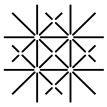

University of Basel

IOTTB: An Automation Testbed for IOT Devices

Bachelor Project

Natural Science Faculty of the University of Basel
Department of Mathematics and Computer Science
Privacy Enhancing Technologies
https://pet.dmi.unibas.ch

Examiner: Prof. Dr. Isabel Wagner
Supervisor: Valentyna Pavliv

Sebastian Lenzlinger
sebastian.lenzlinger@unibas.ch
2018-775-494

30. June 2024

Abstract

To systematically study and assess the privacy and security implications of IoTdevices, it is crucial to have a reliable method for conducting experiments and extracting meaningful data in a reproducible manner. This necessitates the development of a system —referred to as a "testbed"— that includes all the necessary tools, definitions, and automated environment setup required for conduction reproducible experiments on IoT devices.
In this project, I aim to design and implement a testbed that automates and standardizes the collection and processing of network data from IoT devices. The outcome of this project is a Python package that facilitates these tasks, providing a foundation for reproducible IoT device experiments.

Table of Contents

1 Introduction

Internet of Things (IoT) devices are becoming increasingly prevalent in modern homes, offering a range of benefits such as controlling home lighting, remote video monitoring, and automated cleaning [14]. These conveniences are made possible by the sensors and networked communication capabilities embedded within these devices. However, these features also pose significant privacy and security risks [13]. IoT devices are often integrated into home networks and communicate over the internet with external servers, potentially enabling surveillance or unauthorized data sharing without the user's knowledge or consent [15]. Moreover, even in the absence of malicious intent by the manufacturer, these devices are still vulnerable to programming bugs and other security failures [8].

Security researchers focused on the security and privacy of such IoT devices rely on various utilities and tools for conducting research. These tools are often glued together in scripts with arbitrary decisions about file naming and data structuring. Such impromptu scripts typically have a narrow range of application, making them difficult to reuse across different projects. Consequently, useful parts are manually extracted and incorporated into new scripts for each project, exacerbating the problem.

This approach leads to scattered data, highly tailored scripts, and a lack of standardized methods for sharing or reproducing experiments. The absence of standardized tools and practices results in inconsistencies in data collection and storage, making it difficult to maintain compatibility across projects. Furthermore, the lack of conventions about file naming and data structuring leads to issues in finding and accessing the data. For research groups, these issues are further compounded during the onboarding of new members, who must navigate this fragmented landscape and often create their own ad-hoc solutions, perpetuating the cycle of inefficiency and inconsistency.

To systematically and reproducibly study the privacy and security of IoT devices, an easy-to-use testbed that automates and standardizes various aspects of experimenting with IoT devices is needed.

1.1 Motivation

The primary motivation behind this project is to address the challenges faced by security researchers in the field of IoT device security and privacy. The scattered nature of data, the lack of standardized tools, and the ad-hoc methods used for data collection or processing, are an obstacle for researchers who want to produce valid and reproducible results [11]. A standardized testbed, enabling a more systematic approach to collecting and analyzing network data from IoT devices, can help make tedious and error-prone aspects of conducting experiments on IoT devices more bearable, while at the same time enhancing the quality of the data, by adhering to interoperability standards by establishing data collection and storage standards. This bachelor project is specifically informed by the needs of the PET research group at the University of Basel, who will utilize it to run IoT device experiments, and as a foundation to build more extensive tooling.

1.2 Goal

The goal of this project is to design and implement a testbed for IoT device experiments. To aid reproducibility, there are two main objectives:
First, the testbed should automate key aspects of running experiments with IoT devices, particularly the setup and initialization of data collection processes as well as some basic post-collection data processing.
Secondly, the testbed should standardize how data from experiments is stored. This includes standardizing data and metadata organization, establishing a naming scheme, and defining necessary data formats. A more detailed description to how this is adapted for this project follows in Chapter 3.

1.3 Outline

This report documents the design and implementation of an IoT testbed. In the remainder of the text, the typographically formatted string "IOTTB " refers to this projects' conception of testbed, whereas "`iottb` " specifically denotes the Python package which is the implementation artifact from this project.
This report outlines the general goals of a testbed, details the specific functionalities of IOTTB , and explains how the principles of automation and standardization are implemented. We begin by giving some background on the most immediately useful concepts. Chapter 3 derives requirements for IOTTB starting from first principles and concludes by delineating the scope considered for implementation, which is described in Chapter 4. In Chapter 5 we evaluate IOTTB , and more specifically, the `iottb` software package against the requirements stated in Chapter 3. We conclude in Chapter 6 with an outlook on further development for IOTTB .

2
Background

This section provides the necessary background to understand the foundational concepts related to IoT devices, testbeds, and data principles that inform the design and implementation of IOTTB .

2.1 Internet of Things

The IoT refers to the connection of "things" other than traditional computers to the internet. The decreasing size of microprocessors has enabled their integration into smaller and smaller objects. Today, objects like security cameras, home lighting, or children's toys may contain a processor and embedded software that enables them to interact with the internet. The Internet of Things encompasses objects whose purpose has a physical dimension, such as using sensors to measure the physical world or functioning as simple controllers. When these devices can connect to the internet, they are considered part of the Internet of Things and are referred to as **IoT devices** (see Silverio-Fernández et al. [16] and Firouzi et al. [9]).

2.2 Testbed

A testbed is a controlled environment set up to perform experiments and tests on new technologies. The concept is used across various fields such as aviation, science, and industry. Despite the varying contexts, all testbeds share the common goal of providing a stable, controlled environment to evaluate the performance and characteristics of the object of interest.
Examples of testbeds include:

1. **Industry and Engineering**: In industry and engineering, the term *platform* is often used to describe a starting point for product development. A platform in this context can be considered a testbed where various components and technologies are integrated and tested together before final deployment.

2. **Natural Sciences**: In the natural sciences, laboratories serve as testbeds by providing controlled environments for scientific experiments. For example, climate chambers are

used to study the effects of different environmental conditions on biological samples (e.g., in Vaughan et al. [18]). Another example is the use of wind tunnels in aerodynamics research to simulate and study the effects of airflow over models of aircraft or other structures.

3. **Computing**: In computing, specifically within software testing, a suite of unit tests, integrated development environments (IDEs), and other tools could be considered as a testbed. This setup helps in identifying and resolving potential issues before deployment. By controlling parameters of the environment, a testbed can ensure that the software behaves as expected under specified conditions, which is essential for reliable and consistent testing.

4. **Interdisciplinary**: Testbeds can take on considerable scales. For instance, in Hahn et al. [12] provides insight into the aspects of a testbed for a smart electric grid. This testbed is composed out of multiple systems, — an electrical grid, internet, and communication provision — which in their own right are already complex environments. The testbed must, via simulation or prototyping, provide control mechanisms, communication, and physical system components.

2.3 FAIR Data Principles

The *FAIR Data Principles* were first introduced by Wilkinson et al. [19] with the intention to improve the reusability of scientific data. The principles address **F**indability, **A**ccessibility, **I**nteroperability, and **R**eusability. Data storage designers may use these principles as a guide when designing data storage systems intended to hold data for easy reuse. For a more detailed description, see [2].

2.4 Network Traffic

Studying IoT devices fundamentally involves understanding their network traffic behavior. This is because network traffic contains (either explicitly or implicitly embedded in it) essential information of interest. Here are key reasons why network traffic is essential in the context of IoT device security:

1. **Communication Patterns**: Network traffic captures the communication patterns between IoT devices and external servers or other devices within the network. By analyzing these patterns, researchers can understand how data flows in and out of the device, which is critical for evaluating performance and identifying any unauthorized communications or unintended leaking of sensitive information.

2. **Protocol Analysis:** Examining the protocols used by IoT devices helps in understanding how they operate. Different devices might use various communication protocols, and analyzing these can reveal insights into their compatibility, efficiency, and security. Protocol analysis can also uncover potential misconfigurations or deviations from expected behavior.

3. **Flow Monitoring:** Network traffic analysis is a cornerstone of security research. It allows researchers to identify potential security threats such as data breaches, unauthorized access, and malware infections. By monitoring traffic, one can detect anomalies that may indicate security incidents or vulnerabilities within the device.

4. **Information Leakage**: IoT devices are often deployed in a home environment and connect to the network through wireless technologies [14]. This allows an adversary to passively observe traffic. While often this traffic is encrypted, the network flow can leak sensitive information, which is extracted through more complex analysis of communication traffic and Wi-Fi packets [10], [15]. In some cases, the adversary can determine the state of the smart environment and their users [8].

