FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF MASTER'S THESIS

Title:	Network devices and services Identification using passive monitoring
Student:	Bc. Jan Neužil
Supervisor:	Ing. Tomáš Čejka, Ph.D.
Study Programme:	Informatics
Study Branch:	Computer Security
Department:	Department of Information Security
Validity:	Until the end of summer semester 2019/20

Instructions

Study commonly used network protocols and focus on those which can potentially lead to device or service identification from unencrypted network packets.
Study the current state-of-the-art in host/service discovery and focus on passive methods of network traffic analysis.
Based on the research, design a system for host/service discovery that would present the observed information to a network operator.
Implement an analysis tool that will process data from existing monitoring tools (such as flow exporter and flow collector) to gather information about devices/services on the network.
Evaluate the functionality of the created system and compare its results with the available tools for active or passive discovery.

References

Will be provided by the supervisor.

prof. Ing. Róbert Lórencz, CSc.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague October 10, 2018

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master’s thesis

Network Devices and Services Identification Using Passive Monitoring

Bc. Jan Neužil

Department of Information Security
Supervisor: Ing. Tomáš Čejka Ph.D.

July 30, 2020

Acknowledgements

I want to thank my supervisor Ing. Tomáš Čejka Ph.D. who had the patience with me during the stressful and busy times, for leading my steps during the writing of this thesis. I would also like to acknowledge my friends and colleagues, especially Ing. Dominik Soukup, Ing. Jiří Havránek, Ing. Marek Švepeš, and Mgr. Roman Kápl for giving me valuable feedback and council. My gratitude also belongs to my family, girlfriend, and fellow student colleagues for their ongoing support during my studies.

Declaration

In Prague on July 30, 2020 . . . .

Citation of this thesis

Neužil, Jan. *Network Devices and Services Identification Using Passive Monitoring.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2020.

Abstrakt

Viditelnost do síťového provozu je nezbytnou součástí síťové bezpečnosti a bezpečnostní analýzy. Současné stávající nástroje však obvykle poskytují pouze všeobecné informace o komunikaci jednotlivých zařízení. Tato diplomová práce se zabývá možnostmi využití vícero informačních zdrojů v lokálních sítích k identifikaci významu a účelu daného síťového připojení, které jsou srozumitelné pro většinu uživatelů. Tato práce se konkrétně zaměřuje na kombinování informací z "service discovery" protokolů s tradičními daty z IP toků. Výsledkem je vyvinutý softwarový prototyp analyzátoru — modul ACID, který zpracovává několik informačních zdrojů a přiřazuje síťovým spojením štítky, tj. pozorované aktivity. Tento přístup je mnohem slibnější než v současnosti používané nástroje založené na dobře známých portech a protokolech.

Klíčová slova pasivní monitorovaní, analýza síťového provozu, klasifikace aktivit, NEMEA, CESNET, Python

Abstract

Visibility into network traffic is an essential part of network security and security analysis. However, current existing tools usually provide only low-level technical information about the communication of devices. This diploma's thesis explores the possibilities of using several information sources on local networks to deliver high-level meaning and purpose of a network connection that is more understandable by the majority of users. Specifically, this work focuses on combining information from service discovery protocols with traditional IP flow data. As a result, the developed software prototype of the ACID analyzer module processes several information sources and assigns labels, i.e., activities to the network connections. This approach is much more promising than the currently used tools based on just well-known ports and protocols.

Keywords passive monitoring, network traffic analysis, activity classification, NEMEA, CESNET, Python

Contents

List of Figures

List of Tables

Introduction

The analysis of network communication is one of the essential components of any more extensive computer network. Without proper monitoring tools, network operators are blind to what is happening in the networks they manage. Connected users are also at risk as this state creates an ideal playground for attackers that can perform a malicious activity without being detected or leaving a trace. Various tools have been invented and developed to prevent these attacks and gain valuable insights into what is happening in the network.

Active and passive network monitoring and subsequent traffic security analysis are means of minimizing risks and threats. In the ideal world, companies, institutions, and individuals are highly motivated to prevent any security breach, outage, or disruption of any services. Unfortunately, this is almost an impossible task as attackers will always have an advantage over the defenders. Defenders never know when and what type of attack to expect, thus they have to be prepared for everything. Many companies and institutions are still underestimating the role of their network security until they are breached. Network monitoring tools can give the defenders useful information who was seen on the network if network policies are working and mainly the traffic which was exchanged. The crucial element of any monitoring tool is the automation of reporting any event and if the gathered information can be read quickly and efficiently. Additionally, these tools can act based on the gathered input and implement various network policies in case of a network incident. These policies can include denial of access to the reported device, redirecting traffic, changing levels of authorization, and many more. Having this capability can be a crucial element for any institution in the field of network security.

The gathered information about the device that sends or receives network traffic can be read through communication in the local-area network (LAN). Local-area networks are the first point of access, and many networking protocols do not leave this area. We leverage these protocols and individual packets to get information about the device hardware and running software. The analysis of this information can also decide on the role of devices in the network.

It can also create a map of dependencies among other devices. We can see what devices are connected, how they are behaving, and what services they are using or offering. Such information is vital for network administrators or managers. Based on this information, changes can be made to the network to improve communication capabilities or increase security. As many LAN protocols are exploitable, device and service recognition can also be used to find potential attack vectors and prepare the network operator to create relevant countermeasures to prevent these attacks from happening.

Contrary, it is almost impossible to process every packet in the wide-area networks (WAN) or Internet service provider (ISP) environments. As the computer networks gradually accelerate and grow, so does the volume of data that monitoring systems must process. In order to address this problem, network flow records were introduced. A flow is a network stream contains aggregated information from packet headers, reducing the amount of data to be processed. Besides, many protocols work within a single broadcast domain, so we do not observe this type of traffic beyond a default gateway, such as a router. For these reasons, our work focuses on service discovery protocols within LAN, which do not appear in the service provider's environments where the monitoring tools mainly concentrates on traffic signatures, communication dependencies mapping, and accessed services to gather valuable insights. Without these techniques and algorithms over network flows, many of the attacks would not be discovered.

There is a variety of monitoring tools in the industry often divided be their functions complementing each other. One category is the tools that look for security threats, breaches, attacks, and incidents. Another significant category is the ones that monitor the network to classify the network traffic. Based on the results, network operators can determine relevant actions. In our thesis, we will be focusing on this second mentioned category as our goal is to detect and classify various activities in networks just from monitoring network traffic. In the following section, we highlight the goals of this master's thesis and give a brief overview of its chapters.

Goals of this Work

This work deals with network service discovery protocols, which leaks valuable information about available service in the unencrypted payload of network packets. This master's thesis aims to design and implement the activity identification module, which improves the overall network situational awareness of a network operator. The module should be part of the distributed NEMEA [1] system, where it passes the information in the standard data structure for further analysis in the system.

Text Structure of the Thesis

This master's thesis is divided into four chapters:

Chapter 1 - Analysis
This chapter gives a brief overview of the service discovery protocols and their usage in today's networks. It analyzes and describes protocol characteristics and the information we can obtain. This includes the details of the information exchange and current frameworks and monitoring tools in the industry.

Chapter 2 - Design
This chapter portrays the overall architecture of the implemented module for activity identification. To understand the whole concept, we briefly describe the NEMEA system architecture and existing modules, which are essential to our work. Then we formally define the chosen terminology related to our module and introduce a chosen set of activities for classification.

Chapter 3 - Realization
This chapter specifies the realization of the module. It defines its structures and functions, describes all its parameters and configuration, and shows the overall module architecture and workflow. Then we introduce our methods of activity identification for our developed module and introduce the label requirements for the classified network flow.

Chapter 4 - Testing and Evaluation
This chapter explicates the testing methods we have used during the module's development and evaluates our identification methods for activity classification. It introduces the used environment and dataset for the module evaluation. It also shows the module's output for the given set of activities and compares the result of detection on the simulated network traffic data. Lastly, it summarizes all findings during development.

Chapter 1

Analysis

In this chapter, we will describe the related work and existing monitoring solutions.

1.1 Related Work

During the research phase of the thesis, we have not found many publications for the given topic in general. Modern tools rely on the active probing of the network to find potential vulnerabilities and open services that can be exploited. We have also found articles focusing on a small set of information gathered from passive network monitoring as OS fingerprinting, TCP/IP fingerprinting, *etc.* In the industry, there are many proprietary tools where the classification is the company's intellectual property; hence, we have little knowledge about their detection methods. To narrow down the set of possible methods, we have chosen to leverage the service discovery protocols in the LAN networks, which leak valuable information about the running services and host itself immediately after connecting to the network. In the following sections, we will highlight both existing open source and proprietary solutions and the functionality of the service discovery protocols.

1.1.1 Edge Computing

Edge computing is an extension of cloud computing where the computational power is expanded to the network edge. With this concept, traditional network devices like routers and switches provide computing resources besides networking capabilities. These resources can be used for various applications that can leverage distributed architecture [2]. Edge computing is a natural evolution of the Internet of Things (IoT) networks because we can process data faster with lower latency and higher flexibility. The comparison between traditional and edge computing architecture is depicted in Figure 1.1.

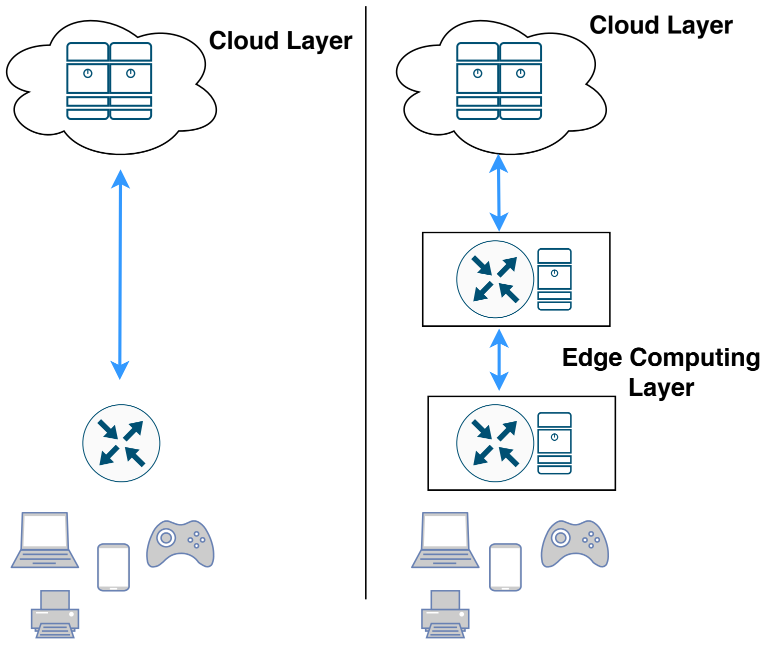

Figure 1.1: Comparison of classic and edge computing architecture

Also, this is a massive benefit for network security because we can have much better insight. Therefore, we can easily solve challenges with visibility behind Network Address Translation (NAT) and central network perimeter. Event detection modules can be presented everywhere and create a more robust network security solution.

This module is designed to operate at the network edge. It leverages information from local service discovery protocols to classify network activities. The first main benefit of this approach is helping network operators to identify better network traffic with human-based description. Currently, this is a crucial area because network traffic is still more and more heterogeneous, which complicates network analysis. The second benefit is the classification of user behavior. Our module can provide preprocessed, and annotated network flows to upper layers in edge computing architecture. This upper layer can leverage more computing resources to do more advanced analysis.

1.2 Existing Monitoring Tools

In this section, we discuss existing solutions to the problem of finding devices on the network, gathering information about them, and finding dependencies among them. Each solution briefly describes its functionality and compares it with our developed module for the NEMEA system. We divide this section into two categories, open-source and proprietary solutions. We list the most common and known solutions. Due to the high industry cost, comparing our module's results with the proprietary solutions was not performed, and it is beyond this master thesis.

1.2.1 Proprietary Solutions

The first category lists a selection of proprietary solutions where their output is comparable to our work. All of them are from the most prominent players in the industry.

Cisco DNA Assurance

Cisco DNA (Digital Network Architecture) [3] is a concept and vision of software-defined networking (SDN) developed by Cisco. It was introduced in 2017 along with platforms for various network environments like campus, data center, WAN and ISP networks. In the campus environments, where our work focuses, Cisco DNA Center (DNAC) platform is the critical component of the entire solution. It controls and manages the entire network from a single pane of glass graphical user interface (GUI). The controller is also a modular platform running various services in the containers. There are two primary components of DNAC. The network controller platform (NCP) automates the configuration and deployment of the network, and the network data platform (NDP) runs the analytics engine for the data. In the following paragraph, we highlight the basic functionality of the NDP component.

