POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Towards Improving the Security and Privacy of Discovery Protocols in IoT

ITZAEL JIMENEZ ARANDA
Département de génie informatique et génie logiciel

Mémoire présenté en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*
Génie informatique

Août 2021

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Ce mémoire intitulé :

Towards Improving the Security and Privacy of Discovery Protocols in IoT

présenté par **Itzael JIMENEZ ARANDA**
en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*
a été dûment accepté par le jury d'examen constitué de :

Jinghui CHENG, président
Pierre LANGLOIS, membre et directeur de recherche
David BARRERA, membre et codirecteur de recherche
Frédéric CUPPENS, membre

DEDICATION

To my family for all the support during the journey of this master's degree

RÉSUMÉ

Les appareils IoT domestiques reposent sur des protocoles de découverte de réseau pour l'intégration et la gestion. Bien que ces protocoles de découverte (à la fois propriétaires et standardisés) offrent une expérience utilisateur transparente pour connecter des appareils aux applications associées, ils peuvent être abusés pour révéler d'autres appareils sur le réseau ou apprendre des informations privées sur l'environnement réseau. Les implications en matière de sécurité et de confidentialité des protocoles de découverte de réseau sont généralement bien comprises sur l'Internet des ordinateurs, mais leur impact sur les réseaux IoT en particulier n'a pas encore été entièrement évalué. Cet memoire présente une analyse empirique sur l'utilisation de protocoles de découverte tels que mDNS, UPnP et des variantes propriétaires sur les appareils IoT grand public. Nous avons analysé un hub domotique, ainsi qu'un petit ensemble d'appareils locaux et un grand ensemble de données publiques de trafic IoT. Nous avons constaté que les protocoles de découverte sont non seulement fréquemment utilisés par les appareils IoT, mais qu'il existe de nombreux cas de mauvaise configuration conduisant à des vulnérabilités. Nous avons conclu en fournissant un ensemble de recommandations de sécurité et d'outils de validation de principe aux administrateurs IoT pour renforcer leurs réseaux sans perdre les avantages offerts par les protocoles de découverte.

ABSTRACT

Home IoT devices rely on network discovery protocols for onboarding and management. While such discovery protocols (both proprietary and standardized) provide seamless user experience for connecting devices to companion apps, they can be abused to reveal other devices on the network or learn private information about the network environment. The security and privacy implications of network discovery protocols are generally well-understood on the Internet of computers, but their impact on IoT networks specifically has not yet been fully evaluated. This thesis presents an empirical analysis on the use of discovery protocols such as mDNS, UPnP, and proprietary variants on consumer IoT devices. We analyzed a home automation hub, as well as a small set of local devices and a large public dataset of IoT traffic. We found that discovery protocols are not only frequently used by IoT devices, but there are numerous instances of misconfiguration leading to vulnerabilities. We concluded by providing a set of security recommendations and proof-of-concept tools for IoT administrators to harden their networks without losing the benefits afforded by discovery protocols.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF APPENDICES

CHAPTER 1 INTRODUCTION

The Internet of Things (IoT) has seen incredible uptake in recent years. It is forecasted that IoT will take up 50 percent of the network connections by 2023, representing an increase of 17 percent over 2018. It is also expected to have 14.7 billion IoT devices connected by 2023, half of them being smart homes devices [1]. Moreover, it is expected to have 500 billion devices at all connected to the internet by 2030 [2], and an increase in the percentage of IoT devices included in this expected quantity.

Lower costs and increased functionality have driven the adoption of IoT in many sectors, including home networks which are often administered by non-technical end users. Given the wide range of functionality of IoT, many end users now find themselves managing a multitude of smart light bulbs, smart gadgets, and smart appliances.

IoT provides users with the control of lighting, heating, ventilation, air conditioning and other appliances, as well as to lock and unlock doors. Moreover, users can receive notifications when a washer/dryer has completed its cycle, when the temperature of a room has changed or when movement in a room has been detected. These features bring users comfort, automation of tedious tasks and reduction of costs. Users can control their devices when they are on bed or siting on a sofa, schedule regular tasks and reduce ongoing costs by scheduling devices operation of considerable power consumption like heating.

IoT devices can be managed wirelessly since many use wireless technology. The most popular technologies used by IoT devices are WiFi, Bluetooth, Radio Frequency Identification (RFID) and ZigBee. Furthermore, IoT devices can be also managed through the internet to provide remote access and status monitoring via smarthpones.

In this thesis our study case are WiFi capable IoT devices, and from now on we will refer to WiFi capable IoT devices as IoT devices. IoT devices are commonly managed through smart phones. IoT vendors make available companion mobile apps that serve as the medium to manage their devices. Even though computers can be used to send commands to IoT devices, it is likely to use smart phones as the way to send such commands through a mobile app due to the practicality and comfort it offers.

IoT devices can be operated regardless of the end-users' knowledge in networking, since these devices allow an easy way of configuration and management. End-users use mobile companion apps to set up and operate their IoT devices. In the set up stage, IoT devices act as a hotspot, and their companion apps connects to it. When this connection is successfully

established, companion apps deliver the network credentials for IoT devices to connect to the network, Figure 1.1 shows this onboarding process.

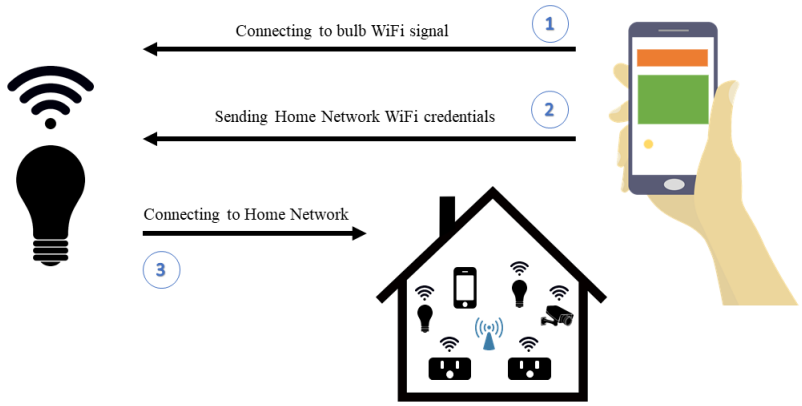

Figure 1.1 Connecting an IoT device to a network

Once end-users set up the IoT devices to connect them to the network, companion apps now serve for the managing stage. In this stage end-users can manage their IoT devices immediately after opening companion apps. Companion apps discover and deliver to end-users the services provided by IoT devices discovered. That is, there is no necessary action from end-users to discover and deliver those services. This process gives end-users the feeling of being able to manage their IoT devices immediately after opening companion apps, without any previous action involved.

Additionally, users can choose to use home automation hubs to manage their IoT devices for enhancing interaction. Home automation hubs can be used to operate IoT devices through the voice. Nevertheless, home automation hubs grant users the opportunity to gather the management of IoT devices in one place. That is, users do not need to have several companion apps to manage devices of different vendors. Home automation hubs keep the ease of configuration and management as IoT devices provide by themselves.

Even though IoT devices bring many benefits, they also come with a lot of risks [3] and with the emerging of IoT, new challenges in security are faced. Since IoT involves a wide variety of devices with diverse functionalities, this broaden the attack surface.

1.1 Discovery protocols

Network discovery protocols are used by devices to search for and/or notify other devices about services on the network. Devices providing a service (e.g., a printer, or a light bulb) will answer to devices searching for that specific service. If a response is received when searching

for a service, the requesting device will take further steps to configure the service on the local system. For example, the network address and port number of the printing service may be stored locally for future use. This mechanism allows users to skip device-by-device configuration of the same target service.