2.5 (Network) Packet Capture

Network *packet capture* [1] fundamentally describes the act or process of intercepting and storing data packets traversing a network. It is the principal technique used for studying the behavior and communication patterns of devices on a network. For the reasons mentioned in Section 2.4, packet capturing is the main data collection mechanism used in IoT device security research, and also the one considered for this project.

2.6 Automation Recipes

) Automation recipes can be understood as a way of defining a sequence of steps needed for a process. In the field of machine learning, *Collective Mind*[2] provides a small framework to define reusable recipes for building, running, benchmarking and optimizing machine learning applications. A key aspect of these recipes some platform-independent, which has enabled wider testing and benchmarking of machine learning models. Even if a given recipe is not yet platform independent, it can be supplemented with user-specific scripts which handle the platform specifics. Furthermore, it is possible to create a new recipe from the old recipe and the new script, which, when made accessible, essentially has extended the applicability of the recipe Friess [10]. Automation recipes express the fact that some workflow is automated irrespective of the underlying tooling. A simple script or application can be considered an recipe (or part of)

[1] also known as *packet sniffing*, *network traffic capture*, or just *sniffing*. The latter is often used when referring to nefarious practices.

[2] https://github.com/mlcommons/ck

3
Adaptation

In this chapter, we outline the considerations made during the development of the IoT testbed, IOTTB . Starting from first principles, we derive the requirements for our testbed and finally establish the scope for IOTTB . The implemented testbed which results from this analysis, the software package `iottb` , is discussed in Chapter 4.

3.1 Principal Objectives

The stated goal for this bachelor project (see Section 1.2), is to create a testbed for IoT devices, which automates aspects of the involved workflow, with the aim of increasing reproducibility, standardization, and compatibility of tools and data across project boundaries. We specify two key objectives supporting this goal:

Objective 1 **Automation Recipes:** The testbed should support specification and repeated execution of important aspects of experiments with IoT devices, such as data collection and analysis (see [11])

Objective 2 **FAIR Data Storage:** The testbed should store data in accordance with the FAIR [2] principles.

3.2 Requirements Analysis

In this section, we present the results of the requirements analysis based on the principal objectives. The requirements derived for *Objective 1* are presented in Table 3.1. Table 3.2 we present requirements based on *Objective 2*.

Table 3.1: Automation Recipes Requirements

R1.1 **Installation of Tools**: Support installation of necessary tools like *mitmproxy* [3], *Wireshark* [7] or *tcpdump* [6]).

Reasoning: There are various tools used for data collection and specifically packet capture. Automating the installation of necessary tools ensures that all required software is available and configured correctly without manual intervention. This reduces the risk of human error during setup and guarantees that the testbed environment is consistently prepared for use. Many platforms, notably most common Linux distributions, come with package managers which provide a simple command-line interface for installing software while automatically handling dependencies. This allows tools to be quickly installed, making it a *lower priority* requirement for IOTTB .

R1.2 **Configuration and Start of Data Collection**: Automate the configuration and start of data collection processes. Specific subtasks include:

a) Automate wireless hotspot management on capture device.

b) Automatic handling of network capture, including the collection of relevant metadata.

Reasoning: Data collection is a central step in the experimentation workflow. Configuration is time-consuming and prone to error, suggesting automating this process is useful.As mentioned in Section 1.1, current practices lead to incompatible data and difficult to reuse scripts. Automating the configuration and start of data collection processes ensures a standardized approach, reducing the potential for user error and thereby increasing data compatibility and efficient use of tools. Automating this process must be a central aspect of IOTTB .

R1.3 **Data Processing**: Automate data processing tasks.

Reasoning: Some network capture tools produce output in a binary format. To make the data available to other processes, often the data must be transformed in some way. Data processing automation ensures that the collected data is processed uniformly and efficiently, enhancing it reusability and interoperability. Processing steps may include cleaning, transforming, and analyzing the data, which are essential steps to derive meaningful insights. Automated data processing saves time and reduces the potential for human error. It ensures that data handling procedures are consistent, which is crucial for comparing results across different experiments and ensuring the validity of findings.

R1.4 **Reproducibility**: Ensure that experiments can be repeated with the same setup and configuration.

Reasoning: A precondition to reproducible scientific results is the ability to run experiments repeatedly with all relevant aspects are set up and configured identically.

R1.5 **Execution Control**: Provide mechanisms for controlling the execution of automation recipes (e.g., start, stop, status checks).

Reasoning: Control mechanisms are essential for managing the execution of automated tasks. This includes starting, stopping, and monitoring the status of these tasks to ensure they are completed successfully.

R1.6 **Error Handling and Logging**: Include robust error handling and logging to facilitate debugging to enhance reusability.

Reasoning: Effective error handling and logging improve the robustness and reliability of the testbed.Automation recipes may contain software with incompatible logging mechanisms. To facilitate development and troubleshooting, a unified and principled logging important for IOTTB .

R1.7 **Documentation**: Provide clear documentation and examples for creating and running automation recipes.

Table 3.2: FAIR Data Storage Requirements

R2.1 **Data and Metadata Inventory**: IOTTB should provide an inventory of data and metadata that typically need to be recorded (e.g., raw traffic, timestamps, device identifiers).

Reasoning: Providing a comprehensive inventory of data and metadata ensures that data remains findable after collection. Including metadata increases interpretability and gives context necessary for extracting reproducible results.

R2.2 **Data Formats and Schemas**: Define standardized data formats and schemas.

Reasoning: Standardized data formats and schemas ensure consistency and interoperability.

R2.3 **File Naming and Directory Hierarchy**: Establish clear file naming conventions and directory hierarchies. for organized data storage.

Reasoning: This enhances findability and accessibility.

R2.4 **Data Preservation Practices**: Implement best practices for data preservation, including recommendations from authoritative sources like the Library of Congress [5].

Reasoning: Implementing best practices for data preservation can mitigate data degradation and ensures integrity of data over time. This ensures long-term accessibility and reusability.

R2.5 **Accessibility Controls**: Ensure data accessibility with appropriate permissions and access controls.

R2.6 **Interoperability Standards**: Use widely supported formats and protocols to facilitate data exchange and interoperability.

R2.7 **Reusability Documentation**: Provide detailed metadata to support data reuse by other researchers.

We return to these when we evaluate IOTTB in Chapter 5.

3.3 Scope

This section defines the scope of the testbed IOTTB . To guide the implementation of the software component of this bachelor project, `iottb` , we focus on a specific set of requirements that align with the scope of a bachelor project. While the identified requirements encompass a broad range of considerations, we have prioritized those that are most critical to achieving the primary objectives of the project.
For this project, we delineate our scope regarding the principal objectives as follows:

- *Objective 1*: `iottb` focuses on complying with *R1.2*, *R1.4*.
- *Objective 2*: `iottb` ensures FAIR data storage implicitly, with the main focus lying on *R2.2*, *R2.1*, *R2.3*.

3.3.1 Model Environment

In this section, we describe the environment model assumed as the basis for conduction IoT device experiments. This mainly involves delineating the network topology. Considerations

are taken to make this environment, over which the `iottb` testbed software has no control, easy reproducible [17].

We assume that the IoT device generally requires a Wi-Fi connection. This implies that the environment is configured to reliably capture network traffic without disrupting the IoT device's connectivity. This involves setting up a machine with internet access (wired or wireless) and possibly one Wi-Fi card supporting AP mode to act as the Access Point (AP) for the IoT device under test [20]. Additionally, the setup must enable bridging the IoT-AP network to the internet to ensure IoT device.

Specifically, the assumed setup for network traffic capture includes the following components:

1. **IoT Device:** The device under investigation, connected to a network.
2. **Capture Device:** A computer or dedicated hardware device configured to intercept and record network traffic. This is where `iottb` runs.
3. **Wi-Fi AP :** The AP through which the IoT device gets network access.
4. **Router/ Internet gateway:** The network must provide internet access.
5. **Switch or software bridge:** At least either a switch or an Operating System (OS) with software bridge support must be available to be able to implement one of the setups described in Fig. 3.1 and Fig. 3.2.
6. **Software:** tcpdump is needed for network capture.