DNA Assurance is the built-in application within DNAC leveraging the NDP analytics engine to process big data gathered from various sources of the network. The sources can be streaming telemetry, NetFlow [4], and various contextual data captured from the devices on the network elements in the network such as switches, routers, *etc.* It uses mathematical algorithms and statistical models to present meaningful information to the network operator. The engine process the data in the (near) real-time future and delivers end-to-end network visibility, actionable insights to proactively respond to various events, guided remediation, the ability to troubleshoot the network for past incidents and visualize real-time application traffic flow.

Compared to our implemented module being developed, Cisco DNA Assurance is a much more complex solution serving mainly to troubleshoot and configure the network based on the occurred events captured by the network devices.

It provides contextual data about the device such as:

- user name,
- IP and MAC address,
- device hostname,

- device type,
- operating system and
- connection statistics on the wired or wireless network.

Our module focuses mainly on service discovery protocols. They announce a list of services and information about connected devices. We correlate this information with network flows to improve situational awareness of the network operators about observed activities.

Aruba Mobile First

Aruba Mobile First (AMF) [5] is a networking architecture for SDN campus networks designed by HP Enterprise. It was published in 2018 as an open, secure, and autonomous solution. This concept is composed of several components, such as underlay, overlay, and management. The management component contains various platforms to allow users to do all the required tasks. One of the major platforms is Aruba Airwave [6] that is responsible for the management and basic visibility operations. To allow network monitoring, Aruba uses a combination of passive and active methods in the Aruba Airwave ecosystem. The data source is NetFlow, contextual data from network devices and identity service, or metadata about the network from dedicated sensors.

The whole solution from Aruba is very complex and composed of many external tools. Our solution leverages information from service discovery protocols and uses it for human-based annotation of network flows. In comparison, Aruba does not track this type of information. The next differentiator is simplicity. The developed module focuses on network service identification and independent on network vendor. Similarly, like DNAC from Cisco, Aruba is an enterprise-grade and paid product. Our module is open-source and allows additional extension or customization with no additional cost.

1.2.2 Open Source Solutions

The second category gives a brief overview of existing free tools for network monitoring. We describe their functions and compare them to our work.

NEMEA

Network Measurements Analysis (NEMEA) [1] system is a stream-wise, flow-based, and modular detection system for network traffic analysis. In the stream-wise concept, data is analyzed in RAM with minimal writing to persistent storage. NEMEA can run both online and offline. In online mode, data is processed and analyzed either from the IP flow information export (IPFIX) [7] collector or directly from the network interface card (NIC). In the offline mode, the system analyzes saved network flows with the independence of the current network traffic. It enables the modular deployment of network monitoring tools and the parallel processing of large amounts of network data. It is developed as an open-source project publicly available for world-wide networking community designed for experimental or production deployments. The system is available as an open-source project developed by CESNET [8] organization, the operator of Czech national research and education network (NREN), in cooperation of Czech universities. Since our developed module is part of this system, we briefly introduce its architecture in the Section 2.1.1.

Zenmap

Zenmap [9] is the official GUI scanner for Nmap. Nmap is an open-source program developed by Gordon Lyon, known as Fyodor. Nmap actively probes the network to find devices and gather information about them, especially open ports on the transport layer of the TCP/IP model. From the information about open ports, Nmap deducts the services which are provided in the network by the devices. The user initiates the scan, and the results about all active devices in the network are presented in a human-readable format.

Nmap can finds the following information about the devices:

- operating system,
- IP and MAC address,
- open ports and
- last boot of the device.

Zenmap is a one-time active network analyzer that finds all currently active devices in the specified network subnet and analyzes them for open services based on the open ports. Our module does not generate any traffic to discover services and information about the devices. Besides, Nmap does not offer any dependency mapping among the devices; however, it is an excellent entry tool to collect much information about the active devices in the local network.

1.3 Service Discovery Protocols

In this section, we describe the most common service discovery protocols we have chosen to leverage for our module. Collecting information from network passive monitoring can be a challenge, as we have described in the introduction. For that reason, we have narrow down the options, and we have chosen a set of service discovery protocols that periodically announce their available services in the network using mainly multicast addresses. In the following subsections, we will describe the implementation of a multicast domain name system (mDNS) and simple service discovery protocol as they are leveraged by most of the operating systems.

1.3.1 Multicast DNS

Multicast DNS was defined in RFC 6762 [10] along with its companion technology DNS-Based Service Discovery defined in RFC 6763 [11] to map AppleTalk's Name Binding Protocol (NBP) into IP networking and to provide easy autoconfiguration with the concepts of Zeroconf and link-local addressing. Multicast DNS leverages the existing structure, syntax, and record types of the DNS protocol syntax.

AppleTalk and Zeroconf

In the times where various networking protocols co-existed with today's standard Internet Protocol (IP), Apple Inc. [12] has developed their suite of networking protocols. On of the AppleTalk's [13] attributes it that it can operate without any manual or automatic configuration. We can easily connect two devices with an Ethernet cable or wirelessly to establish a connection without configuring any addresses. Contrary, TCP/IP networking required manual configuration by the user or automated configuration received from a DHCP server. Both IPv4 and IPv6 now have self-assigned link-local addresses defined in [14, 15], which allow the communication among devices in the same network without the configuration.

mDNS Protocol

The mDNS protocol was defined in the RFC 6762 [10]. In this section, we look at its definition and capabilities. Multicast DNS takes functionality advantages of the standard domain name system (DNS) on the local link without any conventional DNS server. Multicast DNS also leverages the DNS namespaces to be used freely by the local services. The primary benefits of mDNS include infrastructure independence, resiliency during infrastructure failures, and need of no or little configuration. Majority of devices in the LAN networks such as computers, mobile phone, tablets, etc. are not authorized to create names which leave the devices anonymous for various purposes. In order to solve this problem, the mDNS protocol enables devices to elect a unique link-local hostname in the form "unique-host-name.local.". In case of a sporadic hostname conflict, mDNS provides a mechanism to deal with it. If a device queries for the name ending with ".local.", the DNS query must be sent to the mDNS IPv4 link-local multicast IPv4 address *224.0.0.251* or its IPv6 equivalent *FF02::FB* on a UDP port *5353*. This mechanism also applies for link-local reverse mapping, *e.g. "254.169.in-addr.arpa."* for IPv4 or *"8.e.f.ip6.arpa."* for IPv6. These reverse lookups also must be sent to the mDNS multicast address. Multicast DNS creates the underlying framework for DNS-based service discovery.

DNS-Based Service Discovery

The mDNS protocol is the underlying framework for DNS-based service discovery (DNS-SD) defined in RFC 6763 [11]. DNS-SD allow devices to discover the list of named instances of services in the local network. The information about the service instance is obtainable via DNS SRV [16] and DNS TXT [17] record. The SRV record contains the information about the port and target name where the service is reachable in the form of *"<Instance>.<Service>.<Domain>"*. The DNS TXT gives additional information in the structured form of ("key=value") pairs. In order to find all services being advertised in the network, special meta-query service type enumeration *"_services._dns-sd._udp.<Domain>"* was defined. It is a PTR [17] record which returns the set of PTR records in the form of *"<Service>.<Domain>"* in the rdata. This set of service and domain pair can be used to get additional information using subsequent PTR query. The responder to that query should include SRV record(s), TXT record(s) in the PTR rdata and all address A and AAAA records named in the SRV rdata.

A typical DNS-SD workflow we have observed when a device connects to the network is the following, and we illustrate it in Figure 1.2:

1. The device sends a service type enumeration query on mDNS multicast address using UDP and destination port 5353.

2. All other listening hosts in the same multicast DNS group respond with a set of PTR answer records, *e.g., _airplay._tcp.local.*

3. After receiving the set of available services, the device prepares the set of PTR queries for target hosts and services.

4. The device queries the prepared set to the multicast group, *e.g., my-tv._airplay._tcp.local, _airplay._tcp.local.*

5. The host with the queried service responds with the unicast answer to our device with associated SRV, TXT, and A/AAAA record about the service.

6. The device knows that the host offers the SSH server on port 22 and knows the IP address for *my-tv.local.*

One of the significant implementations is the Bonjour [18] service, which comes built-in with Apple's [12] macOS and iOS operating systems. It can also be installed on Windows [19] operating systems. Bonjour works only within a single broadcast domain. In our test environment, we were using devices running Bonjour, and in the design of our module, we leverage the valuable information from SRV and TXT reply records. Another implementation for Linux and BSD distribution is the Avahi [20] software.

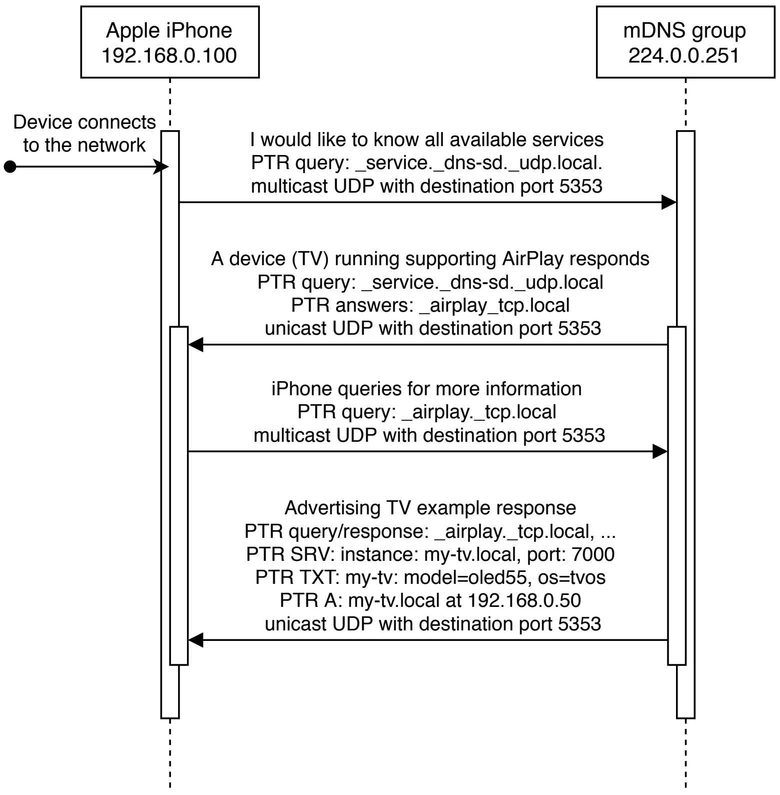
Apple iPhone
192.168.0.100
mDNS group
224.0.0.251
Device connects to the network
I would like to know all available services
PTR query: _service._dns-sd._udp.local.
multicast UDP with destination port 5353
A device (TV) running supporting AirPlay responds
PTR query: _service._dns-sd._udp.local
PTR answers: _airplay_tcp.local
unicast UDP with destination port 5353
iPhone queries for more information
PTR query: _airplay._tcp.local
multicast UDP with destination port 5353
Advertising TV example response
PTR query/response: _airplay._tcp.local, ...
PTR SRV: instance: my-tv.local, port: 7000
PTR TXT: my-tv: model=oled55, os=tvos
PTR A: my-tv.local at 192.168.0.50
unicast UDP with destination port 5353

Figure 1.2: DNS-SD workflow example

1.3.2 Simple Service Discovery Protocol

The Simple Service Discovery Protocol [21] (SSDP) has a similar concept comparing to mDNS but leverages the HTTP over UDP (HTTPU) structures rather than DNS. It also enables the devices to advertise and discover available services without any configuration when connected to the network. It was defined as an Internet Engineering Task Force (IETF) Internet draft in 1999 by Microsoft [19] and Hewlett-Packard [22]. Even though this proposal expired in April 2000, The SSDP creates the building block for Universal Plug and Play (UPnP) protocol stack and architecture [23, 24] similar to mDNS for DNS-SD.

SSDP is a text-based protocol based on HTTP with the usage of UDP as the underlying transport protocol. The host announces the available services using IPv4 multicast address *239.255.255.250* or its link-local IPv6 equivalent *FF02::C* on a UDP port *1900*. SSDP has a hybrid approach to do both discovery and announcements. Once a new device connects to the network, it sends the announcement to the SSDP multicast group about its presence. From this point, no other announcement is needed, unless the device changes state or goes offline. The whole mechanism is event-driven to maintain protocol efficiency and not flood the network stream with periodic announcements.

If a device wants to discover available services, it leverages the *M-SEARCH* method of the HTTP suite. To announce or withdraw service from the network, SSDP uses the *NOTIFY* method. The search queries' responses are sent via unicast back to the originator on the UDP port 1900. Every SSDP discover request must contain a search target (ST) header with a single URI where the requested service is specified. Only the hosts with the matching service sent in the ST header may respond in the SSDP multicast group with the corresponding URL location in the HTTP response. The URL itself contains the port on which is the service accessible.