We argue that one of the critical underlying technologies that has enabled the adoption of IoT devices among a non-technical user base are *network discovery protocols*. Discovery protocols facilitate service and device discovery and management in local networks. IoT devices, apps, hubs and computers can advertise their services, or respond to discovery message requests from other devices on the network. IoT devices often lack displays and identifying IP addresses, port numbers, or other network information for setting them up would be challenging by non-technical users and demanding for technical users. Discovery protocols seamlessly solve this issue, allowing devices that have joined the network to be automatically added to companion apps and hubs. However, messages exchanged via standard discovery protocols are sent in clear text over multicast or broadcast transmissions, allowing any interested party, including attackers on the local network to learn about available services.

1.2 Problem

Usage of discovery protocols is based on the implicit assumption that all participants on the network are known and trusted.

But what if all devices on the network are not trusted? This appears to be increasingly true in many environments as more devices are given network access. IoT device vendors may be curious about what other devices a user might own, perhaps to offer purchasing incentives or loyalty discounts. App developers may want to monetize data about user devices. Friends, family and visitors accessing a network as guests may use compromised (or deliberately malicious) devices to learn details about the IoT devices connected to the network. Collected information generally involves vendor, type and model. These actions can be stealthily executed *with a single packet*, the discovery message, avoiding any noisy network scanning activity, battery drain, or other telling signs that a reconnaissance attack is underway. Moreover, discovery protocols may reveal the device's physical location, as this information is routinely announced in discovery protocols for usability reasons. Users may choose to name their devices according to their physical location (e.g., nursery webcam or basement lights), and these friendly names are transmitted in discovery protocol messages.

Previous work has shown that discovery protocols can be used to perform attacks on traditional IT environments, such as passive enumeration [4–6], which differs from an active

enumeration that common scanners perform, since there is no a direct interaction with the target. The services announced with discovery protocols are prone to be spoofed [7–9]. Discovery protocols also can be used in distributed denial of service (DDoS) attacks [10]. Discovery protocol client libraries have also been subject to buffer overflow attacks [11].

The literature is equally rich with proposals to improve discovery protocols by either adding custom (i.e., non-standard and rarely adopted) security features [12, 13], or replacing the protocols entirely with secure variants [14]. We note that while these secure protocol variants may indeed increase security according to some threat model, they require IoT vendor buy-in, as well as software/firmware updates to the billions of devices already on the market, which is nearly impossible.

Discovery protocols have been thoroughly analyzed in the context of the Internet of computers. However, IoT devices have received less attention in this area. Thus, in this thesis, we investigate the prevalence and security issues of discovery protocols on consumer IoT devices.

We assessed the impact in terms of security concerns of discovery protocols being used in consumer IoT devices through empirical experiments. We also tried to address the discovered risks as an initial attempt, while keeping the usability that discovery protocols provide.

1.3 Research Objectives

The primary objective of this work is to determine security concerns in IoT networks due to the use and misuse of discovery protocols through empirical experiments to provide security recommendations. We have defined the following specific objectives:

- O.1 Identify the use of discovery protocols in IoT devices and assess devices' behaviour
  - O.1.1 Perform a static and dynamic analysis of an open source home automation hub
  - O.1.2 Perform an empirical analysis on both a small local IoT device network as well as a large public dataset
- O.2 Identify the security concerns on using discovery protocols by IoT devices
- O.3 Provide security recommendations and proof-of-concept tools to enhance security while keeping discovery protocols usability benefits

1.4 Organization of the Thesis

This document is organized as follows:

Chapter 2 presents the literature review starting with a technical background on discovery protocols used in IoT devices. Then, we present a taxonomy on discovery protocols, going from abstract concepts to tools that allow the use of discovery protocols. Finally, a discussion on attacks using discovery protocols on traditional IT environments is given, as well as a review of published proposals to improve discovery protocols.

Chapter 3 presents the methodology used to approach the research. First, we show the assumptions of analyzing the internals of a home automation tool. Secondly, we present the design of our test bed and the considerations taken for this. Finally, we expose the data set to be analyzed and the examination approach.

Chapter 4 presents the results of our empirical analysis and findings as well as a discussion about them.

Chapter 5 presents our recommendations to protect users against the security concerns.

Chapter 6 presents a summary of the work and contribution. Also it presents limitations and future work.

CHAPTER 2 LITERATURE REVIEW

This chapter presents a taxonomy on discovery protocols which covers *Zero configuration networking* and the technologies involved. It also covers the implementations of discovery protocols and existing tools making possible to use their benefits. Our primary focus is on the higher-level concepts rather than technical aspects.

The chapter details technical aspects of multicast Domain Name System (mDNS), Universal Plug and Play (UPnP) and vendor-specific protocols. Furthermore, discussion on attacks that use discovery protocols is covered. Finally a discussion on proposals to improve discovery protocols is presented.

2.1 Network data communication methods

There are mainly three network communication methods to transmit data: unicast, broadcast and multicast. Unicast refers to a communication from one point to one point, broadcast from one point to all points, and multicast from one point to a selected group of points. Even though it is possible to find these network communication methods in the Internet Protocol version 6 (IPv6), we will focus the discussion on IPv4. We note that IPv6 is largely unused in the IoT domain, where a vast majority of devices rely exclusively on IPv4 connectivity.

These network communication methods are essential for discovery protocols to work properly. Each communication method has its own characteristics and advantages as described below.

2.1.1 Unicast

In unicast transmissions there is one sender and one receiver in the communication as Figure 2.1 shows. That is, the packet is sent to one destination host, and the host is expected to process the packet. Unicast is used by Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) to transmit packets [15, 16].

Accessing a website through a web browser or making a SSH connection, are examples of unicast communications. Another example is sending a command to turn on or off a smart bulb through a smartphone.

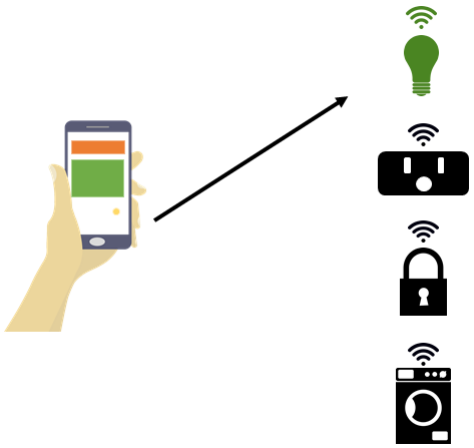

Figure 2.1 Unicast communication method. A single device (a smartphone) communicates with a single device (a lightbulb).

2.1.2 Broadcast

In broadcast transmissions, one sender transmits a packet to all the hosts in the network as Figure 2.2 shows. Broadcast can be seen as the radio signals which are propagated in the air for anyone who wants to catch them. In networking it means sending the packet to all the hosts in a subnet. Broadcast helps when the receivers' IP or MAC address are not known. It avoids hosts sending a unicast packet to each device connected to the network. The network will deliver a broadcast packet to all devices connected to the network itself. Broadcast uses the UDP transport protocol to transmit packets.

In broadcast transmissions, we can find two IPs to be used for sending the packet: 255.255.255 .255 and 0.0.0.0. These IPs means that the packet needs to be sent to all the host connected to the same network. That is, if a host with IP 192.168.1.10 sends a packet to the IP 255.255.255.255 (or 0.0.0.0), the packet will be sent to all the active hosts on the network segment 192.168.1.0 /24. Thus, every active host on that network segment has to check the packet to validate if the host is an expected receiver [16, 17].