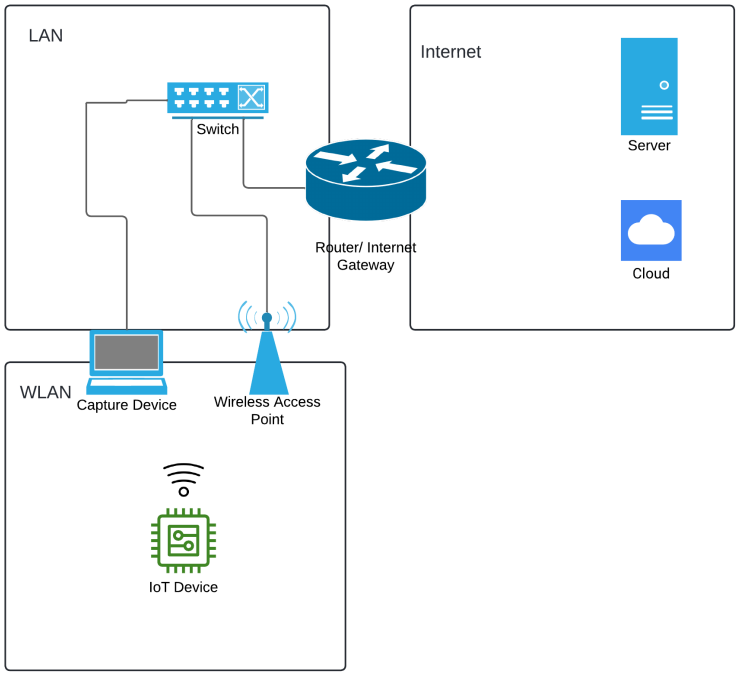

Figure 3.1: Capture setup with separate Capture Device and AP

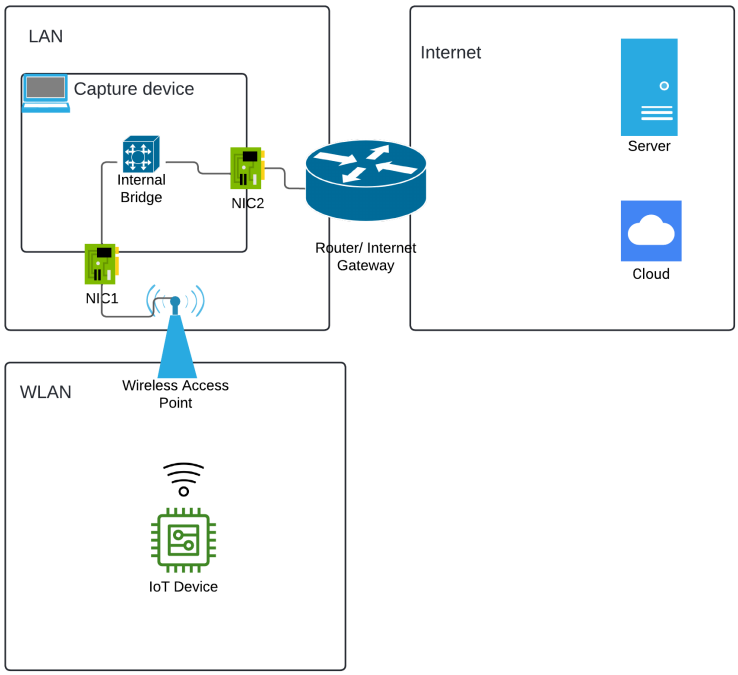

Figure 3.2: Capture setup where the capture device doubles as the AP for the IoT device.

4

Implementation

This chapter discusses the implementation of the IoT device testbed, IOTTB which is developed using the Python programming language. This choice is motivated by Python's wide availability and the familiarity many users have with it, thus lowering the barrier for extending and modifying the testbed in the future. The testbed is delivered as a Python package and provides the `iottb` command with various subcommands. A full command reference can be found at Appendix C.
Conceptually, the software implements two separate aspects: data collection and data storage. The IOTTB database schema is implicitly implemented by `iottb` . Users use `iottb` mainly to operate on the database or initiate data collection. Since the database schema is transparent to the user during operation, we begin with a brief description of the database layout as a directory hierarchy, before we get into `iottb` Command Line Interface (CLI) .

4.1 Database Schema

The storage for IOTTB is implemented on top of the file system of the user. Since user folder structures provide little standardization, we require a configuration file, while gives `iottb` some basic information about the execution environment. The testbed is configured in a configuration file in JSON format, following the scheme in Listing 1. `DefaultDatabase` is a string which represents the name of the database, which is a directory in `DefaultDatabasePath` once initialized. `iottb` assumes these values during execution, unless the user specified otherwise. If the user specifies a different database location as in option in a subcommand, `DatabaseLocations` is consulted. `DatabaseLocations` is a mapping from every known database name to the full path of its parent directory in the file system. The configuration file is loaded for every invocation of `iottb` . It provides the minimal operating information. Now that we understand

```
{
    "DefaultDatabase": "iottb.db",
    "DefaultDatabasePath": $HOME,
    "DatabaseLocations": {
        <str>: <path>
    }
}
```

Listing 1: Schema of the testbed configuration file.

4.2 High Level Description

`iottb` is invoked following the schema below. In all cases, a subcommand be specified for anything to happen. `iottb` is used from the command line and follows the following schema:

```
iottb [<global options>] <subcommand> [<subcommand options>] [<argument(s)>]
```

When `iottb` is invoked, it first checks to see if it can find the database directory in the OS users home directory[3].

4.3 Database Initialization

The IoT testbed database is defined to be a directory named `iottb.db` . Currently, `iottb` creates this directory in the user's home directory (commonly located at the path `/home/<username>` on Linux systems) the first time any subcommand is used. All data and metadata are placed under this directory. Invoking `iottb init-db` without arguments causes defaults to be loaded from the configuration file. If the file does not exist, it is created with default values following Listing 1. Else, the database is created with the default name or the user-suplied name as a directory in the file system, unless a database under that name is already registered in the `DatabaseLocaions` map. The commands described in the later sections all depend on the existence of a IOTTB database. It is neither possible to add a device nor initiate data collection without an existing database. The full command line specification can be found in Appendix C.1.1. Once a database is initialized, devices may be added to that database.

4.4 Adding Devices

Before we capture the traffic of a IoT device, `iottb` demands that there exists a dedicated directory for it. We add a device to the database by passing a string representing the name of the device to the `add-device` subcommand. This does two things:

1. A python object is initialized from the class as in Listing 2
2. A directory for the device is created as `<db-path>/<device_canonical_name>`

[3] Default can be changed

3. A metadata file `device_metadata.json` is created and placed in the newly created directory. This file is in the JSON format, and follows the schema seen in Listing 2.

```
class DeviceMetadata:
    def __init__(self, device_name, description="", model="",
            manufacturer="", firmware_version="", device_type="",
            supported_interfaces="", companion_applications="",
                save_to_file=None):
        self.device_id = str(uuid.uuid4())
        self.device_name = device_name
        cn, aliases = make_canonical_name(device_name)
        self.aliases = aliases
        self.canonical_name = cn
        self.date_added = datetime.now().isoformat()
        self.description = description
        self.model = model
        self.manufacturer = manufacturer
        self.current_firmware_version = firmware_version
        self.device_type = device_type
        self.supported_interfaces = supported_interfaces
        self.companion_applications = companion_applications
```

Listing 2: Device Metadata

The Device ID is automatically generated using a UUID to be FAIR compliant. `canonical_name` is generated by the `make_canonical_name()` function provided in Listing 8. Fields not supplied to `__init__` in Listing 2 are kept empty. The other fields in are currently not used by `iottb` itself, but provide metadata which can be used during a processing step. Optionally, one can manually create such a file with pre-set values and pass it to the setup. For example, say the testbed contains a configuration as can be seen in Listing 9

```
{
    "DefaultDatabase": "showcase",
    "DefaultDatabasePath": "/home/seb",
    "DatabaseLocations": {
        "iottb.db": "/home/seb",
        "showcase": "/home/seb"
    }
}
```

Listing 3: Directory layout after adding device 'default' and 'Roomba'

If we then add two devices `'iPhone 13 (year 2043)'` and `roomba`, the layout of the database resembles Listing 4 and, for instance, the `roomba` devices' will contain the metadata listed in Listing 5. See Appendix A.3.1 for a complete overview.

```
Database Location: /home/seb/showcase
$ tree
iphone-13/
    |--device_metadata.json
roomba/
    |-- device_metadata.json
```

Listing 4: Directory layout after adding device 'default' and 'Roomba'

```
{
    "device_id": "339de2af-c3ef-4c5d-a9c8-a03f7a65cc0a",
    "device_name": "roomba",
    "aliases": [
        "roomba"
    ],
    "canonical_name": "roomba",
    "date_added": "2024-07-01T00:54:34.715850",
    "description": "",
    "model": "",
    "manufacturer": "",
    "current_firmware_version": "",
    "device_type": "",
    "supported_interfaces": "",
    "companion_applications": "",
    "last_metadata_update": "2024-07-01T00:54:34.715859"
}
```

Listing 5: Directory layout after adding device 'default' and 'Roomba'

4.5 Traffic Sniffing

Automated network capture is a key component of `iottb` . The standard network capture is provided by the `sniff` subcommand, which wraps the common traffic capture utility *tcpdump*[6]. Appendix C.1.3 shows usage of the command.