SSDP also has a vulnerability that can be exploitable for distributed denial of service (DDoS) attacks; this weakness is described in [25]. For that reason, many network administrators police SSDP multicast traffic in their networks. In our module, we leverage the SSDP layer to extract the available service and retrieve the information about what type of service has been used.

Universal Plug and Play

Universal Plug and Play (UPnP) is a suite of networking protocols leveraging the SSDP infrastructure. The UPnP Forum [24] published the UPnP technology, a consortium of computer industry vendors to enable simple connectivity among various devices from personal computers to small consumer electronics. The forum no longer exists, and it was replaced by Open Connectivity Foundation (OCF) [26] in 2016.

1.4 Requirements Analysis

An essential part of the final solution's design is the precise determination of requirements. We divide them into functional and non-functional. Functional requirements represent the module's significant functionalities, while non-functional ones instead determine the limitations of system properties and design architecture. In the following subsections, we describe the module's requirements, which are based on the content of the assignment of this work and the performed analysis.

1.4.1 Functional Requirements

Classification

The module's primary goal is to classify which type of activity was observed and present this contextual information to a network operator in a human-readable format or another module in a specified data structure.

Data processing

One of the goals of the module is to process relevant data from service discovery protocols. The module should use an existing framework for network monitoring and expect the input within a selected data structure. The processed data should be stored in memory and passed or stored if requested.

Flow analysis

The module should analyze the given information about announced services and look for patterns revealing a particular activity from the network flows.

Configurability

Program parameters and configuration files should ensure the configurability of the module. The configuration should allow loading saved configuration, and the command-line parameters should allow to extend or overwrite the values from the configuration file.

1.4.2 Non-functional Requirements

Scalability

The module should allow scalability to add more service discovery protocols and their characteristics in the future. It should also allow extending by observable activities in the network. We should implement the resulting module in a modular nature with the concept of object-oriented programming (OOP) to allow future expansion.

Compatibility and development

The module should use an existing framework NEMEA, enabling the interconnection of individual modules to maintain compatibility. It also provides critical components and data structures for receiving predefined network flows. Using the existing framework should ease the implementation and divert focus on the primary goal.

Portability

The deployment of the module should be independent of the host operating system. The source code of the module should use a programming language to fulfill this requirement.

1.5 Selected Solution

This section's content is a description of the selected solution, which was based on the analysis intended for the implementation of the resulting tool. The developed module will focus on service discovery protocols, especially on mDNS and SSDP, to improve situational awareness about captured network traffic. During our analysis, we have not found any existing solution with the same functionality. The module will be part of the distributed system NEMEA from which we leverage existing modules and structures. It will be configurable using a configuration file or parameters. At the same time, it will remain scalable for future expansion of more protocols or a set of activities. Python will be used as the programming language due to its portability and flexibility of deployment. Finally, the module will present results to a network operator in a human-readable format or send the results in a standard data format to other modules for further processing.

Chapter 2

Design

In this chapter, we describe the overall architecture of the implemented module for activity identification, which should improve an overall situational awareness for a network operator. We also define the terminology related to our module and a set of activities that we want to classify. In order to understand the whole concept, we briefly illustrate the NEMEA system architecture and existing modules, which are essential to our work.

2.1 Building Blocks

As mentioned before, our work is a part of the broader modular NEMEA framework; hence we could use many functions from the existing implementations. These functions are essential to our module as we take these prepared building blocks like data structures, packet parsing, and flow processing. In this section, we take a look at the existing NEMEA framework and useful modules.

2.1.1 NEMEA Architecture

In Figure 2.1, we can observe the architecture of the NEMEA system. The basic building blocks are the modules (highlighted in yellow), which run as separate system processes and process the data (export, storage, filter, aggregate, merge, etc.). A particular subset of the modules is detectors (highlighted in red), which detect malicious traffic and traffic anomalies based on the various algorithms. Modules receive various stream of data on their input interfaces, process it, and send to their output interfaces in the specified format. The format is defined by the NEMEA framework, which implements features and data structures for all modules. First of them is the specific type of Traffic Analysis Platform (TRAP) interface (highlighted in blue), which implements functions for sending and receiving messages between interfaces. Types of the messages vary from flow records, alerts, statistics to preprocessed data. The

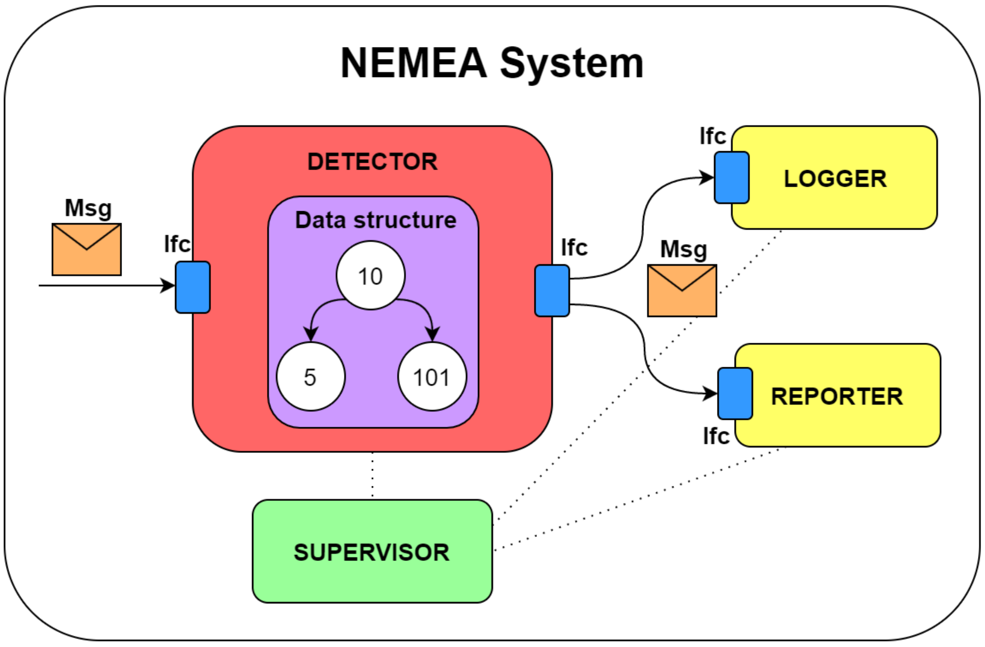

Figure 2.1: NEMEA architecture (source: https://github.com/cesnet/nemea)

second structure is the Unified Record (UniRec), which implements an efficient binary data format for the messages (highlighted in orange). The third is the common library (highlighted in purple), which implements common algorithms and headers used in system modules. The last component of the NEMEA architecture is the supervisor (highlighted in green), which has the role of the central management and monitoring tool of the entire system.

2.1.2 Important Existing Modules

In this section, we highlight the existing modules which preprocess data for our work from network packets captures and flow analysis. We use these auxiliary modules for every iteration over the given data input.

Flow meter

Flow meter [27] is the module from the NEMEA framework which converts IP packets from the NIC interface or network packet capture file to bi-directional network flows. The output flows can be either in a standard IPFIX format or in the NEMEA specific format UniRec. The flow meter also supports plugins for extracting specific information from the packet's application layer. When running more than one plugin, multiple output interfaces have to be specified. During the writing of this thesis, mDNS and SSDP plugins are being developed to optimize the module's data processing. For this reason, we need a temporary preprocessing module for extracting mDNS and SSDP information from packet captures.

Merger

Merger is another auxiliary module from the NEMEA framework use with conjunction to the flow meter. Since the flow meter does not support one output for multiple plugins, we need to merge these outputs from individual plugins into one record to avoid duplicity. Merger helps us to consolidate all network flows to an extended UniRec format enriched with the plugin fields.

Service discovery

The service discovery module [28] is a temporary prototype written in Python to extract application layer information from mDNS and SSDP packets. In the future, it will be replaced by the flow meter plugins for service discovery protocols. It serves as a template for the flow meter plugins to determine the most relevant information for our module. During our research, we have observed the mDNS and SSDP communication to determine the most relevant information for extracting. As we can see in the sample output in Figure 2.2, the module parses all packets based on the source MAC address and add all additional information if the packet from a source device is already in its database.

Based on the observed network traffic, we have defined which information should the module extract. To pass the information to our module, we have chosen the standard JSON structure.. From a device perspective, it parses all IPv4 and IPv6 addresses observed for the given MAC address, all hostnames, and operating systems. It populates the vendor filed by using an external library to map the MAC address to a specific vendor. We do not leverage the key *label* for the time being as we reserve it for future implementations.

In mDNS packets, we look at the information from all queries and response records, which we store in the JSON list format. If the record is already present, we omit it. The most vital information is stored within the PTR response with additional SRV and TXT records. From the SRV record, we extract the port and available service with an instance name and domain. The TXT record contains additional information, e.g., device model and OS, and we save it in the list. To optimize subsequent searching of services, we create an auxiliary list of ports.

We have also observed the behavior for the SSDP traffic. We store the list of searches and user agents from the M-SEARCH headers and check for the duplicity of values. The user agent field can leak us the operating system of the observed device. However, the most crucial information we parse from the HTTP NOTIFY method, which announces the available service with a given port within the location header. We also store the server header for

```
{
  "device": {
   "mac": "20:17:42:e9:f1:c4",
   "ipv4": ["192.168.10.25"],
   "ipv6": ["fe80::2217:42ff:fef9:1df5"],
   "hostname": ["LGwebOSTV.local"],
   "label": [],
   "os": ["WebOS 4.0.0"],
   "vendor": "LG Electronics"
  },
  "services": {
   "mdns": {
    "ports": ["7000", "39891"],
    "services": {
     "7000": ["jn-lgtv55._airplay._tcp.local"]
     "39891": ["LGwebOSTV._hap._tcp.local"]
    },
    "queries": ["_airplay._tcp.local", ...],
    "response": ["LGwebOS._hap._tcp.local", ...],
    "txt": [ "deviceid=AA:BB:CC:", "model=OLED55E9PLA"]
   },
   "ssdp": {
    "ports": ["1446"],
    "notifies": {
     "1446": {
      "urn": ["dial-multiscreen-org:service:dial:1"],
      "server": "WebOS/1.5 UPnP/1.0 webOSTV/1.0"
     }
    },
    "searches": [],
    "useragents": ["Linux/4.4.84-169.gld4tv.4 UPnP"]
   }
  }
}
```

Figure 2.2: Service discovery module sample output

future usage. Similar to mDNS packets, we also create an auxiliary list of ports within the JSON structure.

PassiveAutodiscovery

The PassiveAutodiscovery [29] module is the antecedent work for device discovery from passive monitoring. The goal of this module is to classify devices and assign type labels to them in order to create a dependency mapping. It uses the list of well-known ports and registered services for that given port and a database of vendor's MAC addresses to identify a device in the network. As a result, we receive the information in JSON format about devices and their matching labels, which we use for further analysis.

2.2 Definition of Terms

In this section, we introduce definitions to be consistent within the following sections.

ACID

ACID is the chosen name for our developed module, which comes from the abbreviation of "ACtivity IDentification". We have defined this name to describe the module's principal purpose best. From this part, we refer to ACID as our module developed for the NEMEA system.

Activity

The activity can be any motion of a user or device in the real world observed in network traffic. In Section 2.4, we have defined the type of use cases/scenarios we want to monitor and classify based on the given environment. When we mention an activity, we refer to these defined use cases/scenarios.

Identification

Identification, in the context of this thesis, means the identification/classification of the given activity. If we detect the network traffic matching our defined activity, we refer to this action as an identification. We revisit the details of classification methods in Section 3.4.

Signature

We can identify each activity by matching a particular behavioral pattern. The patterns can vary, from unambiguously series of packet lengths, TCP flags, transferred bytes, specific HTTP headers, or its combination. For each activity, we have defined a specific pattern to be matched. Based on the signature, we define an activity label.

Activity Label

The label is the crucial output of our module. After we unequivocally identify an activity, we assign it to an identified activity for the given flow. The label classifies the flow and enriches the information. This information helps to improve the situational awareness about specific flows observed in the network.

2.3 Overall Module Design

In Figure 2.3, we can see the overall architecture of our resulting module. First, it receives network packet capture either from NIC using TRAP or takes a packet capture file. These captured packets are then sent to the flow meter and merger module (highlighted in red) to create network flows in the UniRec format (highlighted in orange). The intermediate step with the module is important as the flow meter supports different plugins, and we want to consolidate information into one flow. The flow records are stored as a temporary file to be used by the PassiveAutodiscovery (PAD) module (highlighted in pink) for basic dependency, and by ACID to correlate these flow records with information gathered from auxiliary modules. Since processing flows by the PassiveAutodiscovery module might be time-consuming, running this module is optional. We mostly rely on the data from the service discovery module (highlighted in yellow), which we run in a parallel process. This prototype module, written in Python, uses Tshark [30], which parses only mDNS and SSDP packets and serialize them in the JSON format. The example was referenced in Figure 2.2. As discussed in Section 2.1.2, in the future, we will receive the only network flows in the UniRec format containing application layer information about mDNS, SSDP, and other discovery protocols using the flow meter plugins. ACID (highlighted in white) loads information from the service discovery and PassiveAutodiscovery module in the memory structures described in Section 3.1 and after it analyzes this information over the flows from the flow meter module. In the end, the results are presented to a network operator in a human-readable report or serialized into JSON for further processing by other modules.