Moreover, in broadcast a directed broadcast address can be used. This is intended to reach all the hosts but ideally on a remote network. These addresses have the particularity to use 255 in the last octet of the network IP to reach. That is, if the host with IP 192.168.1.10 wants to send a broadcast message to the network segment 10.10.10.0/24, it has to use the directed broadcast IP 10.10.10.255 to send the packet to all the active hosts on that network segment [16, 17]. However, the fact that a device can reach a unfamiliar network to send a message to all the devices belonging to that network, is risky. This exposes the devices

Figure 2.2 Broadcast communication method

to Denial of Service (DoS) attacks, because the devices in the unfamiliar network have to listen to the broadcast messages no matter how many of them arrive. Cisco, a prominent network solutions vendor, recommends to drop directed broadcast messages by default due to the security implications it brings [16].

2.1.3 Multicast

In multicast transmissions one sender transmits a packet to a group of receivers on the network as Figure 2.3 shows. This avoids sending a packet to each of the active hosts on a network but rather to a selected group. This helps hosts spend less resources since they only need to check if the multicast IP is configured to be recognized before continuing. Thus, the packet is not analyzed and it can be ignored [16].

The multicast addresses represent the group that devices belong to, if a multicast address is previously configured. Those addresses are in the range 224.0.0.0 through 239.255.255.255. The addresses between 224.0.0.0 and 224.0.0.255 are reserved by already defined protocols [18].

Discovery protocols such as multicast Domain Name System (mDNS) and Simple Service Discovery Protocol (SSDP) have assigned their multicast IPs (group) by the Internet Assigned Numbers Authority (IANA), 224.0.0.251 [19] and 239.55.55.250 [20] respectively. Thus, hosts that want to consider packets sent to those groups, need to listen and check such IPs addresses. Multicast uses the UDP transport protocol to transmit packets, as broadcast also does.

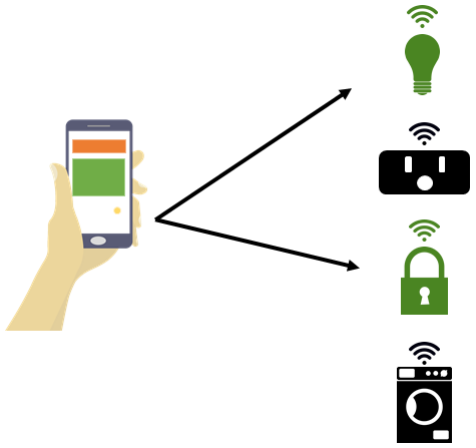

Figure 2.3 Multicast communication method

2.2 Discovery Protocols

Network discovery protocols are used to search for and/or notify other devices about services on the network (e.g., printing, SSH, turn on/off light bulb). If a device is interested in a particular searched/notified service, it can proceed to the configuration/control stage.

When a device enters a network, it might advertise the services it provides to network neighboring devices. Devices connected to the network will receive a notification about the new services available and who provide them. For instance, when a printer is set on a network, it notifies its printing service. Neighboring devices will be aware of the service, and they might proceed to configure it in order to provide printing to end-users. This is achieved without any human intervention to set the printer up in each device interested in the printing service.

Following the same example, if a device wants to know if the printing service is available in the network, it sends a request to search for the service. The printer providing the service will answer to the request. In this case the device interested in printing has to take further steps to provide printing to end-users. In essence, any device connected to the network can use discovery protocols to either notify or search for any type of service.

The process explained above allows avoiding the need of having a network administrator to configure services device-by-device on the network. Thus, the use of discovery protocols grants ease of device configuration on the network since no interaction from the network administrator is needed.

Discovery protocols handle specific networking information that allow them to provide the automatic location of network services, notably IP addresses, port numbers, transport protocol and method of communication. There are also informative aspects, such as information

about how to use the service is shared.

Through source code examination of a popular open source home automation hub (Home Assistant [21, 22]) and an empirical analysis (see Chapter 4) we determined that the main discovery protocols used by IoT devices are mDNS, SSDP that functions as the discovery stage of the Universal Plug and Play (UPnP) protocol [23,24], and a small number of vendor-specific protocols. We provide more details of these protocols in the next sections.

2.2.1 Multicast Domain Name System (mDNS)

mDNS follows the functionality of a conventional human-readable name to network address mapping service (e.g., DNS). The difference with a conventional DNS is that requests to resolve hostnames are sent through a multicast link, whereas the conventional DNS uses unicast.

mDNS can work as a service discovery since *Domain Name System-Service Discovery* (DNS-SD [25]) provides it with that functionality. The location of network services allows devices to know the services other devices provide, such as printing, SSH or turn light on/off to mention some. The protocol was created to provide ease of configuration on IP networks [25, 26].

To discover such services, first, it is necessary to resolve the devices' local hostnames to IPs. Thus, devices can reach devices providing the service of interest. Secondly, information about the service itself, like the transport protocol used, port and additional information to know how to operate the service, must be provided.

To resolve hostnames, locate network services and provide information regarding the service, mDNS uses records that in a conventional DNS are only used to resolve names. To begin with, local hostnames have the characteristic to finish with a dot followed by the word *local* [27]. For practical purposes, most users use their first names, or conventional names, to identify their computers. For example, a Mac configured by someone called *John* will likely have *Mac-John.local* as hostname.

Devices are able to either publish or to discover services through mDNS. Devices can announce that they provide the SSH service, or they can search for devices providing the SSH service for instance. If a device wants to publish its services or wants to discover a service, mDNS mandates to send the requests to the IPv4 multicast address 224.0.0.251 through the UDP port 5353 [25,27]. Either to publish or discover services, mDNS uses the pointer record (PTR), service record (SRV), A/AAAA record and TXT record [25, 27].

Figure 2.4 shows an example of discovering a *HomeKit Accessory Protocol (HAP)* [28] service offered by a smart bulb. When a device wants to discover a service, it needs to query a PTR

Table 2.1 Records used during a mDNS discovery

Type of record	Description	Example query
Pointer Record (PTR)	Asks for a Service in a Domain	_hap._tcp.local
Pointer Record (PTR)	Gives the hostname providing the service asked	LIFX-Mini-D-3CF90D._hap._tcp.local
Service Record (SRV)	Specifies the hostname and port for the service asked	LIFX-Mini-D-3CF90D:80
A/AAAA	Address indicates the IP of the hostname providing the service asked	192.168.1.40
TXT	Additional text notes about the service asked	mac=51:7E:6B:B2:BC:48

record requesting the desired service (step 1 in Figure 2.4). Then, the device that provides the asked service will answer with the PTR response record with the *service instance name* (step 2 in Figure 2.4). The *service instance name* is a representation of the service provided by a device (instance) in the local network. It has the structure *Instance.Service.Domain*, where *Domain* is always *local*, *Service* indicates the service provided and *Instance* is the hostname of the device providing that service.

