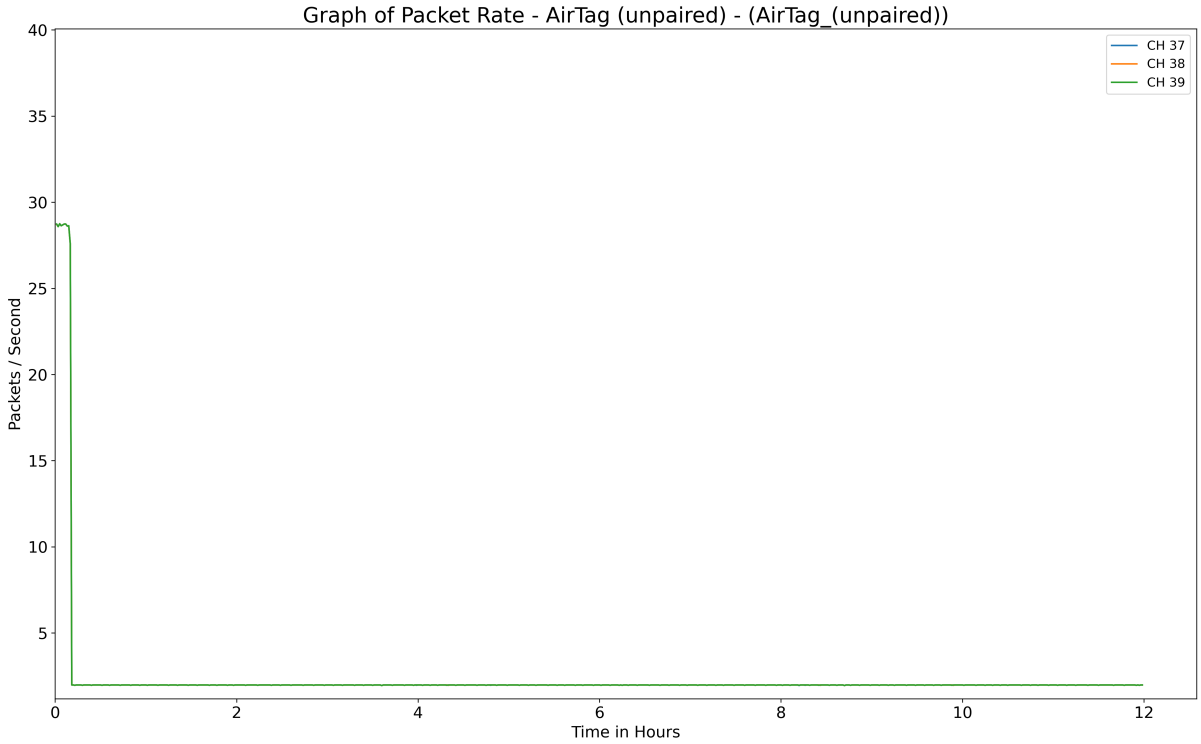

Figure 6.28: Graph of Packet Rate - AirTag (unpaired)

This is undoubtedly by far the most interesting plot of this thesis. As shown before, the AirTag, in its ”unpaired” state, has significant outliers in its packet rate. These occur all at the very beginning of the life cycle of an AirTag. When the security latch of the AirTag is pulled, and the AirTag is started for the first time, it enters a unique state where the packet rate is significantly higher than at any other point in time. The AirTag leaves this state of high packet rate after precisely 600 seconds. This higher packet rate most likely facilitates a faster pairing process.

So far, this state of AirTags is undocumented in the literature. Therefore, I will name this state ”**unboxed**”, as it occurs directly after unboxing an AirTag.

6.2.3 SmartTag

The following subsection contains some interesting plots for Samsung's SmartTag tracker. Not all plots are included; some are omitted for brevity.

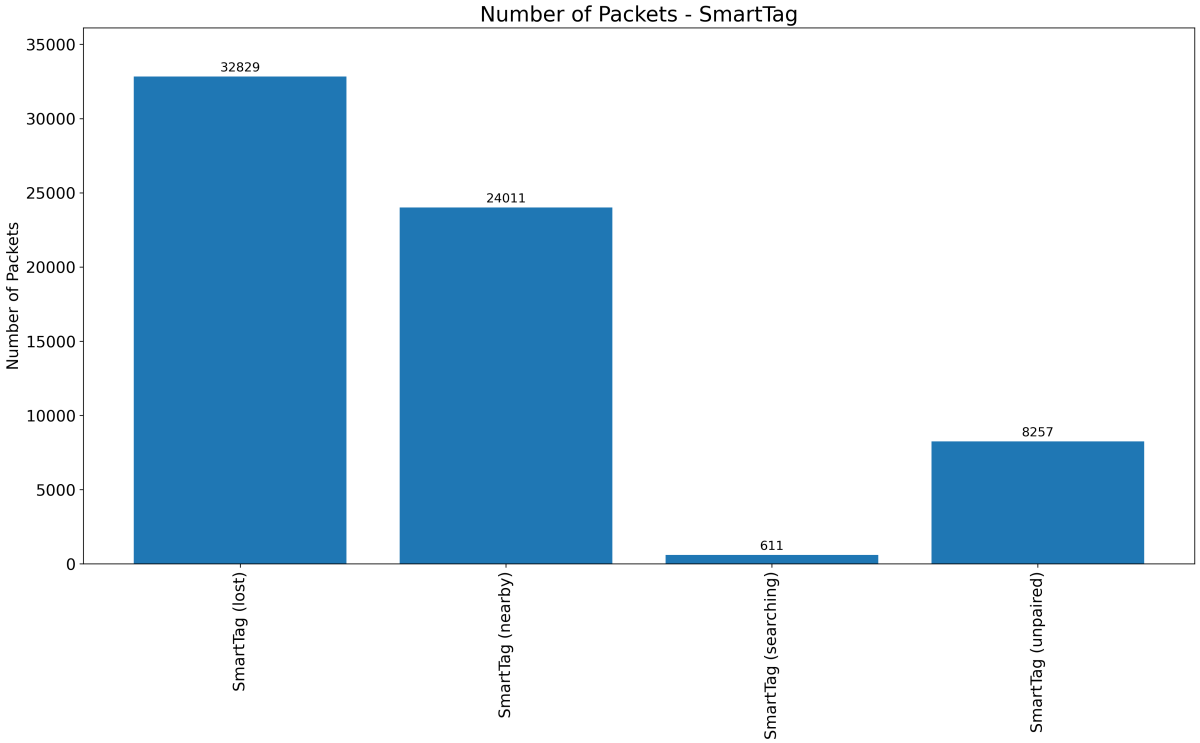

Figure 6.29: Number of Packets - SmartTag

It is evident that both the "searching" state and the "unpaired" state suffer from few packets. This is due to the very short capture time for both states.

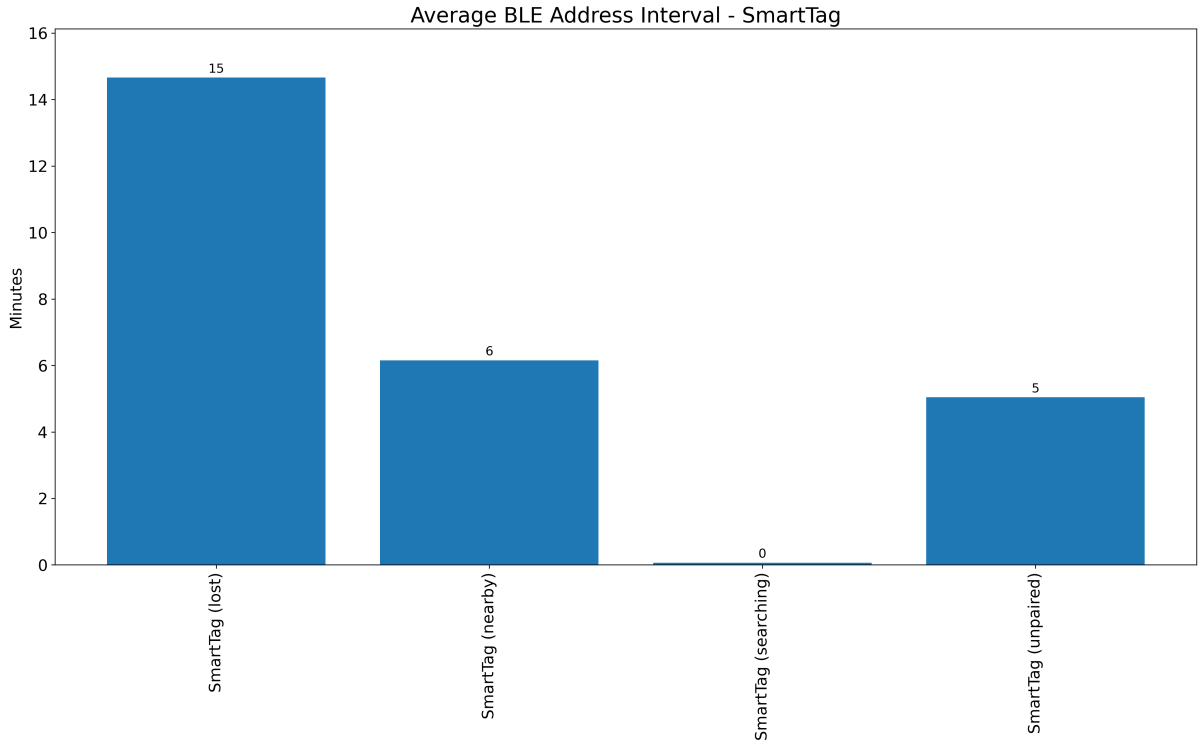

Figure 6.30: Average BLE Address Interval - SmartTag

The BLE address interval in the ”nearby” and ”unpaired” states is relatively short but still sufficient for packet rate modeling. Due to the very short capture time for the ”searching” state, the BLE address interval is equal to the total capture time. This state is not suitable for packet rate modeling.

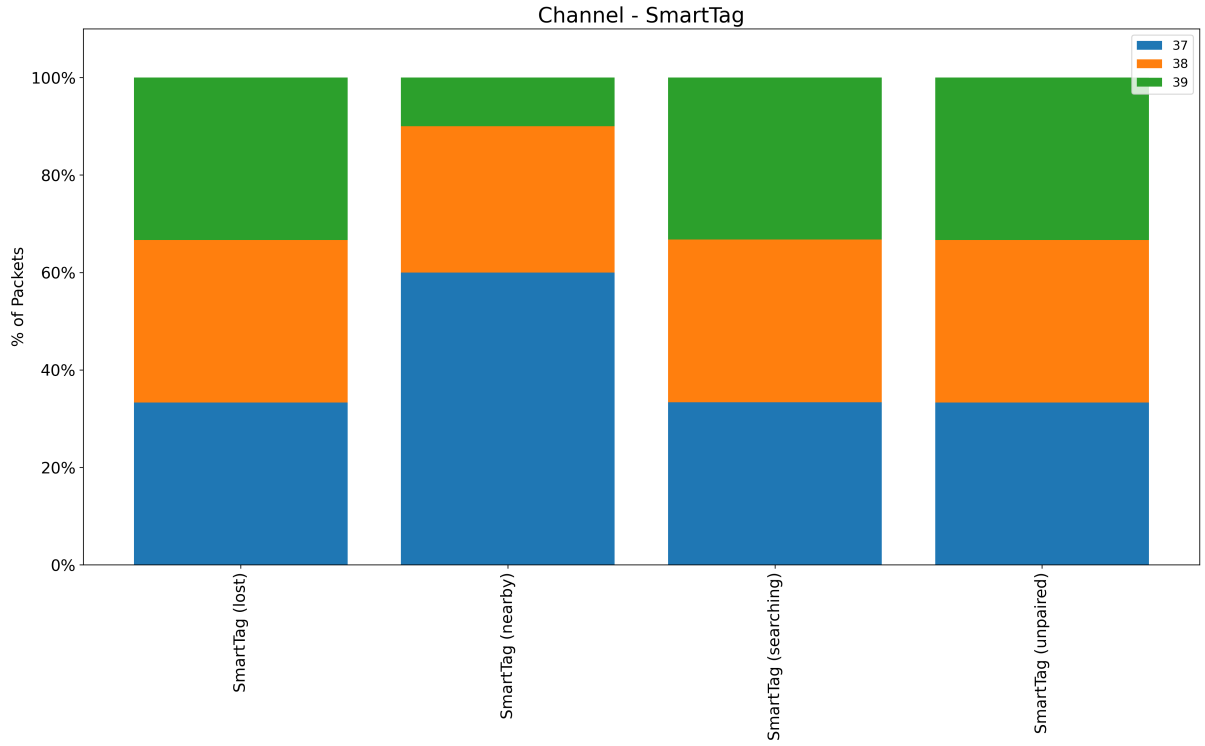

Figure 6.31: Channel - SmartTag

The SmartTag shows peculiar channel usage in the ”nearby” state. The lower the channel number, the higher the relative usage. This also hints at a higher packet rate for lower channels. In all other states, the distribution of packets among channels is roughly uniform.

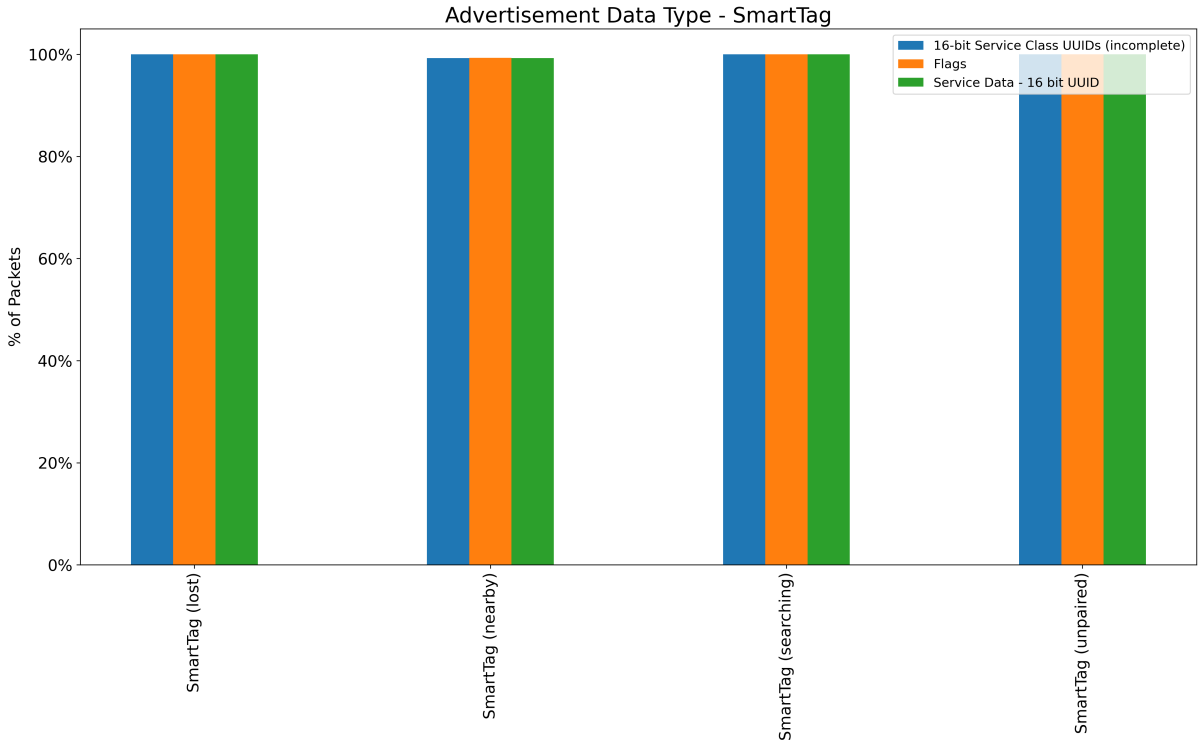

Figure 6.32: Advertisement Data Type - SmartTag

The SmartTag uses Service Data, Service Class UUIDs, and flags as advertisement data types. Therefore, it also carries UUIDs (two from Samsung per packet) and a Service Data payload. The general packet structure is the same across all states. The packets of the SmartTag are, therefore, hard to distinguish.

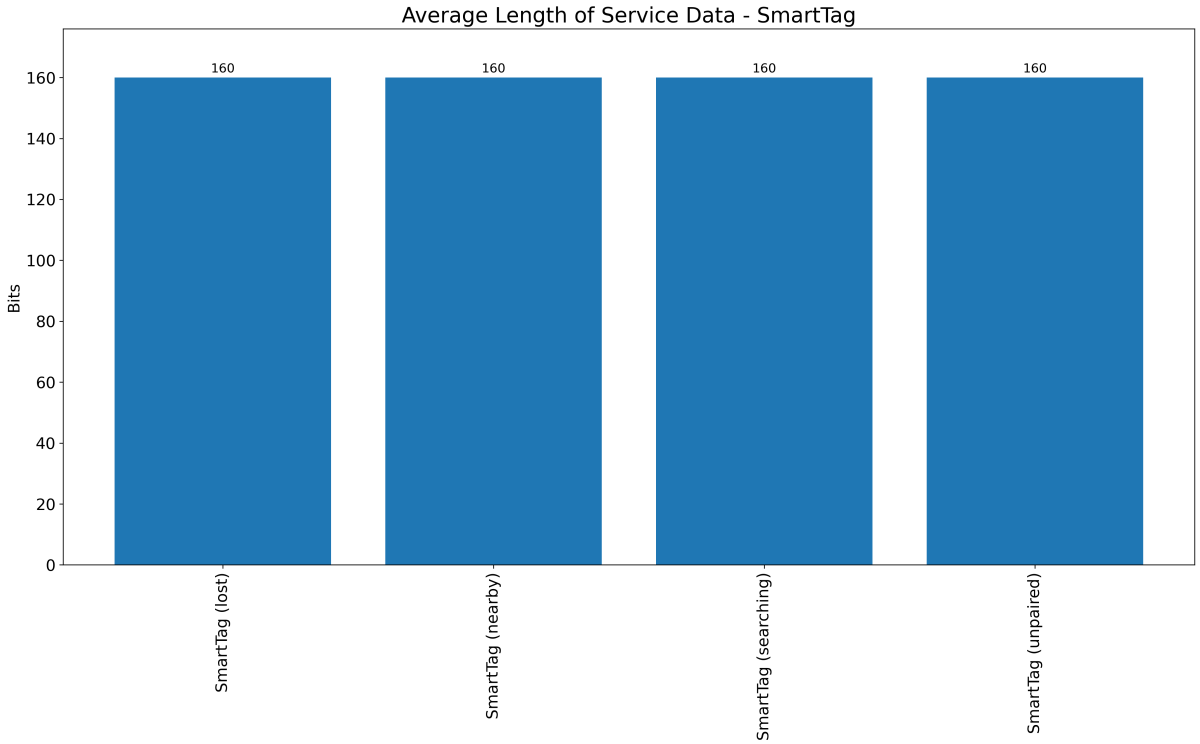

Figure 6.33: Average Length of Service Data - SmartTag

A look at the length of the Service Data further shows how hard the states are to distinguish from each other; unlike the Find My trackers, the SmartTag uses an identical length of 160 bits for its data across all states, making them hard to separate.

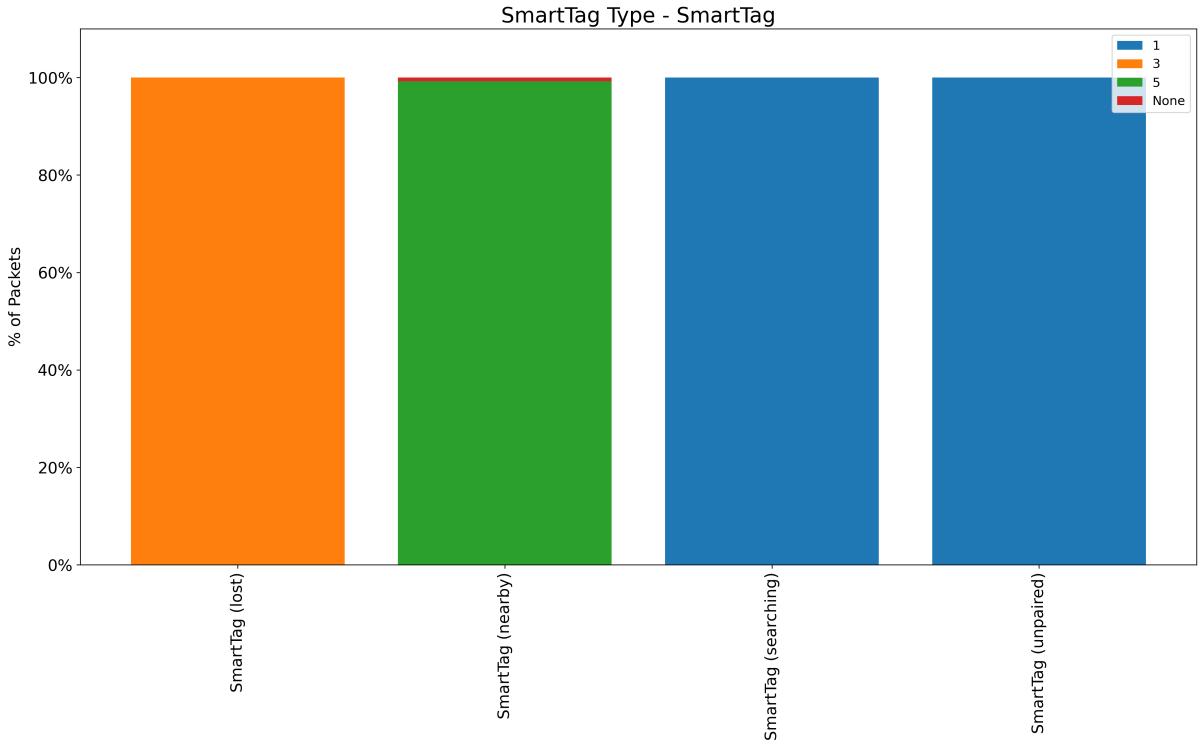

Figure 6.34: SmartTag Type - SmartTag

The only noticeable difference between the SmartTag's states are the status bits in the Service Data. The values differ for all states; only the "unpaired" and "searching" states share the same status bits. The small percentage of None type in the "nearby" state stems from inaccuracies during automatic labeling.

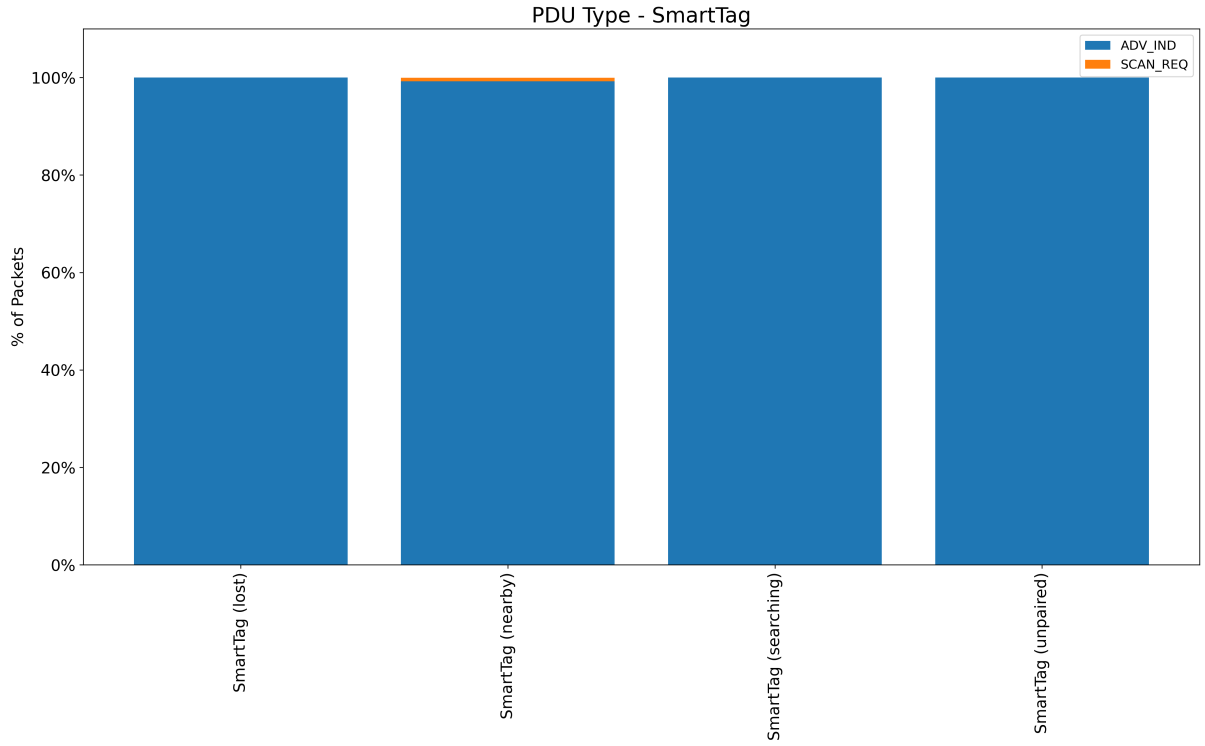

Figure 6.35: PDU Type - SmartTag

The PDU type is the same across all states and also the same as for the Find My trackers. So far, the PDU type is not a particularly interesting feature. The small percentage of SCAN_REQ packets in the ”nearby” state stem from incorrect automatic labeling and should be ignored.

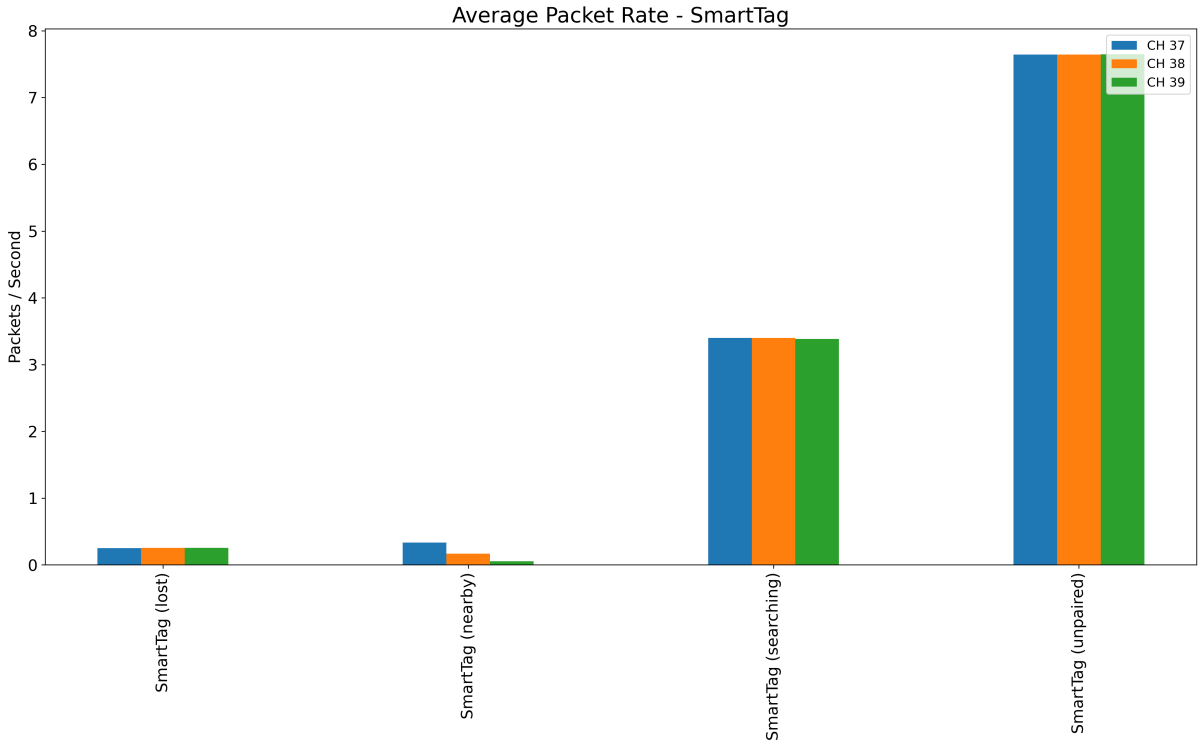

Figure 6.36: Average Packet Rate - SmartTag

Other than the SmartTag type, the packet rate is, in fact, the only distinguishing factor between the various SmartTag states. The packet rate differs significantly between all states. In the ”nearby” state, the packet rate even differs between channels. In all other states, the packet rate is the same across all channels. Due to the short capture time, the packet rate for the ”searching” and ”unpaired” states should be considered with some skepticism.