Unless explicitly allowed by specifying that the command should run in `unsafe` mode, an IPv4, or MAC address *must* be provided. An IP addresses are only accepted in dot-decimal notation [4] and MAC addresses must specify as six groups of two hexadecimal digits[5]. Failing to provide either results in the capture being aborted. The rationale behind this is simple: they are the only way to identify the traffic of interest. Of course, it is possible to retrieve the IP or MAC after a capture. Still, the merits outweigh the annoyance. The hope is that this makes `iottb` easier to use *correctly*. For example, consider the situation, where a student is tasked with performing multiple captures across multiple devices. If the student is not aware of the need of an address for the captured data to be usable, then this policy avoids the headache and frustration of wasted time and unusable data.

To comply with *R1.2* and *R2.1*, each capture also stores some metadata in `capture_metadata.json`. Listing 6 shows the metadata files schema.

[4] e.g., 172.168.1.1
[5] e.g., 12:34:56:78:AA:BB

```
metadata = {
    'device': canonical_name,
    'device_id': device,
    'capture_id': capture_uuid,
    'capture_date_iso': datetime.now().isoformat(),
    'invoked_command': " ".join(map(str, cmd)),
    'capture_duration': delta,
    'generic_parameters': {
        'flags': flags_string,
        'kwargs': generic_kw_args_string,
        'filter': generic_filter
    },
    'non_generic_parameters': {
        'kwargs': non_generic_kw_args_string,
        'filter': cap_filter
    },
    'features': {
        'interface': interface,
        'address': address
    },
    'resources': {
        'pcap_file': str(pcap_file),
        'stdout_log': str(stdout_log_file),
        'stderr_log': str(stderr_log_file)
    },
    'environment': {
        'capture_dir': capture_dir,
        'database': database,
        'capture_base_dir': str(capture_base_dir),
        'capture_dir_abs_path': str(capture_dir_full_path)
    }
}
```

Listing 6: Metadata Stored for sniff command

```
pcap_file_full_path = capture_dir_full_path / pcap_file
stdout_log_file = f'stdout_{capture_uuid}.log'
stderr_log_file = f'stderr_{capture_uuid}.log'
```

Listing 7: Naming scheme for files created during capture.

The `device_id` is the Universally Unique Identifier (UUID) of the device for which the capture was performed. This ensures the capture metadata remains associated even if files are moved. Furthermore, each capture also gets a UUID. This UUID is used as the suffix for the PCAP file, and the log files. The exact naming scheme is given in Listing 7.

4.6 Working with Metadata

The `meta` subcommand provides a facility for manipulating metadata files. It allows users to get the value of any key in a metadata file as well as introduce new key-value pairs. However, it is not possible to change the value of any key already present in the metadata.

This restriction is in place to prevent metadata corruption.
The most crucial value in any metadata file is the `uuid` of the device or capture the metadata belongs to. Changing the `uuid` would cause `iottb` to mishandle the data, as all references to data associated with that `uuid` would become invalid. Changeing the any other value might not cause mishandling by `iottb` , but they nonetheless represent essential information about the data. Therefore, `iottb` does not allow changes to existing keys once they are set.
Future improvements might relax this restriction by implementing stricter checks on which keys can be modified. This would involve defining a strict set of keys that are write-once and then read-only.

4.7 Raw Captures

The `raw` subcommand offers a flexible way to run virtually any command wrapped in `iottb` . Of course, the intended use is with other capture tools, like *mitmproxy*mit [3], and not arbitrary shell commands. While some benefits, particularly those related to standardized capture, are diminished, users still retain the advantages of the database.
The syntax of the `raw` subcommand is as follows:

```
iottb raw <device> <command-name> "<command-options-string>" # or
iottb raw <device> "<string-executable-by-a-shell>" #
```

`iottb` does not provide error checking for user-supplied arguments or strings. Users benefit from the fact that captures will be registered in the database, assigned a `uuid`, and associated with the device. The metadata file of the capture can then be edited manually if needed.
`iottb` does not provide error checking for user-supplied arguments or strings. Users benefit from the fact that captures will be registered in the database, assigned a `uuid`, and associated with the device. The metadata file of the capture can then be edited manually if needed.
However, each incorrect or unintended invocation that adheres to the database syntax (i.e., the specified device exists) will create a new capture directory with a metadata file and `uuid`. Therefore, users are advised to thoroughly test commands beforehand to avoid creating unnecessary clutter.

4.8 Integrating user scripts

The `--pre` and `--post` options allow users to run any executable before and after any subcommand, respectively. Both options take a string as their argument, which is passed as input to a shell and launched as a subprocess. The rationale for running the process in a shell is that Python's Standard Library process management module, `subprocess`[6], does not accepts argument to the target subprocess when a single string is passed for execution.

[6] https://docs.python.org/3/library/subprocess.html

Execution is synchronous: the subcommand does not begin execution until the `--pre` script finishes, and the `--post` script only starts executing after the subcommand has completed its execution. `iottb` always runs in that order.
There may be cases where a script provides some type of relevant interaction intended to run in parallel with the capture. Currently, the recommended way to achieve this is to wrap the target executable in a script that forks a process to execute the target script, detaches from it, and returns.
These options are a gateway for more complex environment setups and, in particular, allow users to reuse their scripts, thus lowering the barrier to adopting `iottb` .

4.9 Extending and Modifying the Testbed

One of the key design goals of `iottb` is easy extensibility. `iottb` uses the Click Library [1] to handle parsing arguments. Adding a new command amounts to no more than writing a function and decorating it according to Click specification.

5 Evaluation

In this sectioned we evaluate `iottb` , paying particular attention to the requirements defined in Section 3.2.

Requirement ID	**Description**	**Status**
R1.1	Installation of Tools	Not Met
R1.2	Configuration and Start of Data Collection	↓
*R1.2*a)	Automate Wi-Fi Setup	Not Met
*R1.2*b)	Automate Data Capture	Met
R1.3	Data Processing	Partially Met
R1.4	Reproducibility	Partially Met
R1.5	Execution Control	Not Met
R1.6	Error Handling and Logging	Partially Met
R1.7	Documentation	↓
*R1.7*a)	User Manual	Met
*R1.7*b)	Developer Docs	Not Met
R2.1	Data and Metadata Inventory	Met
R2.2	Data Formats and Schemas	Met
R2.3	File Naming and Directory Hierarchy	Met
R2.4	Data Preservation Practices	Partially Met
R2.5	Accessibility Controls	Not Met
R2.6	Interoperability Standards	Fully Met
R2.7	Reusability Documentation	Met

Table 5.1: Summary of Requirements Evaluation

Table 5.1 gives an overview of the requirements introduced in Section 3.2 and our assessment of their status. It is important to note that the status "Met" does not imply that the requirement is implemented to the highest possible standard. Furthermore, this set of requirements itself can (and should) be made more specific and expanded in both detail and scope as the project evolves.

Additionally, 5.1 does not provide granularity regarding the status of individual components, which might meet the requirements to varying degrees. For example, while the requirement for data collection automation may be fully met in terms of basic functionality, advanced features such as handling edge cases or optimizing performance might still need improvement. Similarly, the requirement for data storage might be met in terms of basic file organization

but could benefit from enhanced data preservation practices.
Thus, the statuses presented in Table 5.1 should be viewed as a general assessment rather ground truth. Future work should aim to refine these requirements and their implementation to ensure that IOTTB continues to evolve and improve.
To provide a more comprehensive understanding, the following sections offer a detailed evaluation of each requirement. This detailed analysis will discuss how each requirement was addressed, the degree to which it was met, and any specific aspects that may still need improvement. By examining each requirement individually, we can better understand the strengths and limitations of the current implementation and identify areas for future enhancement.