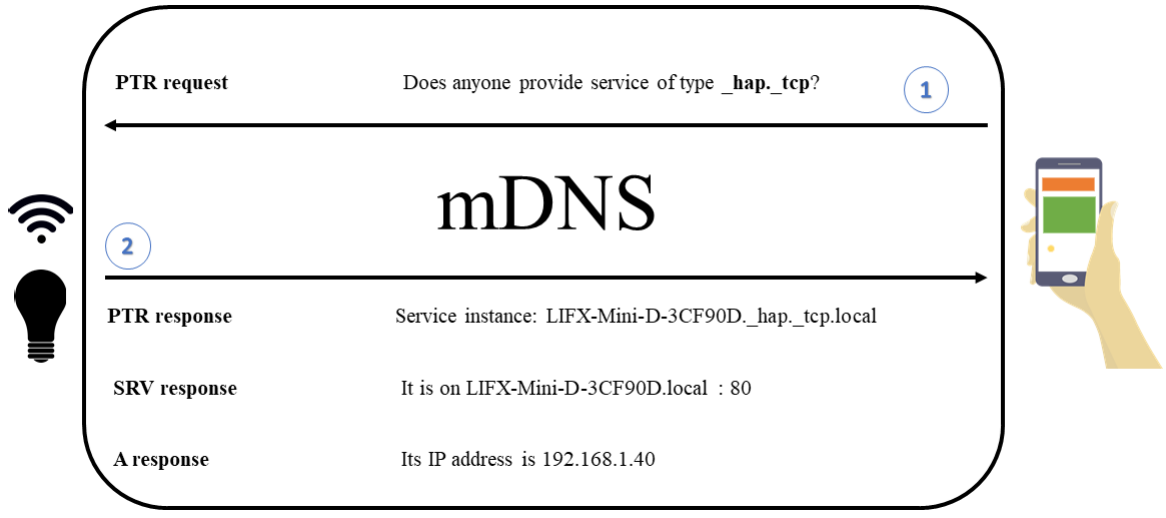

Figure 2.4 Discovering a HAP service with mDNS

Next, the device should add the SRV record in the answer, which gives the hostname and the port where the desired service is provided. Finally, the A/AAAA records should be added in the response, to map hostnames to their IPv4 and IPv6 addresses respectively.

The TXT record is optional and gives additional information about the service instance. It can provide additional useful information for a proper configuration of the service. It is possible to have multiple TXT records to describe a single service instance, and even the TXT record can be left empty.

On the other hand, when a device wants to publish a service, it has to send from two to eight unsolicited responses according to Request for Comments (RFC) of the mDNS [27]. That is, it has to send the PTR response record including the SRV and A/AAAA records all together from two to eight times.

2.2.2 Universal Plug and Play (UPnP) protocol

UPnP defines an architecture for pervasive peer-to-peer network connectivity of devices such as smart appliances. UPnP is designed to provide ease of configuration of devices in the network. A UPnP capable device can dynamically join to the network, announce the services it provides and, at the same time, find other UPnP capable devices and their services [10, 23].

Even though UPnP is an architecture and not only a discovery protocol, the discovery is part of the mandatory stages of UPnP, which are:

- **Addressing**, the stage for obtaining an IP address.
- **Discovery**, the stage is for finding or notifying services, that is, to know the services devices currently in the network can provide or to advertise the services the device itself provides.
- **In description**, the stage to send more information about the device's capabilities is sent.
- **Control**, the stage that allows to interact with the services.
- **Eventing**, the stage for updating the state of a device because of the control stage.
- **Presentation**, the stage where the device's URL can be loaded into a browser allowing the user to control and/or view the device status.

UPnP relies on SSDP for the discovery stage. Devices can either publish or discover services through UPnP. If a device wants to publish its services or wants to discover a service, UPnP discovery stage mandates that the devices has to send a request to the IPv4 multicast address 239.255.255.250 through the UDP port 1900 [23].

UPnP uses labels like NOTIFY, M-SEARCH, ST and LOCATION to mention some [23]. Some labels can have a value. For example, ST means Service Target and its value is normally the service searched. LOCATION indicates the URL where a XML file can be fetched to get information on the service, such as device name, device system information, device management status to mention some.

A UPnP capable device might advertise its services to network neighboring devices sending a NOTIFY message. The message contains the NOTIFY label as well as the label Notify Target (NT) with a value that indicates the service available. Other field added to the message is the LOCATION label that indicates the URL of the XML file with the service information.

The Figure 2.5 shows an example of discovering a *urn:Belkin:service:basicevent:1* service offered by a smart outlet. If a device wants to discover a service, it has to send a M-SEARCH message. The message includes the label ST to indicate the type of service the device wants to discover. Devices providing the searched service will answer in unicast to the device asking for the service. The answer contains the ST label to identify the service found and the URL where the XML file is located. The device who searched for the service needs to fetch the file to get information about the service offered and the device offering it for further steps like management or control.

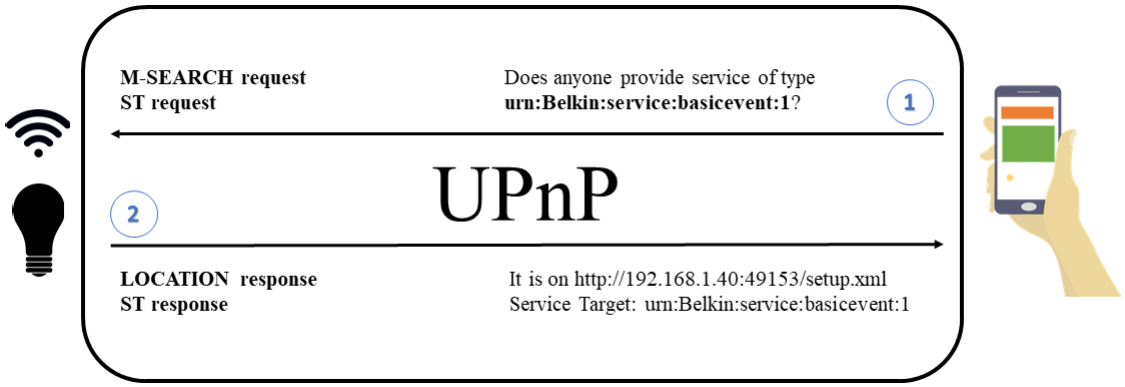

Figure 2.5 Discovering a Belkin service with UPnP

2.2.3 Proprietary protocols

We can consider mDNS and UPnP as standardized protocols since they have well-known structured definitions and are considered by the Internet Assigned Numbers Authority(IANA) [18]. However, there are vendors that decide to develop their own discovery protocols.

Vendor-specific discovery protocols (i.e., proprietary protocols) are usually sent via broadcast addresses. That is, the messages are received by all devices on the broadcast domain. Indeed, only the devices capable of understanding the payload will answer to the sender since payloads are defined by vendors. The IP used is either a local broadcast address (255.255.255.255) or a directed broadcast address (e.g., 192.168.1.255), and the destination UDP port is selected by the vendor. The destination UDP port is in the unprivileged port range (1025-65535) and can be linked to a specific vendors since vendors typically use the same port across their product range.

Devices send a payload that serves as the discovery request to the broadcast IP. If a device understands the payload, it means that the device is able to answer it. The answer is sent in unicast to the requester, and the information about the service is shared in the UDP payload. The discussion about these protocols will be covered later in chapter 4.

Figure 2.6 shows an example of discovering a Lifx bulb. The discovery request is sent towards the port designed by the Lifx vendor. Further, the UDP payload is also designed by the Lifx vendor. Indeed, Lifx devices will understand this proprietary discovery protocol. Once the Lifx bulb receives the discovery request, it will answer it. The answer uses the port designed by the Lifx vendor to send discovery request as the source port. The Lifx bulb's discovery answer (payload) is also vendor-specific (proprietary).