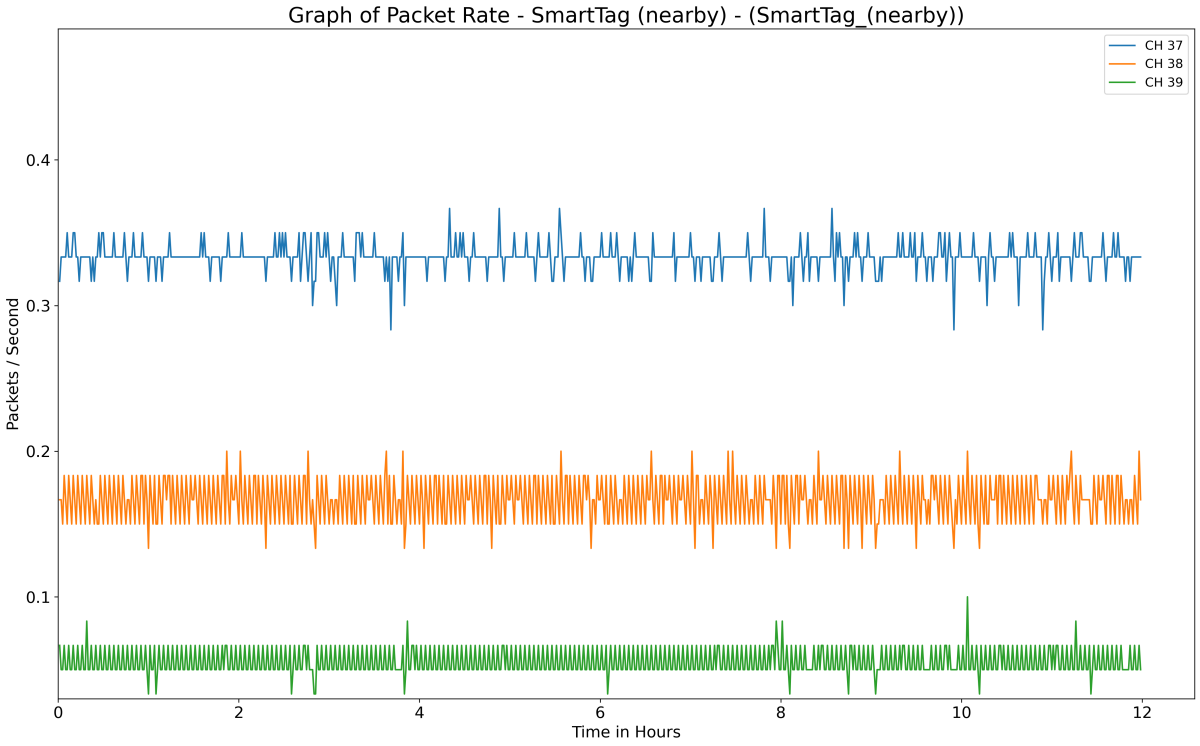

Figure 6.37: Graph of Packet Rate - SmartTag (nearby)

The graph of the packet rate for the ”nearby” state reveals that the packet rate differs between each channel over the entire captured time. This also correlates with the distribution of packets among the channels. The higher the packet rate, the higher the relative share of packets. However, as of writing this thesis, I am unaware of an explanation for this behavior. Based on my understanding of the BLE specification, however, this is a valid behavior, as devices can advertise on **up to** three channels and are **not required** to advertise on all three channels [16].

6.2.4 Tile

The following subsection contains some interesting plots for Tile's Mate tracker. Not all plots are included; some are omitted for brevity.

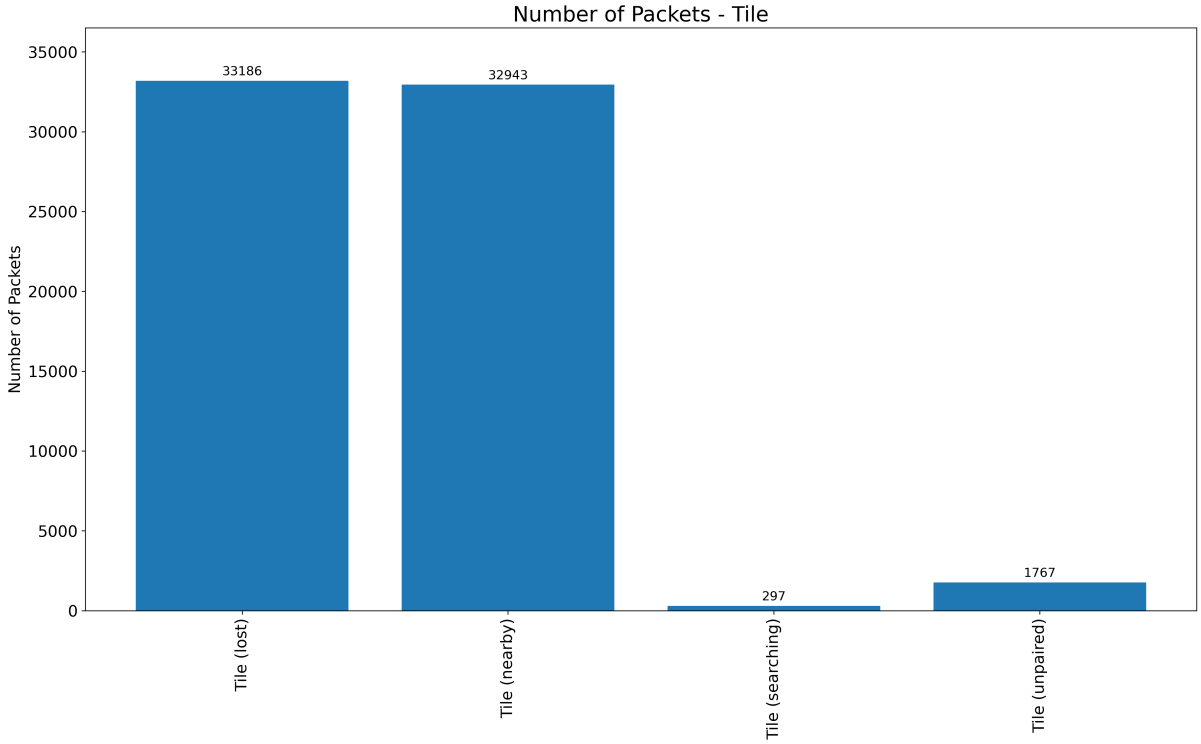

Figure 6.38: Number of Packets - Tile

Due to the short capture time, the number of packets in the "searching" state and the "unpaired" state is very small. Those two states are certainly not usable for modeling, let alone modeling of any packet rate.

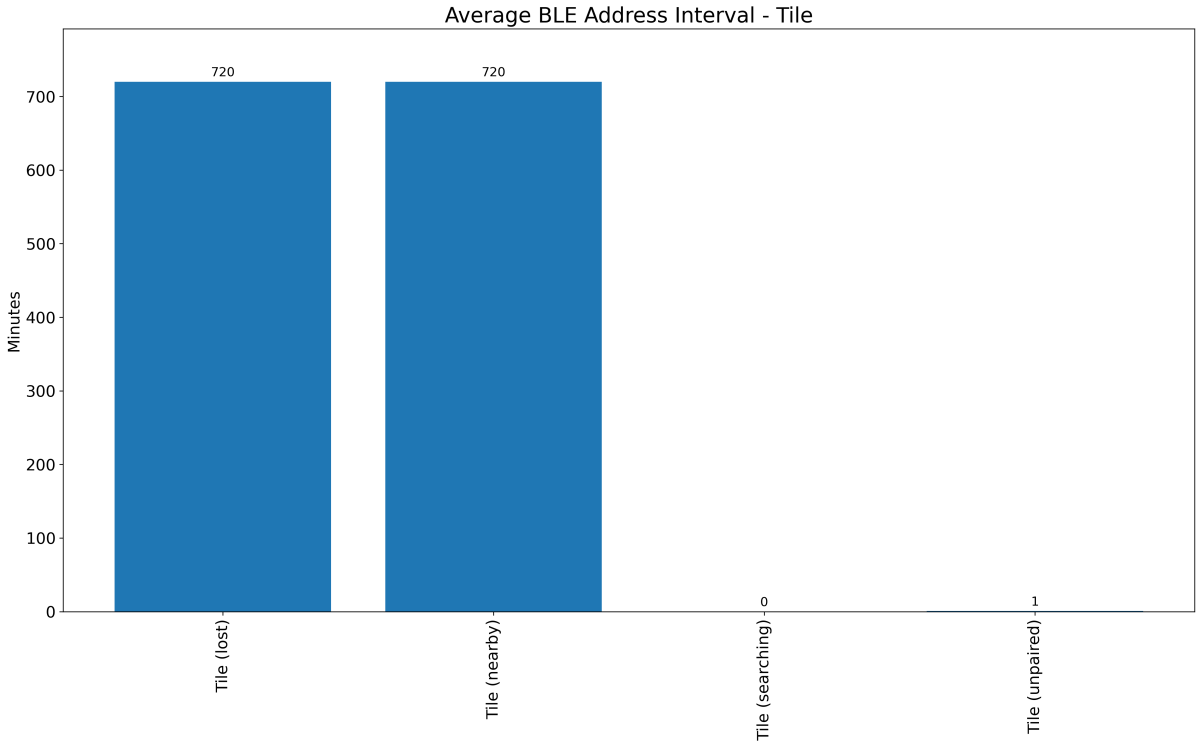

Figure 6.39: Average BLE Address Interval - Tile

The average BLE address interval in the ”lost” and ”nearby” state is equivalent to the capture time of 12 hours. In other words, the source address never changes; it is static. A closer visual inspection of all captured files reveals that the Tile tracker never changes its address. Therefore, the Tile tracker in its ”lost” and ”nearby” states is highly suitable for packet rate modeling.

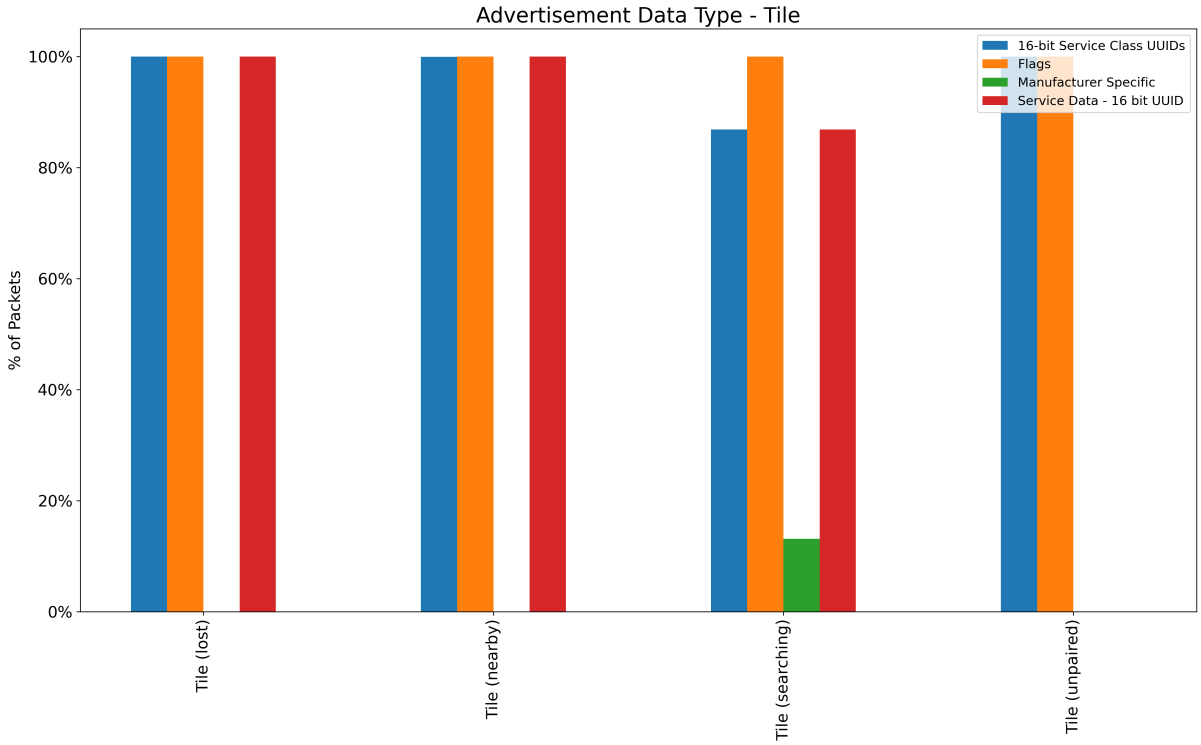

Figure 6.40: Advertisement Data Type - Tile

The usage of advertisement data types indicates two things. First, the structure of the packets in the ”lost” and ”nearby” state is identical. Both use Service Data, flags, and Service Class UUIDs (and therefore have two UUIDs from Tile per packet). Any further inspection of these two states will reveal that the packets are perfectly identical. It seems as if the Tile tracker does not have separate ”lost” and ”nearby” states.

In the ”unpaired” state, the Service Data is omitted. In the ”searching” state, the Tile uses two different types of packets. One carries flags and manufacturer specific data, and the other carries flags, Service Data, and Service Class UUIDs. At this point, it might be interesting to explore the searching state a little further.

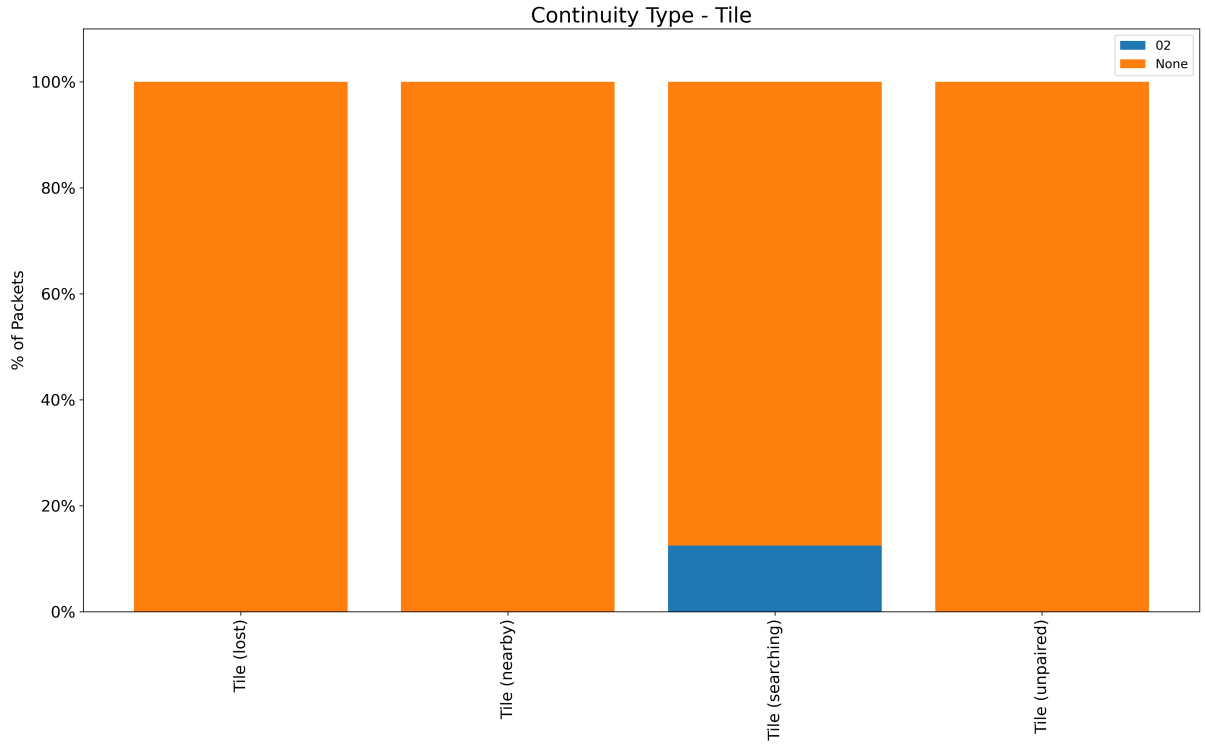

Figure 6.41: Continuity Type - Tile

In the ”searching” state, some of the packets contain manufacturer specific data with a company ID from Apple. Hence, the packets are part of Apple’s continuity protocol. In this case, these packets carry the prefix 0x02. This continuity type seems to be undocumented at the point of writing this thesis [12]. Additionally, as seen before, in all other states, manufacturer specific data is not used. Hence, the continuity type is always none.

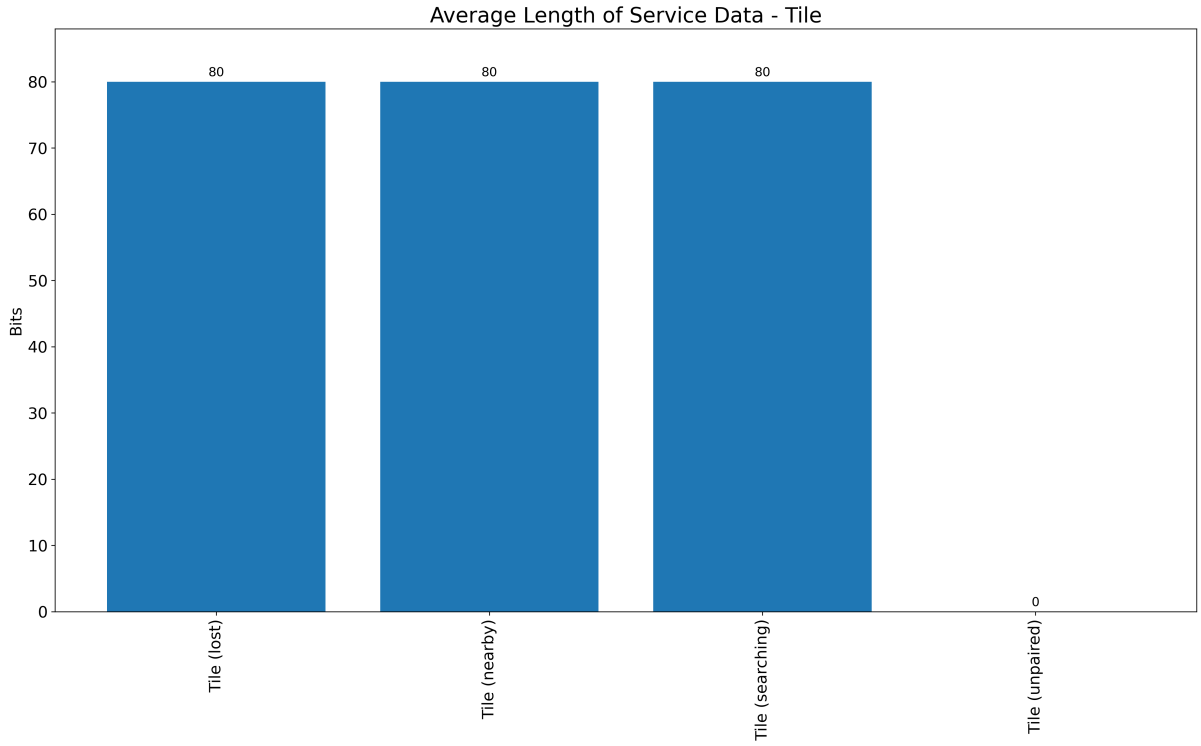

Figure 6.42: Average Length of Service Data - Tile

A look at the average length of the Service Data shows two things. First, in the ”unpaired” state, the length is zero, as expected, because Service Data is not used. Second, the length of the Service Data is 80 bits in the ”lost” and ”nearby” state. This shows, again, that these two states are identical. However, the 80 bits are also different in length from the SmartTag, making this a potentially interesting feature for classification.

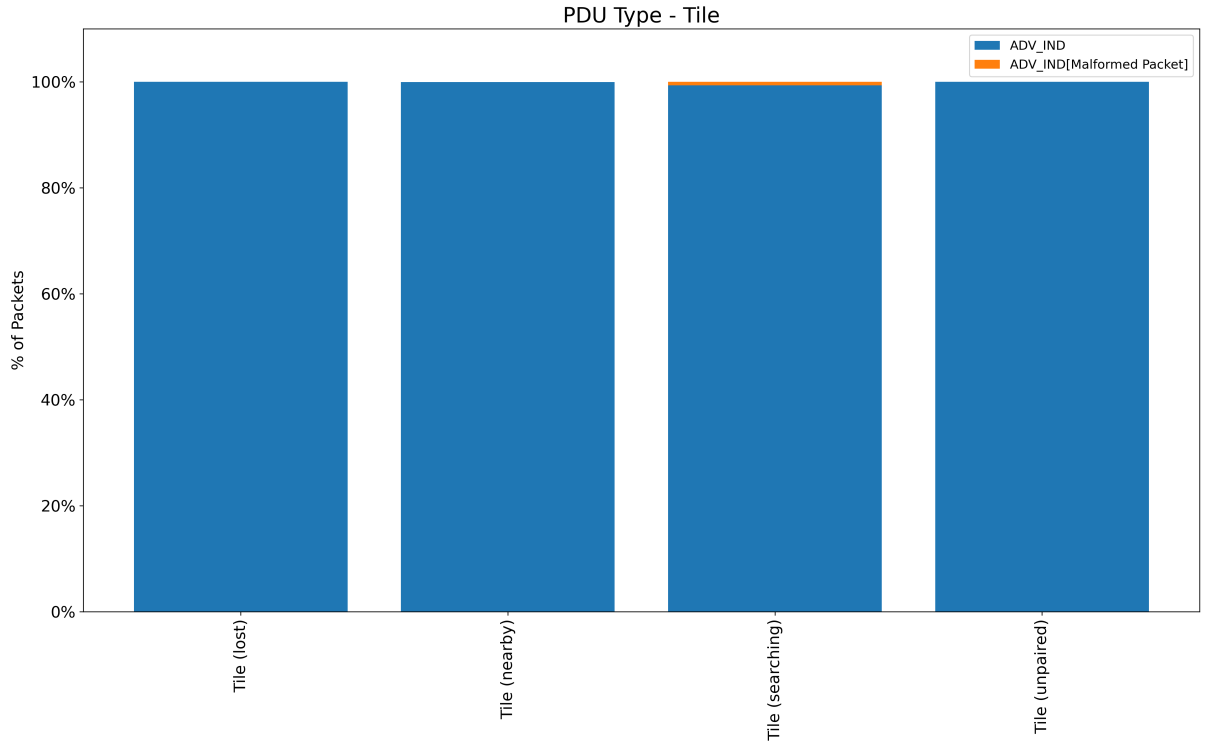

Figure 6.43: PDU Type - Tile

The PDU type for the Tile is again ADV_IND in all states, as with all previous trackers. Generally speaking, the PDU type is not a relevant feature for distinguishing conventional BLE trackers.

6.2.5 iDevices

The following subsections will cover the analysis of the various Apple devices. The iPhone's analysis is a little more extensive. The analysis for the other iDevices is kept as short as possible, as the packets transmitted by these devices are mostly identical.

6.2.5.1 iPhone

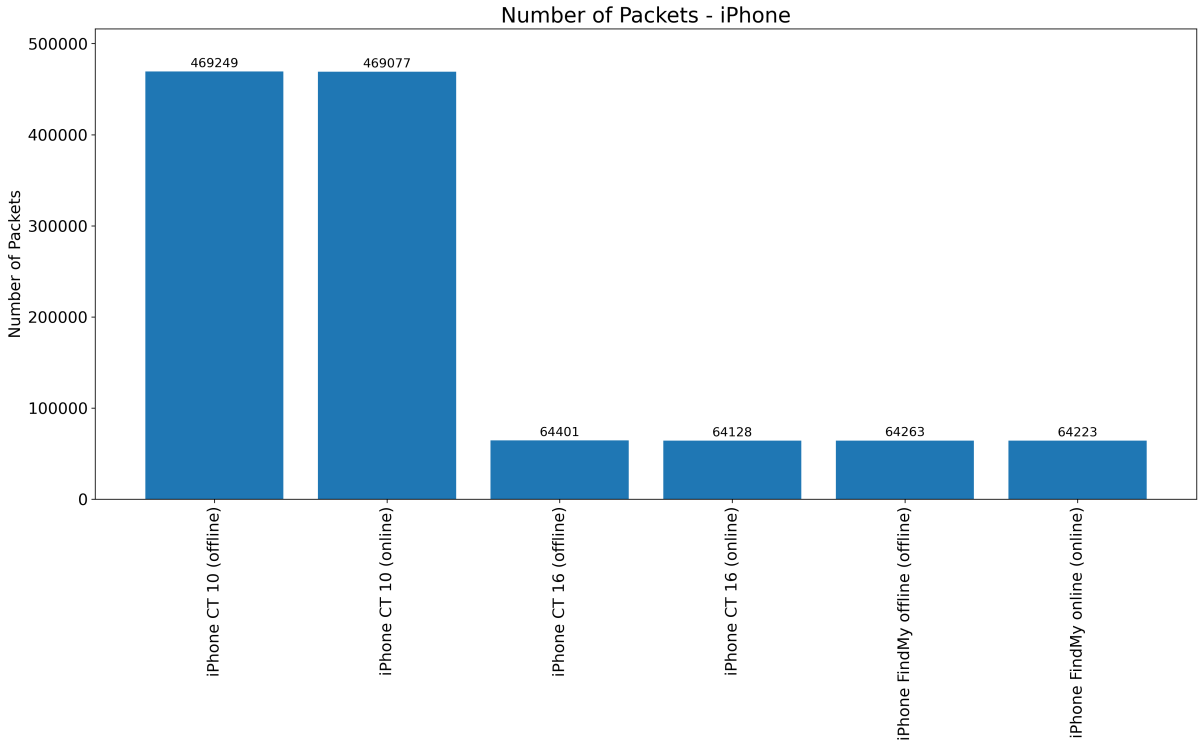

Figure 6.44: Number of Packets - iPhone

The number of packets varies greatly from packet type to packet type. However, it is not state-dependent. Consequently, the same thing applies to the overall packet rate. Interestingly, there is also no difference in the types of packets sent by state. In both states, the same packets are transmitted. Given the vast amount of packets captured, all packet types in all states are highly suitable for modeling.