5.1 *R1.1*: Installation of Tools

Status: Not Met

IOTTB does not install any software or tools by itself. Dependency management for Python packages is handled by installers like PIP, since the Python package declares its dependencies. Tcpdump is the only external dependency, and IOTTB checks if Tcpdump is available on the capture device. If it is not, the user is asked to install it. Our position is that generally it is a good idea to not force installation of software and allow users the freedom to choose. The added benefit to the user of a built-in installer seems low. Adding some installer to IOTTB does not promise great enough improvement in ease-of-use vis-à-vis the higher maintenance cost introduced to maintain such a module. For future work, this requirement could be removed.

5.2 *R1.2*: Configuration and Start of Data Collection

Status: Partially Met

The testbed automates the configuration and initiation of data collection processes, including wireless hotspot management and network capture. This automation reduces setup time and minimizes errors. The testbed automates some aspects of configuring and initializing the data collection process. This project focused on package capture and adjacent tasks. *R1.2*b can be considered *complete* in that packet capture is fully supported thorough Tcpdump and important metadata is saved. Depending on the setup (see Fig. 3.1 and Fig. 3.2) a Wi-Fi hotspot needs to be set up before packet capture is initiated. IOTTB does not currently implement automated setup and takedown of a hotspot on any platform, so *R1.2* a is not currently met. There are scripts for Linux systems bundled with the Python package which can be used with the `--pre` and `--post` options mentioned in Section 4.8. But to consider this task fully automated and supported, this should be built in to IOTTB itself. Furthermore, there are other data collection tools like *mitmproxy*[3] or more complicated setup tasks like setting up a routing table to allow for more capture scenarios, which are tedious tasks and lend themselves to automation. Future work should include extending the set of available automation recipes continuously. New task groups/recipe domains should be added as sub-requirements of *R1.2*. We propose the following new sub-requirement

- *R1.2*c: Testbed should implement automatic setup of NAT routing for situations where AP is connection to the capture device and a bridged setup is not supported.
- *R1.2*d: Testbed should dynamically determine which type of hotspot setup is possible and choose the appropriate automation recipe.

Extending *R1.2* means stating which data collection and adjacent recipes are wanted.

5.3 *R1.3*: Data Processing

Status: Partially Met

While the testbed includes some basic data processing capabilities, there is room for improvement. Currently, the only one recipe exists for processing raw data. IOTTB can extract a CSV file from a PCAP file. The possibilities for automation recipes which support data processing are many. Having the data in a more standardized format allows for the creation of more sophisticated feature extraction recipes with application for machine learning. Before they are available, users can still use the `--post` option with their feature extraction scripts.

5.4 *R1.4*: Reproducibility

Status: Met

Supported automation can be run with repeatedly, and used options are documented in the capture metadata. This allows others to repeat the process with the same options. So in this respect, this requirement is met. But, the current state can be significantly improved by automating the process of repeating a capture task with the same configuration as previous captures. To support this, we propose the following new subrequirement which aids the automated reproduction of past capture workflows

- *R1.4*a: The testbed should be able to read command options from a file
- Item *R1.4*b: The testbed should be able to perform a capture based on metadata files of completed captures.

Taking these requirements promises to seriously increase reproducibility.

5.5 *R1.5*: Execution Control

Status: Not Met

The testbed currently provides no controlled method to interfere with a running recipe. In most cases, `iottb` will gracefully end if the user sends the process a SIGINT, but there are no explicit protections against data corruption in this case. Furthermore, during execution, `iottb` writes to log files and prints basic information to the users' terminal. Extending this with a type of monitoring mechanism would be good steps toward complying with this requirement in the future.

5.6 R1.6: Error Handling and Logging

Status: Met

Robust error handling and logging are implemented, ensuring that issues can be diagnosed and resolved effectively. Detailed logs help maintain the integrity of experiments. It is also possible for the user to control how much output is given in the terminal. Here are four examples of the same command, with just increasing degrees of verbosity specified by the user:

5.6.1 Logging Example

Command: `iottb sniff roomba --unsafe -c 10 <verbosity>` Verbosity can be unspecified, `-v`, `-vv` or `-vvv`

```
$ iottb sniff roomba --unsafe -c 10
Testbed [I]
 Using canonical device name roomba
 Found device at path /home/seb/showcase/roomba
 Using filter None
 Files will be placed in
 ↪ /home/seb/showcase/roomba/sniffs/2024-07-01/cap0000-0214
 Capture has id 62de82ad-3aa2-460e-acd0-546e46377987
 Capture setup complete!
 Capture complete. Saved to
 ↪ roomba_62de82ad-3aa2-460e-acd0-546e46377987.pcap
 tcpdump took 2.16 seconds.
 Ensuring correct ownership of created files.
 Saving metadata.
 END SNIFF SUBCOMMAND
```

Figure 5.1: No verbosity.

On the first verbosity level, only logger warnings are now printed to the standard output. During normal execution we do not expect significantly more output. This is also true for the second verbosity level.

```
$ iottb -v|-vv sniff roomba --unsafe -c 10
<_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>
WARNING - iottb_config - DatabaseLocations are DatabaseLocationMap in the class iot
```

Figure 5.2: Only *additional* output for v or vv.

This changes once we reach the third verbosity level, because now additonally the logger level is set to "INFO". Clearly, Fig. 5.3 contains far more output than Fig. 5.2. It is possible to get even more output printed to standard output by also passing the `--debug` flag. This produces significantly more output as can be seen in Fig. A.1.

```
$ iottb -vvv sniff roomba  --unsafe -c 10
<_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>
INFO - main - cli - 48 - Starting execution.
INFO - iottb_config - __init__ - 24 - Initializing Config object
WARNING - iottb_config - warn - 21 - DatabaseLocations are DatabaseLocationMap in t
INFO - iottb_config - load_config - 57 - Loading configuration file
INFO - iottb_config - load_config - 62 - Config file exists, opening.
INFO - sniff - validate_sniff - 37 - Validating sniff...
INFO - sniff - sniff - 91 - sniff command invoked
INFO - string_processing - make_canonical_name - 20 - Normalizing name roomba
Testbed [I]
 Using canonical device name roomba
 Found device at path /home/seb/showcase/roomba
INFO - sniff - sniff - 152 - Generic filter None
 Using filter None
 Files will be placed in /home/seb/showcase/roomba/sniffs/2024-07-01/cap0003-0309
 Capture has id f1e92062-4a82-4429-996c-97bd7fa57bec
INFO - sniff - sniff - 186 - pcap file name is roomba_f1e92062-4a82-4429-996c-97bd7
INFO - sniff - sniff - 187 - stdout log file is stdout_f1e92062-4a82-4429-996c-97bd
INFO - sniff - sniff - 188 - stderr log file is stderr_f1e92062-4a82-4429-996c-97bd
INFO - sniff - sniff - 246 - tcpdump command: sudo tcpdump -# -n -vvv -c 10 -w /hom
 Capture setup complete!
 Capture complete. Saved to roomba_f1e92062-4a82-4429-996c-97bd7fa57bec.pcap
 tcpdump took 2.12 seconds.
 Ensuring correct ownership of created files.
 Saving metadata.
 END  SNIFF SUBCOMMAND
```

Figure 5.3: Caption

5.7 *R1.7*: Documentation

Status: Partially Met

For users, there is a 'Command Line Reference' (see Appendix C) which details all important aspects of operating the `iottb` CLI . Furthermore, helpful messages are displayed regarding the correct syntax of the commands if an input is malformed. So user documentation does exist and, while certainly can be improved upon, is already helpful. Unfortunately, documentation for developers is currently poor. The codebase is not systematically documented and there is currently no developer's manual. Thoroughly documenting the existing codebase should be considered the most pressing issue and tackled first to improve developer documentation.

5.8 *R2.1*: Data and Metadata Inventory

Status: Fully Met

The testbed organizes data and metadata in a standardized and principled way. The database is complete with respects to the currently primary and secondary artifact which stem from operating `iottb` itself. While complete now, extending `iottb` carries the risk of breaking this requirement if not careful attention is given. Since the database is a central part of the system as a whole, extensions must ensure that they comply with this requirement

before they get built in.