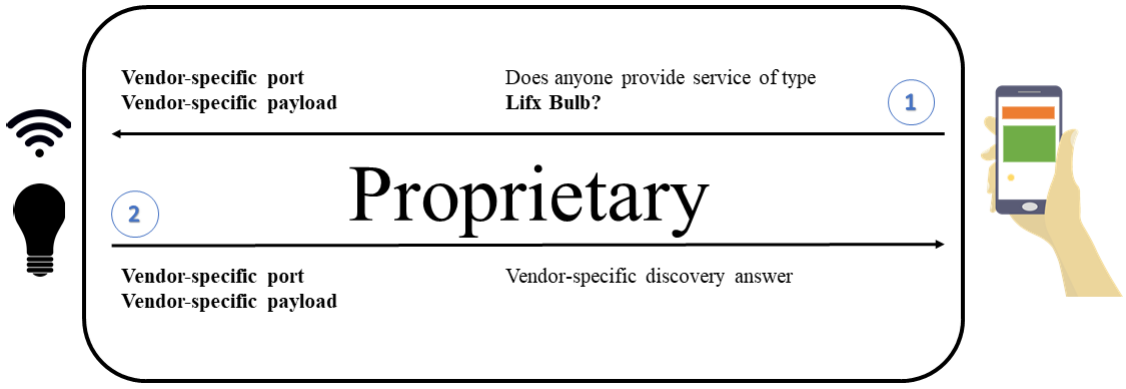

Figure 2.6 Discovering a service - proprietary protocol

2.3 A terminology dissection in the discovery protocols

When studying discovery protocols, we noticed a discrepancy in terminology definitions and use, which lead to confusion and misuse of terms. This section seeks to clarify some of this misunderstanding by creating a taxonomy of terms and applications used to broadly define

"zero-configuration networking". The taxonomy is presented in Figure 2.7 and explained below.

One of the first terms encountered while reading about discovery protocols is *zero configuration networking*. In general, *zero configuration networking* is the idea of having network devices easy to set up. That is, giving the sense of plugging devices in, turn them on and you will have them working without any further configuration. To achieve its goal, *Zero configuration networking* relies on three technologies: automatic IP address selection, automatic hostname resolution, and automatic location of network services [26, 29].

Discovery protocols play a significant role in *zero configuration networking* since they mainly help to automatically locate network services. The most popular discovery protocols are mDNS, SSDP, NetBIOS and WS-Discovery. In this thesis the discovery protocols of interest are mDNS, SSDP and vendor-specific since they were found to be used in our testbed and dataset (details in chapter 4). None of the devices in our data used NetBIOS, WS-Discovery or others.

mDNS follows the idea to turn conventional human-readable names into network addresses. mDNS operates in local networks and when no conventional Domain Name System (DNS) is installed on the network for name resolution [30]. mDNS cannot provide service discovery by itself so, Domain Name System-Service Discovery (DNS-SD) [25] extends mDNS to provide that.

SSDP operates in the discovery stage of UPnP stack [23]. UPnP has six stages detailed in section 2.2), however for practical terms we refer to UPnP as a discovery protocol.

Vendor-specific discovery protocols are tied to vendor decisions. However, in general they follow the idea of automatic location of network services.

There are software tools that serve to use discovery protocols. For instance, Avahi [31] and Bonjour [32] are the common well-known tools used for mDNS. Avahi is the version for Linux and BSD systems, and Bonjour is mainly the version for Mac OS and Windows systems. UPnP and vendor-specific protocols do not have common well-known tools. Vendors are responsible developing the elements to fulfill the required aspects to have a discovery protocol. However, UPnP client libraries can be found for a wide range of systems (e.g., UPnPy [33]), while the development of proprietary protocols tends to be closed source.

Figure 2.7 shows a taxonomy from upper to lower level of concepts, technologies, protocols or tools that have a relation with discovery protocols. Items shaded in blue represent the elements involved in discovery protocols in IoT IP networks. White items represent additional elements that have a relation with discovery protocols but there is no relation with IoT IP

networks. The blue squares were defined as per the results in chapter 4.

At the top level of the taxonomy the *zero configuration networking* is placed. The second level represents the three necessary technologies to provide *zero configuration networking*. Discovery protocols lied on the technology automatic location of network services. In the third level we can find the actual protocols stood for discovery protocols. Last level gives the most well-known tools that allow to leverage the benefits of discovery protocols.

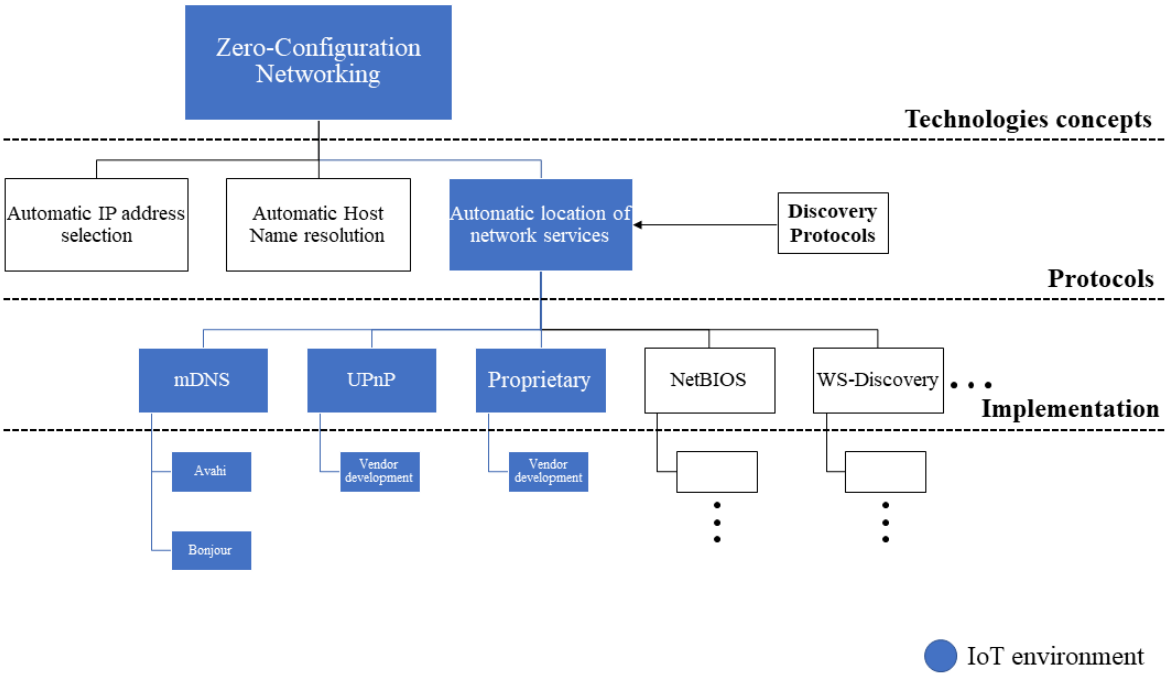

Figure 2.7 Discovery protocols taxonomy

2.4 Security concerns in IoT devices

IoT has seen incredible uptake in recent years, hence they are a modern-day method used by cybercriminals to perform cyber attacks. In 2016, a cyber attack relying on IoT devices exposed the impact of using IoT devices to produce a distributed denial of service attack since the amount of network traffic produced was not seen before [34].

IoT devices are deployed in a wide variety of environments, and homes are not the exception. Kumar *et al.* [35] showed that more than 70% of homes in North America have an IoT device. Even though this differs among regions and type of IoT devices, the easy way of management of IoT devices is fundamental for the adoption tendency in different regions and the variety deployment of IoT devices in homes.