2.4 Defined Activities

The goal of our module is to identify activities from passive network passive monitoring. However, countless scenarios exist with a set of activities initiated by a user, spontaneously by different devices, varying in protocol specifications, and many more variables to observe in the real network traffic. For this reason, we have defined and simulated a set of activities that can be observed and tested in our limited environment from packet captures. The chosen approach was a good start for the realization of the ACID module.

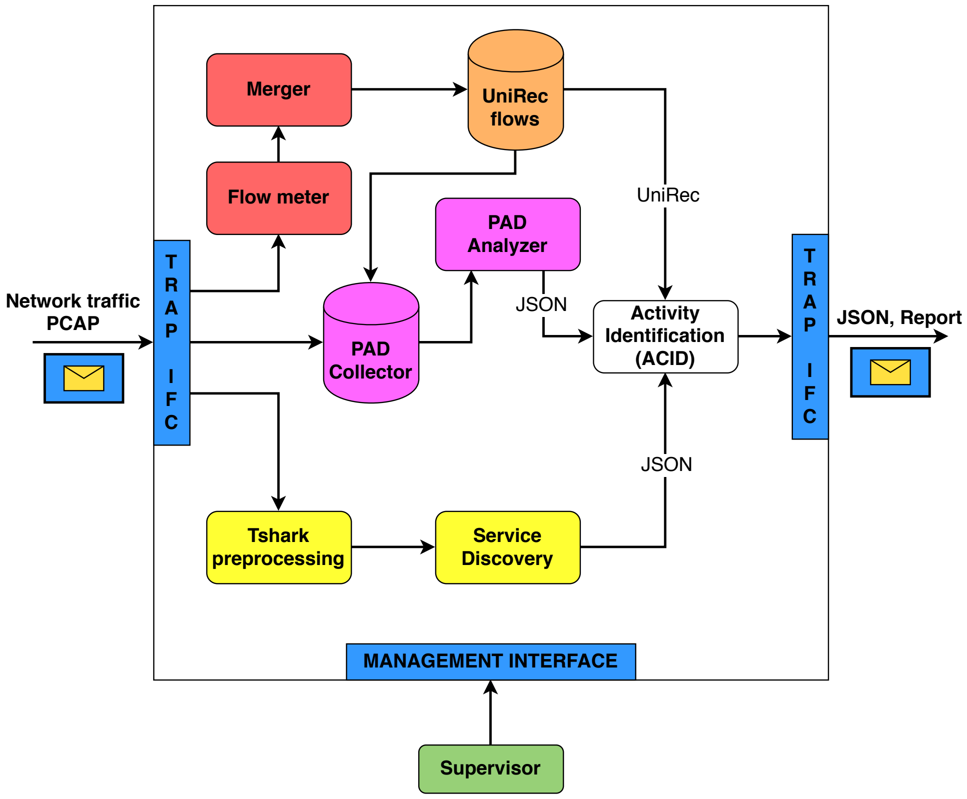

Figure 2.3: Module architecture of the ACID module consists of packet preprocessing, existing NEMEA framework, and its modules.

These defined activities can be seen and identified using several different information sources in the network traffic, including service discovery protocols. We have created a dataset to test our module by simulating these activities. Due to our implementation's modular nature as one of the non-functional requirements, we plan to expand the current set of activities in the future. The following activities are the examples that our work can detect and classify from the data captured by passive monitoring.

Screen mirroring and multimedia streaming

With the expansion of smart televisions and screens capable of connecting to the network via Ethernet [31] cable or Wi-Fi [32], now we can seamlessly stream media and mirror screens from various type of devices like laptops, phones, tablets and many more. The streaming can be done using Bluetooth technology or direct ad-hoc wireless network at proximity level, or remotely using network infrastructure. Apple AirPlay [33] is one of the service allowing multimedia streaming and screen mirroring via network infrastructure. Air-

Play leverages Bonjour to announce the service to the mDNS multicast group. After the connection request is initiated, the device pairing uses TCP over the announced port via HTTP. The data stream itself is encrypted and uses TCP at a random port. We have observed a signature of specific TCP flags and packet lengths when devices form a connection. We can also approximate the type of activity based on the transferred bytes in the given time frame, e.g., the user was mirroring a screen or user was streaming media over AirPlay.

Remote session and file transfer

One of the ubiquitous advertised services via mDNS is the Secure Shell (SSH) and SSH File Transfer Protocol (SFTP). They both operate at the same port 22, but in some cases, the chosen port might differ. As the name of the protocols reveals, the data transfer is encrypted; hence we do not see in the payload. To accurately distinguish between a file transfer and a remote session, is a discussion for another thesis. In [34], they use statistical traffic analysis techniques to classify the SSH traffic. In our approach, we use simple techniques based on the simulated traffic from our datasets, like transferred bytes per second. From that estimate, we can approximate with a certain probability whether the user was transferring data or was connected to a remote host via SSH. This rough estimate might be vital for a network operator as he can detect suspicious sessions or data exfiltration from an unknown device.

Printing and scanning over IPP

Another activity we can normally observe in the traffic is whether a user was using the connected network printer, and what type of action was requested. Most standard network printers use the Internet Printing Protocol (IPP) to perform the desired action. IPP was defined as an experimental protocol in RFC 2910 [35] and became a worldwide standard short after. It uses HTTP structure over TCP on a port 631. We can extract information about the document, size, and desired printer action from the application layer. However, this approach can not be used within its secure version called IPP over HTTPS (IPPS) (defined in RFC 7472 [35]) where the information is encrypted. From a security perspective, new devices use IPPS as a default option; nonetheless, many printers still do not support this functionality. For our module, we want to classify the printing or scanning activity.

Smart remote control

Nowadays, we connect smart fridges, air-conditioning, thermostats, doors, boilers, bulbs, and many more into the home or enterprise network within buildings. In most cases, no segmentation of these devices is configured, which leaves the possibility of entry for an attacker who can exploit any vulnerabilities to gain access in the network. Besides, many IoT protocols are not secure or left without any encryption. In our environment, we have a connect IoT gateway to control various accessories via Apple HomeKit integration. Similar to AirPlay, the gateway announces its availability in the network using the Bonjour protocol. Users with a capable smartphone can afterward remotely control the lights, the sunblinds, or power outlets. We wanted to observe and identify these activities. We have captured and analyzed the traffic.

2.4.1 Discussion

In the beginning, we have defined a set of activities that we had observed in our lab environment. Still, the activity was not captured in the traffic. These findings are due to various implementations of services where some of them might use service discovery protocols to announce the availability of a service, but the actual data traffic is using different technology or network. For example, we have observed that the smart television can be controlled from Apple iPhone using Apple HomeKit technology and the television did announce *_hap._tcp.local*, hence the smartphone sees the television either turned on or off. If we decide to turn the television on or off remotely, the smartphone uses Bluetooth to execute this action. However, if the smartphone is not connected wirelessly in the network, the television remote is not available as it was not discovered using mDNS.

Another example is the Apple AirDrop technology, which is also announced using the Bonjour service in the local network. It enables quick sharing of documents and multimedia among Apple devices. The data transfer is via Apple Wireless Direct Link (AWDL), a special interface using the same hardware Wi-Fi chip for low latency and high speed wireless peer-to-peer connection. The real benefit is the independence over the network infrastructure. The first implementations of AWDL in Apple products used Bonjour over the AWDL which significantly degraded the wireless performance as the generated mDNS traffic interfered with regular Wi-Fi traffic [36]. For these reasons, Apple products announce their services using Bonjour in local networks. The design of our work is modular; hence we plan to define and add more activities to discover in the future.

Lastly, many services announce their services using service discovery protocols with a specific port, but the stream of communication occurs on a different port or interface. During our observation, many services were using encrypted communication to exchange crucial information about the next steps. Regardless of this obstacle, some services follow a specific pattern like the same packet length. There are methods for detection avoidance, even within encrypted communication. Padding, which dynamically changes the packet length, could be one of them.

2.5 Module Configuration

In this section, we design the options for ACID from the non-functional requirements of the module analysis. We highlight the most crucial options for the realization and show configurable parameters from the configuration file and command line. All input from the CLI or configuration file should be thoroughly checked for validity, permissions, and authorization rights.

2.5.1 ACID Parameters

One of the functional requirements of the work should be the integration into the existing system NEMEA. As described in Section 2.1.1, the NEMEA system is a CLI-based framework where the supervisor module manages the running of individual components. The supervisor depends on a configurable set of parameters for each module. For that reason, our work should take similar options for corresponding auxiliary structures. Thus, the supervisor can create subsequent functional blocks of individual modules for analysis and detection.

- **-c/--config**: a path to a configuration file
- **-C/--clean**: a flag whether to delete auxiliary files
- **-d/--pcap_dir**: a path to a directory with packet captures files to be analyzed
- **-h/--help**: prints a help message about usage of the module
- **-i/--ifc**: a specification of the input and output interfaces for the NEMEA
- **-j/--json**: an option to serialize results into JSON data format
- **-f/--file**: a path to an individual file(s), delimited by ":"
- **-o/--output**: a path to a file for storing results, otherwise standard output is used
- **-p/--print**: an optional argument to present the result in a human-readable format, the results are printed to standard output
- **-P/--run_pad**: an optional argument to analyze flows with the PassiveAutodiscovery module which assigns additional labels to devices and creates dependency mapping
- **-v/--verbose**: an optional argument to print current progress

2.5.2 Configuration File

As the only required argument for running the module should be the path to a configuration file, we determine the mandatory arguments to analyze the given data successfully. In the following list, we define the necessary key/value pairs:

- **pcap_dir**: a path to a directory or directories with packet captures, can be overwritten from the CLI with respective parameter
- **sdp_dir**: a path to a directory with service discovery module prototype
- **pad_dir**: an optional path to the PassiveAutodiscovery module, if specified the module will be used
- **tmp_dir**: a path to a directory for storing temporary auxiliary results
- **nfm_input_spec**: a required header format for UniRec messages including the flow meter plugins

Chapter 3

Realization

In this chapter, we explain the module's functionality and realization of the most vital parts. To ensure easy portability as one of the non-functional requirements, we have chosen Python version 3.7 as the programming language. As the module does not require to process and analyze real-time traffic, and we work with JSON data structures, Python was an ideal candidate for the implementation. As mentioned in the previous chapter, our module depends on the NEMEA framework and its existing modules.

3.1 Module Structure

The source code of the module consists of nine files. This section will briefly explains the structure and classes used in source code files. More information about classes will follow in Section 3.2.

- **acid.py**: The file contains the principal class *ACID*, which holds all configuration attributes, information about communications and devices. It is the main file of the entire solution.
- **device.py**: The file contains the class *Device*, which stores all information about individual devices observed in the network, their communications, and advertised services.
- **service.py**: The file contains the class *Service*, which is the consolidation of information from service discovery protocols. It holds pointers to class *MDNS* and *SSDP*. The protocol classes are loaded from the information received from the service discovery module.
- **activity.py**: This file contains the most interesting classes. Class *Communication* helps to store unidirectional information about exchanged communication between a pair of devices. Class *Activity* is attached to the communication flow when a particular activity from the defined list

is identified. Class *Flow* is used as a key to a Python dictionary, and class *UNIREC* is an auxiliary class to store UniRec binary data into an object.

- **identification.py**: The file contains the class *Identification* for detection of a defined activity. Based on our measurements, we have implemented specific algorithms to identify the given activity, which are described later in this chapter. In this file, we have also implemented the auxiliary class *Label*, which assigns the label to the identified activity.

- **config.py**: The file contains the class *Config*, which verify and check all configuration parameters, and stores them in its attributes for further usage.

- **helper.py**: The file contains the auxiliary class *Messages*, which stores all messages and formatting for the output. It also defines tables of various services for the human-readable output.

3.2 Class Diagram

As we have described the module source code structure above, we look now in detail for each class and its purpose. In Figure 3.1, we can observe the class dependencies, their attributes, and methods. In the following pages, we take a look at each of the classes and explain their function in the final implementation.

Class Config

The first class, which is initialized. It takes the program parameters and configuration file as the input. The configuration file is in the standard INI format, where key/value string is parsed and loaded. Then it thoroughly verifies all parameters and the permissions for all specified files and directories. The exception class *ConfigError* is raised if any of the parameters are incorrect or user does not have sufficient permissions to read or write in specified paths.