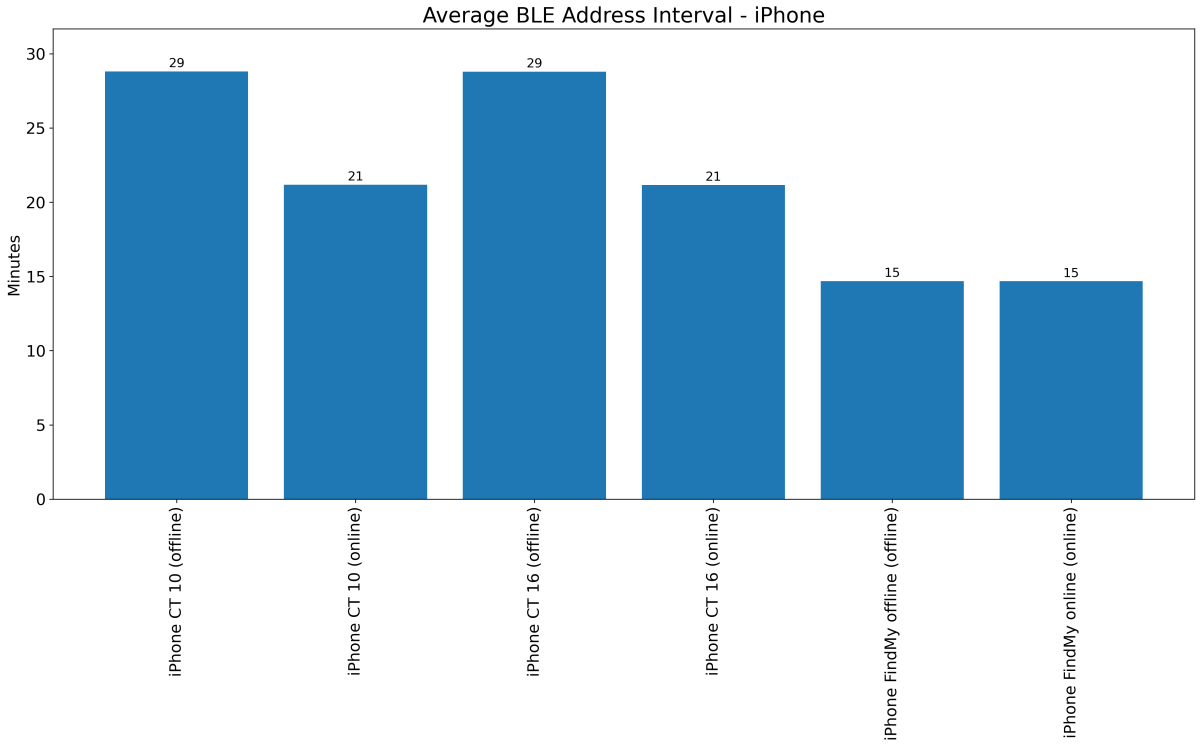

Figure 6.45: Average BLE Address Interval - iPhone

Interestingly, the source address interval varies based on the device state and the packet type. For all packet types and states, the interval is long enough for modeling the packet rate.

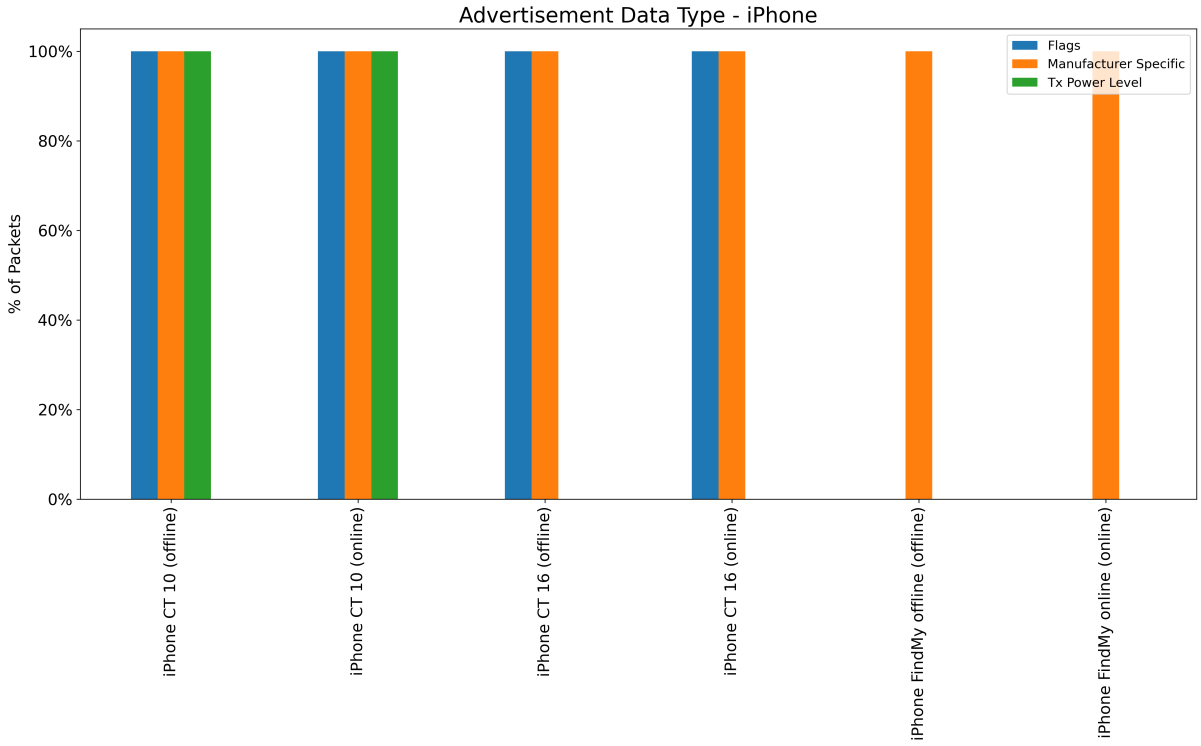

Figure 6.46: Advertisement Data Type - iPhone

The usage of advertisement data types varies greatly from packet type to packet type. However, it is state-independent. Unsurprisingly, all packets use manufacturer specific data as they are part of the Apple continuity protocol stack. The continuity type 0x10 packets additionally use flags and a TX power level, while the 0x16 packets use only flags in addition. Overall, the packet structure is very different among the various continuity packet types.

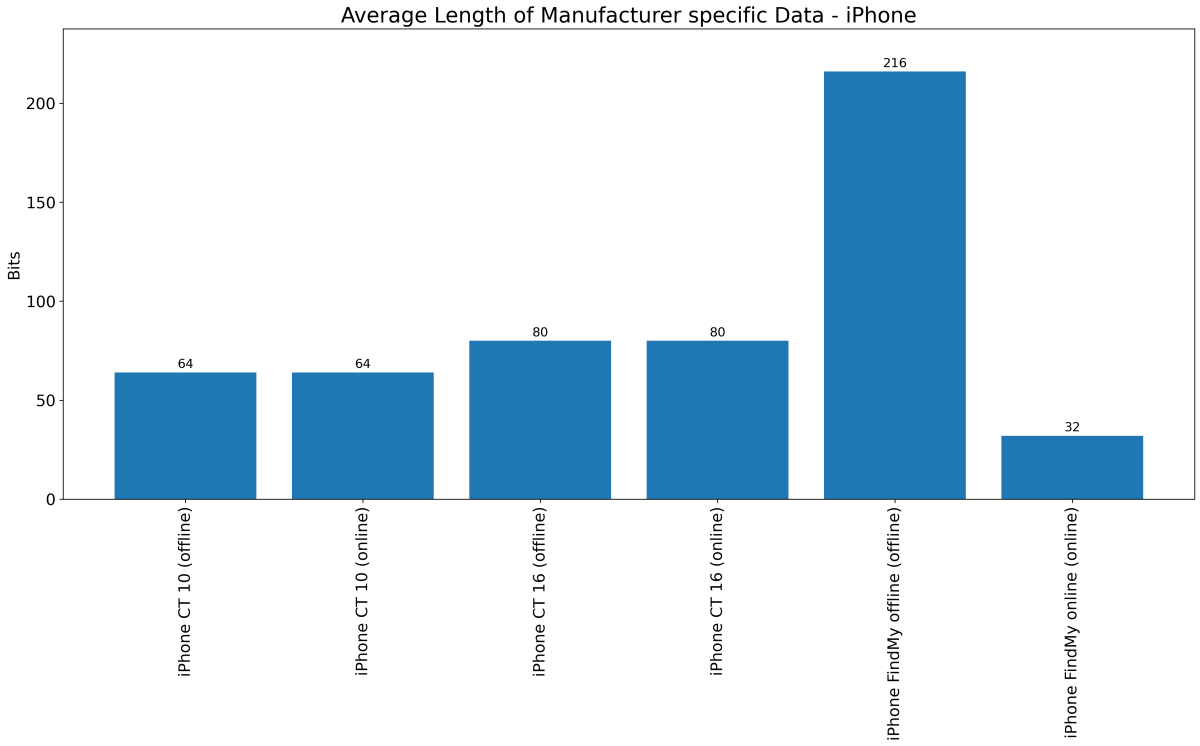

Figure 6.47: Average Length of manufacturer specific Data - iPhone

Interestingly, the length of the manufacturer specific data varies from packet type to packet type. In the case of the 0x10 and 0x16 packets, it is state-independent. In the case of the 0x12 Find My packets, the length is state-dependent. This behavior is identical to the behavior shown by the AirTag. In the case of the Find My online packet, the public key does not have to be transmitted; hence, the manufacturer specific data is shorter. The overall lengths, however, are identical to those of the AirTag.

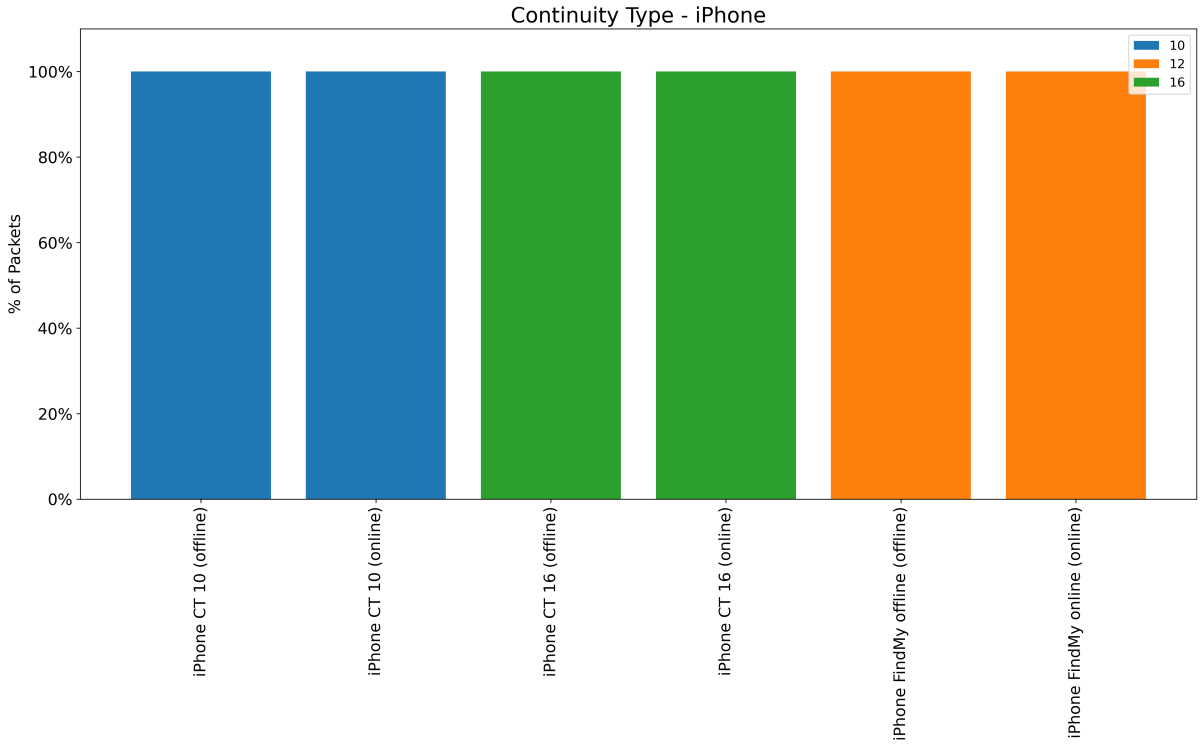

Figure 6.48: Continuity Type - iPhone

Obviously, the packets' continuity types align with their labels. Therefore, the continuity type is suitable for separating all continuity-type packets independent of any other features. However, it is not possible to differentiate the device states "online" and "offline" with the continuity type.

An important note here is that this plot verifies that the packets of continuity type 0x16 actually have type 0x16. As of the writing of this thesis, this continuity type is entirely unknown. Furthermore, testing with various iPhones sometimes resulted in the complete lack of this packet type. Based on my testing, this continuity type was introduced with the software update iOS 17 in 2023. The iPhones tested all transmitted this continuity type only after updating to iOS 17.

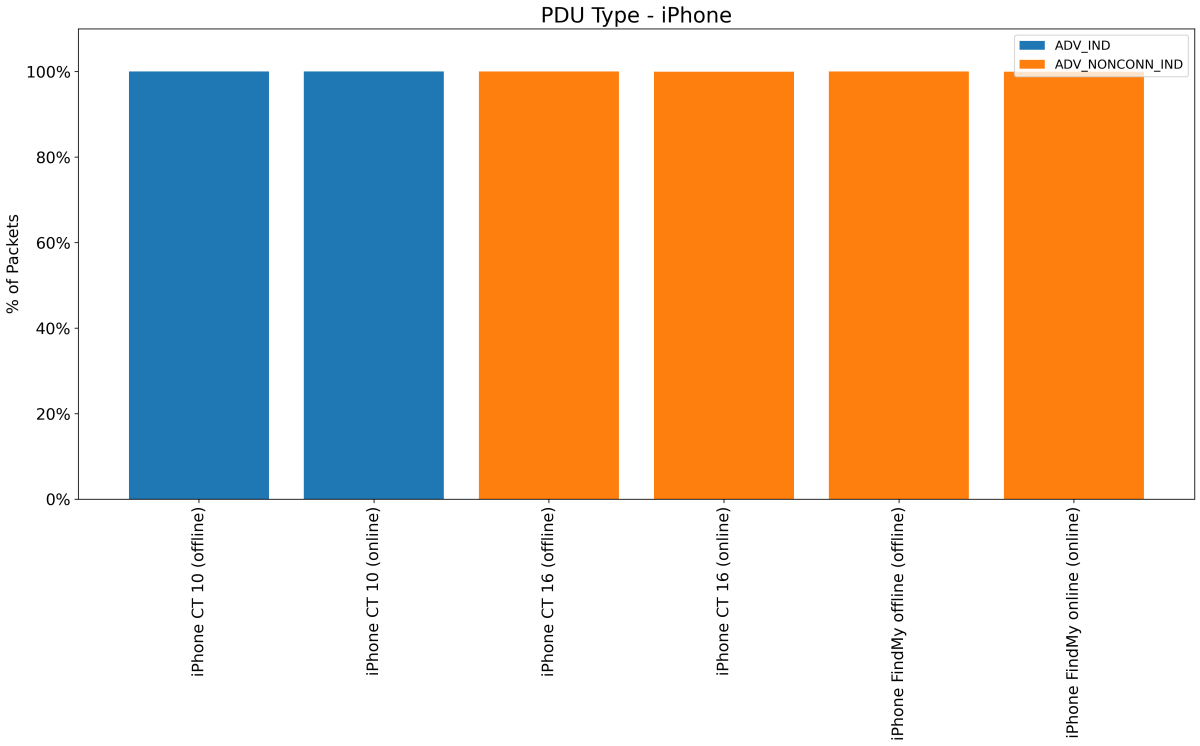

Figure 6.49: PDU Type - iPhone

The PDU type is a very interesting feature in the case of the iPhone (and Apple devices in general). The PDU type varies among the different continuity-type packets. Most importantly, the PDU type of both Find My packets is ADV_NONCONN_IND and, therefore, different from the PDU type ADV_IND used by the Find My trackers. This is the only feature that allows for differentiation between the Find My packets from a Find My tracker and the Find My packets of other Apple devices (iDevices).

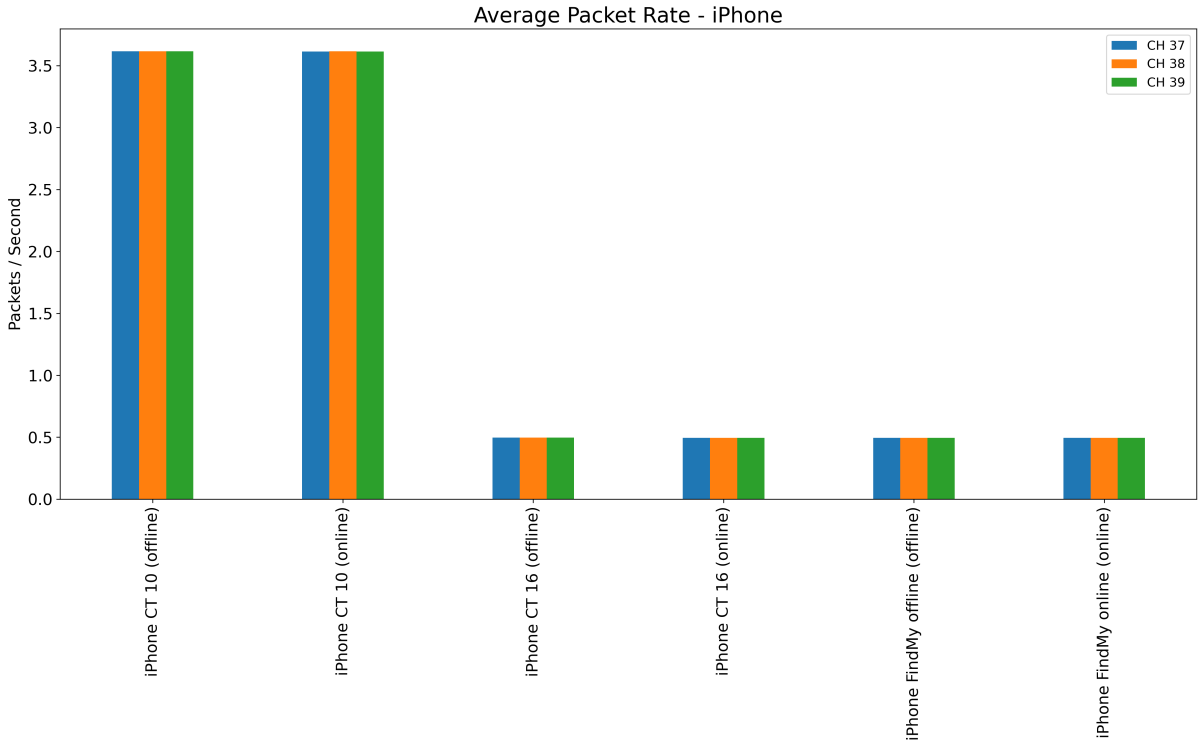

Figure 6.50: Average Packet Rate - iPhone

The average packet rate is again state-independent. However, it varies between packet types, as implied by the different number of packets captured. It is certainly possible to attempt to use the packet rate as a differentiating factor among continuity-type packets. However, the marginal success might be underwhelming, given that other features can already separate these packets.

6.2.5.2 iPad

The iPad is a device very similar to the iPhone. Therefore, there is little point in discussing the iPad to any extent. All packets used by the iPad are identical to the ones used by the iPhone. The only interesting plot is the one covering the number of packets captured.

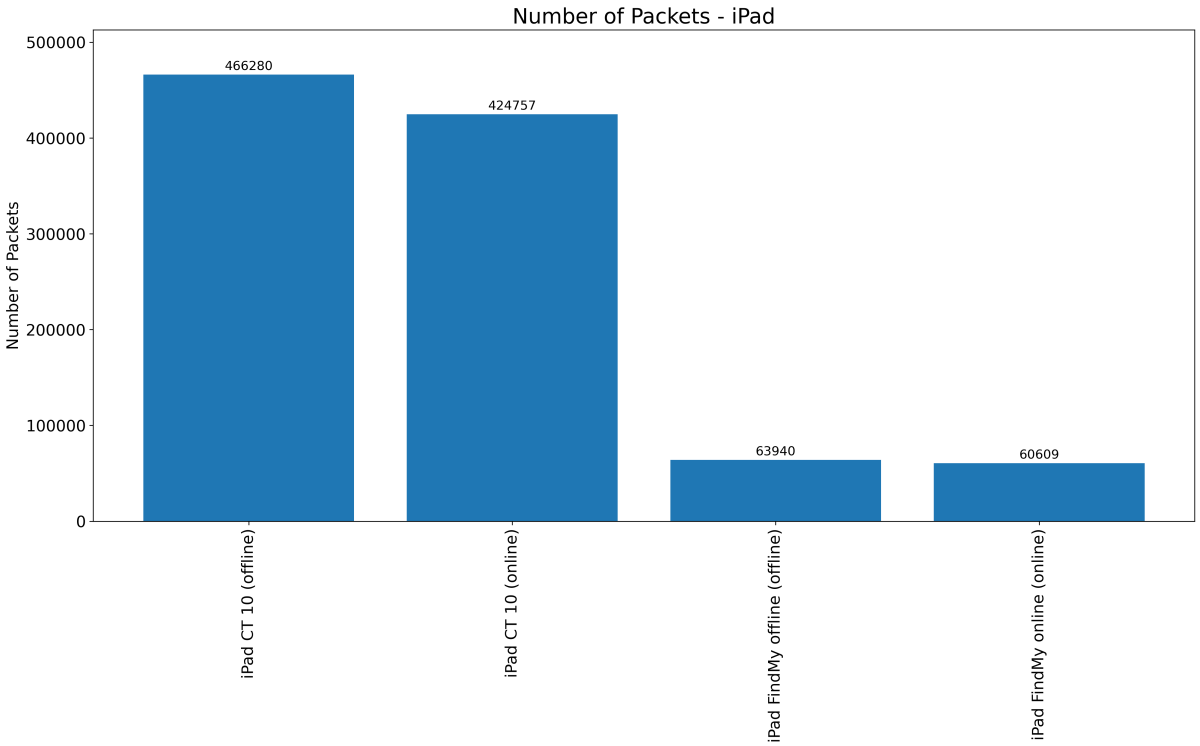

Figure 6.51: Number of Packets - iPad

The overall plot looks very similar to the iPhone in terms of the number of packets captured. All packet types have extensive data and are suitable for modeling. What is not visible in the plot is much more interesting. The iPad does not use packets of continuity type 0x16, even when updated to iOS 17. The reason for this is unknown to me.

6.2.5.3 MacBook

As for the iPad, the MacBook is not different from the iPhone in any meaningful way. Therefore, only two plots are discussed.

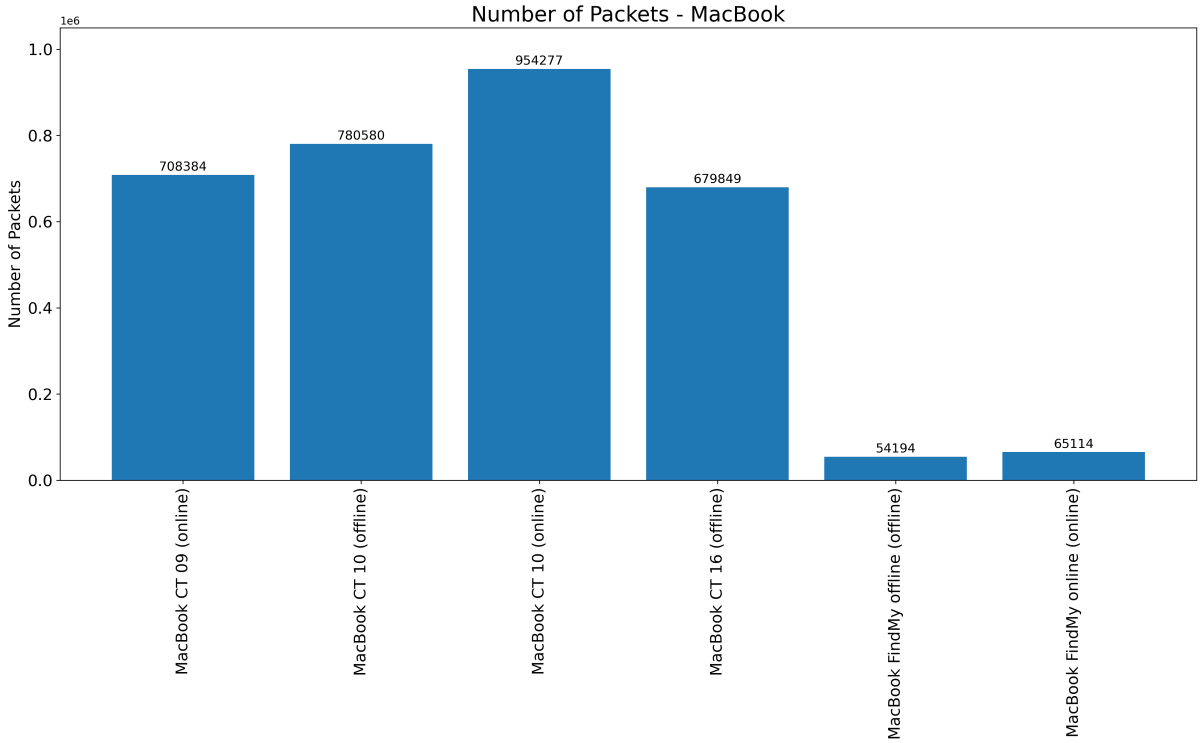

Figure 6.52: Number of Packets - MacBook

What becomes apparent at first sight is that the number of packets transmitted for all packets other than the Find My packets is significantly higher than in the case of the iPhone. Overall, the MacBook transmits many more packets; hence, the packet rate must be higher. Additionally, it should be apparent that the types of packets sent are state-dependent.

The 0x09 packets are only used when online, and the 0x16 packets are only used when offline. In the case of the 0x09 packets, this is explainable. These are packets used for Airplay, which is used for WiFi-based streaming of music and video [12]. When the MacBook is connected to WiFi, it is also online. Hence, these packets only appear in the ”online” state.

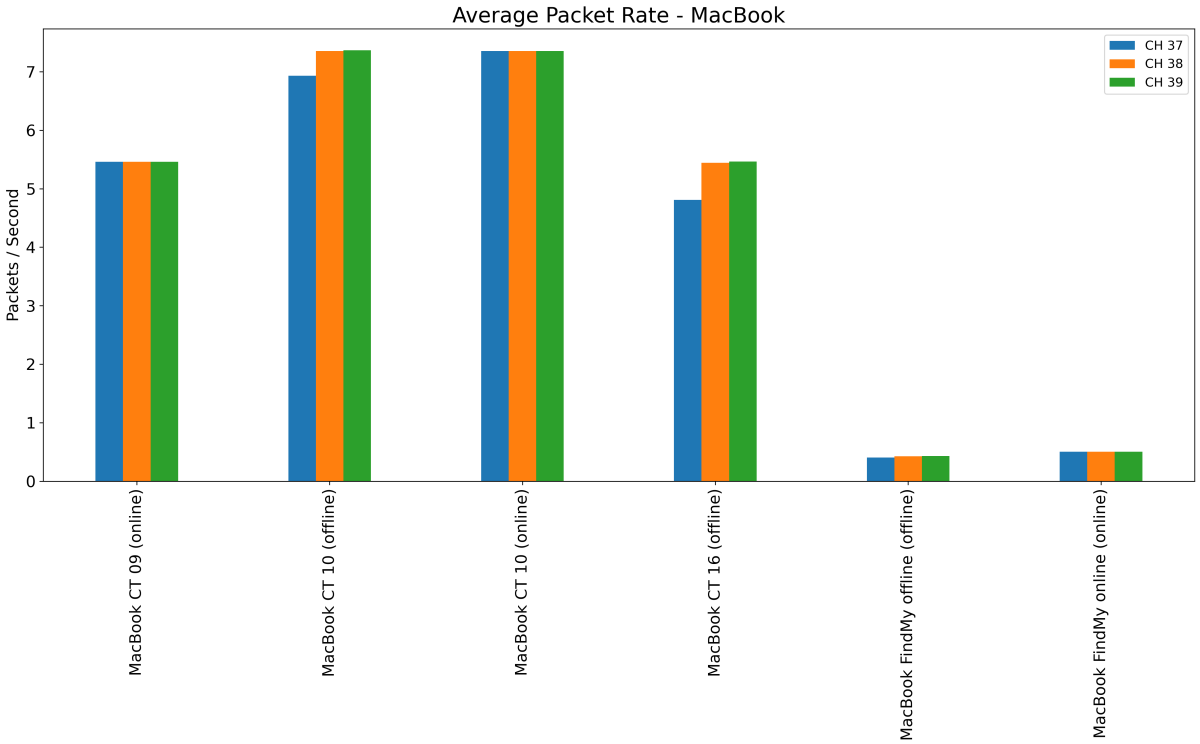

Figure 6.53: Average Packet Rate - MacBook

As implied by the previous plot, the packet rate differs greatly between the continuity types and is significantly higher than that of the iPhone or iPad. This would make it possible to distinguish between packets from MacBooks and packets from other iDevices.