5.9 *R2.2*: Data Formats and Schemas

Status: Met

The testbed standardizes directory and file naming. All metadata is in plain test and in the JSON format. This makes them very accessible to both humans and machines. Currently, the only binary format which IOTTB creates are PCAP files. Luckily, the PCAP format is widely known and not proprietary, and widely available tools (e.g., Wireshark[7]) exist to inspect them. Furthermore, the data in the PCAP files can be extracted in to the plaintext CSV format, this further improves interoperability. Consistence is currently implicitly handles, that is, there are no strict schemas [7] `iottb` should generally not corrupt data during operation. But plaintext files are manually editable and can inadvertently be corrupted or made invalid (e.g. accidentally deleting a few digits from a UUID). It is important to keep this in mind when extending IOTTB and the types of files residing in the database become more heterogeneous.

5.9.1 *R2.3*: File Naming and Directory Hierarchy

Status: Met

`iottb` currently names all files which it creates according to a well-defined schema. In all cases, the file name is easily legible (e.g., metadata files like Listing 6) or the context of where the file resides provides easy orientation to a human reviewer. For instance, raw data files, which currently only are PCAP files, are all named with a UUID. This is not helpful to the human, but the metadata file, which resides in the same directory, provides all the needed information to be able to understand what is contained within it. Furthermore, these files reside in a directory hierarchy which identifies what devices the traffic belongs to, the date the capture file was created. Finally, capture files reside in a directory which identifies where in the sequence of capture of a given day it was created. Automation recipes expanding the range of data types collected can just follow this convention. This ensures interoperability and findability between various capture methods.
Chapter 4 4.4 already showed examples of the naming convention when adding devices.

5.10 Item *R2.4*: Data Preservation Practices

Status: Partially Met

Specific data preservation practices are not taken. `iottb` already follows the Library of Congress recommendations on data formats (see rec [5]). Most data is stored in plain text, and the binary formats used are widely known within the field and there is no access barrier. To enhance the testbeds' compliance with this requirement, automation recipes which back-up the data to secure locations periodically can be developed. The need for

[7] Strict schemas for metadata file briefly were introduced, but then abandoned due to the lack of knowledge surrounding the PYdantic library [4].

built-in preservation should be balanced with the goal of not introducing dependencies not related to the core aim of automated collection and FAIR storage. One way is just to have a repository of scripts which are not built in to the `iottb` executable, but which users can use and adapt to their needs[8].

5.11 Item *R2.5*: Accessibility Controls

Status: Deferred

While the `iottb` executable is aware what data it can and cannot access or change, there are currently no wider access controls implemented.

[8] For instance rsync scripts with predefined filters appropriate for the database.

6
Conclusion

IOTTB is an attempt for at an automation testbed for IoT devices. The `iottb` package can be considered somewhat feature limited and incomplete for a proper testbed, but it provides a foundation on which to build a more fully fledged system. `iottb` currently automates the setup and configuration of network packet capture and saves relevant database. The testbed uses the file system as a database such that it is also navigable by humans, not just machines. Data is stored in a predictably named hierarchy, and files which are produced as a result of operating `iottb` are both uniquely identifiable and interpretable for humans. This is achieved by using the file system paths to provide some context, such that file names must only contain minimal information to make it meaningful to humans. Additionally, all created resources are identified by a UUID which ensures that even if data is accidentally moved, their data is linked at least in principle. In summary, IOTTB is a testbed which takes the first step toward a future where data is FAIR and experiments are reproducible.

Acronyms

AP Access Point. 9, 10, 20

CLI Command Line Interface. 11, 22

IoT Internet of Things. 1–6, 8–10, 12, 25

OS Operating System. 9, 12

UUID Universally Unique Identifier. 15, 23, 25

Bibliography

[1] Welcome to click — click documentation (8.1.x). URL https://click.palletsprojects.com/en/8.1.x/.

[2] FAIR principles. URL https://www.go-fair.org/fair-principles/.

[3] mitmproxy - an interactive HTTPS proxy. URL https://mitmproxy.org/.

[4] Welcome to pydantic - pydantic. URL https://docs.pydantic.dev/latest/.

[5] Recommended formats statement – datasets | resources (preservation, library of congress). URL https://www.loc.gov/preservation/resources/rfs/data.html.

[6] Home | TCPDUMP & LIBPCAP. URL https://www.tcpdump.org/.

[7] Wireshark · go deep. URL https://www.wireshark.org/.

[8] Abbas Acar, Hossein Fereidooni, Tigist Abera, Amit Kumar Sikder, Markus Miettinen, Hidayet Aksu, Mauro Conti, Ahmad-Reza Sadeghi, and Selcuk Uluagac. Peek-a-boo: I see your smart home activities, even encrypted! In *Proceedings of the 13th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, pages 207–218. doi: 10.1145/3395351.3399421. URL http://arxiv.org/abs/1808.02741.

[9] Farshad Firouzi, Bahar Farahani, Markus Weinberger, Gabriel DePace, and Fereidoon Shams Aliee. IoT fundamentals: Definitions, architectures, challenges, and promises. In Farshad Firouzi, Krishnendu Chakrabarty, and Sani Nassif, editors, *Intelligent Internet of Things: From Device to Fog and Cloud*, pages 3–50. Springer International Publishing. ISBN 978-3-030-30367-9. doi: 10.1007/978-3-030-30367-9_1. URL https://doi.org/10.1007/978-3-030-30367-9_1.

[10] Kristof Friess. Multichannel-sniffing-system for real-world analysing of wi-fi-packets. In *2018 Tenth International Conference on Ubiquitous and Future Networks (ICUFN)*, pages 358–364. doi: 10.1109/ICUFN.2018.8436715. URL https://ieeexplore.ieee.org/abstract/document/8436715. ISSN: 2165-8536.

[11] Grigori Fursin. Collective knowledge: organizing research projects as a database of reusable components and portable workflows with common interfaces. 379(2197): 20200211. doi: 10.1098/rsta.2020.0211. URL https://royalsocietypublishing.org/doi/full/10.1098/rsta.2020.0211. Publisher: Royal Society.

[12] Adam Hahn, Aditya Ashok, Siddharth Sridhar, and Manimaran Govindarasu. Cyber-physical security testbeds: Architecture, application, and evaluation for smart grid. 4(2):847–855. ISSN 1949-3061. doi: 10.1109/TSG.2012.2226919. URL https://ieeexplore.ieee.org/abstract/document/6473865. Conference Name: IEEE Transactions on Smart Grid.

[13] Md. Milon Islam, Sheikh Nooruddin, Fakhri Karray, and Ghulam Muhammad. Internet of things: Device capabilities, architectures, protocols, and smart applications in healthcare domain. 10(4):3611–3641. ISSN 2327-4662. doi: 10.1109/JIOT.2022.3228795. URL https://ieeexplore.ieee.org/abstract/document/9983826/references#references. Conference Name: IEEE Internet of Things Journal.

[14] Deepak Kumar, Kelly Shen, Benton Case, Deepali Garg, Galina Alperovich, Dmitry Kuznetsov, Rajarshi Gupta, and Zakir Durumeric. All things considered: An analysis of IoT devices on home networks. In *28th USENIX security symposium (USENIX security 19)*, pages 1169–1185. USENIX Association. ISBN 978-1-939133-06-9. URL https://www.usenix.org/conference/usenixsecurity19/presentation/kumar-deepak.

[15] Jingjing Ren, Daniel J. Dubois, David Choffnes, Anna Maria Mandalari, Roman Kolcun, and Hamed Haddadi. Information exposure from consumer IoT devices: A multidimensional, network-informed measurement approach. In *Proceedings of the Internet Measurement Conference*, IMC '19, pages 267–279. Association for Computing Machinery. ISBN 978-1-4503-6948-0. doi: 10.1145/3355369.3355577. URL https://dl.acm.org/doi/10.1145/3355369.3355577.

[16] Manuel Silverio-Fernández, Suresh Renukappa, and Subashini Suresh. What is a smart device? - a conceptualisation within the paradigm of the internet of things. 6(1):3. ISSN 2213-7459. doi: 10.1186/s40327-018-0063-8. URL https://doi.org/10.1186/s40327-018-0063-8.

[17] Benjamin Andreas Ulsmåg. Private information exposed by the use of robot vacuum cleaner in smart environments.

[18] TL Vaughan, SC Battle, and KL Walker. The use of climate chambers in biological research. 39(14):5121–5127. Publisher: ACS Publications.

[19] Mark D. Wilkinson, Morris A. Swertz, and et al. The FAIR guiding principles for scientific data management and stewardship. 3(1):160018. ISSN 2052-4463. doi: 10.1038/sdata.2016.18. URL https://www.nature.com/articles/sdata201618. Publisher: Nature Publishing Group.