Class Acid

The main class of the module which contains all information and references in its attributes. We initialize it after we successfully parse and verify all parameters. It serves as our internal database of all devices and communications stored in the class attributes. These attributes are dictionaries, where its MAC address uniquely identifies a device, and a pair of source and destination IP address identifies a communication. These structures help us to find an existing record quickly in our database. In case of a runtime error, the class raises exception class *ACIDEror* with the corresponding message.

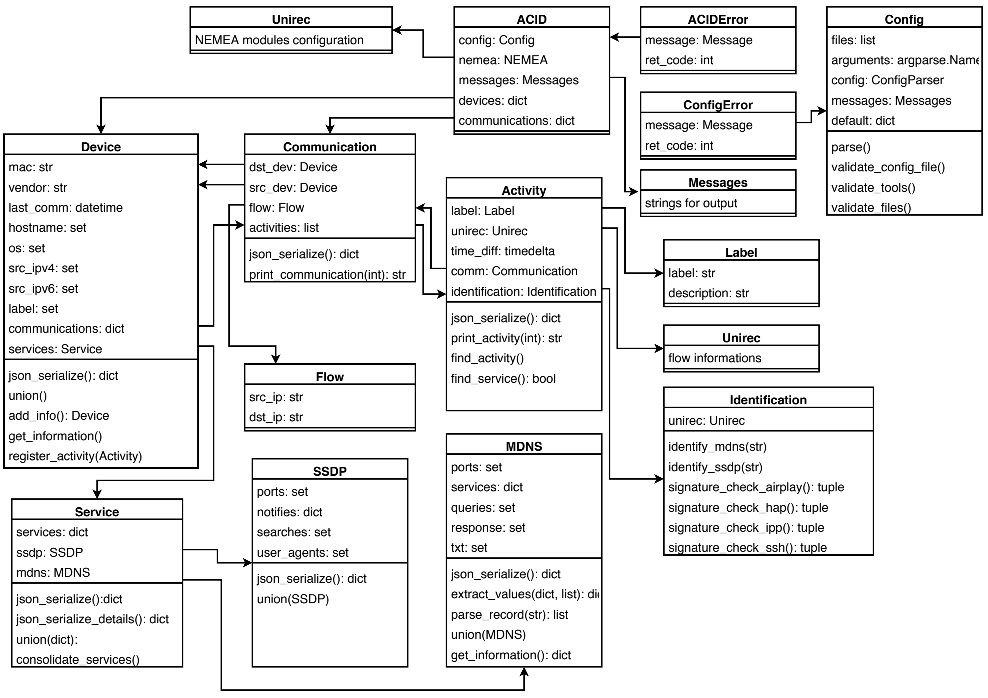

Figure 3.1: Module class diagram

Class Device

This class contains information about a single device. It loads the information from the service discovery module, and additionally from the PassiveAutodiscovery module if requested. Afterward, it consolidates all the services in one structure regardless of the source. In the case of an existing record, the class has builtin functions to merge the information. It contains the following list of attributes:

- **mac**: a string containing MAC address which identifies the device
- **vendor**: a string containing vendor determined from first three bytes of the MAC address
- **last_comm**: a time of a last seen communication from or to the device
- **model**: a string containing the model of a device
- **hostname**: a set of observed hostnames for a device
- **os**: a set of observed operating systems for a device
- **src_ipv4**: a set of observed IPv4 addresses for a device

- **src_ipv6**: a set of observed IPv6 addresses for a device
- **label**: a set of labels obtained from the PassiveAutodiscovery module
- **protocols**: a pointer to the auxiliary class containing information from service discovery protocols
- **services**: a pointer to consolidated observed services for effective processing
- **communications**: a dictionary of all communications from the device underlying identified activities

Some of the class attributes are updated with the information from the service discovery protocol. For example, it is not very accurate to specify the vendor from the MAC address as the device usually has a NIC from a third-party vendor. We also update a model type from the mDNS TXT record. The reason for having multiple IP addresses, hostnames, and operating systems are that we assume that the device can run multiple instances (e.g., a server with running virtual machines or containers) or obtain a different IP address from a DHCP server.

Class Communication

This class holds information about all unidirectional communication from a single device to another device. It holds the pointer to a source and destination device if a device exists in the database. The class *Flow* is its attribute and uniquely identifies the given communication. The most important attribute is the list of activities for the given pair of devices.

Class Activity

This class takes a given UniRec message and, based on the information, tries to identify the activity from our set of defined activities. It uses auxiliary methods to decide how to process the flow's information. It holds these essential attributes:

- **label**: a label of identified activity, e.g., AirPlay screen mirroring
- **description**: a description of the given activity in a human-readable format
- **time_diff**: a calculated time difference of the given communication
- **unirec**: a pointer to an instance of the UniRec message
- **communication**: a pointer to the communication class of the device pair

- **identification**: a pointer to a class for an activity identification algorithms

Class Identification

This class is bound to the class *Activity* and helps to identify the type of activity based on the input UniRec message. It uses decision tree-like algorithms to determine the observed activity successfully. The identification might include the traffic patterns in terms of packet lengths, transferred bytes, or parsing the HTTP headers. These classification algorithms are later described in Section 3.4.

Class Label

We have implemented this auxiliary class to assign the label for an identified activity, and it is bound to the class *Identification*. It holds descriptions and defined labels for the activities. We assign the label only after the successful classification of the flow.

Class Service

This auxiliary class was made for future expansion to accomplish the scalability as one of the non-function requirements. At the moment, it holds pointers to protocol classes *MDNS* and *SSDP*. The main purpose of this class is to consolidate information from service discovery protocols into one structure to optimize searching and code scalability.

Class MDNS

This class holds the essential information received from the service discovery module. The class loads the JSON into predefined structures. The sample of the received JSON is displayed in Figure 2.2. The attributes are the following:

- **ports**: a set of ports on which a device announces services using mDNS
- **services**: a dictionary of observed SRV records for the given port used as the key
- **queries**: a set of observed mDNS queries from a device
- **response**: a set of observed mDNS responses from a device
- **txt**: a set of observed mDNS TXT records with additional information about a device

Class SSDP

Similar to the class *MDNS*, this class holds attributes specific to SSDP and is loaded from the service discovery module. The attributes are the following:

- **ports**: a set of ports on which a device announces services using SSDP
- **notifies**: a dictionary of observed HTTP headers for the given port used as the key
- **searches**: a set of observed SSDP queries using HTTP M-SEARCH
- **user_agents**: a set of observed user agents with additional information about an OS

Class NEMEA

This class holds information specific to the NEMEA system. It initialize the auxiliary functions used for receiving the network traffic and prepares the UniRec template with information fields specified in the configuration file.

Class Flow

This auxiliary class serves as the key for a dictionary of communications, whether in class *ACID* or class *Device*. This is achieved by the Python data class attribute *frozen*, which generates a *__hash__()* method for the class. We use this hash to find the requested communication.

Class UNIREC

This class loads the flow from the binary format into the Python structure. Each instance of this object is assigned to the class *Activity*. This instance is used in the activity analysis, and later in the output.

Class Messages

This auxiliary class is a set of output messages containing various predefined text lines. We use it when the verbosity level is set to be true.

3.3 Module Workflow

Now, as we understand the module structure and its classes, we can highlight the module workflow, which is portrayed in Figure 3.2. After starting the *acid.py* module with Python 3.7 interpreter and mandatory configuration file argument, the program parses arguments and configuration file parameters.

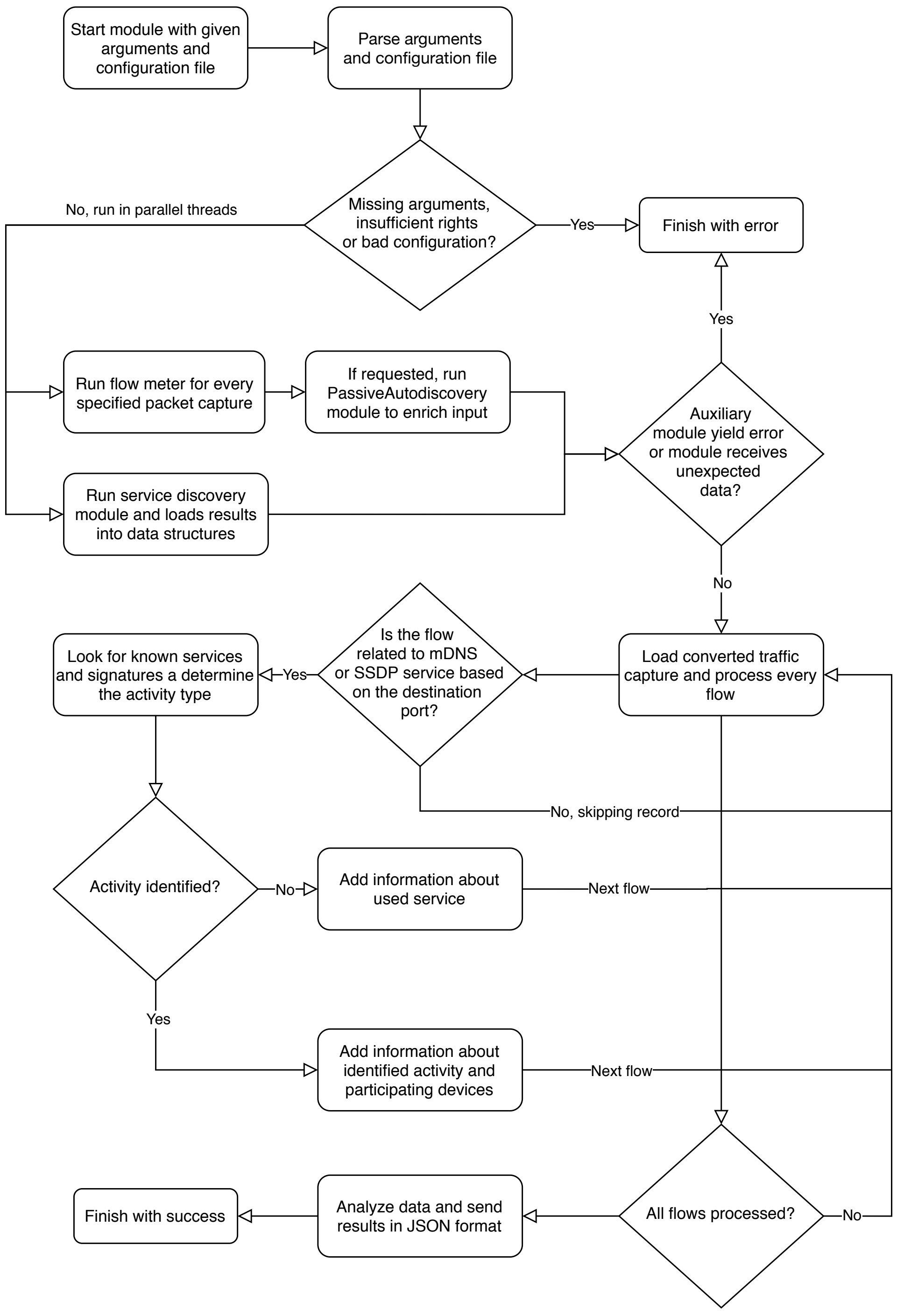
Start module with given arguments and configuration file
Parse arguments and configuration file
Missing arguments, insufficient rights or bad configuration?
Yes
Finish with error
No, run in parallel threads
Run flow meter for every specified packet capture
If requested, run PassiveAutodiscovery module to enrich input
Run service discovery module and loads results into data structures
Auxiliary module yield error or module receives unexpected data?
Yes
No
Load converted traffic capture and process every flow
Is the flow related to mDNS or SSDP service based on the destination port?
Yes
Look for known services and signatures a determine the activity type
No, skipping record
Activity identified?
No
Add information about used service
Next flow
Yes
Add information about identified activity and participating devices
Next flow
All flows processed?
No
Analyze data and send results in JSON format
Finish with success

Figure 3.2: Module flow chart

If the module detects any missing parameter or insufficient rights in the host's file system, it ends with an error.

If all initial checks are successful, the module creates two independent threads to process input data with the auxiliary modules. The first thread runs the service discovery module for each specified file. If a directory is specified, it processes every file within that directory. If a file in a directory is not a packet capture, the module ends with an error. In the future, we plan to check and add only PCAP files in a directory.

The second thread processes the packet captures with the flow meter module with selected plugins and stores the binary UniRec messages into a temporary file where its path is specified in the configuration. If analysis with PassiveAutodiscovery is requested, this thread loads the flow in this module and process the flows. The output is stored in the temporary file using a JSON format. The module loads the obtained information in the local database after the first thread finishes as we want to avoid any concurrent writing in the data structures.