Consumer IoT devices, the ones commonly found in homes, also bring security concerns. Williams *et al.* [36], Liu *et al.* [37] and Ling *et al.* [38] showed that consumer IoT devices bring security concerns in homes. Moreover, Chu *et al.* [39] and Shasha *et al.* [40] demonstrated that even toys are a concern.

The deployment of IoT devices in homes includes companion apps and sometimes IoT hubs rather than only IoT devices. All of them have an impact in the security of the home as Alrawi *et al.* [41] and Wang *et al.* [42] presented. End-users use the mobile companion apps and IoT hubs for managing their IoT devices.

IoT devices are managed remotely (out the home network) or locally (in the home network). The remote management commonly depends on cloud services provided by the vendors themselves. Commands are sent from companion apps to the cloud service, then from the cloud service to the IoT device. Local management depends on discovery protocols where, once the IoT device is discovered, the commands are sent directly to the IoT device.

IoT devices and their companion apps rely on discovery protocols to provide ease of management locally. That is, giving the sense of opening the companion app and interacting with the IoT devices right away. Discovery protocols have been thoroughly analyzed in the context of the Internet of computers, whereas IoT devices have received less attention in this area.

2.5 Security concerns in discovery protocols

The use of mDNS and UPnP in devices allows an exchange of information and they might transmit private information without the user's consent. These protocols can be used to perform a passive reconnaissance, and the network traffic generated would be considered benign. It differs from commonly used reconnaissance applications, such as Nmap, which performs an active reconnaissance through a direct interaction with the device.

Aura *et al.* [4] showed that devices have a false sense of anonymity when they are connected to a network, because they share information *accidentally* to perform many tasks automatically without the user interaction. This shared information can be used by a passive observer to perform reconnaissance, and discovery protocols contribute to this fact.

Konings *et al.* [5] showed how mDNS shares information that can be use to identify the owner's name. Moreover, information to identify device's brand and type is also shared. They did an analysis where the 58% of their dataset showed the type of device, allowing to build a list of targets non-invasively. Furthermore, the 25.5% of their dataset gave the owner's first name and the 14.2% gave the owner's first and last name. This demonstrates

that mDNS can be used for passive reconnaissance, approaching the information gathering stage of a penetration testing which leads to exploitation [43].

Also, Konings *et al.* [5] warned about the risk of sharing this type of information in a place with free WiFi, because it might be easy to identify the name of someone. If there is only one person using a specific hardware such as an iPad in that place, the information that mDNS transmits allows to identify the person, which can lead to stalking or scams.

Kaiser *et al.* [6] mentioned that the ability of mDNS to discover services allows anyone to know about the service itself without restriction. Hostname, service name and service type is the information shared during the process of discovering services. Since the service type is a synonym of the port number, enough information is shared to draw up an attack without performing an invasive and detectable port scan. The authors also demonstrated that mDNS offers a hostname resolution (to get the IP) even when the device is not offering a service in particular. Taking the example of only one person (the target) connected to a network with a particular device like a Mac, the IP address of the person's device can be obtained from the device's hostname (as per this example it might be Mac-Name.local) and a directed attack can be drawn up.

UPnP shares the same characteristic as mDNS in providing information which might be considered risky. In the discovery stage, a URL is shared under the LOCATION label. It can be used to fetch the setup file to gather information about the device, like firmware version, hardware, model and device name.

Further, Bai *et al.* [9] showed a spoofing attack. They demonstrated that when a device searches for a service, the attackers can reply the same answer as the legitimate device but using the SRV or A records with the attacker's device hostname. Atlasis [7] as well as in [8] showed how mDNS leads to spoofing attacks, since there is no mechanism to identify the device that allegedly provides the service.

Bai *et al.* [9] provided a good demonstration on how mDNS can be used to perform a man-in-the-middle (MitM) attack by spoofing a service. They showed how a malicious device can impersonate a printer. When the malicious device selects the same mDNS hostname as the real printer, it generates a conflict, and the real printer resolves that conflict by changing its mDNS hostname by appending the string *"(2)"* to its original hostname. Thus, *HP LaserJet 3600* becomes *HP LaserJet 3600(2)*, and the devices connected to network do not know about this change. Therefore, when devices send a mDNS query to search for the printer service, the malicious device will answer to it. As a result, the malicious device will receive the documents to print, and then forward the documents to the real printer, performing the MitM attack.

UPnP has been also analyzed, Haque *et al.* [11] showed that it is possible to execute a buffer overflow. The vulnerability relies on the size of the device description sent by the NOTIFY header message, since UPnP does not do any boundary checking.

Wang *et al.* [42] and Al Hasib *et al.* [10] mentioned that UPnP might be used to perform Distributed Denial of Service (DDoS) attacks. A device receiving a request from an alleged control point will answer to the supposed requester which might be spoofed.

Discovery protocols lack of security controls, and many authors have warned that this leads to use them for attacks. Besides of the security concerns discussed in this section, this might also have a physical impact since IoT devices rely on discovery protocols.

2.6 Securing discovery protocols

Some solutions have been proposed to resolve the security concerns involved in discovery protocols in the context of the Internet of computers. For instance, Kaiser *et al.* [12] proposed an architecture for pairing devices to allow hiding instances name and service type and encrypt information shared with mDNS. The proposed solution implies modifying Avahi, the tool that allows to use mDNS in Linux systems.

A similar solution is proposed by Wu *et al.* [14] using cryptography. However, they proposed a totally new protocol to cover private service discovery and private mutual authentication, which involves a certified authority scheme and the Diffie Helman algorithm.

Pang *et al.* [13] also proposed an architecture that enhances the confidentiality of discovery. The solution proposed also implies the implementation of cryptography primitives for paring and authentication. They argue the solution works regardless of the discovery protocol.

Due to the processing limitations of IoT devices, it will be difficult to install any solution that implies high cryptographic processing. In addition, vendors would need to deploy the solution in their devices from manufacturing. In case the processing would not be a problem, it would be necessary to update the software/firmware of the billions of devices already on the market, which is not a feasible option. Further, obsolete IoT devices already deployed would not receive the update. Last but not least, even though there are standard discovery protocols, IoT devices use also vendor-specific discovery protocols as mentioned in the section 2.2. This fact makes it hard to analyze diverse protocols to identify their security weaknesses and then improve each protocol.

Conventional security mechanisms to limit/control the network traffic, such as firewalls, are not able to limit the use of discovery protocols on local networks since discovery queries are transmitted locally and they do not cross the firewall. Firewalls are also inadequate to see

the cross-device and companion app-device interactions that are commonly found in local IoT communications.

Software Defined Networking (SDN) provides a facility to see and control local network traffic. SDN separates the control plane from the data plane, enabling policy-based management of network resources. SDN has been used in network security research proposals to implement access control [44, 45]. SDN has been used in IoT security research too.

For instance, Vandana *et al.* [46] shared an idea to use SDN in an IoT environment to have different segments, like domains, where each segment has a controller with segment's policies. The controllers can have a communication among them to share policies. Thus, the controllers behave as security guards in the segments, controlling communication among IoT devices and among segments, authorizing or denying communications.

Flauzac *et al.* [47] proposed a similar architecture but in Ad-Hoc networks, by splitting devices into domains. Each domain has a controller and policies, but in this case each node in the domain has the data plane integrated. However, IoT devices do not support the data plane due to hardware limitations. Hence, they propose to have a neighbour device, as part of the IoT device, which can support the data plane.

Yu *et al.* [48] proposed a solution to protect IoT devices using a middlebox for each IoT device, where a controller handled by SDN can enforce policies in the middlebox, protecting the IoT devices separately or individually. However, that will imply users to have a pair for each IoT device they bought, which will have a financial impact causing the user to prefer not to protect their devices. The advice presented herein can be partially implemented at little-to no cost to the administrator.