6.2.5.4 AirPod

The AirPod is a particularly interesting device as it is a hybrid between an AirTag and other Apple devices such as iPhones, as the analysis of its packets will reveal.

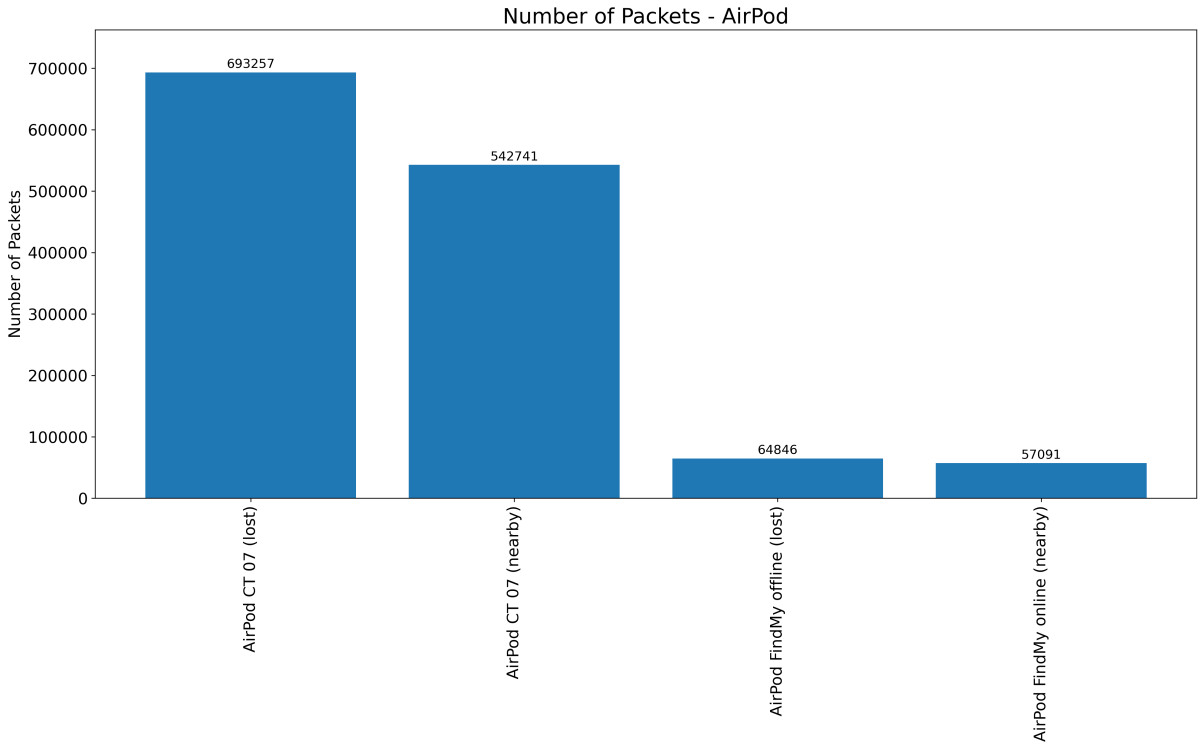

Figure 6.54: Number of Packets - AirPod

First and foremost, the number of packets for all packet types is sufficient for modeling. However, much more interesting is the fact that the AirPod uses both Find My packets and packets of another continuity type, in this case 0x07 (proximity pairing). This makes the AirPod relatively similar to the other Apple devices (iDevices). On the other hand, the 0x07 continuity type itself makes the AirPod similar to an AirTag, as only the AirTag makes use of continuity type 0x07 (in its ”unpaired” state). Other Apple devices never use the continuity type 0x07.

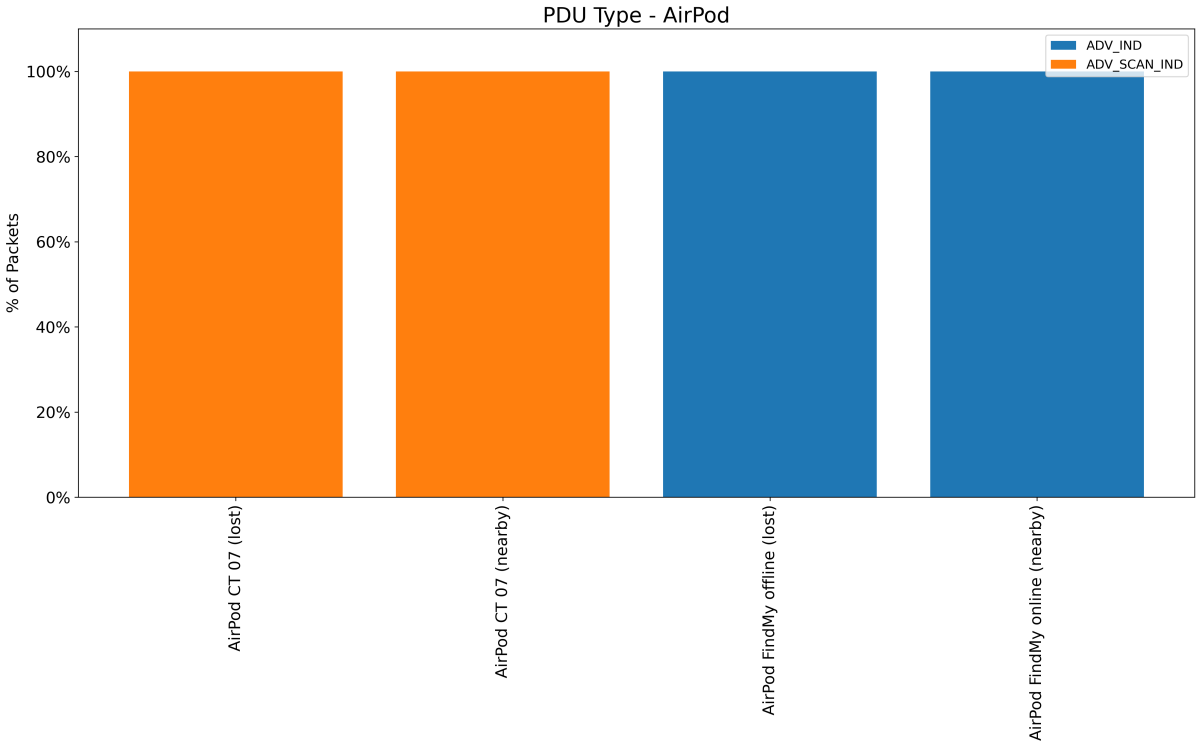

Figure 6.55: PDU Type - AirPod

When looking at the PDU types used by the AirPod, it becomes clear that it is, in fact, a hybrid between an AirTag and an iDevice. For the Find My packets, the AirPod uses the ADV_IND PDU type, like the AirTag but unlike an iDevice. For the continuity type 0x07 packets, the AirTag uses the ADV_SCAN_IND PDU type, unlike the AirTag, which uses the ADV_IND PDU type for its 0x07 packet, making it more similar to other Apple devices.

On the one hand, the AirPod can, therefore, be considered a Find My tracker, and on the other hand, it is obviously not a traditional tracker and can, therefore, also be considered an iDevice. This distinction will become necessary during modeling. It is not possible to put the AirPod strictly into one of these two categories. Its Find My packets are identical to the ones of an AirTag, but the fact that the AirPod simultaneously also transmits packets of another continuity type makes it more similar to other Apple devices, such as the iPhone.

6.2.6 Other Devices

For the other devices, i.e., the garbage class, the analysis is very brief as there is not much to see. The important aspect is that the packets, on average, are very different from anything seen so far.

With around 1.8 million packets captured and an average BLE address interval of 3 minutes, the other Device file is perfectly suitable for modeling, including the packet rate. The plots are omitted for brevity, as they are not especially pleasing to look at.

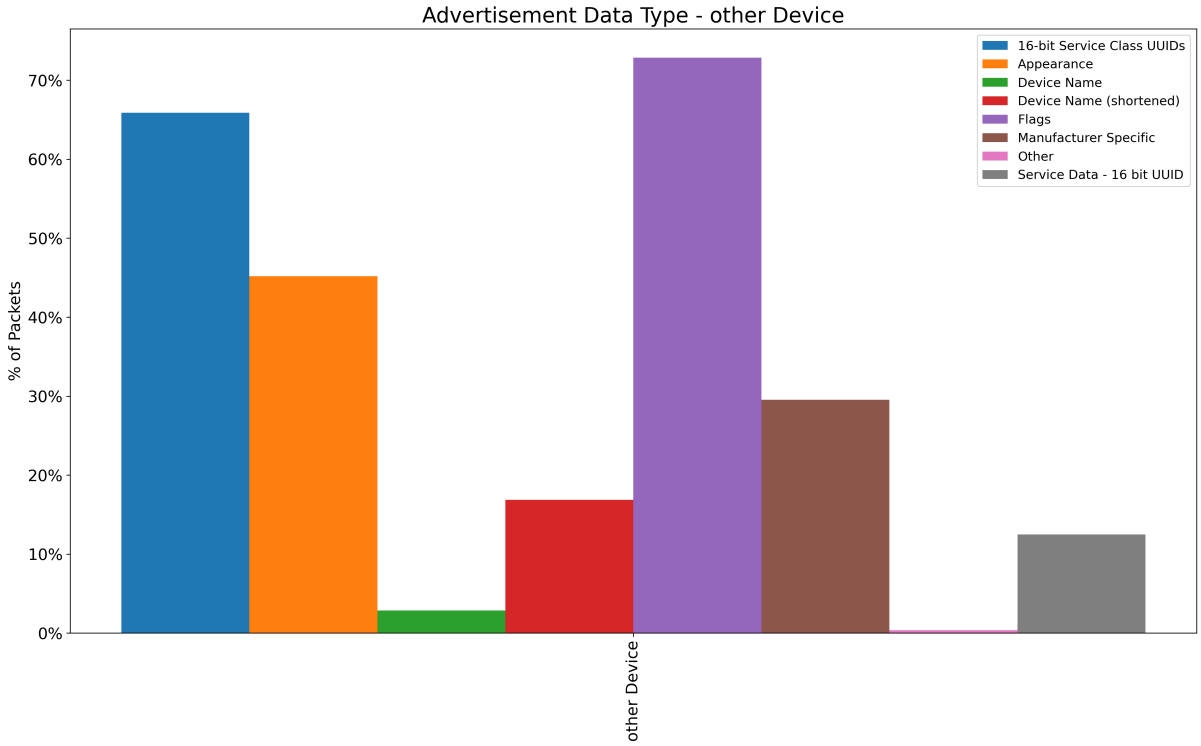

Figure 6.56: Advertisement Data Type - other Device

When looking at the advertisement data types used by the other devices, the large variation becomes apparent. Some of the advertisement data types previously seen are present. However, there are also many new ones. Solely based on the distribution of these advertisement data types, the other devices are very much distinguishable from all the previously analyzed tracking devices.

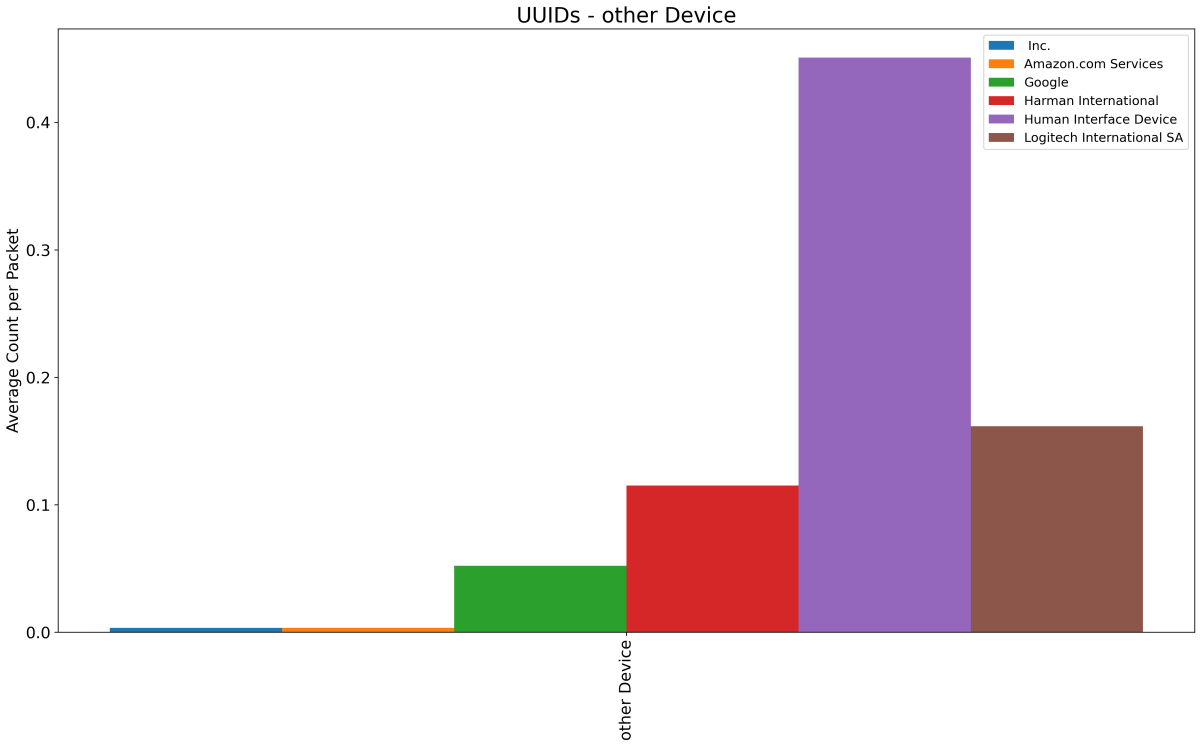

Figure 6.57: UUIDs - other Device

A very similar picture is painted by the plots for the UUIDs. There is a lot of variation, and the UUIDs used are noticeably different from the ones used by the tracking devices. There is little point in looking at many more plots; based on these two plots, it should already be relatively apparent that the other devices are, on average, very distinguishable from the tracking devices.

6.3 Insights for Modeling

The core insights gained during the analysis of the captured data for the modeling are the following:

- Some features are irrelevant for modeling, as they hardly differ between devices and states. These features include the percentage of malformed packets, the percentage of broadcast packets, and the protocol type. These features should not be considered for modeling.
- The three Find My trackers are very hard to distinguish. Their technical implementation is principally identical, except for the ”unpaired” state, where the implementation differs between the AirTag and the other two trackers. In conclusion, these three trackers should be put into the same class for modeling. However, a distinction between the states is perfectly possible.
- The datasets of the Chipolo and SkyTag in their ”unpaired” state are too short for more modeling. These two states should be discarded entirely for these two trackers.
- The Samsung SmartTag uses a very similar implementation for all its states, except for the status bits in the Service Data, i.e., the SmartTag type. A distinction between the ”lost” and ”nearby” state should therefore be possible. However, the datasets for the ”unpaired” and ”searching” states are too short for modeling. These two states should be discarded entirely for this tracker.
- The Tile tracker does not have a separate ”nearby” state. The ”lost” and ”nearby” states are identical and should be put into the same class for modeling. Due to the shortness of the datasets for the ”searching” and ”unpaired” states, these two states should be discarded entirely for this tracker.
- The packets transmitted by the iDevices (iPhone, iPad, and MacBook) are all part of the continuity protocol stack. Principally speaking, it does not seem possible to distinguish between different devices. A packet of a certain continuity type will generally look the same across all devices. Therefore, the classes for these devices should be formed based on the continuity type and not the device.
- For the Find My packets of type 0x12, a separation between the AirTag and other Apple devices based on the PDU type is possible. Additionally, a distinction between ”online” and ”offline” is possible based on the length of the manufacturer specific data. Therefore, FindMy online and FindMy offline packets can be divided into two separate classes for modeling in the case of the iDevices. This is especially interesting, as these packets indicate that the transmitting device is trackable via Bluetooth, and the device is, therefore, a tracker under the definition of this thesis.
- The AirPod is a hybrid between the AirTag and the iDevices. Its Find My packets should be in the same class as the AirTag’s, and all the other packets should be in the corresponding class of the iDevices based on the continuity type. For example, the 0x07 packets of the AirTag should be in a class labeled as ”iDevice” with continuity type 0x07.

- All the other devices form a separable garbage class, which is perfectly usable for modeling.

6.4 Summary of Analysis

The most relevant insights from this chapter are:

- Complex data preprocessing is required prior to the analysis of the data. This preprocessing is enabled by a data pipeline built with the Task-Group-Framework.
- The feature extraction process differs from feature to feature. There are classic BLE features such as packet length or PDU type but also more advanced features such as the continuity type or the SmartTag type.
- Categorical variables are converted to dummy variables to enable automatic labeling for trackers in the ”nearby” state.
- In addition to the device labels, device states and the continuity type of packets were considered for the analysis. The goal of the analysis was to generate separable classes, hence the distinction.
- All devices were extensively analyzed using a wide range of plots, at least one for every extracted feature. These plots highlight key differences between the various devices and states.
- Not all devices are distinguishable from each other. For successful modeling, it is of utmost importance to form separable classes based on the analyzed features. Similar devices, such as Find My trackers, are especially hard to distinguish and should be put into the same classes for modeling.
- Some features, such as the Bluetooth protocol type, are irrelevant for modeling, as they never differ between devices and states.

Chapter 7

Modeling

The modeling chapter covers the creation of the actual machine learning models. The chapter after covers the inference, i.e., the application of the models to real-world data. This chapter is split into the following sections:

- The first section covers the classes formed for the classification, i.e., how the devices in each state were labeled.
- The data preprocessing section covers the data preprocessing for the modeling, i.e., the feature extraction. This data preprocessing makes use of the Task-Group-Framework.
- The analysis section covers the analysis of the previously formed modeling classes. The goal is to verify that the classes are separable based on the extracted features.
- The packet modeling section covers various machine learning models using the simple packet modeling approach.
- The packet rate modeling section covers various machine learning models using more advanced packet rate modeling approaches.
- The comparison section compares the various models with each other, discusses their advantages and disadvantages, and discusses potential improvements such as hyperparameter optimization. Ultimately, one of the many machine learning models is picked for the inference in Chapter 8.

This chapter's ultimate goal is to create a machine learning model that best suits the task of classifying BLE devices in high-traffic environments.

7.1 Modeling Classes

As a first step, it is important to define the classes for modeling. The classes were set as follows based on the insights gained during analysis:

- **FindMy Tracker (lost):** This class includes the three Find My trackers AirTag, Chipolo, and SkyTag in their ”lost” state.
- **FindMy Tracker (nearby):** This class includes the three Find My trackers AirTag, Chipolo, and SkyTag in their ”nearby” state.
- **FindMy Tracker (unpaired):** This class includes only the AirTag in its ”unpaired” state. For the other two Find My trackers, there is not enough data for this state.
- **SmartTag (lost):** This class includes the SmartTag in its ”lost” state.
- **SmartTag (nearby):** This class includes the SmartTag in its ”nearby” state.
- **Tile (lost):** This class includes the Tile in its ”lost” state. As the Tile tracker does not have a distinctive ”nearby” state, all packets captured in the ”nearby” state were labeled with the ”lost” state and put into this class.
- **iDevice:** This class includes all packets from the iPhone, iPad, and MacBook that are not Find My packets. Additionally, it includes packets from the AirPod in its ”lost” state that are not Find My packets.
- **iDevice FindMy offline:** This class includes all Find My packets from the iPhone, iPad, and MacBook that do not include the public key, i.e., were transmitted by an offline iDevice.
- **iDevice FindMy online:** This class includes all Find My packets from the iPhone, iPad, and MacBook that include the public key, i.e., were transmitted by an online iDevice.
- **other Device:** This class includes all data from the other devices.

One might add that forming even more granular modeling classes would be possible. For instance, separating all packets from the iDevices would be possible based on their continuity type. Additionally, as shown in the analysis, it would be possible to distinguish between the iOS devices (iPhone and iPad) and the MacBook based on the packet rate. However, the goal was to have at maximum 10 different classes for modeling, hence this generalization.

Additionally, some devices and states are not included in the list above. The captured data of these devices and states was not used for modeling:

- The Chipolo and SkyTag tracker in the ”unpaired” state due to a lack of data.
- The SmartTag and the Tile tracker in the ”searching” and ”unpaired” state due to a lack of data.
- The AirPod in its ”nearby” state and all the Find My packets from the ”lost” state. The Find My packets are no different from the ones transmitted by the AirTag; hence, they are not interesting to model. And because the other non-Find My packets do not differ between the ”lost” and ”nearby” state, the dataset from the latter was dropped entirely.

7.2 Data Preprocessing

The data preprocessing for the modeling chapter is analogous to the preprocessing in the analysis chapter. The implemented data pipeline uses the Task-Group-Framework. However, this pipeline is longer as it involves specific processing steps for modeling purposes. The steps in the processing pipeline are the following:

1. **General data preprocessing:** This includes feature extraction and simple processing steps such as filling NULL values.
2. **Conversion to dummy variables:** All non-numerical features, e.g., categorical features, must be converted to numerical values, i.e., one-hot encoded.
3. **Labeling:** All packets need to be labeled. This can happen manually, where every packet is assigned the same label, or automatically using the machine learning approach.
4. **States and Continuity Type:** The class labels must be extended to class state labels. For instance, in the case of an AirTag in its "lost" state, the previously assigned class label "AirTag" is extended to "AirTag (lost)".
5. **Drop Labels:**: Some labels for some datasets need to be dropped. For example, in the case of a tracker in the "nearby" state, the packets of the owner device should not be part of the final dataset for modeling.
6. **Modeling:**: For the packet rate modeling approaches, the datasets need to be resampled and aggregated to make them suitable for the packet rate modeling, as described in Chapter 2.

The following subsections will detail some of the most relevant aspects of these individual pipeline steps.

7.2.1 General Data Preprocessing

The general data preprocessing and feature extraction are very similar to how this was done for the analysis in Chapter 6. For every feature, the individual processing steps are the same. However, some features were excluded as they are irrelevant for classification, as shown and seen in the analysis section. The resulting features for the modeling are:

- Channel
- Length of packet
- Length of header
- Advertisement data type

- Company ID
- Length of manufacturer specific data
- UUIDs
- Length of Service Data
- PDU type
- Continuity type
- SmartTag type

The features broadcast, malformed packet, and protocol type were removed as the analysis showed that they are irrelevant, given that they are the same across virtually all packets.

7.2.2 Conversion to Dummy Variables

As in the analysis pipeline, the categorical variables must be converted to dummy variables. The approach taken is also exactly the same. The custom executor for dummy variable creation based on the Task-Group-Framework was used. This ensures a deterministic selection and order of the dummy variables.

The specific dummy variables, i.e., which categorical values would receive their own dummy variable columns and which would be aggregated into an other column, were selected based on the insights gained in the analysis. The dummy variables that seemed most promising based on the analysis were selected. In other words, dummy variables that might allow for device distinction were selected. The goal was to limit the number of columns, i.e., features, to speed up training and inference on the machine learning models.

All categorical features except one have a dummy variable column for other values to indicate values other than the selected dummy variable columns. The channel does not use such a column as there are exactly three channels: 37, 38, and 39. It is important to note that absent values (or NULL) values will result in zeros in all dummy variable columns. This is important, as the packet rate modeling can create zero rows, which should be equivalent to the complete absence of any packets.

7.2.3 Labeling

The labeling section of the pipeline labels all packets. This can happen either with simple labeling, where all packets receive the same label, or with the machine learning-based approach. It is important to note that the machine learning models for automatic labeling had to be trained again from scratch for the modeling pipeline. Reusing these models across pipelines is impossible as the feature extraction differs between pipelines. The resulting feature vector of the modeling pipeline has a different number and order of features. Therefore, the machine learning models from the analysis pipeline are incompatible.

The class labels for the modeling section differ from those used in the analysis, so the labeling section of the pipeline had to be slightly adapted. However, the underlying implementation is very much the same as in the analysis pipeline.

7.2.4 States and Continuity Type

As within the analysis pipeline, states must be added to the class label for some devices. The required states differ from the ones needed during analysis. The analysis showed that modeling might not be useful for some states, such as the ”searching” states. Therefore, those states can be omitted.

Additionally, the defined modeling classes do not require packet labeling of the continuity type except for the Find My packets. This tremendously simplifies the process of state labeling. In the end, the state is again appended with brackets to the class label, i.e., a SmartTag in the ”lost” state would become ”SmartTag (lost)”.

7.2.5 Drop Labels

Some class labels needed to be dropped from the training data set as they were not needed for modeling. First and foremost, this affects all owner devices for trackers in the ”nearby” state. The packets from these devices are not needed for modeling and can be removed. In other words, all packets with the label from the owner device can be dropped from a labeled dataset of trackers in the ”nearby” state.

This applies similarly to the AirPod, where all the Find My packets were removed from the dataset. The Find My packets are indistinguishable from the ones used by the Find My trackers (such as the AirTag) and would, therefore, cause havoc during modeling.

At this point, it is very important to remember that the automatic labeling for trackers in the ”nearby” state is not perfect. It can and does happen that packets are falsely labeled, even with test set accuracies approaching 100%. Therefore, when the class label of the owner device is dropped, it is perfectly possible that there are still some packets from the owner device left in the dataset, as these were falsely labeled with the label from the tracker and, therefore, not dropped. This will become highly important during the packet rate modeling.

7.2.6 Modeling

In the modeling section of the pipeline, the datasets are finally prepared for modeling. This means that unnecessary columns, such as the time column, are removed, and if requested by setting an appropriate Flag, the datasets are resampled and aggregated for packet rate modeling.

Picking the resampling interval for packet rate modeling is difficult. As discussed in previous chapters, there is a trade-off between the model’s applicability and the presumed

accuracy. The longer the interval, the more accurate the model is; the shorter the model, the higher the practical applicability. If a device is not seen by the BLE sniffer for at least the resampling interval, packet rate modeling is not applicable. Therefore, the resampling interval had to be set at the shorter end of the spectrum.

During analysis, this value was 15 seconds; for modeling, the resampling interval is 10 seconds. From a practical point of view, it might be preferable to go even lower, even though this must hurt accuracy. But based on some analysis of datasets captured at Zürich central station, an interval of 10 seconds will already cause most devices (ca. 60%) not to be classifiable as they are not seen long enough by the packet sniffer.

7.3 Analysis of Modeling Classes

After defining the modeling classes and creating the pipeline, it might be worth it to take a brief look at the plots to ensure and validate that the classes are, in fact, separable. The plots shown are the same ones used for analysis. However, only a selection of relevant plots are shown, and the rest is omitted for brevity. All plots can be found on GitHub. Additionally, the "other Device" class is not shown on the plots for additionally clarity. It is already known from analysis that the other devices are distinctively different from all tracking devices.