If both threads successfully finish their operations, the module loads the temporary file with UniRec data prepared by the flow meter. For each bidirectional flow, we look at the destination device and determine if the destination port was announced using the service discovery protocols. If the destination port matches an announced service, we analyze the flow information. Based on known traffic different signatures, transferred bytes, or HTTP header information, we are able to determine the defined activity. If the flow does not match any defined activity, we store information about the usage of this service. All other records are skipped. Once we process every flow, the module outputs the results in a human-readable format to a network operator or pass the information in JSON to other modules of the NEMEA system.

3.4 Identification Methods

This section describes the used detection methods for identifying our defined set of activities in the ACID. As described in Section 2.4, we were focusing on a small subset of activities to demonstrate its functionality and maintain the space for future expansion due to the module's scalable nature. We were using decision-tree algorithms based on observed behavior we have simulated in our environment. Currently, the chosen thresholds are set manually, but we plan to automate the decisions using machine learning techniques. Our testing environment and devices we have used for the simulation will be more described in Section 4.1. For each following activity, we will describe its behavior used for the identification by our module.

Table 3.1: AirPlay RTSP Pairing

#	Source	Size	TCP flags	RTSP header
1	client	82	SYN	—
2	server	78	SYN, ACK	—
3	client	70	ACK	—
4	client	346	PSH, ACK	GET /info RTSP/1.0
5	server	70	ACK	—
6	server	1518	ACK	RTSP/1.0 200 OK
7	server	426	PSH, ACK	—
8	client	70	ACK	—
9	client	357	PSH, ACK	POST /pair-verify RTSP/1.0
10	server	327	PSH, ACK	RTSP/1.0 200 OK
11	client	70	ACK	—

Apple AirPlay activities identification

One of the first activities we wanted to observe and analyze is media streaming and screen sharing using the Apple Airplay service. It uses the Bonjour service to announce its presence in the network on a TCP port 7000. The service discovery module captures this announcement, and our module stores this information within the *Service* class for later.

When we process the network flows and match the AirPlay's destination port, we check for a known device pairing signature. In the pairing process, the devices agree on the random port to stream the screen or media. Unfortunately, this information is encrypted, and we see only the headers of the pairing exchange. Both for pairing and data exchange, the AirPlay leverages the stateful real-time streaming protocol (RTSP) [37], where its structure is very similar to HTTP.

If we detect this kind of pairing, we can expect another flow between these devices within a specified time range. When we observe this flow, the TCP handshake matches the packet lengths of the TCP handshake of respective device pairing. Even though this communication is fully encrypted, we can approximate based on the time of the connection and transferred bytes and packets, whether a user was sharing a screen or streaming media.

As discussed at the beginning of this section, we set this threshold manually, which works in our environment with various devices, but needs further testing and tweaking. For illustration, we list the RTSP exchange pairing with obtained information in Table 3.1. The client is the device that wants to stream data, and the server is the receiving screen. The size of the larger packets will depend on the maximum transmission unit (MTU). For example, the encrypted communication from the server in the sixth and seventh packet might fit within one packet if the MTU is larger than standard 1500 bytes.

SSH activities identification

In the beginning, determining the SSH activity seemed like a quite straightforward job. However, this problem is much more complex, and the topic for another article. There are so many variables that can influence the proper classification. As we do not process every packet of communication, but we use the aggregate flow information, we can not use any statistical methods. The most vital information for us, in this case, is the packets count transferred from and to the server, total transferred bytes from client or server, and time of the communication. Thus, we have manually set the threshold based on the annotated simulated dataset from the observation. We will revisit the other possibilities and methods of detection in the future.

IPP activities identification

As we mentioned earlier, IPP or IPPS operates on a TCP port 631. Whenever we see the flow matching this TCP port, we subject it for further inspection. The flow meter module can enrich the flow information for HTTP headers such as HTTP request method, request host, request URL, request agent, request referer, response code, and response content-type. We leverage these fields, especially the request method, with a specified URL to determine the printer's job. We also look at the sent or received packets to see the direction of the communication. We can assume that if the printer sends a large amount of data towards the client, we classify this job as scanning, where the copy is sent to the client.

Similarly, if the client sends a significant amount of bytes to the printer, we observe a printing job. To determine whether the device is a printer or not, we use the PassiveAutodiscovery module, which assigns label types for observed devices. We would see more information in the unencrypted payload; however, any content parsing plugin has not yet been implemented in the flow meter module.

Apple HomeKit activities identification

Similar to Apple AirPlay, Apple HomeKit leverages the Bonjour service to announce its presence in the network. It uses the stateless HTTP for device pairing over TCP on a port 80, and then a random port for communication exchanged during the pairing. However, we can not see the encrypted payload, which would tell us more information. To classify the flow as the HomeKit activity, we are leveraging a similar successful pairing signature with Apple Airplay. In Table 3.2, we exhibit the observed communication signature. Our module determines the pairing signature and classifies the flow as the HomeKit pairing. Based on this information, we expect a subsequent flow from the same source device to the destination controller. Since the payload is encrypted, we cannot determine the desired activity, only its execution.

Table 3.2: HomeKit HTTP Pairing

#	Source	Size	TCP flags	HTTP header
1	client	102	SYN	—
2	server	86	SYN, ACK	—
3	client	78	ACK	—
4	client	247	PSH, ACK	POST /pair-verify HTTP/1.1
5	server	298	PSH, ACK	HTTP/1.1 200 OK
6	client	78	ACK	—
7	client	336	PSH, ACK	POST /pair-verify HTTP/1.1
8	server	78	ACK	—
9	server	159	PSH, ACK	HTTP/1.1 200 OK
10	client	78	ACK	—

3.5 Label Requirements

Once we identify an activity based on the selected identification method, the main requirement is to assign a corresponding label with a human-readable description. This information is vital for further processing by a sequential module or a network operator. We have implemented the auxiliary class *Label*, which gives the contextual information to the given flow. If both devices are present in the database, we classify the flow for both directions. Thus, the server will have a different label than a client. For example, if a device A shared the screen to the device B, we assign a label "A user (with more information from the flow and the database) was mirroring a screen using AirPlay towards the television B" to the device A. On the contrary, the device B will receive a label with the information "I was projecting the device A's screen for 10 minutes". This contextual information is the main output of our module and can be used to be further processed within the NEMEA system.

Chapter 4

Testing and Evaluation

This chapter describes the testing environment and procedures for the created module. In the beginning, we introduce a testing environment we have created to study behaviour in the real network traffic. Furthermore, individual activities from Section 2.4 are verified. We also check other defined functionalities. Finally, the behavior of individual items from the profile is measured.

4.1 Testing Environment

Before we started implementing the final solution, we have simulated several scenarios in our home environment. We could easily control and capture the network traffic without any network administrator's restrictions. We have observed the network traffic, thoroughly analyzed the packet captures in WireShark [30], and compare our findings with the learned theory. We have tried to connect as many devices with different operating systems as we could to observe service discovery announcements. In Figure 4.1, we show all devices connected during our research.

Our network infrastructure consist of a small router for home environment, eight port switch with Power over Ethernet (PoE) [38] and wireless access point. All components are from the cloud-managed Cisco Meraki [39] family. Connected devices for observation were following:

Apple MacBook Pro

A first device is an Apple laptop with macOS 10.15, which we have used to capture all the network traffic. We set up a port mirroring on the Cisco Meraki switch to send us all packets from all ports, including the Cisco Meraki access point. The only caveat of this approach is that if two devices communicate wirelessly, the traffic is only processed on the access point. For this reason, we have connected most of the devices via an Ethernet cable. This scenario

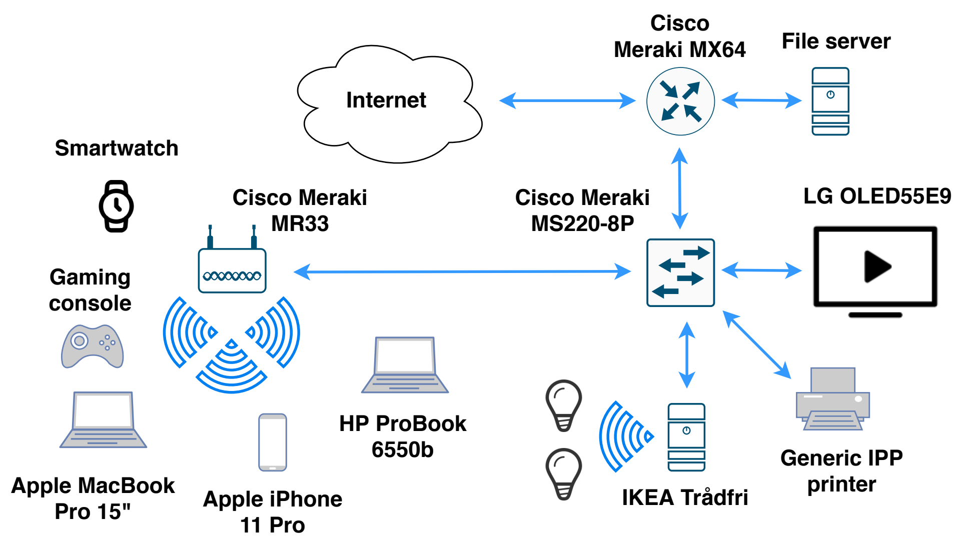

Figure 4.1: Testing environment

is specific to a small home network environment. In large campuses, network access points send all network traffic to the central wireless controller(s).

Apple iPhone 11 Pro

Mobility and working from anywhere become essential in our daily lives. Smartphones and tablets are more and more replacing standard computers. To represent this category of devices, we chosen an Apple smartphone with iOS 13.6 to monitor and capture various activities. We have also connected smartphones with Android [40] operating systems, but we have not observed any announced services using service discovery protocols fitting our scenarios.

Television LG OLED55E9

To simulate and capture the traffic from screen monitoring and screen sharing, we have connected an LG [41] smart television, which enables the media streaming and screen mirroring, among other protocols, using Apple AirPlay. From the network observations, this device is the most talkative in the network announcing its services using mDNS and SSDP. It also continually queries for other services in the network.

IKEA Trådfri

To represent the family of IoT devices, we decided to monitor the smart gateway from IKEA [42]. This gateway can control various products from IKEA portfolios such as light bulbs, sunblinds, power plugs, etc. using Zigbee [43] technology. The gateway announces Apple HomeKit technology in the network, and with a compatible device like Apple iPhone, we can control various things. In our environment, we were using smart bulbs to identify what actions are sent to the gateway.

HP ProBook 6550b

An older laptop from HP [22] to observe the behavior of Microsoft Windows 10. From the observations, unless Bonjour is installed, which is not by default, the device is not very talkative about running services.

File server

A simple network-attached storage (NAS) system which provides NFS, FTP, and SFTP services in the network. It runs Ubuntu [44] Linux Server 20.04. It runs Avahi daemon to announce its file services using mDNS. In this thesis, we were mainly focusing on SFTP file transfer. In future work, we want to discover and observe other protocols for file services.

Generic network printer

When we were defining our set of activities, we thought about the network printers, which are, in most cases, left without any proper security configured in the computer networks. We have chosen the standard IPP without which sends all the information in a plain text. However, we did not have this printer in our environment. For this reason, we have simulated all printer's jobs using the common UNIX printing system (CUPS) software to observe the exchanged communication. This observation provided us essential information on how to identify the printing jobs for the activity classification.

Other devices

As shown in Figure 4.1, we have connected and observed traffic from other sources as specified above, like a sports smartwatch or gaming console. However, we have not captured any particular traffic using service discovery protocols leading to activity identification. Many modern devices are synchronized via a central server in the cloud where the payload is encrypted. This observed synchronization for various services is now beyond our work.

4.2 Activity Evaluation

An essential part of the testing was the verification of defined activities from captured packets. We have simulated all defined scenarios in our environment while we were capturing the network traffic. Our initial observation's findings

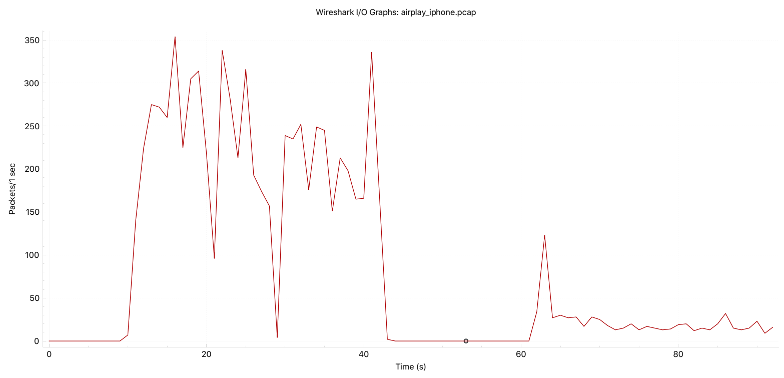

Figure 4.2: AirPlay screen sharing and multimedia streaming using Apple iPhone (source: Wireshark [30])

were critical to determining which information we need to obtain either from the auxiliary modules or the UniRec flows. This section describes the tests and simulations for each defined set of activities.