Goutman *et al.* [49] proposed a solution based on SDN to enforce the principle of least privilege on each IoT device. They classify IoT devices into two types of devices: controllers and non-controllers. They mentioned that non-controllers, such as sensors or actuators, need only communication through a controller, such as a hub, or user devices like a laptop. However, IoT devices are not always controlled through a hub, as they can be controlled by the user's smartphone. The solution enforces a policy scheme over the IoT devices, and Goutman *et al.* [49] considered the discovery protocols as part of the policy scheme. The policy scheme only considered the no-controllers to be restricted in the discovery. However, a controller can abuse the discovery to know more about the user network in a marketing way for instance.

We view SDN as a plausible technology to enable the deployment of discovery protocol filtering on IoT networks, as SDN provides the functionality to monitor and control local

network traffic. Moreover, SDN has been already used in IoT environments as a security control providing good results.

2.7 Summary

This chapter presented a technical explanation of network communication methods (unicast, broadcast and mutlicast) that operate in the discovery protocols. Then we gave a technical explanation about the discovery protocols mDNS, UPnP and proprietary (vendor-specific) going from networking aspects to discovery messages content.

Moreover, we presented a taxonomy of discovery protocols. The discussion covers the zero networking concept and the technologies it depends on. Discovery protocols as a dependent technology of zero networking as well as the protocols involved and the actual common tools that grant the use of discovery protocols in different platforms.

Next, we discussed the actual security concerns involved in using those protocols in the Internet of computers. Finally, we presented proposed solutions to approach the security concerns involved in discovery protocols.

The presented proposed solutions add security, such as authentication and authorization, to discovery protocols, however such solutions are mainly focused in modify existent discovery protocols or create new ones, but in IoT environment it is not feasible to be implemented. As well was presented SDN as a solution in an IoT environment where traditional security tools such as firewalls are not suitable. Despite that, the proposed solutions do not cover the discovery problem at all or in its whole.

CHAPTER 3 EMPIRICAL ANALYSIS OF DISCOVERY PROTOCOLS IN IOT DEVICES

This chapter describes a set of empirical experiments performed to understand the prevalence, usage patterns, and security considerations of discovery protocols in IoT.

We address the analysis of discovery protocols in IoT devices starting by reviewing the code of a well-know open-source home automation hub and then inspecting the network traffic generated by the hub. Moreover, we analyzed two IoT environments: (1) we deployed a small local IoT network with multiple device types from multiple vendors, and (2) IoT traffic from the public dataset of Alrawi *et al.* [41]. In our small IoT network, we controlled the interaction among devices and companion apps. This allowed us to examine in detail the network traffic and interaction patterns of discovery protocols. On the other hand, the public dataset provided a larger sample of IoT devices than our small local devices network. The dataset permitted to observe what discovery protocols are more used by vendors and device types, as well as the interaction among vendors and devices at a larger scale. However, the public dataset does not contain information about companion apps or interaction between devices, if any.

3.1 Home automation hub

Home Assistant [50] (HA) is an open source home automation hub able to integrate IoT devices in a central point, either to monitor, control or both. HA can integrate more than 1500 devices and services (weather, currency price, calendar, etc.) combined together. This allowed us to have a first approach to understand the use and prevalence of discovery protocols in IoT devices.

First, we performed a code review since HA is open source and we could download its source code from GitHub [51]. We performed a code review to identify the module that HA uses to discover devices. Moreover, this allowed us to understand how a home automation hub can manage several IoT devices. Nevertheless, we could identify the trend of discovery protocols used by IoT devices.

Besides the code review, we examined in detail network traffic and interaction patterns when using discovery protocols by HA. HA provides many options to be installed in a system. We used the Virtual Machine (VM) version of HA and we set it up in VirtualBox in a computer running Windows 10. Furthermore, we assigned a dedicated physical WiFi interface to HA

and we set the interface up as Bridged Interface in VirtualBox. This configuration permitted HA to have its own IP to be able to be connected to any network independently of the computer host.

We set an Access Point (AP) up on a Kali Linux running Wireshark [52] for network traffic collection. Next, we connected HA to the AP since it allowed us to capture all the traffic generated by HA without restriction. This allowed us to have a perspective of HA discovery behaviour when connected to the network.

Finally, we defined two topologies to examine HA behaviour as well as the interaction with IoT devices. The details are provided in next sections.

3.1.1 Topology 1 - Home Assistant individually connected to the network

The first topology was composed of having HA connected to the AP in order to capture network traffic as soon it is connected to the AP. This allowed us to gather only the network traffic generated by HA.

We recorded the network traffic, and then we performed an empirical analysis of HA's behaviour. In the recorded data we searched for the presence of discovery protocols sent by HA. In order to gather discovery queries sent, if any, and patters when sending them. This allowed us to have a perspective on the HA's behaviour once connected to the network.

3.1.2 Topology 2 - Home Assistant interacting with IoT devices

The second topology was composed of having HA connected to the AP simultaneously with a Wemo outlet, a Lifx bulb and a Tp-Link. We could observe the interaction between HA as a home automation hub and the IoT devices. This setup helped us to see how HA discovers IoT devices and any interaction between them using discovery protocols.

We recorded the network traffic while HA is booting and when we manually integrated the IoT device into the HA's GUI. Then we performed an empirical analysis of the interaction between HA and the IoT devices. In the recorded data we searched for the discovery protocol and the discovery query used by HA to discover the devices.

3.2 Testbed

We built a small testbed to connect IoT devices and their companion mobile apps to perform empirical experiments. We built a testbed since it provided us the ease to control IoT devices and companion apps behaviour in terms of end-user interaction. We could examine in detail

network traffic and interaction patterns when using discovery protocols. We built a testbed composed of retail IoT devices, an Android smartphone and an Access Point (AP) as follows.

We used seven IoT devices: three smart bulbs, two smart outlets and two IP cameras from well-known IoT vendors [53]. We defined them as specific purpose devices since they do not provide any add-on app installation, and they perform a specific task like turning on/off a light. Furthermore, they receive commands from end-user through their companion apps rather than sending commands to any endpoint.

We used an Android smartphone to install the companion app recommended by the IoT vendor of each device. Companion apps give end-users a way to interact with their IoT devices. Table 3.1 provides the information about the IoT devices and their companion apps used. Such information specifies IoT vendor, type of device, model and the companion app used for each IoT device.

Finally, the AP was set up on a Kali Linux computer running Wireshark [52] for network traffic collection. The AP allowed to capture all the traffic generated by any device connected to it without restriction.

Once the testbed was built, we defined seven different topologies to examine IoT devices and companion apps behaviour as well as the interaction among all of them. The details are provided in next sections.

Table 3.1 IoT devices and their respective companion apps used in the testbed

Brand	Type	Model	Companion app (Android)
Lifx	Bulb	Mini Day & Dusk	Lifx
Wemo	Outlet	Mini	Wemo
TP-Link	Bulb	LB120	Kasa
TP-Link	Outlet	HS110	Kasa
TP-Link	Camera	NC250	tpCamera
Yeelight	Bulb	Unknown	Mi Home
D-Link	Camera	DCS-2132L-ES	mydlink

3.2.1 Topology 1 - IoT devices individually connected to the network

The first topology was composed of each IoT device individually connected to the AP as Figure 3.1 shows. That is, we connected each device one at a time in order to capture network traffic as soon as the device was connected to the AP. This avoided any network traffic noise, and allowed us to get only the interested data, i.e., the IoT device network traffic. Moreover, this allowed us to have a perspective of the device's behaviour once it

connected to the network.