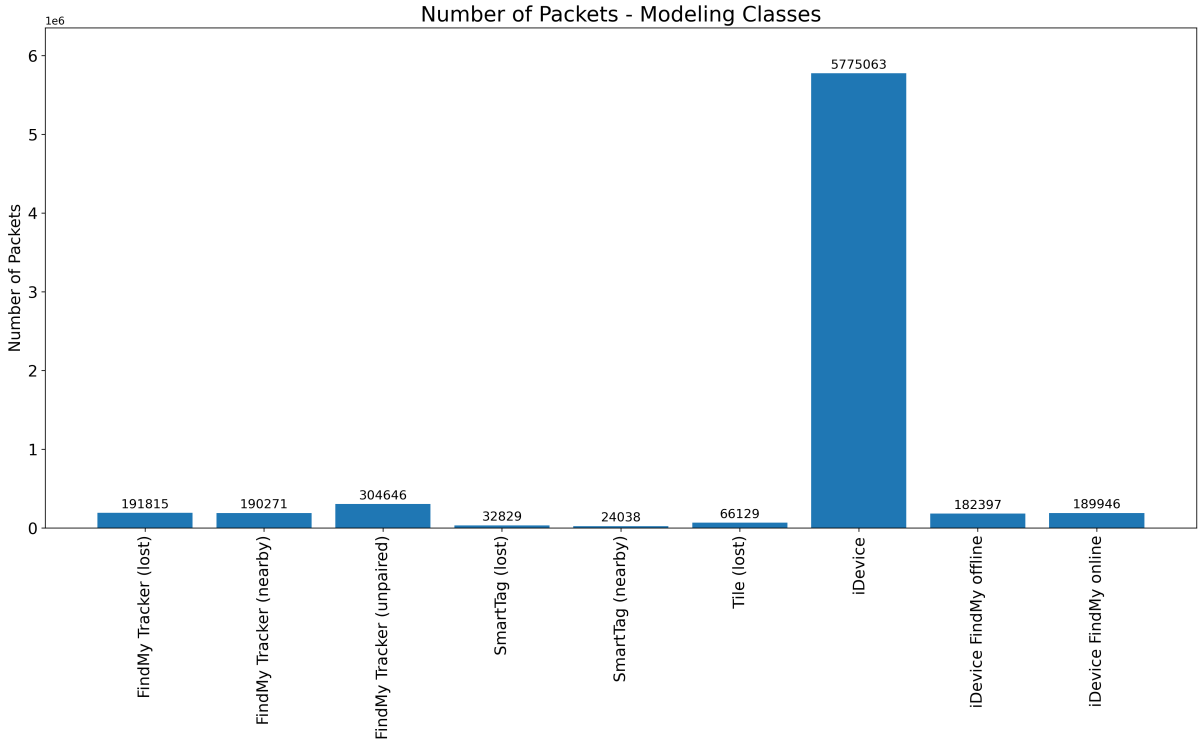

Figure 7.1: Number of Packets - Modeling Classes

The number of packets is sufficient for modeling all classes. From a practical point of view, the smallest and largest numbers of packets are particularly interesting. There are two techniques for balancing the dataset for modeling: undersampling and oversampling. In the case of undersampling, the class with the lowest number of samples is decisive. This number of samples is picked at random from every class, and in the final dataset, every class has exactly the same number of samples, i.e., the number of samples from the class with the fewest samples.

In this case, the class with the fewest samples is the "SmartTag (nearby)" class, with roughly 24'000 samples. Therefore, the size of the entire dataset, balanced with undersampling, would be roughly 240'000 samples large, which is more than enough for training any kind of machine learning model.

With oversampling, the entire process is reversed, and the class with the most samples is decisive. In this case, the class with the most samples is the "other Device" class, with

roughly 1.8 million samples (not shown on the plot). This would result in a balanced dataset with roughly 18 million samples (10 classes with 1.8 million samples each). Such a large training dataset would certainly be infeasible given the sheer amount of computing performance and memory required for training.

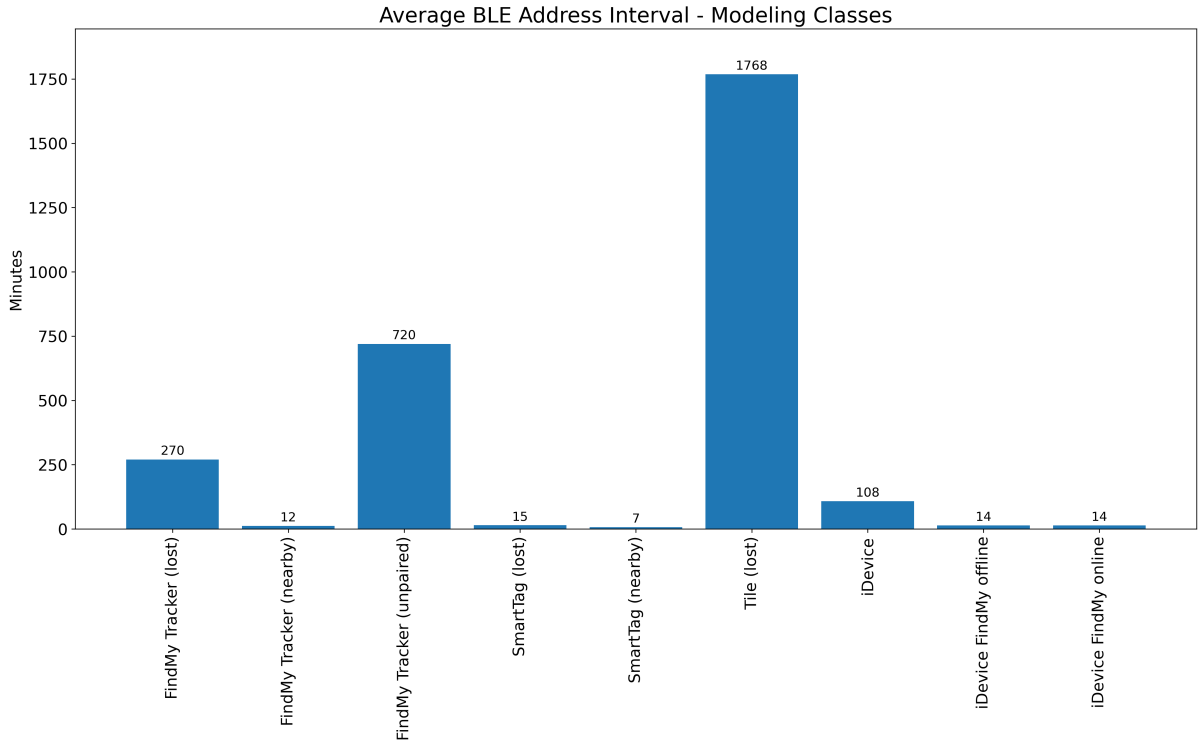

Figure 7.2: Average BLE Address Interval - Modeling Classes

The average source address interval is sufficiently large for all classes to do packet rate modeling. Even the roughly 7 minutes from the SmartTag in its ”nearby” state should not pose any problem.

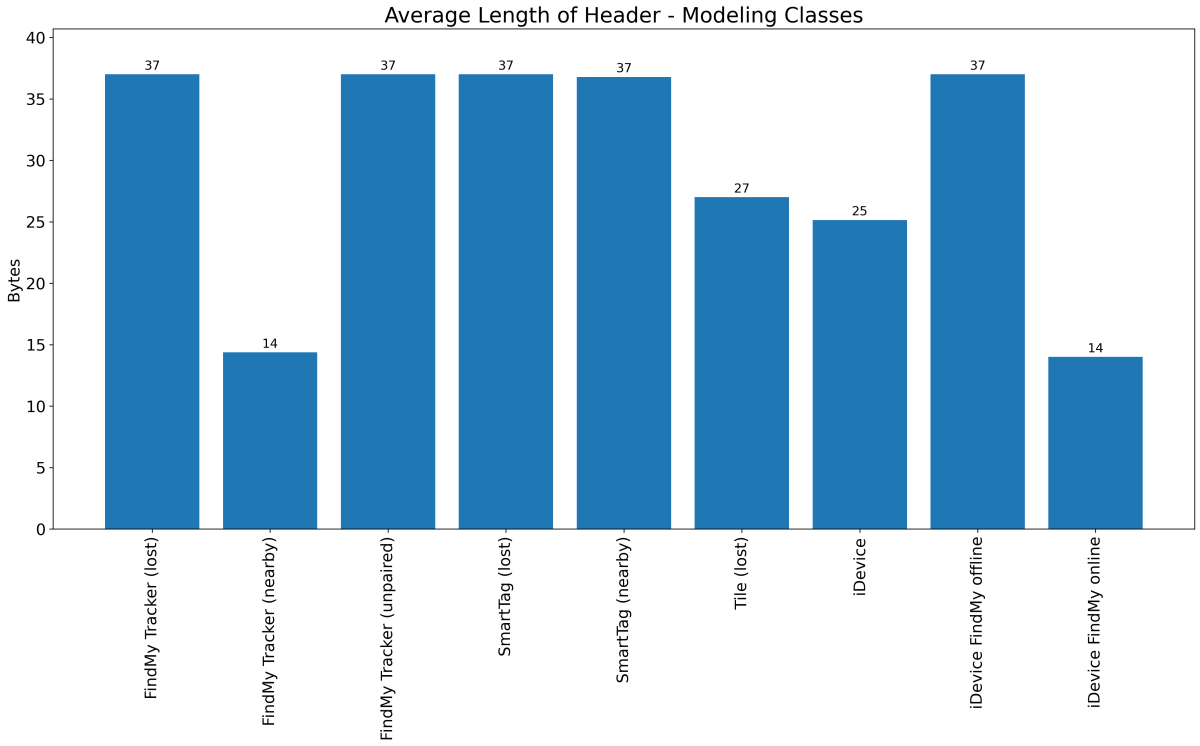

Figure 7.3: Average Length of Header - Modeling Classes

When looking at the header length, i.e., the packet length on the BLE link layer, it becomes clear that the ”lost” state and ”nearby” state for the Find My trackers are separable. The same applies to the ”online” and ”offline” Find My packets for the iDevices. This behavior is expected and stems from the fact that only ”offline” and ”lost” Find My packets carry the long private key. However, the header length alone is not sufficient to separate the other classes.

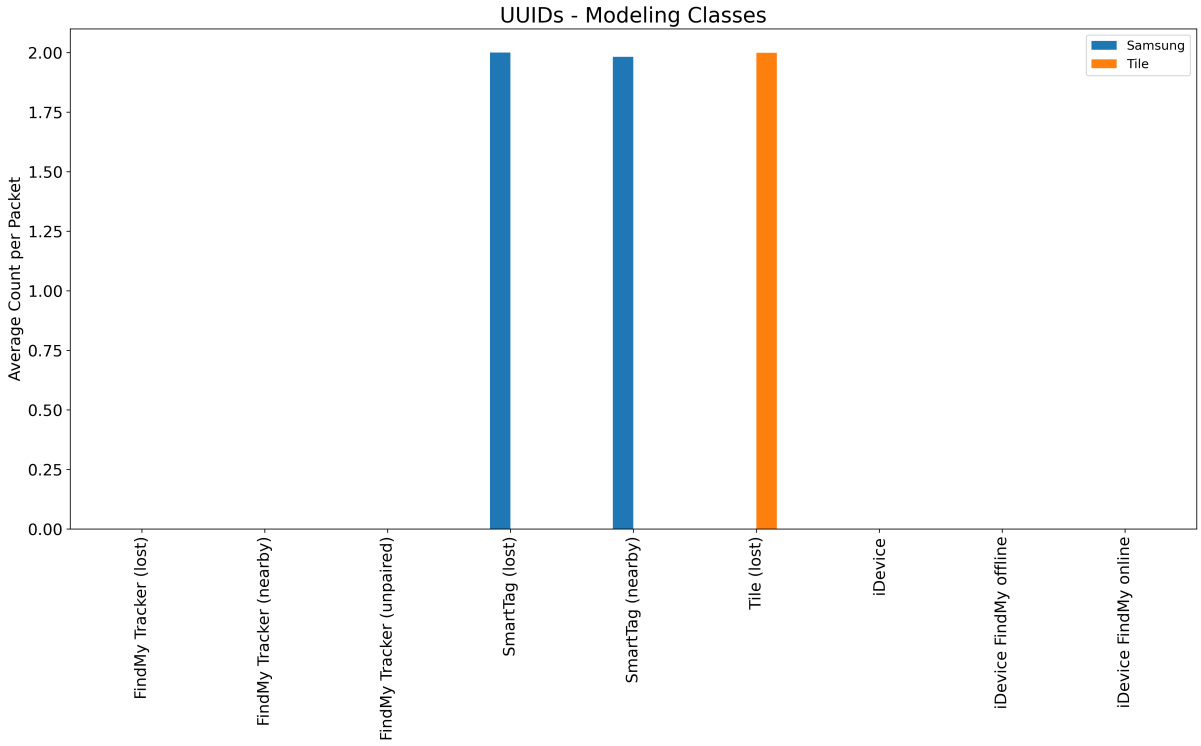

Figure 7.4: UUIDs - Modeling Classes

At first glance, the SmartTag and Tile tracker are unique in their respective UUIDs. No other device carries either of these two UUIDs. However, it is not possible to distinguish the SmartTag's two states with just the UUID. The UUID is only good for identifying the SmartTag as a device.

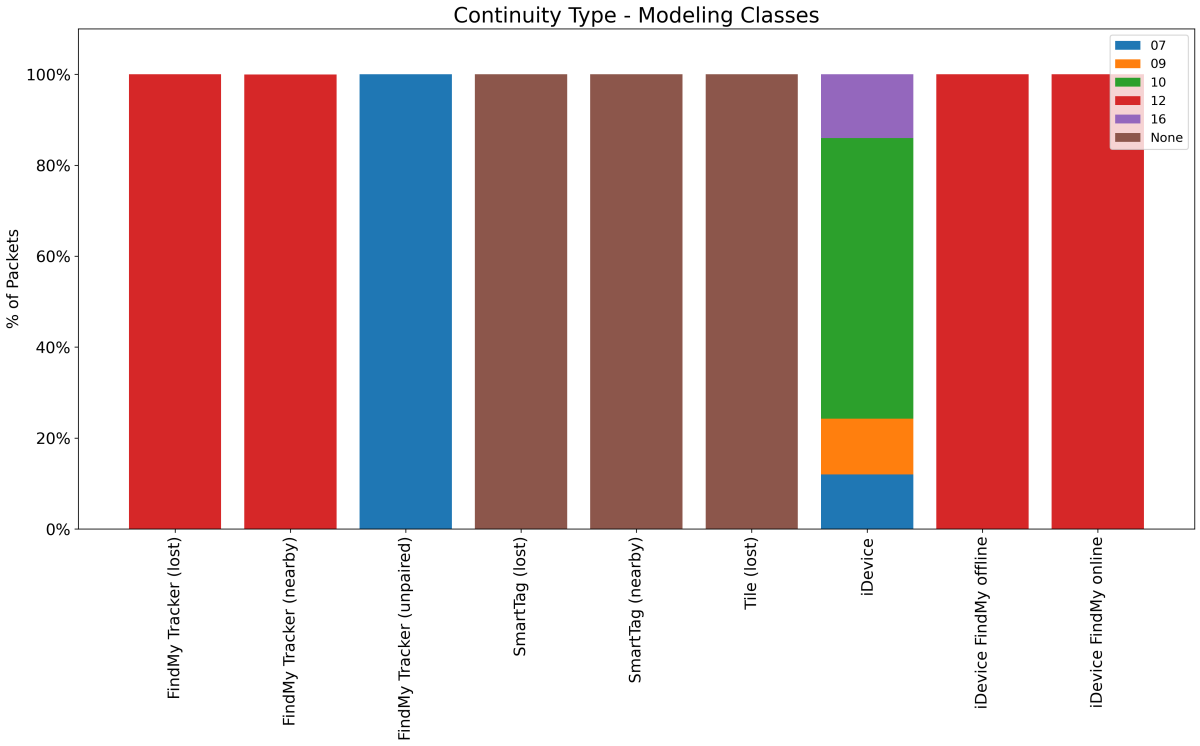

Figure 7.5: Continuity Type - Modeling Classes

The continuity type is one of the most important features. All variants of Find My packets share the continuity type 0x12. The ”iDevice” class shows a wide variety of continuity types. This is expected, as the iDevice class serves as a kind of garbage class for all Apple device packets that are not Find My packets. Devices that are not related to Apple, such as the SmartTag, do not carry a continuity type, as expected. Potentially problematic is the ”FindMy Tracker unpaired” class that shares its continuity type 0x07 with some of the packets from the ”iDevice” class. However, we already know that these packets in the ”iDevice” class stem from the AirPod and have a distinct PDU type.

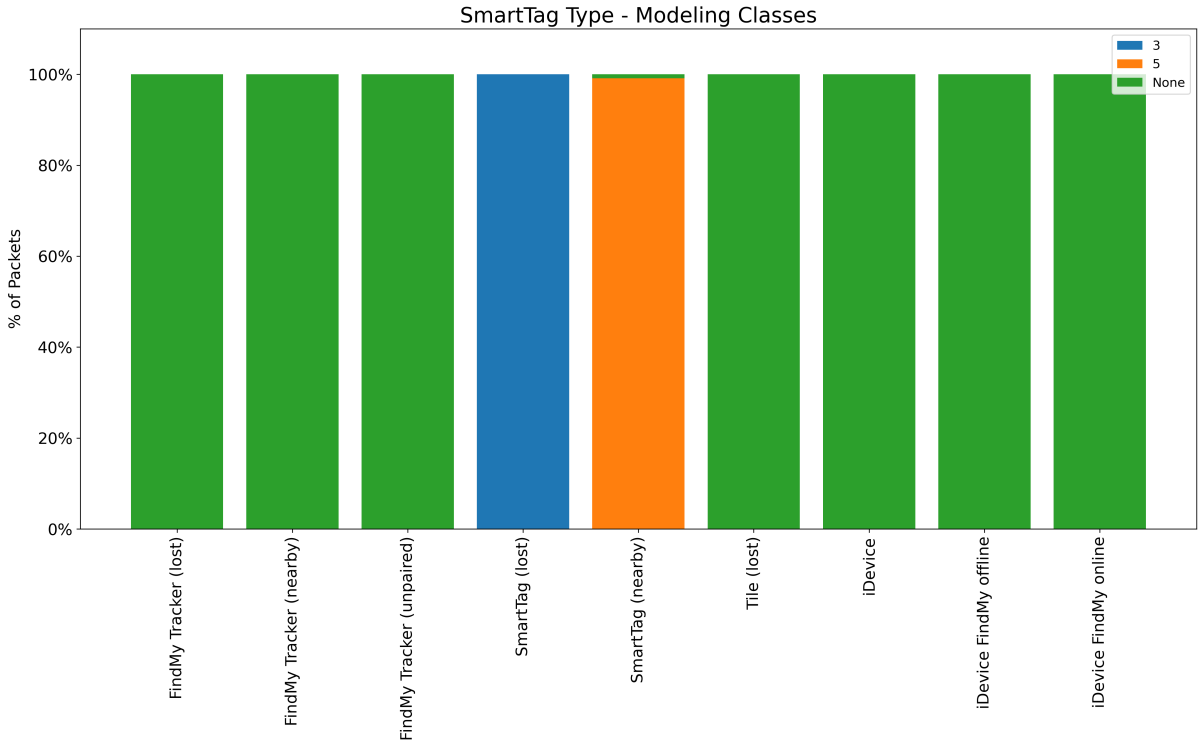

Figure 7.6: SmartTag Type - Modeling Classes

The SmartTag type is a highly relevant feature, albeit for the SmartTag only. This is the only feature that allows for distinction between the two states of the SmartTag. All other classes never carry a SmartTag type.

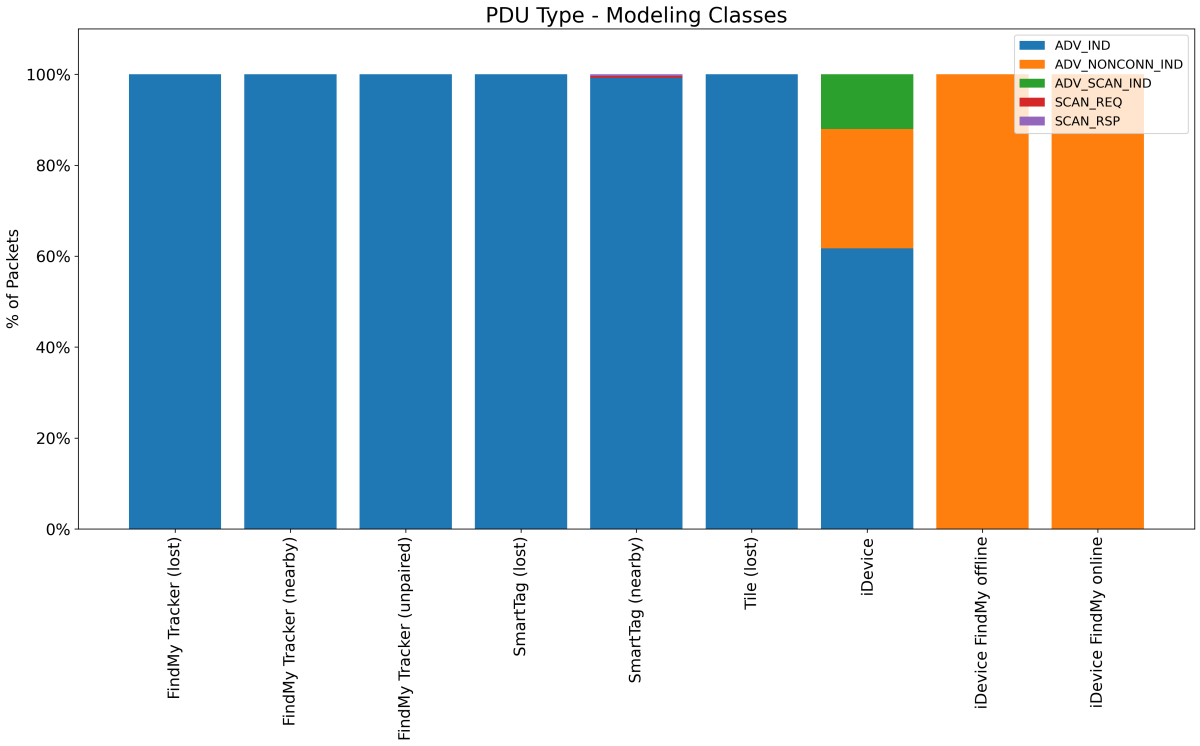

Figure 7.7: PDU Type - Modeling Classes

The PDU type is a feature especially relevant for the Find My packets. It is the only feature that allows for distinction between Find My packets from an iDevice and the Find My trackers. Other than that, his feature is presumably not very relevant.

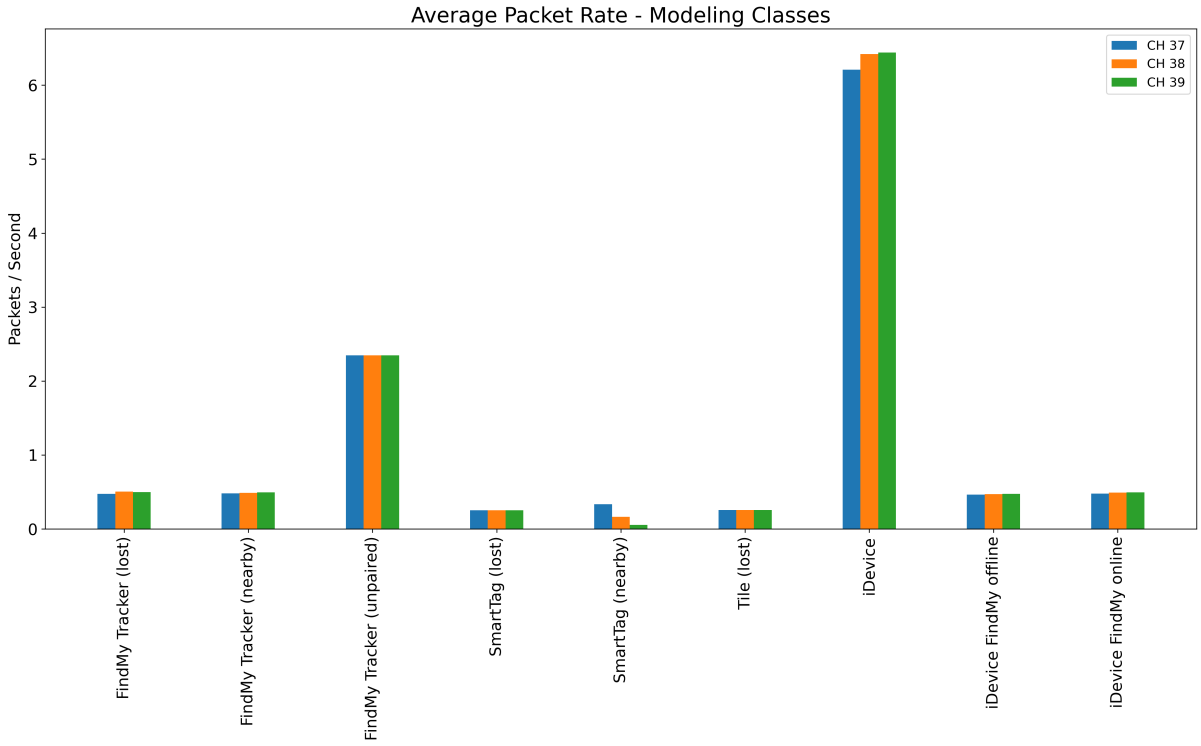

Figure 7.8: Average Packet Rate - Modeling Classes

The packet rate is the essential feature for packet rate modeling. However, the variance in the packet rate among the classes is not particularly large for most of them. Therefore, the marginal benefit of packet rate modeling could be negligible.

7.4 Packet Modeling

The following section covers three machine learning models that were applied to simple packet modeling, i.e., the classification of packets. The data preprocessing for these models was done using the previously described modeling pipeline. On a high level, the procedure for packet modeling followed the following steps:

1. Import, process, and label the datasets using the Task-Group-Framework and the purpose-built modeling pipeline.
2. Concatenate all datasets into one large dataset.
3. Balance this large dataset with undersampling. This results in a dataset with ca. 240'000 samples. As described in the analysis section of this chapter, oversampling is not feasible as it would result in roughly 18 million samples.
4. Split the balanced dataset into both a training and a test set with a 75/25 split. In other words, the test set consists of 25% of samples, roughly 60'000 samples, and the training set of the other 75%, roughly 180'000 samples. Both the training and test sets are also balanced datasets. For every class, there are about 18'000 samples in the training set and 6'000 samples in the test.
5. Scale the training set with a min-max scaler. The scaler is trained on the training set and then applied to both the training and the test set. This prevents data leakage between the test and training set. The scaler is then saved to a pickle file for later use, i.e., inference.
6. The scaled training data is used to train the machine learning model. The resulting model is saved in a pickle file for later use.
7. The machine learning model is evaluated on the test set with a confusion matrix and a classification report.

For packet modeling, three machine learning models were trained: a neural network, a semi-supervised self-training classifier based on a neural network, and a decision tree. The sci-kit-learn library was used to implement all three models. The above-described procedure applies to all three of these models. The following subsections will discuss the results obtained from the various models.

7.4.1 Neural Network

The neural network trained is the base implementation of the MLP classifier from the sci-kit learn library. This neural network has one hidden layer with 100 neurons and a RELU activation function. The second fully connected layer uses softmax to obtain the probabilities for the classes.

The resulting network should have roughly 4'000 parameters: 3'000 for the first fully connected layer (30 features * 100 hidden neurons) plus 1'000 for the second fully connected layer (100 hidden neurons * 10 classes). The bias neurons were ignored for this back-of-the-envelope calculation. Only the order magnitude is relevant, and 4'000 parameters are perfectly acceptable in the case of 180'000 training samples. Even though the training was done on the CPU only, the training time did not exceed a few seconds, as early stopping was used to prevent overfitting.