[20] Shicheng Zhu, Shunkun Yang, Xiaodong Gou, Yang Xu, Tao Zhang, and Yueliang Wan. Survey of testing methods and testbed development concerning internet of things. 123(1):165–194. ISSN 1572-834X. doi: 10.1007/s11277-021-09124-5. URL https://doi.org/10.1007/s11277-021-09124-5.

Appendix A

A.1 Command Line Examples

A.1.1 Pre and post scripts

In this example, the `--unsafe` option allows not to specify a IP or MAC address. `default` is the device name used and `-c 10` tells `iottb` that we only want to capture 10 packets.

```
# Command:
$ iottb sniff --pre='/usr/bin/echo "pre"' --post='/usr/bin/echo "post"' \
          default --unsafe -c 10
# Stdout:
Testbed [Info]
 Running pre command /usr/bin/echo "pre"
pre
 Using canonical device name default
 Found device at path /home/seb/iottb.db/default
 Using filter None
 Files will be placed in /home/seb/iottb.db/default/sniffs/2024-06-30/cap0002-2101
 Capture has id dcdf1e0b-6c4d-4f01-ba16-f42a04131fbe
 Capture setup complete!
 Capture complete. Saved to default_dcdf1e0b-6c4d-4f01-ba16-f42a04131fbe.pcap
 tcpdump took 2.12 seconds.
 Ensuring correct ownership of created files.
 Saving metadata.
 END  SNIFF SUBCOMMAND
 Running post script /usr/bin/echo "post"
post
```

The contents of the 'sniff' directory for the default device after this capture has completed:

```
sniffs/2024-06-30/cap0002-2101
$ tree
.
```

```
|-- capture_metadata.json
|-- default_dcdf1e0b-6c4d-4f01-ba16-f42a04131fbe.pcap
|-- stderr_dcdf1e0b-6c4d-4f01-ba16-f42a04131fbe.log
L__ stdout_dcdf1e0b-6c4d-4f01-ba16-f42a04131fbe.log
```

and the metadata file contains (\ only used for fitting into this document):

```
# capture_metadata.json
```


```
{
"device": "default",
"device_id": "default",
"capture_id": "dcdf1e0b-6c4d-4f01-ba16-f42a04131fbe",
"capture_date_iso": "2024-06-30T21:01:31.496870",
"invoked_command": "sudo tcpdump -# -n -c 10 -w \
    /home/seb/iottb.db \
        /default/sniffs/2024-06-30 \
            /cap0002-2101/default_dcdf1e0b-6c4d-4f01-ba16-f42a04131fbe.pcap",
"capture_duration": 2.117154359817505,
"generic_parameters": {
    "flags": "-# -n",
    "kwargs": "-c 10",
    "filter": null
},
"non_generic_parameters": {
    "kwargs": "-w \
        /home/seb/iottb.db/default/sniffs/2024-06-30 \
            /cap0002-2101 \
                /default_dcdf1e0b-6c4d-4f01-ba16-f42a04131fbe.pcap",
    "filter": null
},
"features": {
    "interface": null,
    "address": null
},
"resources": {
    "pcap_file": "default_dcdf1e0b-6c4d-4f01-ba16-f42a04131fbe.pcap",
    "stdout_log": "stdout_dcdf1e0b-6c4d-4f01-ba16-f42a04131fbe.log",
    "stderr_log": "stderr_dcdf1e0b-6c4d-4f01-ba16-f42a04131fbe.log",
    "pre": "/usr/bin/echo \"pre\"",
    "post": "/usr/bin/echo \"post\""
},
"environment": {
    "capture_dir": "cap0002-2101",
```

```
    "database": "iottb.db",
    "capture_base_dir": "/home/seb/iottb.db/default/sniffs/2024-06-30",
    "capture_dir_abs_path": \
        "/home/seb/iottb.db/default/sniffs/2024-06-30/cap0002-2101"
}
}
```

A.2 Canonical Name

```python
def make_canonical_name(name):
    """
    Normalize the device name to a canonical form:
    - Replace the first two occurrences of spaces
    - transform characters with dashes.
    - Remove remaining spaces.
    - Convert to lowercase.
    """
    aliases = [name]
    # We first normalize
    chars_to_replace = definitions.REPLACEMENT_SET_CANONICAL_DEVICE_NAMES
    pattern = re.compile('|'.join(re.escape(char) for char in chars_to_replace))
    norm_name = pattern.sub('-', name)
    # Remove non ascii chars
    norm_name = re.sub(r'[^\x00-\x7F]+', '', norm_name)

    aliases.append(norm_name)
    # Lower case
    norm_name = norm_name.lower()
    aliases.append(norm_name)
    # canoncial name is only first two tokens
    parts = norm_name.split('-')
    canonical_name = canonical_name = '-'.join(parts[:2])
    aliases.append(canonical_name)
    aliases = list(set(aliases))
    return canonical_name, aliases
```

Listing 8: Shows how the canonical name is created.

A.3 Add Device Example

A.3.1 Configuration File

A.4 Debug Flag Standard Output

```
 iottb -vvv --debug sniff roomba  --unsafe -c 10
<_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>
INFO - main - cli - 48 - Starting execution.
INFO - iottb_config - __init__ - 24 - Initializing Config object
WARNING - iottb_config - warn - 21 - DatabaseLocations are DatabaseLocationMap in t
INFO - iottb_config - load_config - 57 - Loading configuration file
INFO - iottb_config - load_config - 62 - Config file exists, opening.
DEBUG - main - cli - 52 - Verbosity: 3
DEBUG - main - cli - 54 - Debug: True
INFO - sniff - validate_sniff - 37 - Validating sniff...
INFO - sniff - sniff - 91 - sniff command invoked
DEBUG - sniff - sniff - 98 - Config loaded: <iottb.models.iottb_config.IottbConfig
DEBUG - sniff - sniff - 104 - Full db path is /home/seb/showcase
INFO - string_processing - make_canonical_name - 20 - Normalizing name roomba
DEBUG - string_processing - make_canonical_name - 38 - Canonical name: roomba
DEBUG - string_processing - make_canonical_name - 39 - Aliases: ['roomba']
Testbed [I]
 Using canonical device name roomba
 Found device at path /home/seb/showcase/roomba
INFO - sniff - sniff - 152 - Generic filter None
 Using filter None
DEBUG - sniff - sniff - 160 - Previous captures <generator object Path.glob at 0x7f
DEBUG - sniff - sniff - 162 - Capture count is 4
DEBUG - sniff - sniff - 165 - capture_dir: cap0004-0310
 Files will be placed in /home/seb/showcase/roomba/sniffs/2024-07-01/cap0004-0310
DEBUG - sniff - sniff - 172 - successfully created capture directory
 Capture has id 59153b53-c49d-44de-99d2-b5a3490df29a
DEBUG - sniff - sniff - 185 - Full pcap file path is /home/seb/showcase/roomba/snif
INFO - sniff - sniff - 186 - pcap file name is roomba_59153b53-c49d-44de-99d2-b5a34
INFO - sniff - sniff - 187 - stdout log file is stdout_59153b53-c49d-44de-99d2-b5a3
INFO - sniff - sniff - 188 - stderr log file is stderr_59153b53-c49d-44de-99d2-b5a3
DEBUG - sniff - sniff - 191 - pgid 260696
DEBUG - sniff - sniff - 192 - ppid 12862
DEBUG - sniff - sniff - 193 - (real, effective, saved) user id: (1000, 1000, 1000)
DEBUG - sniff - sniff - 194 - (real, effective, saved) group id: (1000, 1000, 1000)
DEBUG - sniff - sniff - 209 - Flags: -# -n
DEBUG - sniff - sniff - 217 - verbosity string to pass to tcpdump: -vvv
DEBUG - sniff - sniff - 228 - KW args: -c 10
DEBUG - sniff - sniff - 237 - Non transferable (special) kw args: -w /home/seb/show
INFO - sniff - sniff - 246 - tcpdump command: sudo tcpdump -# -n -vvv -c 10 -w /hom
 Capture setup complete!
DEBUG - sniff - sniff - 259 -
stdout: <_io.TextIOWrapper name='/home/seb/showcase/roomba/sniffs/2024-07-01/cap000
stderr: <_io.TextIOWrapper name='/home/seb/showcase/roomba/sniffs/2024-07-01/cap000

 Capture complete. Saved to roomba_59153b53-c49d-44de-99d2-b5a3490df29a.pcap
 tcpdump took 1.11 seconds.
 Ensuring correct ownership of created files.
 Saving metadata.
 END  SNIFF SUBCOMMAND
```