Apple AirPlay activities evaluation

We have already described how Apple AirPlay works, how the service is discovered, how the devices pair, and how the data stream is transferred. Now, we take a look, how to distinguish a specific type of activity, e.g., a media stream or screen mirroring. We have simulated these activities while capturing the traffic. For simulation, we have used devices capable of AirPlay streaming. In these tests, we have used the Apple iPhone and the Apple MacBook as the clients, and the LG television as the receiving server. We have performed several measurements to determine the type of activity from the encrypted data stream.

For illustration, we can take a look at Figure 4.2. We have connected the smartphone to the television at the mark of 10 seconds and shared the screen for 30 seconds. At the one minute mark, we have started to stream the high-resolution from the Internet for another 30 seconds. In the graph, we can see the high volume of traffic with screen sharing, but not with the high-resolution video. This is probably due to the RTSP *Content-Location* [45] header which contains the URL for the target server. This functionality is embedded within the smartphone's video player. We have confirmed this theory as the video traffic was not streamed from the smartphone, but the television connected to the video's location. The smartphone was then only controlling the playback of the video; thus, we can see the constant stream of traffic.

In Figure 4.3, we have tried to simulate a similar activity using the laptop. At the mark of 10 seconds, we have again started to share the screen for 30 seconds. In the first 20 seconds, we remained idle, and then we were changing

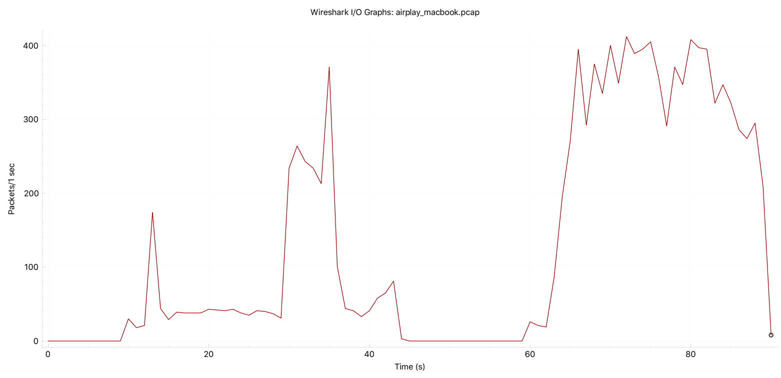

Figure 4.3: AirPlay screen mirroring with Apple MacBook (source: Wireshark [30])

screens and generating display changes. At the mark of one minute, we have started to share the screen and playing video in the full-screen mode for 30 seconds. In the beginning, the graph shows us the pairing and initial screen packets are sent to the television, creating a small spike. Then the traffic remains constant when we were idle, and for the last 10 seconds, we observe traffic spikes as the changes are sent to the screen. Lastly, playing the video generates a high peak of traffic, as we did not use any AirPlay supportive video player; hence, all the screen changes have to be continuously streamed to the television. Due to latency and delay, this might be a significant challenge for the user experience.

In these examples, we have described the identification of a specific activity. In our implementation, we rely on the aggregate flow information. Even though the UniRec record supports individual packet lengths statistics, this field is limited. In our module, we identify the activity based on the transferred bytes per second. We have established the threshold based on our simulations; hence we can only approximate whether a user was sharing a screen or streamed media.

SSH activities evaluation

In our environment, we have used a file server with SFTP capabilities running Ubuntu Linux. We could easily simulate the remote connection to the server or initiate a file transfer. Similar to Airplay, we have simulated both activities in one packet capture. In Figure 4.4, we can observe two different network flows to the SSH/SFTP server. At the mark of 10 seconds, we have initiated a remote session to the server for 40 seconds. In the beginning, we were mostly idle, listing directories, manual pages, and executing basic shell commands. After 20 seconds, we were outputting large log files, which generated the second peak illustrated in the graph. After one minute, we have initiated

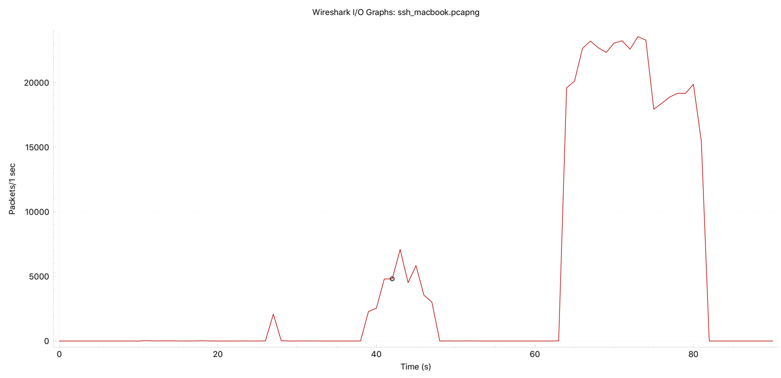

Figure 4.4: SSH session and SCP file transfer (source: Wireshark [30]

a file transfer of a large file for 20 seconds towards the server. The most towering peak shown in the graph is the representation of this file transfer. In this simple way, we can quickly determine the desired activity. Our module assigns the correct label based on the transferred bytes, direction, and packet count. As we have mentioned in Section 3.4, we are aware that this approach might not be accurate, and it would require more tuning.

IPP activities evaluation

As we mentioned in the previous section, we have simulated the network printer over IPP in our environment. We configured a pretended printer on the Apple MacBook laptop using built-in CUPS software. With the second laptop, we have added this network printer in the OS as the default printer. It was announced using the DNS-SD service; hence, we did not need to add its address manually. Within the observed communication, we have seen that simple network management protocol (SNMP) [46] is used to get information about the printer, its functions, and its capabilities. After, we have requested a print job of a sample document. From that information, we could determine the behavior of the IPP traffic. With a combination of the IPP documentation, we have implemented signatures for printing and scanning traffic patterns. We have successfully tested this activity identification within the simulated dataset.

Apple HomeKit activities evaluation

We wanted to leverage all available types of devices we have set up in our environment. One of them was the IoT gateway from IKEA, enabling the smart home remote control of various accessories via Zigbee like light bulbs via a smartphone application. The gateway can be integrated with modern home voice assistants and Apple HomeKit technology. For our module testing, we wanted to capture and classify the control of the lights. We have successfully identified the pairing from the smartphone and the subsequent action. As the service was designed with security as one of its main requirements, we can not classify the relevant action from the network flow information. We evaluate this finding as a very positive step in the IoT security domain. As the next step, we want to discover and evaluate other similar services that do not put security first.

4.3 ACID Output

After the module processes all the captured flows and identifies the given activities, it outputs the results in the desired format in the standard output or in a specified file. The results' format can be either human-readable or in a JSON data structure for further analysis. In Figure 4.5, we display the most interesting excerpt from the output. We have modified the values to ensure anonymity and we have omitted certain records for better readability. The output tells a network operator or to a subsequent analysis module that device with IP address 192.168.10.10 was communicating with two devices where with the first one the user was sharing screen for 30 seconds at a given time (omitted), and with the second, the device

4.4 Discussion

Our work aims to successfully identify a specific set of activities, which we can classify and assign corresponding labels for further analysis. Given the waste majority of options to detect, we focused on the small subset of these activities we can simulate in our environment. We have defined these sets of activities announcing their presence using service discovery protocols. Based on our initial observations, we considered them to be easily identifiable. However, we stumbled over more complex problems and decision making. Even though we cannot ensure the perfect accuracy, this module is the functional prototype of the future subsequent work. We have not seen any similar-like tool analyzing the network flows during our research, which outputs and presents the high-level description of the observed activity.

```
"device": {
  "mac": "f0:18:98:db:ca:fe",
  "vendor": "Apple, Inc.",
  "model": "MacBookPro15,1",
  "hostname": ["macbook.local"],
  "os": ["MacOS 19"],
  "ipv4": ["192.168.10.10"],
  "ipv6": ["fe80::10cb:fe14:409:3361"],
  "label": ["End Device","MacOS"],
  "last_comm": "2020-06-28 10:20:11.388000"},
"services": {
  "22": {
   "service": ["macbook._ssh._tcp.local"],
   "source": "mdns"
  },
},
"communications": [{
  "src_ip": "192.168.10.10",
  "dst_ip": "192.168.10.25",
  "activities": [
   {"time": "0:00:00.015000",
    "label": "airplay_pairing",
    "description": "The device has paired using Ai..."},
   {"time": "0:00:29.8175600",
    "label": "airplay_screen_mirroring",
    "description": "User was sharing screen for 3..." },
  },
  {
  "src_ip": "192.168.10.10",
  "dst_ip": "192.168.10.11",
  "activities": [
   {"time": "0:00:06.817000",
    "label": "ipp_printing",
    "description": "The user has requested a pri..." },
   {"time": "0:00:20.913000",
    "label": "ssh_remote_session",
    "description": "Client initiated an SSH remot..."},
}]
```

Figure 4.5: ACID output

Conclusion

One of the main pillars of network security is network monitoring that plays an essential role in getting visibility into network traffic. However, the currently existing tools provided rather low-level information about active devices and sent/received network packets, which are usually aggregated into network IP flows. This information is beneficial for experienced network operators and security analysts, who can identify security threats based on their knowledge about legitimate network behavior of their operated devices. Unfortunately, this professional manner of information is not easily understandable for the majority of users.

For the maximal comfort of users in the application scope, many network services run on background and exchange messages for automatic "self-discovery". In this way, the incorporation of new devices or discovering existing services on the local network is done automatically and independently from a user. As a result, it is much more straightforward for users to find their devices and use them. However, we studied such protocols, and we discovered how much useful information the devices leak in local networks.

This work aims to analyze the network traffic in combination with information from the service discovery protocol (SDP) messages to estimate the real purpose of the communication. Contrary to the existing monitoring approaches, this work goes beyond the current interpretation of network flows based on IP addresses, protocols, and ports, which are useful information for experienced network experts. This work focuses on the identification of high-level events, i.e., activities from the users' perspective. The main goal is to utilize announced information by devices and services to assign the meaning of the IP flows.

This thesis listed some example activities of real network devices that interacted directly with a user or via another device. The main contribution is higher reliability of the assigned labels to the communication than can be reached by straightforward checking port numbers of well-known services.

During this work, we have developed a software prototype of an analyzer called ACID. This module receives information from several information sources, such as SDP messages (of 2 main protocols) and IP flow data. The ACID software implements the described scenarios and estimates the labels for the traffic, where the labels represent user activities. As a result, the module provides information about "shared screen from a laptop on a smart TV" instead of low-level information about communication protocol and port. We believe this kind of information is much more helpful for both network operators and security analysts who try hard to understand the traffic and achieve situational awareness.

To evaluate the functionality of the developed software module, we have created and set up a new laboratory environment. This environment consists of various types of real devices that can be used by a user. IP flow tools simultaneously monitored the whole network infrastructure, and we have captured some traffic samples of SDP traffic in the form of packets. This effort focused on comprehensive testing of the designed and developed ACID module. Also, we used the created infrastructure to create datasets to evaluate the developed methods for identification. In this thesis, we have evaluated the achieved results of the tests and the required hardware resources for the module's deployment.

Future work

Nowadays, there are countless types of devices with different services connecting to a network. Hence, it is hard to keep track of all different vendors and specific services. For the future work, we would like to automate the discovery of new services and their related activities by leveraging machine learning techniques.

Another possible improvement is a simplification of the entire workflow, thus reducing the module's run time. As the module depends on the preprocessed information from mDNS and SSDP packets, the per-packet analysis slows down the entire workflow. Plugins for the flow meter module are being developed based on the prototype created in this thesis. In the future, both mDNS and SSDP application layer information will be exported in UniRec format directly, so our module will not require to run the service discovery module.

Bibliography

[1] Bartoš, V.; Žádník, M.; et al. Nemea: Framework for stream-wise analysis of network traffic. *CESNET, ale, Tech. Rep*, 2013. Available from: <https://github.com/CESNET/NEMEA>

[2] Statista. Fog Computing and the Internet of Things: Extend the Cloud to Where the Things Are. Cisco. [online], 2015 [cit. 2020-07-23]. Available from: <https://www.cisco.com/c/dam/en_us/solutions/trends/iot/docs/computing-overview.pdf>

[3] Szigeti, T.; Zacks, D.; et al. *Cisco Digital Network Architecture: Intent-based Networking for the Enterprise*. Cisco Press, 2018.