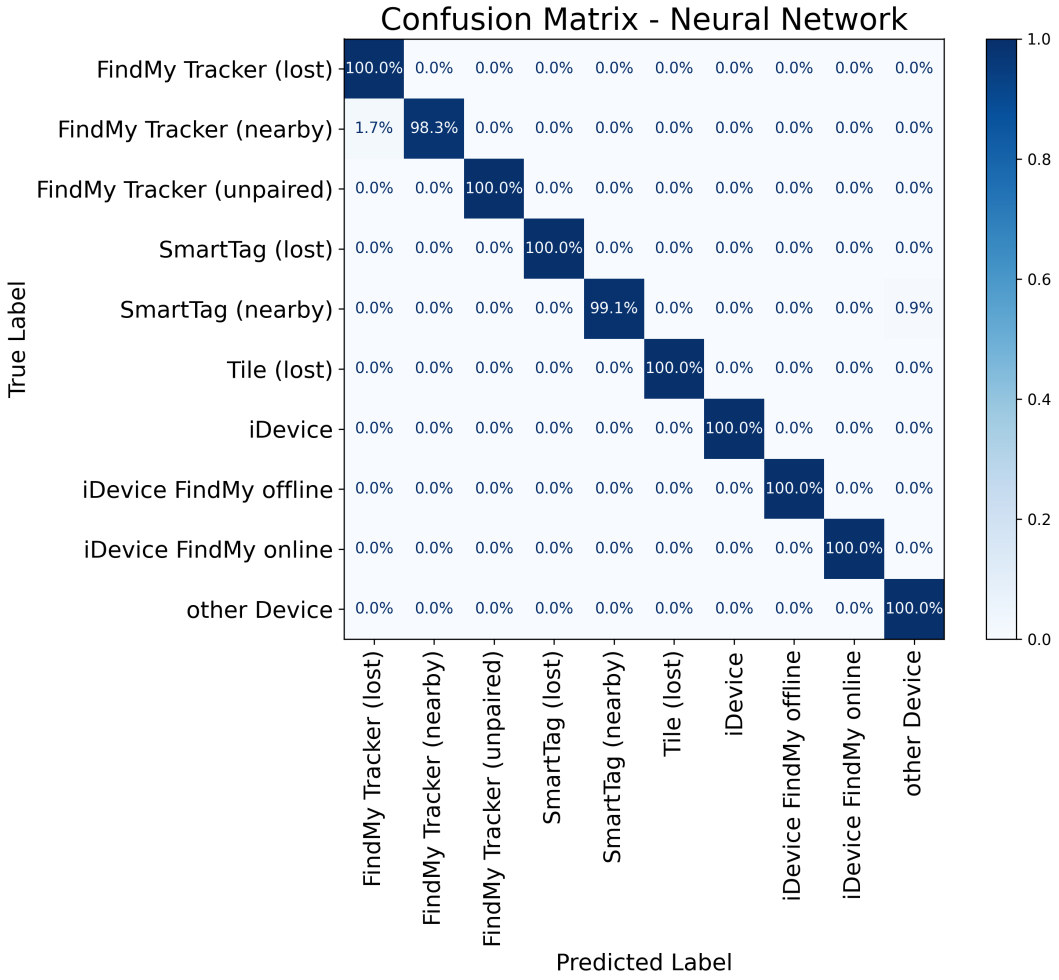

Figure 7.9: Confusion Matrix - Neural Network

The resulting confusion matrix (Figure 7.9) provides a clear view of the model's performance. For most classes, the accuracy is at a perfect 100%. The two major exceptions are the Find My trackers in their "nearby" state and the SmartTag in its "nearby" state.

The explanation for the Find My trackers is relatively straightforward. Sometimes, the Find My trackers in their ”nearby” state switch into their ”lost” state, as discussed in the analysis chapter. This produces confusion between the two states.

In the case of the SmartTag, the confusion stems most likely from incorrectly labeled packets, as automatic labeling is not perfect. Possibly, some of the owner device packets were confused with those of an ”other Device”, as the owner device for the SmartTag, a Samsung Galaxy S23 Ultra, is also part of the dataset for the other devices.

7.4.2 Self Training Classifier

The previously created neural network can be extended with semi-supervised learning to improve the robustness in real-world inference. The semi-supervised method used in this thesis is a self-training classifier built upon a neural network, i.e., the network created in the previous subsection. The implementation stems again from the sci-kit learn library.

Semi-supervised learning aims to improve the model's capability to generalize and, therefore, improve its performance during real-world inference. However, the performance on the fixed test set does not necessarily have to improve. The idea is solely to improve the robustness. This works best when the additional unlabeled training data is similar to the data seen during inference. For this purpose, there are two datasets captured at Zürich central station. The shorter one is for semi-supervised learning, and the longer one is for inference. Both were captured under identical conditions shortly after each other, as described in the chapter about dataset generation.

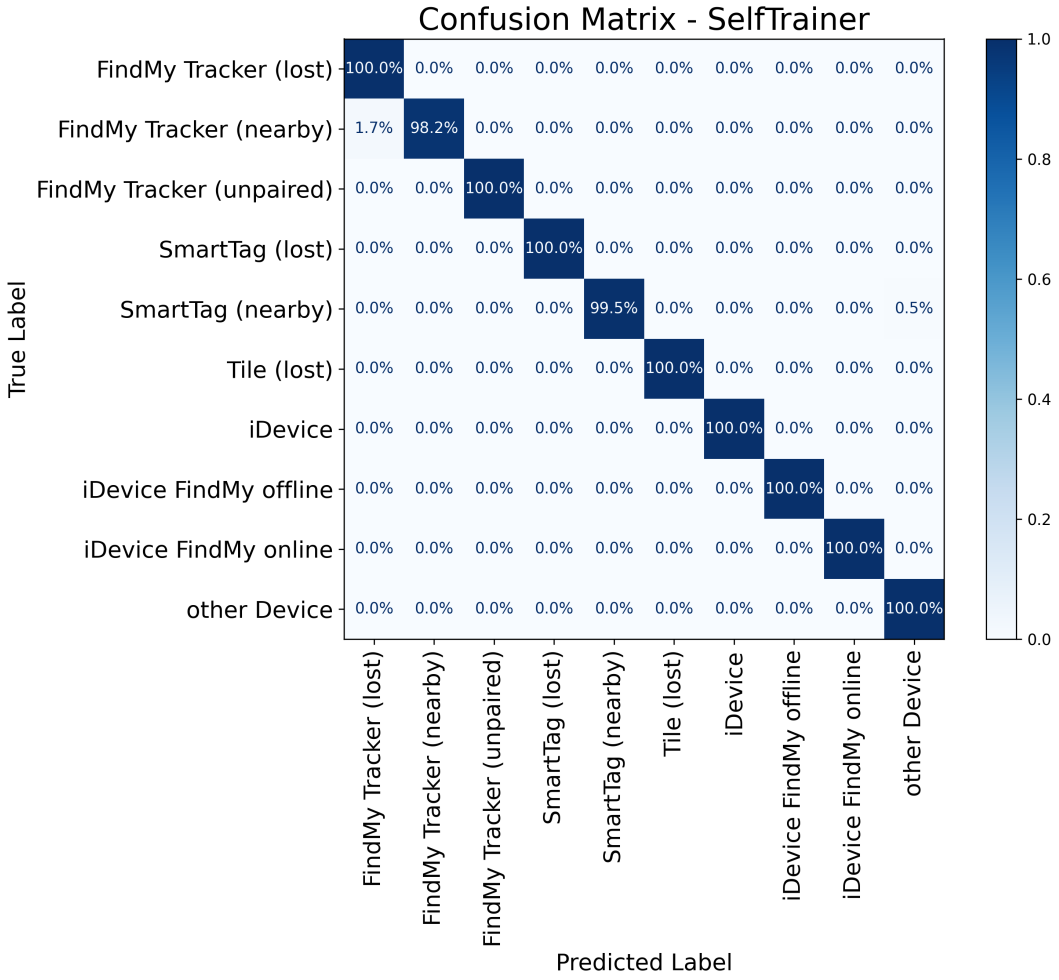

Figure 7.10: Confusion Matrix - SelfTrainer

As seen in the confusion matrix (Figure 7.10), the model's performance is more or less identical to that of the base neural network. Therefore, this model is preferred over the

base neural network as it presumably performs better on real-world data and verifiably not worse on the test data.

7.4.3 Decision Tree

The final model for simple packet modeling is a simple decision tree. The insights gained during the analysis of the data suggest that it should not be all that difficult for a decision tree to separate the classes as most of them have a unique feature attached to them. Additionally, decision trees are quick to train and highly interpretable.

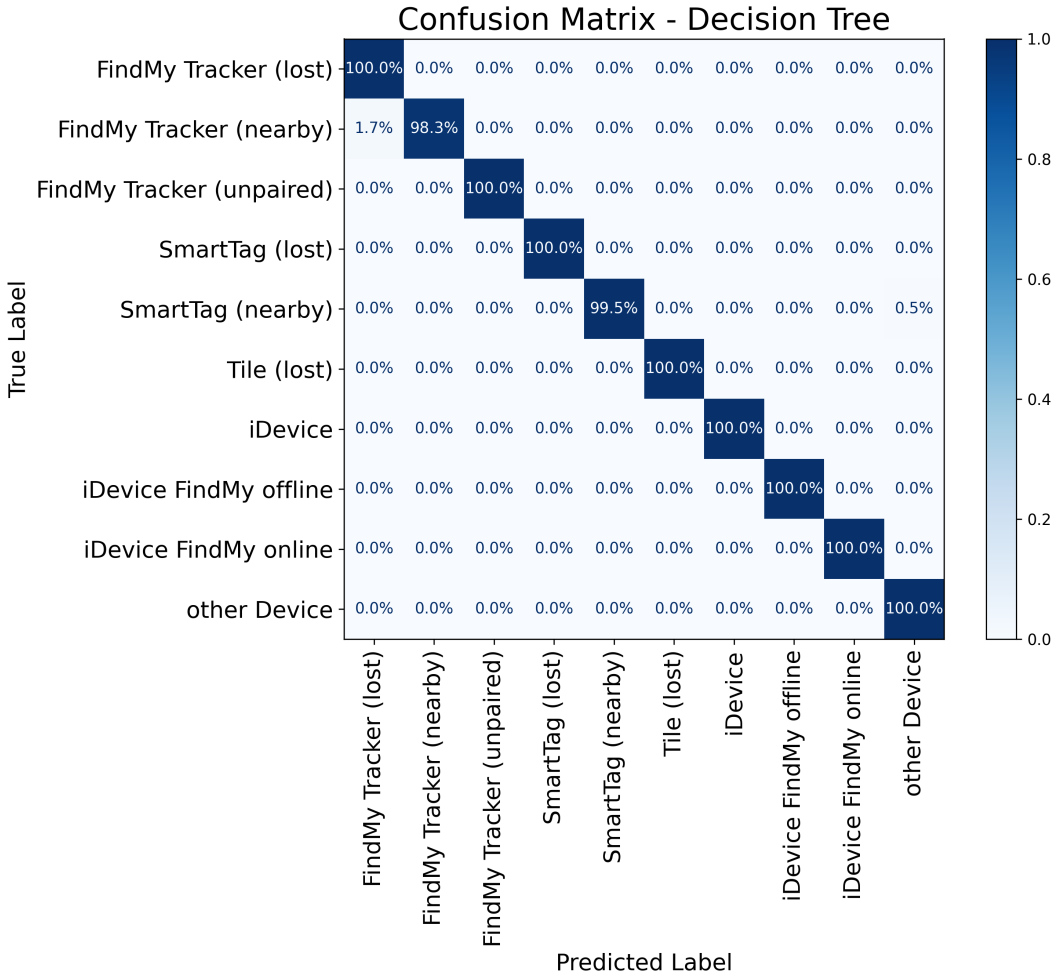

Figure 7.11: Confusion Matrix - Decision Tree

The decision tree used is the base implementation from the sci-kit learn library, and no hyperparameters are changed. The decision tree's training terminates after a few seconds. The obtained confusion matrix (Figure 7.11) confirms that the decision tree, as simple as it might be, is perfectly suitable for BLE device classification. The overall accuracy is comparable to that of the neural network.

However, it is also possible to further exploit a decision tree's two main advantages: its quick training and interpretability. The latter can be examined using a plot of the relative feature importance. In theory, the most important features should be roughly the ones that were identified during the analysis as highly relevant for the classification. In

other words, the SmartTag type (ST), which is absolutely necessary to separate the two SmartTag states, should be much more important than, for instance, the channel number.

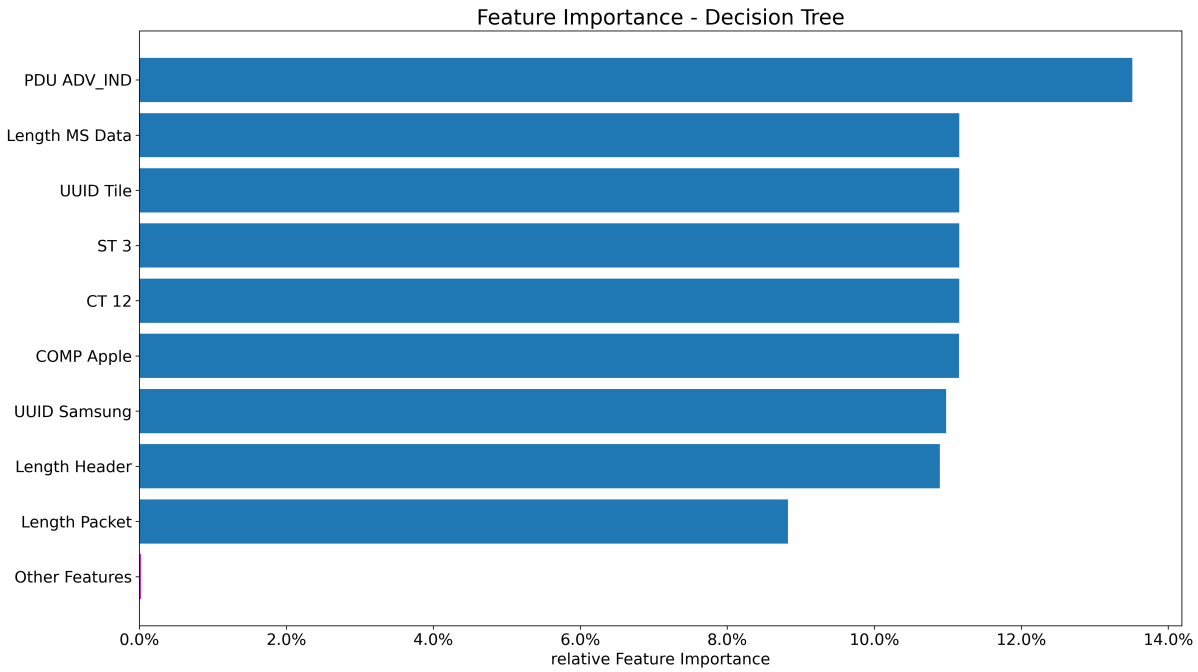

Figure 7.12: Feature Importance - Decision Tree

Plotting the importance of features (Figure 7.12) reveals that the findings during the analysis were, in fact, correct. All of the presumably relevant features are also features that are relevant to the decision tree classification. This indicates that the decision tree learned to classify devices based on their unique features.

One other interesting question to ask is whether the amount of training data has an influence on accuracy. For this purpose, one can train a model multiple times on training sets of various sizes and compare the results by evaluating each model on a fixed test set. For this purpose, a total of seven decision trees were trained on training datasets ranging from 1/64 of the training data to the full dataset. All decision trees were then evaluated on a fixed test set. The resulting accuracy plot (Figure 7.13) clearly reveals that the amount of training data has little to no influence on the overall model's accuracy. Therefore, it is highly likely that capturing over 30 million packets would not have been necessary for accurate BLE device classification.

Finally, it is also possible to plot the decision tree as a tree. Therefore, there is a highly detailed plot of over 300 megapixels on GitHub showing the entire decision tree. Due to the sheer size of this plot, it was not possible to show it in this thesis.

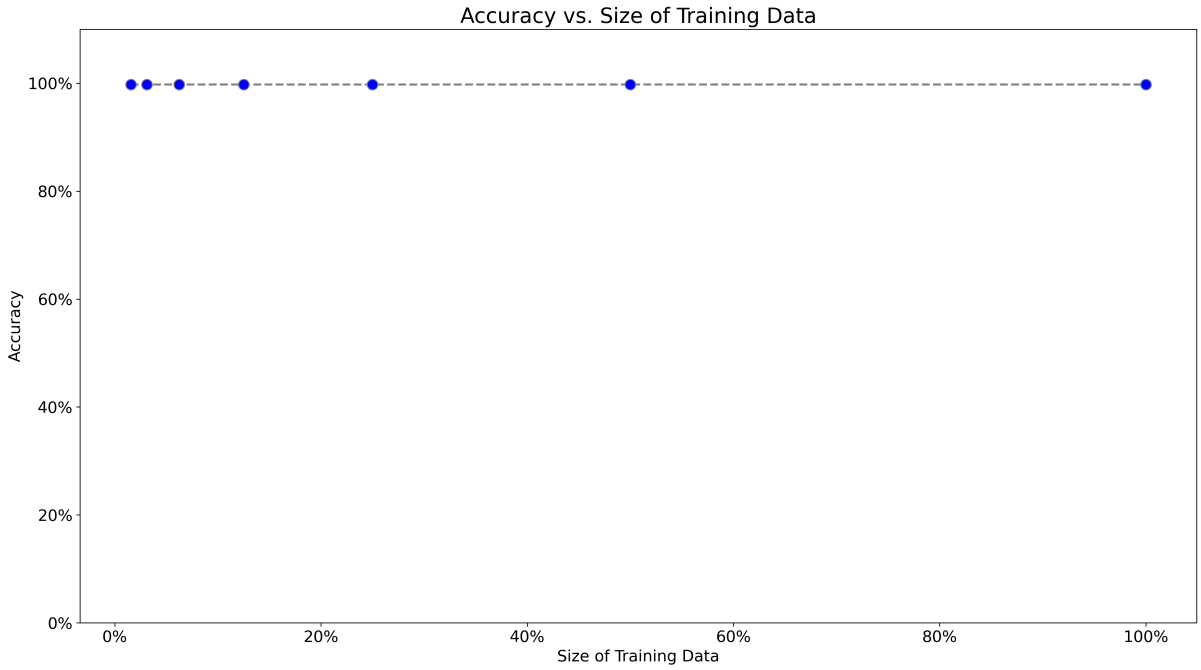

Figure 7.13: Accuracy vs. Size of Training Data

7.5 Packet Rate Modeling

The following section covers the packet rate modeling as described in Chapter 2. The data preprocessing was done using the previously described modeling pipeline. On a high level, the procedure for packet rate modeling followed the following steps:

1. Import, process, and label the datasets using the Task-Group-Framework and the purpose-built modeling pipeline. Additionally, in the final step of the pipeline, the packets are resampled and aggregated to generate the packet rate samples. Every 12-hour dataset will result in 4'320 samples (43'200 seconds divided by the resampling period of 10 seconds).
2. Concatenate all datasets into one large dataset.
3. Balance this large dataset with undersampling. This results in a dataset with 43'200 samples.
4. Split the balanced dataset into a balanced training and test set with a 75/25 split. The resulting training set has 32'400 samples, and the test set has 10'800.
5. Scale the training set with a min-max scaler. The scaler is trained on the training set and then applied to both the training and the test set. This prevents data leakage between the test and training set. The scaler is then saved to a pickle file for later use, i.e., inference.
6. The scaled training data is used to train the machine learning model. The resulting model is saved in a pickle file for later use.
7. The machine learning model is evaluated on the test set with a confusion matrix and a classification report.

The machine learning model picked for the packet rate modeling is again a decision tree. The evaluation of the trained decision tree on the test set reveals a rather mediocre accuracy (Figure 7.14). For most classes, the model is highly accurate. However, the SmartTag and Find My tracker in their "nearby" state and the "iDevice" class cause problems and are sometimes confused with the garbage class, i.e., the "other Device" class. Principally speaking, one would expect the exact opposite. The added information in the form of the packet rate should increase the model's accuracy and not decrease it. On top of that, the packets from the SmartTag and the iDevices are not similar in any shape or form, so confusion should not take place.

However, after a close inspection of the model's classifications and the underlying test set, there seems to be a rather logical explanation for this. Remember, automatic machine learning-based labeling for trackers in the nearby state is not perfect. Sometimes, a packet from the owner's device can be falsely labeled as the tracker. These packets are also not removed in the preprocessing pipeline, as they are falsely labeled. During the final pipeline step, where the packet rate modeling takes place, the data is then grouped by the source address to perform the packet rate modeling per source address.

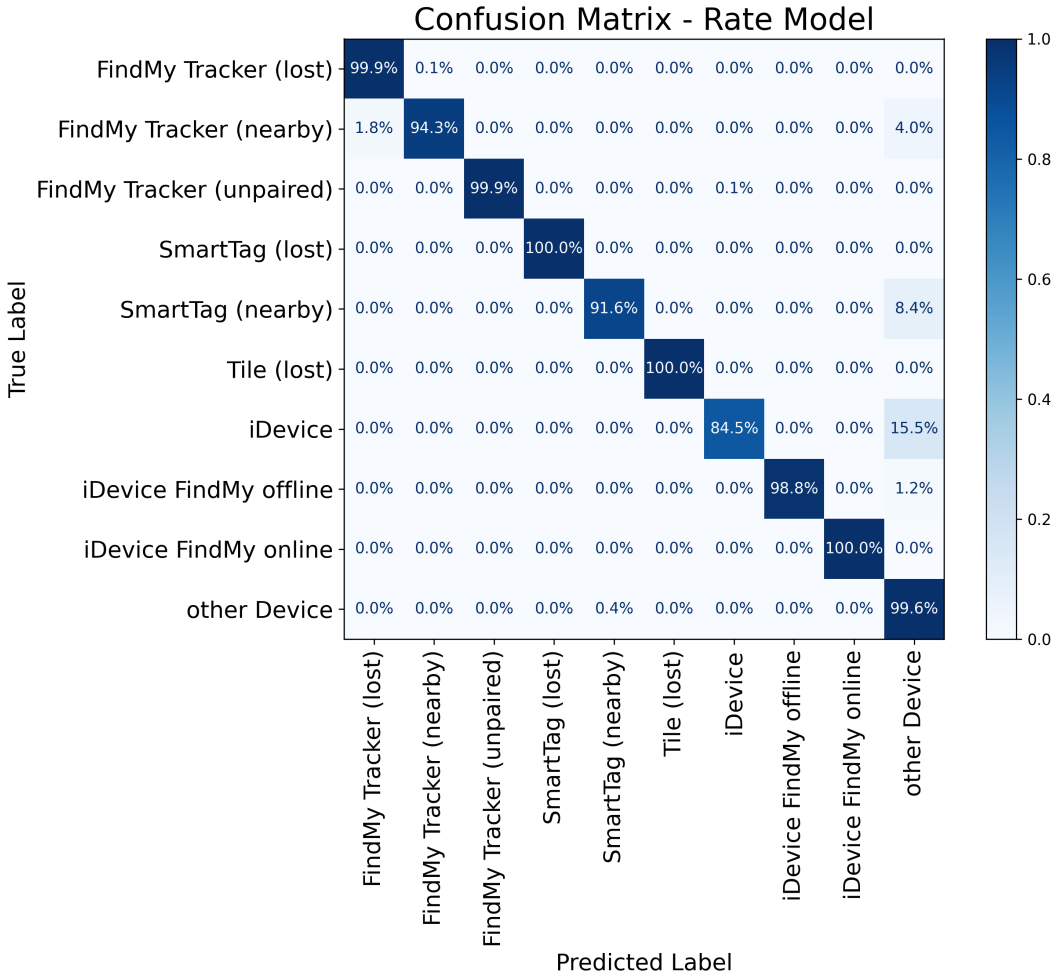

Figure 7.14: Confusion Matrix - Rate Model

Even if only a few of the owner device's packets were falsely labeled, there are potentially large gaps between the individual falsely labeled packets. Packet rate modeling will fill these gaps with zero rows, indicating the absence of packets. The same thing happens when a source address is "dormant" for a while and then suddenly restarts transmitting packets. This will also result in zero rows, i.e., the absence of packets, for the entire time the source address was dormant. This is presumably what happened in the case of the "iDevice" class and the "other Device" class. And because zero rows from one device are identical to the zero rows from any other device, it is not possible to separate them.

Therefore, filtering the zero rows prior to training the model should result in much higher overall accuracy. When looking at the second confusion matrix (Figure 7.15), it is apparent that the zero rows were indeed the issue and that removing them increases the accuracy almost across all classes.

However, the resulting accuracy is still underwhelming at best. It is slightly worse overall than the accuracies obtained by simple packet modeling. As of writing this thesis, I do not exactly know why this is the case. What should be clear is that packet rate modeling is most likely not the most fabulous idea. Given the relatively high resampling interval

of 10 seconds, most devices in a real-world environment cannot be labeled. Therefore, the applicability is much worse than that of simple packet modeling. Additionally, the accuracy is only just as good, if not slightly worse. Increasing the resampling interval might increase accuracy, but it would further limit the applicability. Therefore, packet rate modeling is great from a theoretical point of view but not very applicable in practice.

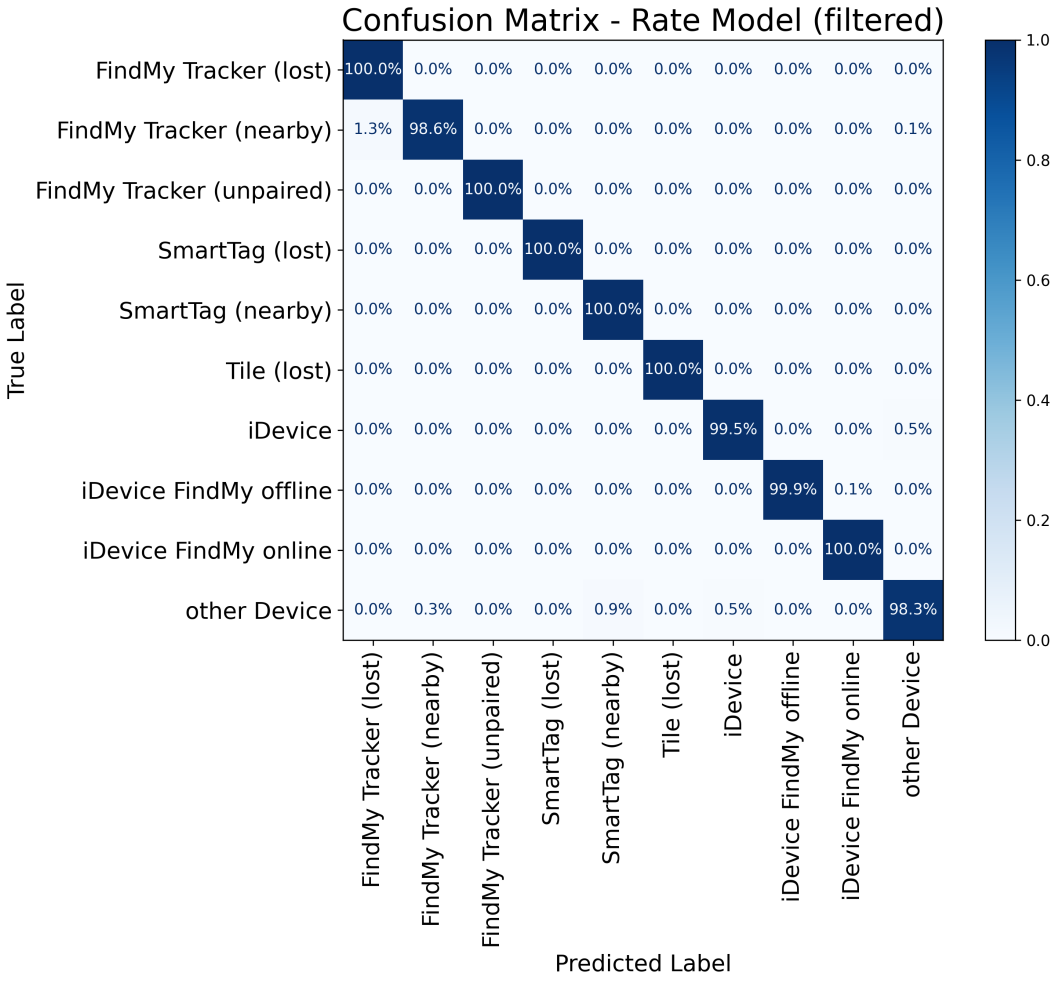

Figure 7.15: Confusion Matrix - Rate Model (filtered)

7.6 Comparison of Models and Improvements

Overall, all models show comparable accuracies on the test set. From an accuracy point of view, there is no reason to choose one model over the other for inference. The limiting factors are practical applicability and presumed performance during inference.