Figure A.1: Output with max verbosity and debug flag set.

```
 {
    "DefaultDatabase": "showcase",
    "DefaultDatabasePath": "/home/seb",
    "DatabaseLocations": {
        "iottb.db": "/home/seb",
        "showcase": "/home/seb"
    }
}

Database: showcase
Database Location: /home/seb/showcase
$ tree
iphone-13/
    |--device_metadata.json
roomba/
    |-- device_metadata.json

{
    "device_id": "a2158407-2b73-428d-9f94-cc8f3a497478",
    "device_name": "iPhone 13 (year 2043)",
    "aliases": [
        "iphone-13--year-2043-",
        "iPhone-13--year-2043-",
        "iPhone 13 (year 2043)",
        "iphone-13"
    ],
    "canonical_name": "iphone-13",
    "date_added": "2024-07-01T00:54:56.655710",
    "description": "",
    "model": "",
    "manufacturer": "",
    "current_firmware_version": "",
    "device_type": "",
    "supported_interfaces": "",
    "companion_applications": "",
    "last_metadata_update": "2024-07-01T00:54:56.655719"
}

{
    "device_id": "339de2af-c3ef-4c5d-a9c8-a03f7a65cc0a",
    "device_name": "roomba",
    "aliases": [
        "roomba"
    ],
    "canonical_name": "roomba",
    "date_added": "2024-07-01T00:54:34.715850",
    "description": "",
    "model": "",
    "manufacturer": "",
    "current_firmware_version": "",
    "device_type": "",
    "supported_interfaces": "",
    "companion_applications": "",
    "last_metadata_update": "2024-07-01T00:54:34.715859"
}
```

Listing 9: Directory and file contents after adding two devices.

B

Appendix B

B.1 Software Requirements

IOTTB was developed on the *Linux*[9] operating system *Fedora 40*[10]. It has not been tested on any other platform. IOTTB is implemented in a Python[11] package `iottb` , which has been developed with Python version 3.12.

B.1.1 Runtime Dependencies

- Poetry[12], version 1.8.3. Used for packaging and dependency management.
- Click[13], version 8.1, is a library which enables parameter handling through decorated functions.

B.1.2 Testing Dependencies

- Pytest[14], versions 8.2. Although not many exist.

[9] kernel.org
[10] https://fedoraproject.org/workstation/
[11] python.org
[12] https://python-poetry.org/
[13] https://click.palletsprojects.com/en/8.1.x/
[14] https://docs.pytest.org/en/8.2.x/

C
Appendix D

C.1 iottb

```
Usage: iottb [OPTIONS] COMMAND [ARGS]...

Options:
  -v, --verbosity  Set verbosity  [default: 0; 0<=x<=3]
  -d, --debug      Enable debug mode
  --dry-run        [default: True]
  --cfg-file PATH  Path to iottb config file  [default:
                   $HOME/.config/iottb/iottb.cfg]
  --help           Show this message and exit.

Commands:
  add-device           Add a device to a database
  init-db
  rm-cfg               Removes the cfg file from the filesystem.
  rm-dbs               Removes ALL(!) databases from the filesystem if...
  set-key-in-table-to  Edit config or metadata files.
  show-all             Show everything: configuration, databases, and...
  show-cfg             Show the current configuration context
  sniff                Sniff packets with tcpdump
```

C.1.1 Initialize Database

```
Usage: iottb init-db [OPTIONS]

Options:
  -d, --dest PATH                 Location to put (new) iottb database
  -n, --name TEXT                 Name of new database.  [default: iottb.db]
  --update-default / --no-update-default
                                  If new db should be set as the new default
```

```
                                [default: update-default]
  --help                        Show this message and exit.
```

C.1.2 Add device

```
Usage: iottb add-device [OPTIONS]

  Add a device to a database

Options:
  --dev, --device-name TEXT   The name of the device to be added. If this
                              string contains spaces or other special
                              characters               normalization is
                              performed to derive a canonical name  [required]
  --db, --database DIRECTORY  Database in which to add this device. If not
                              specified use default from config.  [env var:
                              IOTTB_DB]
  --guided                    Add device interactively  [env var:
                              IOTTB_GUIDED_ADD]
  --help                      Show this message and exit.
```

C.1.3 Capture traffic with *tcpdump*

```
Usage: iottb sniff [OPTIONS] [TCPDUMP-ARGS] [DEVICE]

  Sniff packets with tcpdump

Options:
  Testbed sources:
    --db, --database TEXT  Database of device. Only needed if not current
                           default.  [env var: IOTTB_DB]
    --app TEXT             Companion app being used during capture
  Runtime behaviour:
    --unsafe               Disable checks for otherwise required options.
                           [env var: IOTTB_UNSAFE]
    --guided               [env var: IOTTB_GUIDED]
    --pre TEXT             Script to be executed before main  command is
                           started.
    --post TEXT            Script to be executed upon completion of main
                           command.
  Tcpdump options:
    -i, --interface TEXT   Network interface to capture on.If not specified
                           tcpdump tries to find and appropriate one.
                           [env var: IOTTB_CAPTURE_INTERFACE]
```

```
    -a, --address TEXT      IP or MAC address to filter packets by.
                            [env var: IOTTB_CAPTURE_ADDRESS]
    -I, --monitor-mode      Put interface into monitor mode.
    --ff TEXT               tcpdump filter as string or file path.
                            [env var: IOTTB_CAPTURE_FILTER]
    -#, --print-pacno       Print packet number at beginning of line. True by
                            default.  [default: True]
    -e, --print-ll          Print link layer headers. True by default.
    -c, --count INTEGER     Number of packets to capture.  [default: 1000]
  --help                    Show this message and exit.
```

C.2 Utility commands

Utility Commands mostly for development and have not yet been integrated into the standard workflow.

C.2.1 Remove Configuration

```
Usage: iottb rm-cfg [OPTIONS]

  Removes the cfg file from the filesystem.

  This is mostly a utility during development. Once non-standard database
  locations are implemented, deleting this would lead to iottb not being able
  to find them anymore.

Options:
  --yes   Confirm the action without prompting.
  --help  Show this message and exit.
```

C.2.2 Remove Database

```
Usage: iottb rm-dbs [OPTIONS]

  Removes ALL(!) databases from the filesystem if they're empty.

  Development utility currently unfit for use.

Options:
  --yes   Confirm the action without prompting.
  --help  Show this message and exit.
```

C.2.3 Display Configuration File

```
Usage: iottb show-cfg [OPTIONS]

  Show the current configuration context

Options:
  --cfg-file PATH  Path to the config file  [default:
                   /home/seb/.config/iottb/iottb.cfg]
  -pp              Pretty Print
  --help           Show this message and exit
```

C.2.4 ”Show All”

```
Usage: iottb show-all [OPTIONS]

  Show everything: configuration, databases, and device metadata

Options:
  --help  Show this message and exit.
```

University of Basel

Faculty of Science

Declaration on Scientific Integrity

(including a Declaration on Plagiarism and Fraud)
Translation from German original

Title of Thesis: IOTTB: An Automation Testbed for IOT Devices

Name Assessor: Isabel Wagner

Name Student: Sebastian Lenzlinger

Matriculation No.: 19-775-494

I attest with my signature that I have written this work independently and without outside help. I also attest that the information concerning the sources used in this work is true and complete in every respect. All sources that have been quoted or paraphrased have been marked accordingly.

Additionally, I affirm that any text passages written with the help of AI-supported technology are marked as such, including a reference to the AI-supported program used. This paper may be checked for plagiarism and use of AI-supported technology using the appropriate software. I understand that unethical conduct may lead to a grade of 1 or "fail" or expulsion from the study program.

Place, Date: Amsterdam, 17.07.2024 Student: _____

Will this work, or parts of it, be published?

☐ No

☐ Yes. With my signature I confirm that I agree to a publication of the work (print/digital) in the library, on the research database of the University of Basel and/or on the document server of the department. Likewise, I agree to the bibliographic reference in the catalog SLSP (Swiss Library Service Platform). (cross out as applicable)

Publication as of: _____

Place, Date: _____ Student: _____

Place, Date: _____ Assessor: _____

Please enclose a completed and signed copy of this declaration in your Bachelor's or Master's thesis.