[4] Claise, B. Cisco Systems NetFlow Services Export Version 9. RFC 3954, IETF, Oct. 2004. Available from: http://tools.ietf.org/rfc/rfc3954.txt

[5] Networks, A. *The Aruba Mobile First Architecture*. Aruba Networks, 2018.

[6] Networks, A. *Aruba AirWave*. Aruba Networks, 2019.

[7] Claise, B.; Trammell, B.; et al. Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of Flow Information. RFC 7011, IETF, Sept. 2013. Available from: http://tools.ietf.org/rfc/rfc7011.txt

[8] CESNET, z.s.p.o: CESNET, z.s.p.o. 1996–2014. Available from: <https://www.cesnet.cz>

[9] Lyon, G. Zenmap-official cross-platform nmap security scanner gui. *URL ¡http://nmap.org/zenmap¿*, 2011.

[10] Cheshire, S.; Krochmal, M. Multicast DNS. RFC 6762, IETF, Feb. 2013. Available from: http://tools.ietf.org/rfc/rfc6762.txt

[11] Cheshire, S.; Krochmal, M. DNS-Based Service Discovery. RFC 6763, IETF, Feb. 2013. Available from: `http://tools.ietf.org/rfc/rfc6763.txt`

[12] Apple Inc.: Apple. 1976–2020. Available from: <`https://apple.com`>

[13] Cheshire, S.; Krochmal, M. Requirements for a Protocol to Replace the AppleTalk Name Binding Protocol (NBP). RFC 6760, IETF, Feb. 2013. Available from: `http://tools.ietf.org/rfc/rfc6760.txt`

[14] Cheshire, S.; Aboba, B.; et al. Dynamic Configuration of IPv4 Link-Local Addresses. RFC 3927, IETF, May 2005. Available from: `http://tools.ietf.org/rfc/rfc3927.txt`

[15] Thomson, S.; Narten, T.; et al. IPv6 Stateless Address Autoconfiguration. RFC 4862, IETF, Sept. 2007. Available from: `http://tools.ietf.org/rfc/rfc4862.txt`

[16] Gulbrandsen, A.; Vixie, P.; et al. A DNS RR for specifying the location of services (DNS SRV). RFC 2782, IETF, Feb. 2000. Available from: `http://tools.ietf.org/rfc/rfc2782.txt`

[17] Mockapetris, P. Domain names - implementation and specification. RFC 1035, IETF, Nov. 1987. Available from: `http://tools.ietf.org/rfc/rfc1035.txt`

[18] Apple Bonjour Open Source Software. 2002–2020. Available from: <`http://developer.apple.com/bonjour/`>

[19] Microsoft Corporation: Windows OS. 1975–2020. Available from: <`https://microsoft.com`>

[20] Avahi: Avahi. 2004–2020. Available from: <`https://www.avahi.org`>

[21] Goland, Y. Y.; Cai, T.; et al. Simple service discovery protocol. 1999.

[22] HP Inc.: Hewlett-Packard. 1939–2020. Available from: <`https://hp.com`>

[23] Goland, Y.; Schlimmer, J. Multicast and unicast UDP HTTP messages. *IETF draft, internet engineering task force*, 1999.

[24] Donoho, A.; Bryan, R.; et al. UPnP device architecture 2.0. *Architecture*, 2015: pp. 1–196.

[25] Majkowski, M. Stupidly simple ddos protocol (SSDP) generates 100 gbps ddos. 2017. Available from: <`https://blog.cloudflare.com/ssdp-100gbps/`>

[26] Open Connectivity Foundation: OCF. 2016–2020. Available from: `<https://openconnectivity.org>`

[27] Havránek, J. *Exportér síťových toků s podporou aplikačních informací.* B.S. thesis, České vysoké učení technické v Praze. Vypočetní a informační centrum., 2017.

[28] Sedláček, O. service-discovery-prototype. 2020. Available from: `<https://github.com/xsedla1o/service-discovery-prototype>`

[29] Koumar, J. *Automatické rozpoznávání síťových zařízení a jejich závislostí.* B.S. thesis, České vysoké učení technické v Praze. Vypočetní a informační centrum., 2020.

[30] Combs, G. Tshark—dump and analyze network traffic. *Wireshark*, 2012.

[31] Plummer, D. An Ethernet Address Resolution Protocol: Or Converting Network Protocol Addresses to 48.bit Ethernet Address for Transmission on Ethernet Hardware. RFC 826, IETF, Nov. 1982. Available from: `http://tools.ietf.org/rfc/rfc0826.txt`

[32] Alliance, W.-F. Wi-Fi Protected Access: Strong, standards-based, interoperable security for today's Wi-Fi networks. *White paper, University of Cape Town*, 2003: pp. 492–495.

[33] Larsen, R. How Apple AirPlay & AirPlay Mirroring Works. 2012.

[34] Dusi, M.; Este, A.; et al. Using GMM and SVM-based techniques for the classification of SSH-encrypted traffic. In *2009 IEEE International Conference on Communications*, IEEE, 2009, pp. 1–6.

[35] Herriot, R.; Butler, S.; et al. Internet Printing Protocol/1.0: Encoding and Transport. RFC 2565, IETF, Apr. 1999. Available from: `http://tools.ietf.org/rfc/rfc2565.txt`

[36] Ciabarra, M. WiFried: iOS 8 WiFi Issue. Medium.com. [online], 2014 [cit. 2014-11-25]. Available from: `<https://medium.com/@mariociabarra/wifried-ios-8-wifi-performance-issues-3029a164ce94>`

[37] Schulzrinne, H.; Rao, A.; et al. Real Time Streaming Protocol (RTSP). RFC 2326, IETF, Apr. 1998. Available from: `http://tools.ietf.org/rfc/rfc2326.txt`

[38] Group, I. . W.; et al. IEEE standard for broadband over power line networks: Medium access control and physical layer specifications. Technical report, Tech. Report, 2010.

[39] Meraki, C. Cloud managed networks that simply work. *meraki.cisco.com [online], [cit. 2013/08/15].* Available from: `<http://meraki.cisco.com/products/systems-manager>`

[40] DiMarzio, J. F. *Android.* Tata McGraw-Hill Education, 2008.

[41] LG Corporation, LG Display. 1947–2020. Available from: `<https://www.lg.com/>`

[42] IKEA: Trådfri. 1943–2020. Available from: `<https://ikea.com/>`

[43] Kinney, P.; et al. Zigbee technology: Wireless control that simply works. In *Communications design conference*, volume 2, 2003, pp. 1–7.

[44] Canonical Ltd.: Ubuntu OS. 2004–2020. Available from: `<https://ubuntu.com/>`

[45] Vasseur., C. Unofficial AirPlay Protocol Specification. 2012 [cit.2012-03-20]. Available from: `<https://nto.github.io/AirPlay.html#videon>`

[46] McCloghrie, K.; Rose, M. Structure and identification of management information for TCP/IP-based internets. RFC 1065, IETF, Aug. 1988. Available from: `http://tools.ietf.org/rfc/rfc1065.txt`

Appendix A

Acronyms

ACID Activity Identification

AMF Aruba Mobile First

AWDL Apple Wireless Direct Link

BSD Berkeley Software Distribution

CLI Command-Line Interface

CTU Czech Technical University

CUPS Common UNIX Printing System

DDoS Distributed Denial of Service

DHCP Dynamic Host Resolution Protocol

DNA Digital Network Architecture

DNAC DNA Center

DNS Domain Name System

DNS-SD DNS-Based Service Discovery

FTP File Transfer Protocol

GUI Graphical User Interface

HP Hewlett-Packard

IDS Intrusion Detection System

IETF Internet Engineering Task Force

IFC Interface

IP Internet Protocol

IPFIX IP Flow Information Export

ISE Identity Services Engine

ISP Internet Service Provider

JSON JavaScript Object Notation

LAN Local-Area Network

MAC Media Access Control

mDNS Multicast Domain Name System

NAS Network Attached Storage

NAT Network Address Translation

NBD Name Binding Protocol

NCP Network Controller Platform

NDP Network Data Platform

NEMEA Network Measurement Analysis

NFS Network File System

NREN National Research and Education Network

OOP Object-Oriented Programming

OS Operating System

PAD PassiveAutodiscovery

PoE Power over Ethernet

RAM Random Access Memory

RFC Request for Comments

RTSP Real Time Streaming Protocol

SDA Software-Defined Access

SDN Software-Defined Networking

SDP Service Discovery Protocols

SFTP SSH File Transfer Protocol

SNMP Simple Network Management Protocol

SSDP Simple Service Discovery Protocol

SSH Secure Shell

ST Search Target

TCP Transport Control Protocol

TRAP Traffic Analysis Platform

UDP User Datagram Protocol

UniRec Unified Record

URI Uniform Resource Identifier

URL Uniform Resource Locator

UPnP Universal Plug and Play

WAN Wide-Area Network

Wi-Fi Wireless Fidelity

Appendix B

Installation Manual

B.1 NEMEA System Installation

We have tested and developed our module on the Debian Linux distribution system. To successfully run and verify our module, we need to download and install package dependencies to compile and install the NEMEA system.

GIT installation:

```
apt install git
```

NEMEA recursive cloning from GitHub:

```
git clone --recursive https://github.com/CESNET/nemea
```

NEMEA dependencies installation:

```
apt install -y bc bison autoconf automake gcc gcc-c++ flex \
               libidn11-dev libpcap-dev libtool libxml2-dev \
               make pkg-config python3-devel python3-pip
```

Python installation:

```
apt install python3
```

NEMEA installation:

```
cd nemea/
./bootstrap.sh
./configure --enable-repobuild --prefix=/usr \
            --bindir=/usr/bin/nemea --sysconfdir=/etc/nemea \
            --libdir=/usr/lib64
make
make install
```

NEMEA-Framework installation:

```
cd nemea-framework
./bootstrap.sh
./configure
make
make install
mkdir -p /usr/local/lib64/python3.6/site-packages/
cd pytrap
python3 setup.py install
mkdir -p /usr/local/lib/python3.6/site-packages/
cd ../pycommon
python3 setup.py install
```

B.2 ACID Installation

After we successfully install the NEMEA system, we need to install specific Python package dependencies. We use the Python builtin pip package system. The required packages are listed in the *requirements.txt* file.

NEMEA installation:

```
cd acid
pip3 install -r requirements.txt
```

Appendix C

Module Usage

C.1 Module Usage

Running the ACID module is straightforward, we go in the source code directory and runs the module with the configuration file. In the configuration file we set the directories for auxiliary modules and a path to the directory with packet captures.

```
cd acid
python3 acid.py -c ../acid.ini
```

C.2 Module Help

The ACID is a standard a command-line tool with configurable arguments to run the program. In the following example, we can see the help output for module usage.

Help

```
usage: acid.py [-h] -c CONFIG [-d PCAP_DIR] [-f FILE] [-j]
               [-o OUTPUT] [-p] [-P PAD_DIR] [-S SDP_DIR]
               [-T TMP_DIR] [-v] [-vv]

Analyze flows from service discovery protocols
and identify activities from communications.

optional arguments:
  -h, --help            show this help message and exit
  -c CONFIG, --config CONFIG
                        Configuration file.
  -C, --clean           Clean temporary files.
  -d PCAP_DIR, --pcap_dir PCAP_DIR
                        Path to a directory with PCAP files
                        to be analyzed. Multiple directories
                        allowed delimited by ':'.
  -f FILE, --file FILE  Specification of the input file(s)
                        in PCAP format. Multiple files
                        allowed delimited by ':'.
  -j, --json            Stores results in JSON format.
  -o OUTPUT, --output OUTPUT
                        Output file to store results.
  -p, --print           Print results in the standard output.
  -P PAD_DIR, --pad_dir PAD_DIR
                        A path for the passive
                        auto-discovery module.
  -S SDP_DIR, --sdp_dir SDP_DIR
                        A path for the service discovery module.
  -T TMP_DIR, --tmp_dir TMP_DIR
                        A path for a temporary directory
                        to store auxiliary files.
  -v, --verbose         Be verbose.
  -vv                   Be more verbose.
```

Appendix D

Contents of Enclosed CD

```
acid ............................... main module source code directory
 └─ acid ............................... Python source code directory
     └─ __init__.py ..............................................
     └─ acid.py ..................................................
     └─ activity.py ..............................................
     └─ config.py ................................................
     └─ device.py ................................................
     └─ helper.py ................................................
     └─ identification.py ........................................
     └─ service.py ...............................................
 └─ acid.ini ......................... configuration file of the module
 └─ README.md ........................ module description and manual
 └─ LICENSE ................................ used source code license
 └─ requirements.txt ......... archive with the module implementation
└─ thesis .................................. thesis source code directory
 └─ thesis.tar.gz ......... archive with LaTeX source codes of the thesis
```

Figure D.1: Contents of enclosed CD