When it comes to practical applicability, the packet rate model should be discarded. Due to its inherent limitations, it can only classify a fraction of devices. On the other hand, the three simple packet models can classify 100% of devices as they are not subject to these limitations. Therefore, the tie-braking criteria is the presumed accuracy during inference.

Here, the two neural networks have an intrinsic advantage over the decision tree. The softmax activation produces a confidence output for every class, i.e., the model predicts the likelihood of a sample belonging to the respective classes. This allows for something called softmax thresholding. If the confidence is similar across all classes, the model is uncertain about which class the sample belongs to. This can be exploited to detect potentially misclassified samples. In other words, if the neural network indicates uncertainty with equal confidence across all classes, it is possible to discard such samples as the classification is most likely incorrect. This discarding can happen based on a confidence threshold, hence the name thresholding. If the maximum confidence is below a certain threshold (typically 90% or higher), the model is uncertain, and the sample can be discarded.

Decision trees can't do this as they are unable to predict probabilities for classes. Hence, neural networks are much better suited for real-world inference. Finally, between the plain neural network and the semi-supervised trained neural network, i.e., the self-raining classifier, the latter is presumably more robust in real-world inference. Hence, the self-training classifier is the ideal choice for inference.

7.7 Summary of Modeling

The most relevant insights from this chapter are:

- The Task-Group-Framework was used again to build a robust and flexible data pipeline.
- The valuable insights from the analysis chapter allowed for the construction of separable classes for modeling.
- The analysis of these modeling classes showed that they are indeed separable based on the features extracted.
- All machine learning models were trained on a balanced training set and evaluated on a smaller balanced test set.
- The simple packet modeling leads to high accuracies across various machine learning models. It is, in fact, possible to classify BLE devices based on their packets.
- The dataset is unnecessarily large. It would not have been needed to capture so many packets. Even with a training data set 1/64 the size of the original training data set, the accuracy is still largely the same.
- The packet rate modeling is much more difficult and can lead to inferior results compared to simple packet modeling. To increase the accuracy to an acceptable level, generated zero rows must be removed.
- From a practical point of view, the model to pick for the inference is the self-training classifier as it offers the best of two worlds: high practical applicability and presumably high accuracy due to semi-supervised training and the potential usage of softmax confidence thresholding.

Chapter 8

Inference

The final content chapter covers the inference, i.e., applying the previously generated machine learning models to a dataset collected in a real-world high-traffic environment, Zürich central station. As discussed in the previous chapter, the model picked for inference is the self-training classifier, as it is highly applicable and presumably offers best-in-class accuracy with advanced techniques such as softmax confidence thresholding.

The chapter is split into three major sections. First, the preprocessing, the actual inference, and the softmax confidence thresholding, i.e., the selection of devices, are discussed. Next, the inference result is evaluated using the plots known from the analysis chapter to get an idea of the success of the inference. Finally, some limitations and potential improvements are discussed.

8.1 Preprocessing and Device Selection

The data preprocessing for inference is equivalent to the preprocessing for modeling, except that the data is not labeled. The preprocessing pipeline is the same as for modeling. The Task-Group-Framework allows for the flexibility of omnitting the labeling step.

Next, the data is scaled with the pre-trained scaler and finally fed through the selected model. The result is a matrix with the softmax probabilities for each packet. However, this result can still be improved.

At this point, it is possible to exploit a property of BLE packets. Every BLE packet comes from a source address. For each source address, the label should be the same across all packets. Therefore, different packets from the same source address should not have differing labels. Additionally, the softmax probabilities can be exploited to detect uncertain classifications. The goal is to keep only classifications that are correct with a high degree of certainty and discard potentially wrong classifications.

Therefore, in the first step, the source addresses of the packets are joined onto the output matrix with the softmax probabilities and the predicted class label (Table 8.1). Next, the

Source Address	Softmax Confidence	Predicted Label
2E-B0-D0-63-C2-26	80%	other Device
2E-B0-D0-63-C2-26	60%	iDevice
2E-B0-D0-63-C2-26	100%	other Device
C9-2D-8B-7F-9B-A6	30%	SmartTag
C9-2D-8B-7F-9B-A6	100%	FindMy Tracker
C9-2D-8B-7F-9B-A6	40%	Tile
C9-2D-8B-7F-9B-A6	30%	SmartTag

Table 8.1: Simplified Prediction Table after joining the Source Addresses

Source Address	Predicted Label	Average Softmax Confidence	Score
2E-B0-D0-63-C2-26	other Device	90%	75%
2E-B0-D0-63-C2-26	iDevice	60%	25%
C9-2D-8B-7F-9B-A6	FindMy Tracker	100%	50%
C9-2D-8B-7F-9B-A6	SmartTag	30%	30%
C9-2D-8B-7F-9B-A6	Tile	40%	20%

Table 8.2: Prediction Table with average Softmax Confidence and Score per Class Label and Source Address

average softmax confidence is computed for every source address and class label. Additionally, for every source address and class label, the relative weighted softmax confidence is calculated. This value shall be called score and is represented by the "Score" column in the tables (Table 8.2). The idea is that the most likely class label for a source address is a class label that is relatively frequent and has a high average softmax confidence.

Next, for every source address, the class label with the maximum score is picked. The intuition behind this is that the label most likely correct is the label with a high score, i.e., a relatively frequent occurrence and high average softmax confidence. After this step, the score column can be discarded (Table 8.3).

Source Address	Predicted Label	Average Softmax Confidence
2E-B0-D0-63-C2-26	other Device	90%
C9-2D-8B-7F-9B-A6	FindMy Tracker	100%

Table 8.3: Device Table with predicted Labels and average Softmax Confidence per Source Address

Finally, the softmax confidence thresholding is applied. All source addresses where the average softmax confidence of the highest score label was below the threshold of 98% were removed. The intuition is to keep only source addresses where the label was predicted with high certainty, i.e., the label is likely correct. At this point, the softmax confidence can be discarded. The resulting device table contains all the source addresses and labels for which a prediction with high certainty is possible (Table 8.4).

In practice, this softmax confidence thresholding led to 20% of source addresses being discarded. In other words, the final device table contained 80% of all source addresses

Source Address	Predicted Label
C9-2D-8B-7F-9B-A6	FindMy Tracker

Table 8.4: Device Table with predicted Labels per Source Address

of the initial dataset. Most of the discarded source addresses were assigned the ”other Device” class label. For the evaluation of the inference, the original dataset was fed again through the first step of the modeling pipeline. Finally, the device table was joined onto the dataset with an inner join. In other words, the dataset was filtered for the source addresses in the device table, and the labels were assigned based on the labels in the device table.

8.2 Evaluation

The evaluation of the inference is not trivial, as there are no target labels against which to compare the predicted labels, so generating a confusion matrix or something similar is impossible. Therefore, a more qualitative approach was chosen by plotting and analyzing the results with the plots from the analysis chapter. The intuition is that the plots should look relatively similar to those seen during analysis. For instance, a packet labeled ”SmartTag” should contain UUIDs from Samsung and not UUIDs from other vendors.

This section is split into two parts. The first contains the qualitative analysis of the inference results, and the second contains a discussion of the results, where the quality of the classification is discussed for each class.

8.2.1 Qualitative Analysis

Some of the more relevant plots are shown for the evaluation of the inference. Most are omitted for brevity.

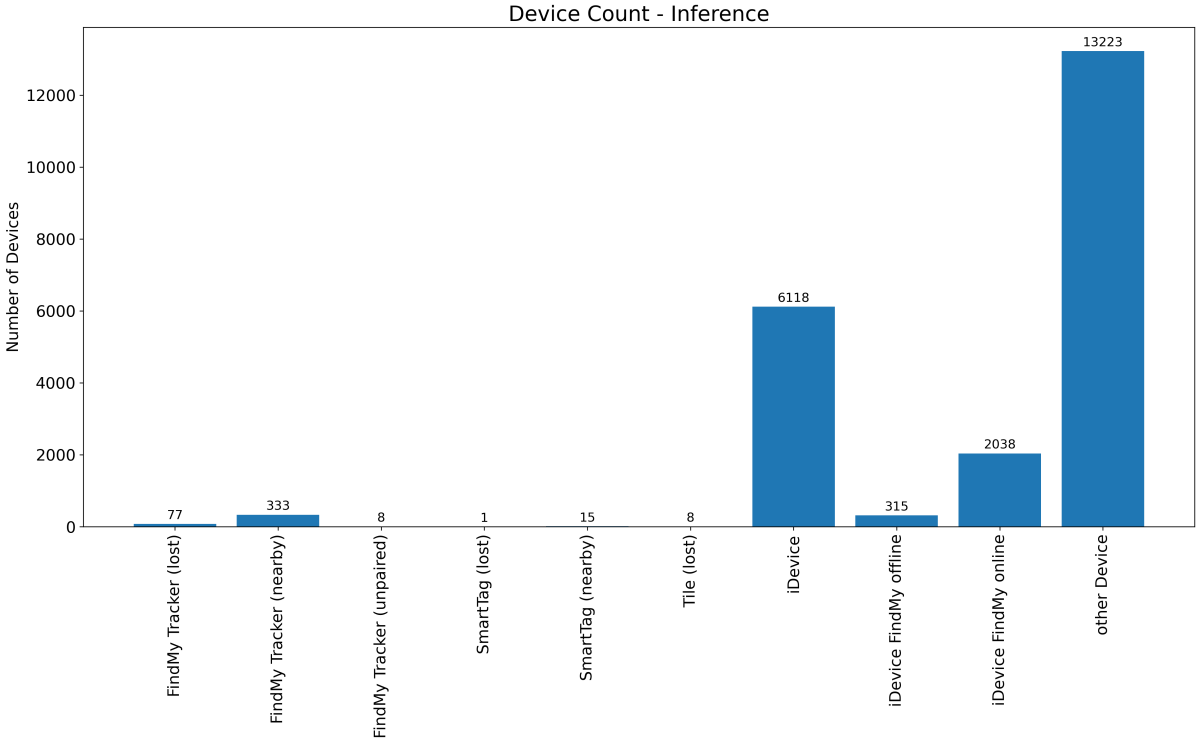

Figure 8.1: Device Count - Inference

This plot shows the number of source addresses/devices found per class label. The most interesting aspect of this plot is the relative size of the individual bars. The by far most frequent device type is ”other Device”. Well over half of the devices found are other devices. This is plausible simply given the fact that there are many, many more

non-tracking devices out there than tracking devices (even considering the fact that the iDevices are counted as tracking devices). Next in line is the "iDevice" class. As seen during analysis, the iDevices transmit many packets, most of which are continuity types other than 0x12, i.e., Find My packets. Therefore, it is perfectly plausible why the bars for the Find My packets are so much smaller.

The "FindMy online" bar is substantially higher than the "FindMy offline" bar. This can be explained by the environment of the BLE capture. At a train station, there are presumably more iDevices online than offline. Every iPhone is connected to the cellular network and, therefore, online. The remaining iDevices are iPads, MacBooks, and so on, which, for the most part, are not cellular-capable and, therefore, offline at the train station. And because there are presumably many more iPhones than MacBooks and iPads carried around at train stations, it is perfectly sensible that there are more online than offline Find My packets.

All the BLE trackers (excluding iDevices) are detected significantly less frequently. The SmartTag and Tile only have single-digit and low two-digit device counts, which is a little unexpected, to be honest. Given their high ranking in online stores, I would have expected more of these devices to be found, especially when comparing the numbers to the detections of the Find My trackers.

Find My trackers are found much more often, albeit less frequently, than iDevices. This is reasonable, as there are many, many more iDevices out there than Find My trackers. It is also reasonable that there are more Find My trackers in the "nearby" state than in the "lost" state, as most people presumably carry their iPhones with them, making the trackers "nearby". And finally, there are few Find My trackers in the "unpaired" state, presumably because most people register their trackers as soon as they take them out of the packaging box.

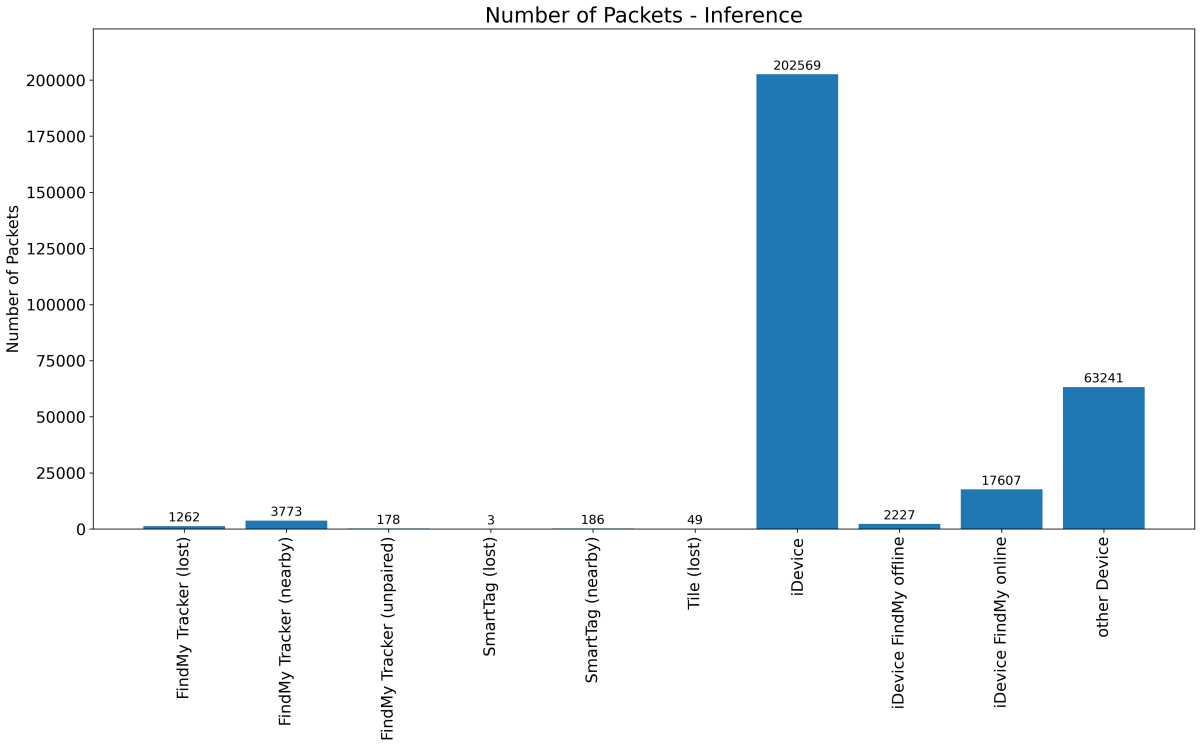

Figure 8.2: Number of Packets - Inference

The previous plot about the device count and this plot about the number of packets are both connected via the packet rate. In other words, similar relative packet rates result in similar relative heights of the bars. The higher the relative packet rate of a class, the higher the relative height of the bar in this plot.

Principally, the relative heights of the bars did not change significantly compared to the previous plot. But probably the most notable exception to this is the bar from the ”iDevice” class. In the previous plots, the ratio between ”iDevice” and ”other Device” was about 1:2 and is now roughly 3:1. In other words, the relative height of the ”iDevice” bar changed by a factor of 6. This is explainable by the simple fact that iDevices have a relatively high average packet rate compared to many other Bluetooth devices.

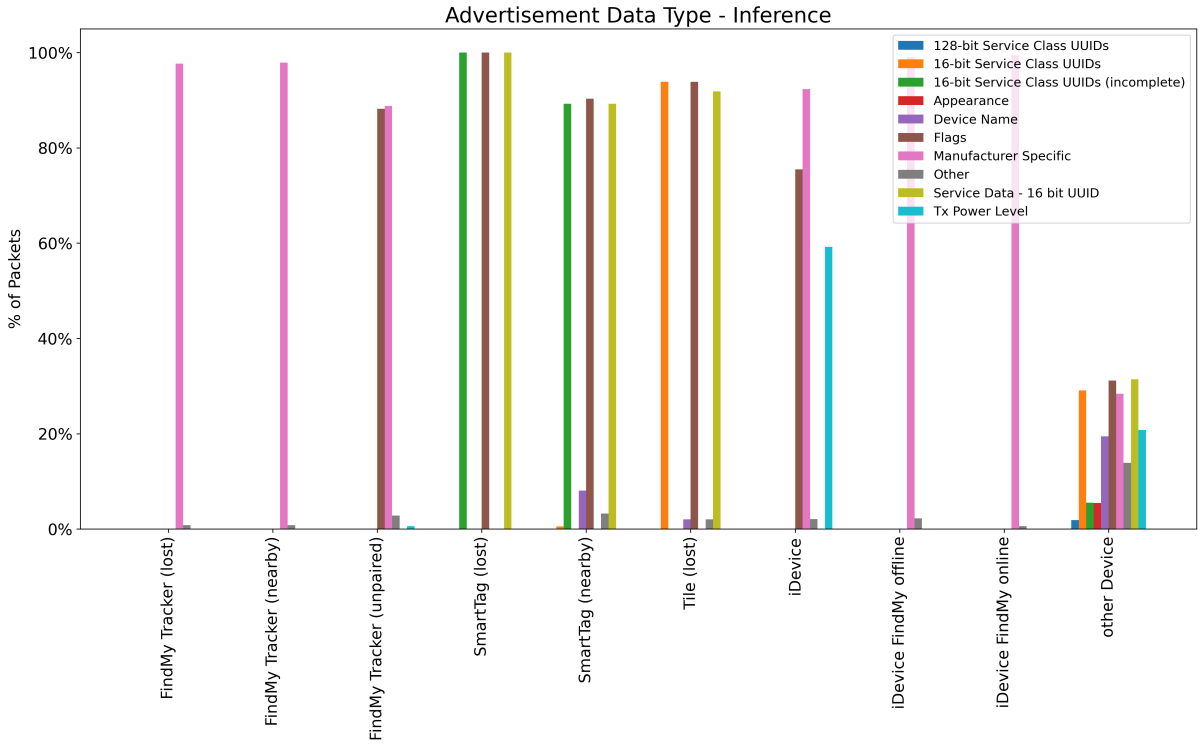

Figure 8.3: Advertisement Data Type - Inference

In the case of almost all the Find My packets (both from Find My trackers and iDevices), the distribution of the advertisement data types is as expected, with almost 100% of packets containing only manufacturer specific data. In the ”unpaired” state, however, the distribution is nowhere where it should be. The high relative presence of flags indicates that the vast majority of packets in this class are, in fact, not packets coming from a Find My tracker (in any state).

For both SamrtTag states and the Tile, the distribution of advertising data types is relatively close but not perfect. The correct advertisement data types are present in most packets, but not all. Sometimes, as in the case of the SmartTag in the ”nearby” state, incorrect advertising data types are present, such as the Device Name type.

Most interesting is the ”iDevice” class, where the distribution is surprisingly close to the distribution found in the analysis of the modeling classes. This indicates that the selection of Apple devices for modeling was not too far-fetched from what’s found in real-world environments.

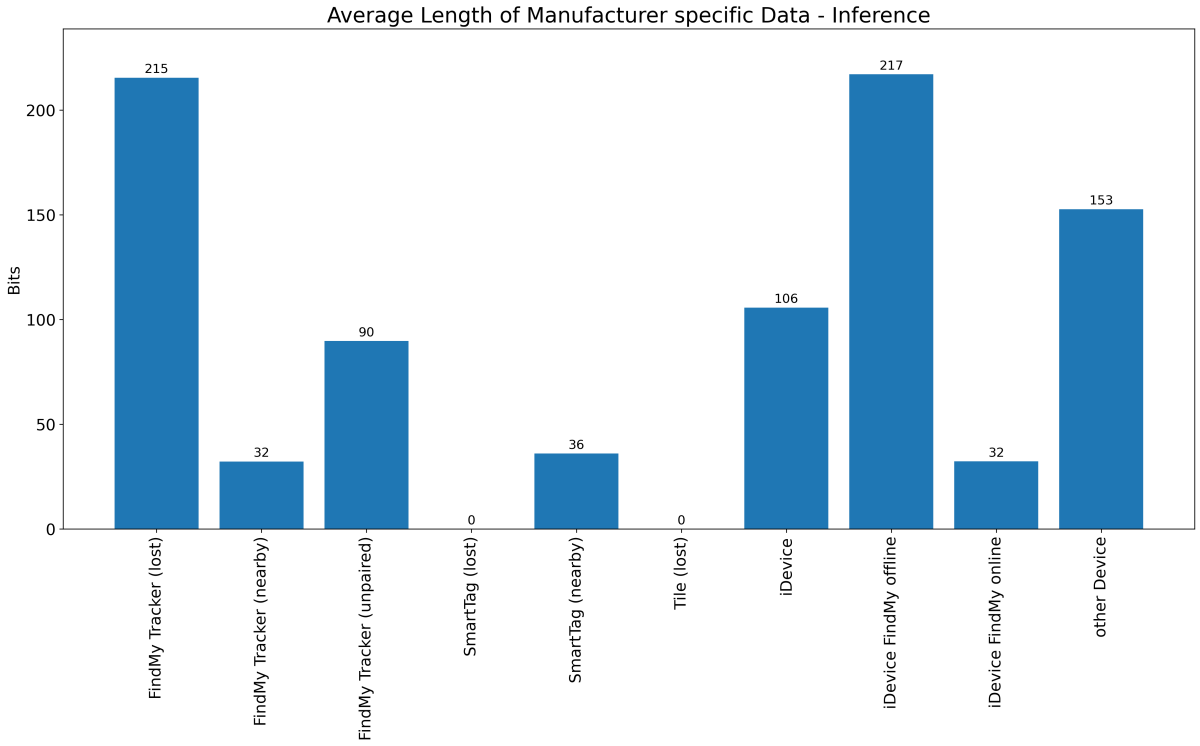

Figure 8.4: Average Length of Manufacturer specific Data - Inference

The length of the manufacturer specific data is especially interesting for Find My trackers and iDevices. For the Find My tracker in their ”lost” state and the Find My offline packets, the average length is only one bit off from where it should be. In the case of the Find My trackers in their ”nearby” state and the Find My online packets, the average value is even spot on. In the case of the Find My tracker in the ”unpaired” state, the value is nowhere where it should be. This further indicates that the detection of Find My trackers in the ”unpaired” state does not work very well.

However, the values for the SmartTag ”nearby” are most concerning. Here, the average should be zero, as this device and state never use manufacturer specific data. Remember: The average length of the manufacturer specific data is only computed over the packets where manufacturer specific data is present. However, as seen in the prior plot about advertising data types, manufacturer specific data is a rare occurrence for the SmartTag.

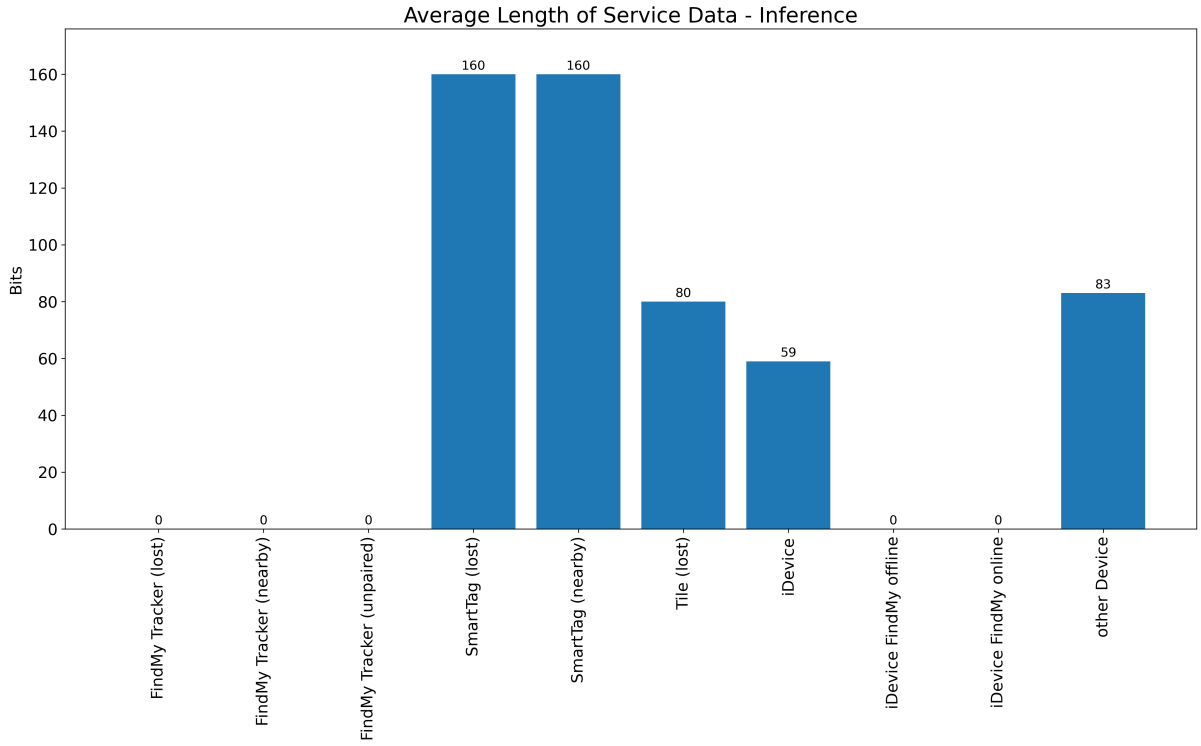

Figure 8.5: Average Length of Service Data - Inference

For both states of the SmartTag and the Tile, the average length of the Service Data is spot on, indicating that the detection of these packets works well when Service Data is present. In other words, the model learned that these devices and states are attributed to a very specific Service Data length.

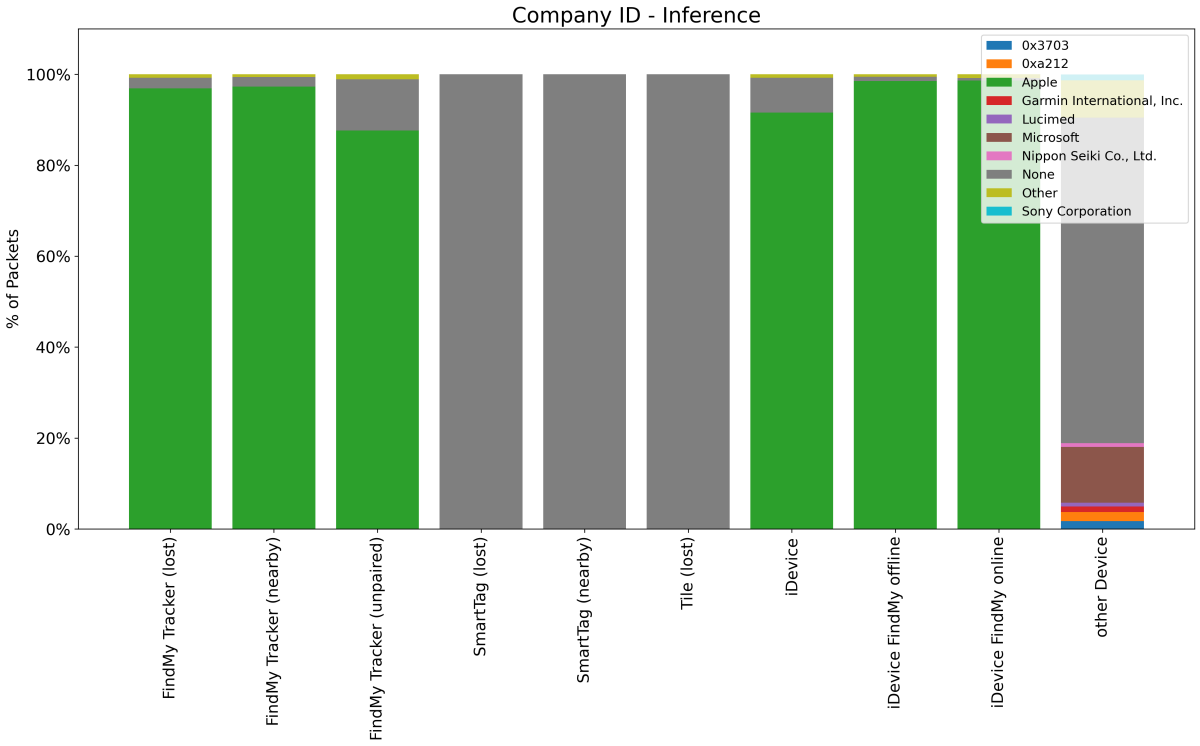

Figure 8.6: Company ID - Inference

The distribution of the company ID is mostly relevant for Find My trackers and iDevices. For these devices and states, the presence of the company ID ”Apple” should be very close to 100%. However, this is not entirely correct. In the preprocessing step, soft max confidences and source addresses were used to improve the model's accuracy. This included assigning the same class label to all packets of a source address. Sometimes, devices in the real world transmit packets other than the ones captured in laboratory-like conditions, such as SCAN_REQ or SCAN_RSP packets. Both of these cannot contain advertising data. However, these packets were also assigned one of the class labels based on the source address.

This drags down the share of packets containing the correct company ID. Hence, the percentage cannot be expected to hit the full 100%. Therefore, the observed presence of company IDs is as good as it will get. Especially in the case of the SmartTag and the Tile, the model learned that these do not contain manufacturer specific data, and hence, company IDs are not present.

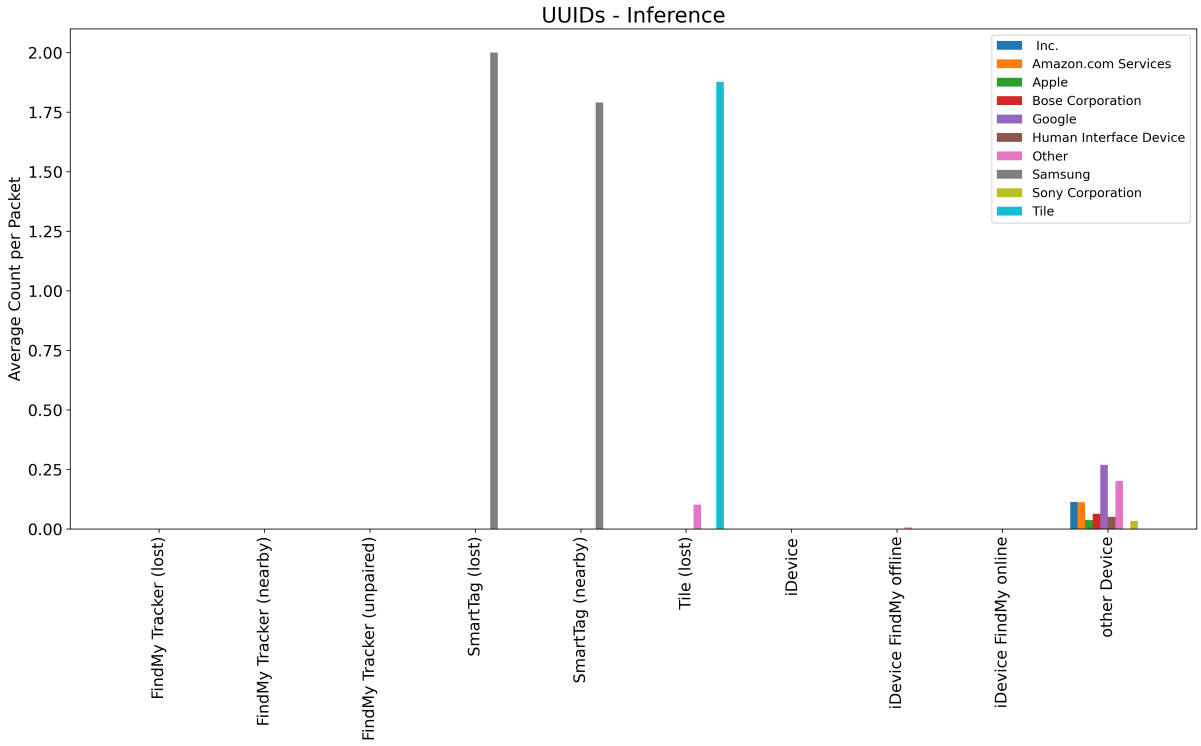

Figure 8.7: UUIDs - Inference

The presence of UUIDs is as expected in the case of the SmartTag in its ”lost” state, with an average count of 2. In the ”nearby” state, however, the bar does not quite reach an average of 2, hence the detection is not perfect. The same thing applies to the Tile tracker, even though the bar is ever so slightly closer to 2. For the Find My trackers and iDevces, the model correctly learned that these do not contain UUIDs and hardly ever assigned a packet with UUIDs to one of those classes.

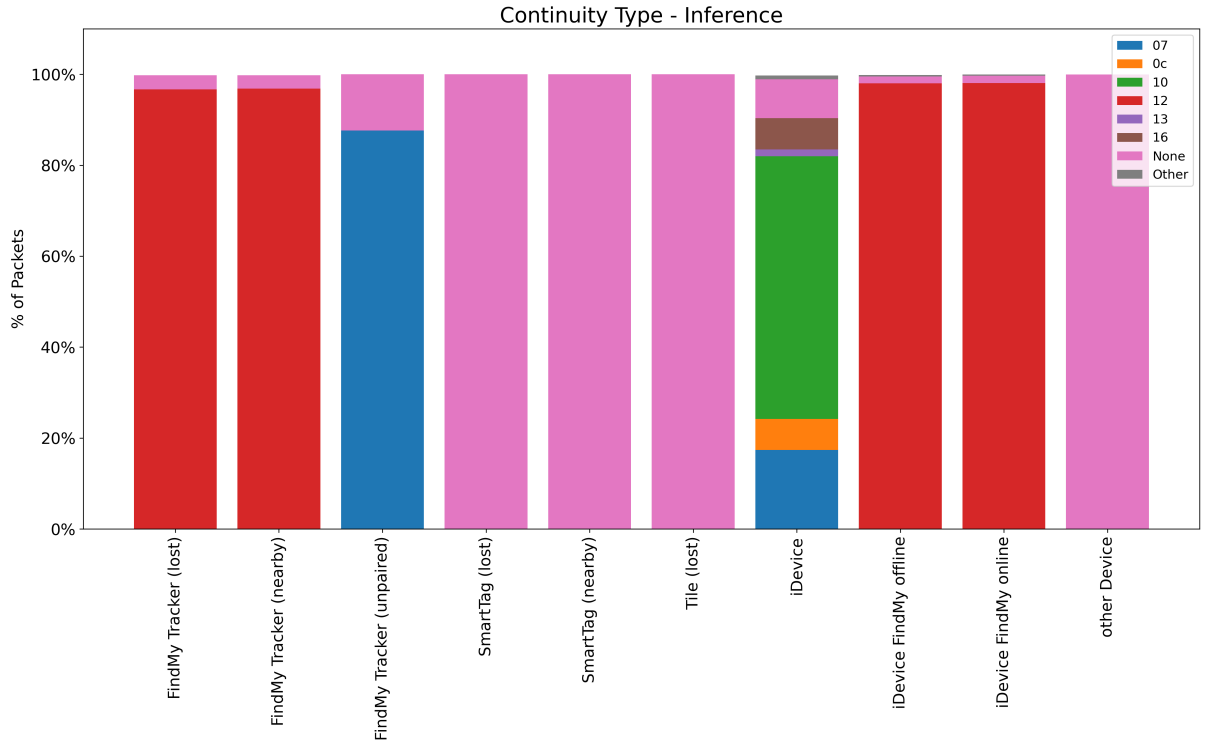

Figure 8.8: Continuity Type - Inference

The distribution of the continuity types is presumably the most interesting plot. First, the model learned that only Find My trackers and iDevices have a continuity type. In all other cases, the continuity type is always none.

Second, for all Find My packets, the continuity type is almost always 0x12. The small percentage of none type mostly stems from SCAN_REQ or SCAN_RSP packets coming from a source address transmitting Find My packets.

Third, for the Find My tracker in the ”unpaired” state, the continuity type 0x07 is present in most packets, which indicates a correct classification. However, this is misleading, as previous plots have shown that most of the packets in this class are probably incorrectly classified.

Finally, the ”other Device” class always has a continuity type of none, which is a strong indicator that the classification of this class works rather well.

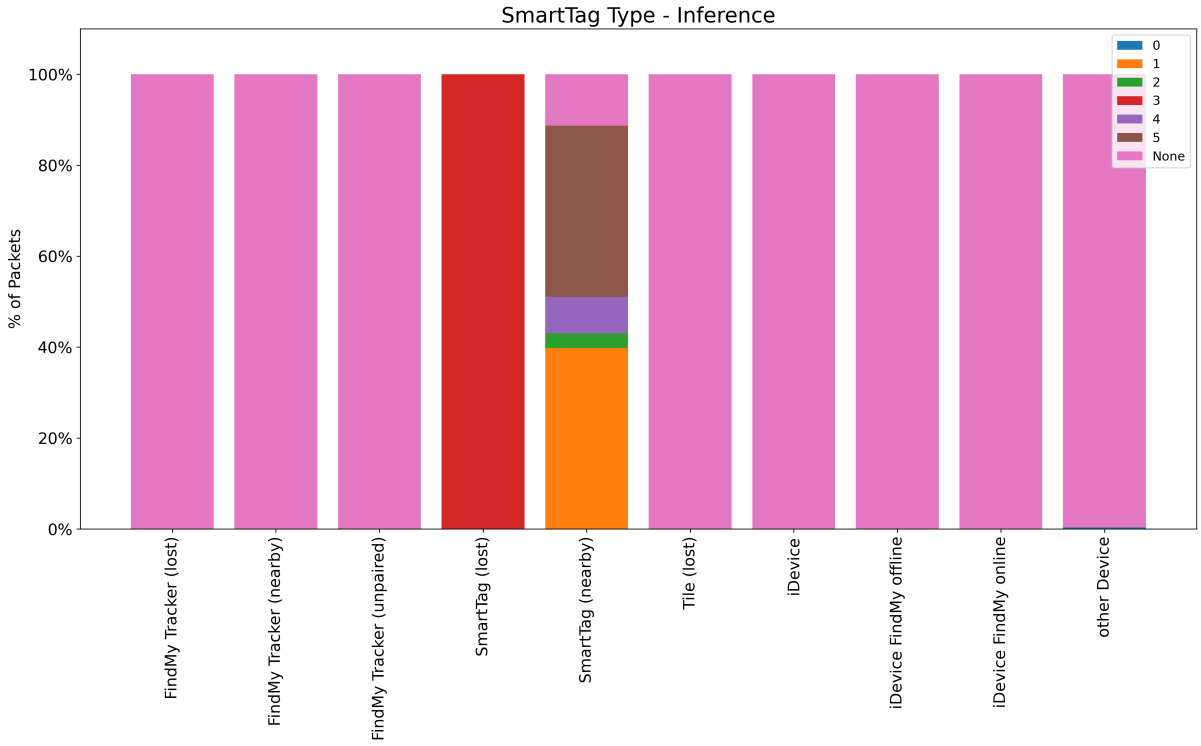

Figure 8.9: SmartTag Type - Inference

The distribution of the SmartTag types is most interesting for the SmartTag. For all other classes, the SmartTag state is always none, as it should be. In the SmartTag's "lost" state, the type is always correct: 3. In the "nearby" state, however, there are a multitude of types that should not be in there. The SmartTag type should be 5 for all packets in the "nearby" state. This issue presumably stems from insufficient training data. The SmartTag type 1, for instance, is attributed with the "unpaired" state of the SmartTag and not the "nearby" state. The data from the "unpaired" state was not used to train the model. Hence, the model could not learn that the SmartTag type 1 is not associated with the "nearby" state.

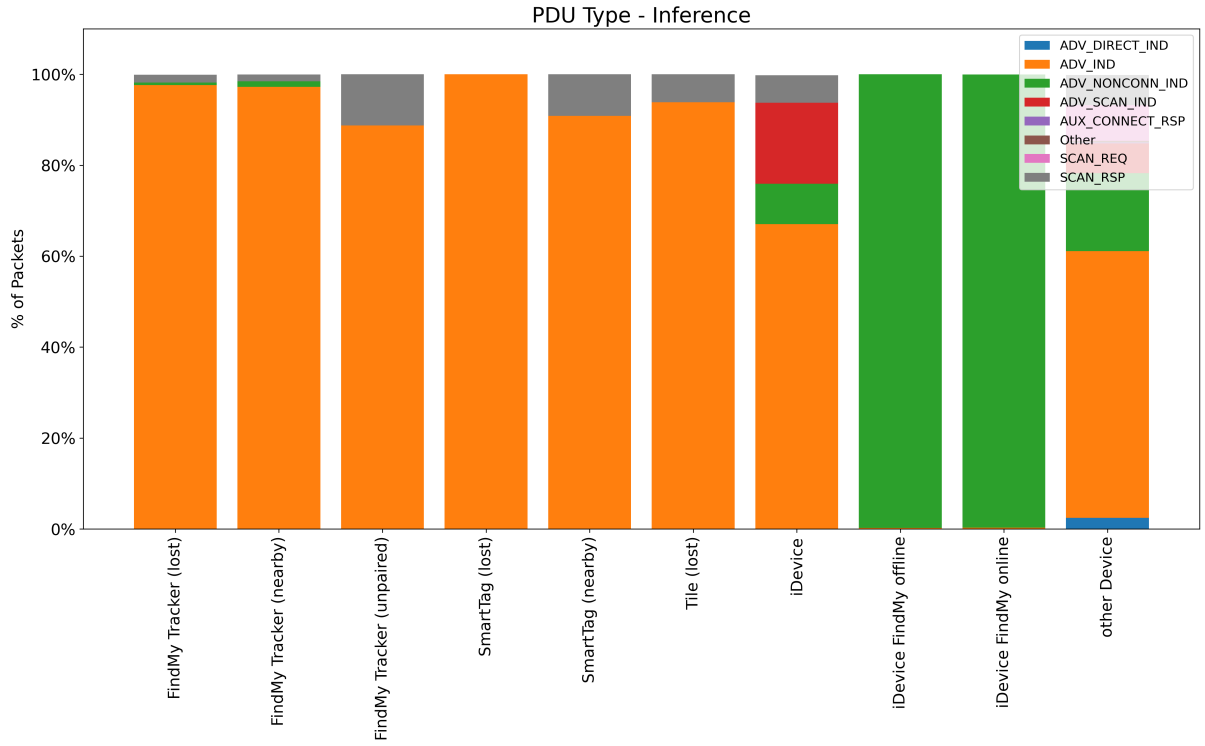

Figure 8.10: PDU Type - Inference

The distribution of PDU types reveals two things. First, in most cases, the PDU type is correct. For all states of all conventional BLE trackers, most packets are of PDU type ADV_IND as they should be. The same applies to the Find My packets from iDevices, where the PDU type is correct for almost all packets.

However, the most interesting is the presence of the SCAN_RSP PDU type in some of the classes. These packets are mostly assigned based on the softmax confidences and the source addresses in the preprocessing and are not ”detected” by the model, as there are hardly any such packets in the training data set. This is a great example of the difference between data collected in the laboratory and data collected in real-world environments. These SCAN_RSP packets are also the reason why, for instance, the distribution of company IDs does not look as expected.

8.2.2 Discussion of Results

In summary, the classification of BLE packets in the real world works in most cases. On a per-class basis, the result can be interpreted as follows:

- **FindMy Tracker (lost):** The classification is correct for almost all packets. This is especially clear when looking at the PDU type, the continuity type, and the length of the manufacturer specific data.
- **FindMy Tracker (nearby):** The classification is correct for almost all packets. This is especially clear when looking at the PDU type, the continuity type, and the length of the manufacturer specific data.
- **FindMy Tracker (unpaired):** The classification is almost never correct, as shown evidently by the high presence of the advertising data type flag. Most of these packets should be in the garbage class "iDevice", as indicated by the continuity type 0x07.
- **SmartTag (lost):** The classification is correct for all packets. This is especially evident when looking at the SmartTag type, which is always correct. However, given the low number of 3 classified packets, this result should be taken with a grain of salt.
- **SmartTag (nearby):** The classification is correct for about half of the packets. This is most evidently seen by the distribution of the SmartTag type. In many cases, the packet belongs to a SmartTag but in the "unpaired" rather than the "nearby" state. The model should better have been trained on data from this state, too.
- **Tile (lost):** The classification is correct for most of the packets. This is clear when looking at the average count of the UUIDs and the presence of the advertising data types, the most notable features of the Tile.
- **iDevice:** The classification is correct for most of the packets. Garbage classes are always difficult to evaluate by nature. However, the strong presence of continuity types other than 0x12 and the absence of these types in other classes indicate a somewhat successful classification. The exception is the Find My tracker in its "unpaired" state, where confusion took place.
- **iDevice FindMy online:** The classification is correct for almost all packets. This is evident when looking at the PDU type, the continuity type, and the length of the manufacturer specific data.
- **iDevice FindMy offline:** The classification is correct for almost all packets. This is evident when looking at the PDU type, the continuity type, and the length of the manufacturer specific data.
- **other Device:** The classification is correct for almost all packets. As with the "iDevice" class, the evaluation is challenging for a garbage class. However, none of the prominent features of the other classes, such as continuity types, SmartTag

types, certain UUIDs, or company IDs, are ever present in packets of this class. This indicates a relatively successful classification.

In conclusion, the classification of BLE devices/source addresses/packets works very well in practice with few obvious misclassifications.

8.3 Limitations and Improvements

The main limitation of the inference is the necessity of applying the softmax thresholding in the preprocessing. The model is only able to make accurate classifications for 80% of packets. For the other 20%, the classification is to be improved upon. However, as the evaluation showed, for these 80% of packets, where the classification is confident, it is also correct in the vast majority of cases. This inevitably leads to the question of how the classification of the remaining 20% could be brought up to this level. Principally speaking, there are many ways to achieve this, some of which are discussed below.

- **More training data:** Collecting more training data would certainly increase the model's accuracy. However, given the practical constraints of resources such as time or finances, this doesn't seem too viable, to be honest. A potential approach could be the generation of synthetic data in addition to the collection of real data.

- **Improved feature extraction:** Any improvements to the feature extraction are almost certainly not going to improve the result. As the analysis of the inference showed, the relevant features were extracted. The misclassifications are not related to insufficiencies on the features side. The incorrectly classified packets were clearly misclassified based on the extracted features. The model could have been able to correctly classify these packets if trained properly.

  The same logic applies to feature selection, too. The problem of misclassifications is not feature-related. In the case of categorical features, simple feature selection algorithms such as principal component analysis wouldn't work anyway. Much more advanced techniques would be necessary, such as wrapper methods. However, these would probably be very slow given the size of the feature space (ca. 30 features lead to over a billion possible feature selections).

- **More advanced models:** Improving the complexity of the models, such as hyperparameter optimization, will not help. The performance on the test set is already outstanding, with accuracies exceeding 99%. The fundamental limitation is that there is no more to learn for the models from the training data available. Hence, more training data is necessary, and optimizing any existing models is not helpful at this point. The improvements achievable with better models are negligible in the greater scheme of things and would only increase the performance on the test set and not in real-world environments.

8.4 Summary of Inference

The most relevant insights from this chapter are:

- The softmax confidences and distribution of class labels among source addresses can be used effectively to identify the correct class label for a source address. Furthermore, softmax confidence thresholding can be used to eliminate source addresses where the assigned class labels are likely incorrect, which primarily affects the "other Device" class.
- A quantitative evaluation of the inference with a confusion matrix is not possible due to the lack of target labels, so a more qualitative approach with plots was chosen.
- The qualitative evaluation showed that for the vast majority of packets/source addresses, the assigned class label is correct. The classes most affected by the incorrect classification are "FindMy (unpaired)" and "SmartTag (nearby)."
- The result could be most improved by training the models on more diverse training data, i.e., training data of more different devices. However, this could be difficult due to time and financial constraints. A potential solution could be synthetic data.

Chapter 9

Conclusion

To conclude this thesis, one can evaluate how the goals set in the introductory chapter were met.

- **Collect Bluetooth packet data from Bluetooth devices, both trackers and non-trackers.**

  BLE packets were captured extensively for various tracking and non-tracking devices. The dataset provided on GitHub is over 3.7 GB large and contains both PCAP and CSV files of the BLE capture.

- **Train a machine learning model on the collected data.**

  Numerous machine learning models following different approaches were trained on the generated dataset. The high classification accuracy of over 99% on the test sets across all models is a testament to the success of the training of the machine learning models.

- **Use the machine learning model to identify Bluetooth trackers in real-world high-traffic environments and evaluate the result.**

  The qualitative evaluation showed conclusively that identifying BLE trackers in real-world high-traffic environments with machine learning models is possible with a high degree of accuracy.

Overall, all of the goals set for this thesis have been met with overwhelming success. In my opinion, the next step for personal tracker identification in high-traffic real-world environments would be to enrich the training dataset with vast amounts of synthetic data to improve the models' ability to generalize across an even larger number of devices and, therefore, enable the model to correctly classify even more devices in real-world environments.

Bibliography

[1] "Apple tracker detect," February 2022, accessed 20-07-2024. [Online]: https://play.google.com/store/apps/details?id=com.apple.trackerdetect&hl=de

[2] A. Heinrich, M. Stute, T. Kornhuber, and M. Hollick, "Who can find my devices? security and privacy of apple's crowd-sourced bluetooth location tracking system," *Privacy Enhancing Technologies Symposium*, Vol. 3, pp. 227–245, 2021, accessed 20-07-2024. [Online]: https://doi.org/10.2478/popets-2021-0045

[3] G. LLC, "Jetzt neu: "mein gerät finden"-netzwerk," private Email received from Google, May 2024.

[4] C. S. W. Group, "Bluetooth core specification v5.4," Bluetooth Special Interest Group, Tech. Rep., January 2023, accessed 20-07-2024. [Online]: https://www.bluetooth.org/DocMan/handlers/DownloadDoc.ashx?doc_id=556599

[5] ——, "Supplement to the bluetooth core specification v9," Bluetooth Special Interest Group, Tech. Rep., December 2019, accessed 25-07-2024. [Online]: https://www.bluetooth.org/docman/handlers/DownloadDoc.ashx?doc_id=480305

[6] J. Wong, "Advertising payload format on ble," August 2019, accessed 20-07-2024. [Online]: https://jimmywongiot.com/2019/08/13/advertising-payload-format-on-ble/

[7] S. Malakar, S. Goswami, B. Ganguli, A. Chakrabarti, S. S. Roy, K. Boopathi, and A. G. Rangaraj, "Designing a long short-term network for short-term forecasting of global horizontal irradiance," *SN Appl. Sci.*, Vol. 3, March 2021, accessed 21-07-2024. [Online]: https://doi.org/10.1007/s42452-021-04421-x

[8] L. Bai, L. Yao, S. S. Kanhere, X. Wang, and Z. Yang, "Automatic device classification from network traffic streams of internet of things," *Conference on Local Computer Networks (LCN)*, Vol. 43, 2018, accessed 20-07-2024. [Online]: https://doi.org/10.1109/LCN.2018.8638232

[9] J. Liao, "Bluetooth low energy device classifier," Master's thesis, Universität Zürich, Switzerland, August 2023, accessed 20-07-2024. [Online]: https://www.merlin.uzh.ch/publication/show/24042

[10] A. Catley, "Apple airtag reverse engineering," February 2022, accessed 20-07-2024. [Online]: https://adamcatley.com/AirTag.html

[11] A. Inc., "Continuity features and requirements on apple devices," May 2024, accessed 26-07-2024. [Online]: https://support.apple.com/en-us/108046

[12] F. research group, "An apple continuity protocol reverse engineering project," September 2023, accessed 20-07-2024. [Online]: https://github.com/furiousMAC/continuity

[13] T. Yu, J. Henderson, A. Tiu, and T. Haines, "Privacy analysis of samsung's crowd-sourced bluetooth location tracking system," October 2022, accessed 20-07-2024. [Online]: https://arxiv.org/abs/2210.14702

[14] A. Stevens, "Smart phone studie 2020," 'November' 2020, accessed 20-07-2024. [Online]: https://www.comparis.ch/-/media/images%202nd%20level%20page/download-center/smartphone-report-2020/comparis_smartphonestudie_2020_de.pdf

[15] N. S. ASA, "nrf sniffer for bluetooth le v4.1.0," Nordic Semiconductor ASA, Tech. Rep., 2024, accessed 26-07-2024. [Online]: https://infocenter.nordicsemi.com/pdf/nRF_Sniffer_BLE_UG_v4.1.0.pdf

[16] M. Woolley, "The bluetooth low energy primer v1.0.4," Bluetooth Special Interest Group, Tech. Rep., June 2022, accessed 20-07-2024. [Online]: https://www.bluetooth.com/wp-content/uploads/2022/05/Bluetooth_LE_Primer_Paper.pdf

Abbreviations

BLE	Bluetooth Low Energy
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
CRISP-DM	Cross-industry standard process for data mining
CSV	Comma-separated Values
CT	Continuity Type
GPS	Global Positioning System
ID	Identifier
IoT	Internet of Things
IP	Internet Protocol
MLP	Multi Layer Perceptron
PCAP	Packet Capture
PDU	Protocol Data Unit
RELU	Rectified Linear Unit
RSSI	Received Signal Strength Indicator
ST	SmartTag Type
USB	Universal Serial Bus
UUID	Universally Unique Identifier

List of Figures

List of Tables

Listings

Appendix A

Repository on GitHub

The GitHub repository can be found under the following URL: `www.github.com/stsaxe/Bachelor-Thesis-Stefan-Richard-Saxer`

The GitHub repositories' directories contain the following files:

- **notebooks:** The Jupyter Notebooks that are covered in the written thesis.
- **notebooks_final_presentation:** The Jupyter Notebooks for the final presentation on 15th July 2024 (these notebooks are not covered in the written thesis).
- **plots:** All plots generated by the Jupyter Notebooks (from the notebooks folder).
- **slides:** Slides and images used in the presentations.
- **src:** Various Python functions and classes used in the Jupyter Notebooks, such as the Task-Group-Framework.
- **tables:** The classification reports for the evaluation of the machine learning models.
- **wireshark_profile:** The Wireshark profile used to export from PCAP to CSV.

The folder containing the data (including both PCAP and CSV files) can be found under the following URL on Kaggle, as it is too large for GitHub. Link to data: `www.kaggle.com/datasets/stefansaxer/ble-packets-from-tracking-devices`. Include the "data" folder as is on the top level directory to execute the Jupyter Notebooks locally.

Note: Local execution requires Python Version 3.12 or later.