Figure 3.1 Connecting an IoT device to the network

We recorded the network traffic for thirty minutes, and then we performed an empirical analysis of the IoT device's behaviour. In the recorded data we searched for the presence of discovery protocols sent by each IoT device individually, in order to gather the discovery queries sent, if any, and patters when sending them. As we used specific purpose IoT devices, we ideally expected not to observe any discovery query sent by any IoT device.

3.2.2 Topology 2 - Only companion apps connected to the network

The second topology was composed of the smartphone connected to the AP as Figure 3.2 shows. The smartphone had installed all the companion apps of the IoT devices (see table 3.1 for reference). To avoid network noise generated by others apps installed in the smartphone, both from system apps or third-party apps, we used a mobile firewall app. There are plenty of mobile firewalls, but we particularly used *Mobiwol* as it does not need root privileges to work. In this way, we controlled which companion app was allowed to send network traffic in a specific moment. Thus we could get only the interesting data for each companion app. This allowed us to have a perspective of the companion app's behaviour once it connected to the network.

Figure 3.2 Connecting a companion app installed in a smartphone to the network

We recorded the network traffic for thirty minutes, and then we performed an empirical

analysis of the companion apps' behaviour. In the recorded data we searched for the presence of discovery protocols sent by each companion app individually to gather discovery queries sent, if any, and patters when sending them. We expected to see the discovery protocols and discovery queries sent by companion apps to find their respective IoT devices.

3.2.3 Topology 3 - All devices simultaneously connected to the network

The third topology was composed of all the IoT devices connected to the AP at the same time as Figure 3.3 shows. This allowed us to observe the behaviour of different types of IoT devices from different vendors connected together to the same network.

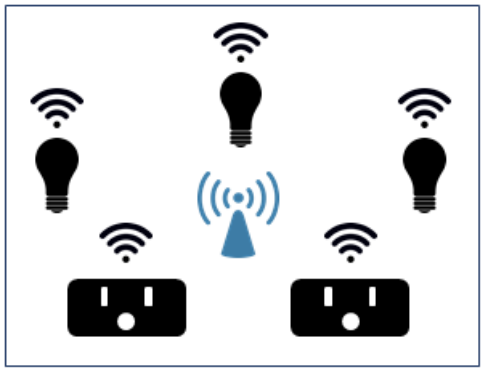

Figure 3.3 Connecting all the IoT devices to the network

We recorded the network traffic for thirty minutes, and then we performed an empirical analysis of the behaviour of the IoT devices connected all together to the same network. In the recorded data we searched for the presence of discovery protocols sent by the IoT devices when having neighboring IoT devices. We expected not to see discovery protocols and discovery queries sent by IoT devices since they were specific purpose devices.

3.2.4 Topology 4 - IoT devices interacting with their companion apps

The fourth topology was composed of each IoT device and the smartphone with the respective device's companion app connected to the AP as Figure 3.4 shows. To avoid network noise generated by others apps installed in the smartphone, both from system apps or third-party apps, we used the same mobile firewall app we used in topology 2. Thus, we could observe the interaction between IoT devices and their companion app, pairwise. This setup helped us determine how a companion app discovers its IoT devices and any interaction between them using discovery protocols.

Figure 3.4 Connecting an IoT device and its companion app to the network

We recorded the network traffic while the IoT device was connected to the network, and then we opened the companion app. Then we performed an empirical analysis of the interaction between the device and its companion app. In the recorded data we searched for the discovery protocol and the discovery query used by the companion app to discover its device.

3.2.5 Topology 5 - All devices and companion apps connected to the network simultaneously

The fifth topology was composed of all the IoT devices and the smartphone with all the companion apps connected to the AP simultaneously as Figure 3.5 shows. To avoid network noise generated from system apps installed in the smartphone, we used the same mobile firewall app we ussed in topologies 2 and 4. This setup allowed us to observe the behaviour of IoT devices and companion apps when different vendors are connected together to the same network.

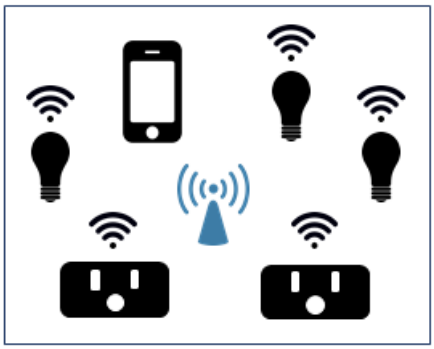

Figure 3.5 Connecting different IoT device and companion apps to the network

We recorded the network traffic while IoT devices were connected to the network, while we were opening and closing the companion apps, and while we put them in foreground and background. Then, we performed an empirical analysis of the interaction of IoT devices and

companion apps from different vendors. In the recorded data we searched for discoveries made by companion apps towards IoT devices that are from a different vendor. We expected not to see any cross-vendor interaction and only see discoveries towards proper IoT devices.

3.2.6 Topology 6 - Companion apps logged in with different user accounts

The sixth topology was similar to topology 4, that is, it was composed of each IoT device and the smartphone with the respective device's companion app connected to the AP. However, in the previous topologies we used companion apps with the devices owner's account, since this account set the IoT devices up on the network. In this topology we used companion apps set up with a new user account to simulate guest users joining the network. To avoid network noise generated by others apps installed in the smartphone, both from system apps or third-party apps, we used the same mobile firewall app we used in topology 2.

Thus, we could observe the interaction between the IoT device and the companion app with the new user account, pairwise. This permitted us to observe the behaviour of companion apps when discovering an IoT device already configured with another account.

We recorded the network traffic while each IoT device was connected to the network and then we opened the companion app with the new user account. Then we performed an empirical analysis of the interaction between each IoT device and its companion app with the new account. In the recorded data we searched for answers from the IoT device to the companion app. Then we observed what the companion apps showed to end-user.

3.2.7 Topology 7 - IoT devices interacting with our selected discovery queries

The seventh topology was similar to topology 1, that is, it was composed of each of the IoT devices individually connected to the AP. In this case the AP could send discovery queries selected by us. Since the AP was running on the top of an OS, we could install Avahi and Python to send crafted mDNS and UPnP discovery queries through the AP interface.

We generated a generic discovery query in both mDNS and UPnP [23, 25, 27] that we called *root* discovery (see Table 3.2). mDNS-compatible devices should answer to mDNS root discovery and UPnP-compatible devices should answer to UPnP root discovery. This implies that the sender does not need to know the specific discovery query to discover a specific device. This permitted us to observe the behaviour of IoT devices when receiving a root discovery regardless on how their companion app discovered them.

We recorded the network traffic while the IoT device is connected on the network and while sending the crafted discovery queries to such IoT device. We searched in the data recorded

if the IoT device answers to the root discovery query to analyze if the answer is expected as per the discovery made by companion apps.

Table 3.2 mDNS and UPnP *root* discovery queries

Discovery protocol	*root* query
mDNS	_services._dns-sd._udp.local
UPnP	ssdp:all upnp:rootdevice

3.3 Public Dataset

A larger sample of recorded data can provide a wide overview of the use of discovery protocols by IoT devices. IoT traffic from the public dataset of Alrawi *et al.* [41] provides a larger sample of IoT devices than our small local devices network.

Yourthings [41] is a dataset of network traffic recorded over a period of 13 days. The network traffic was recorded in the following dates (considering full days of recording): March 20, 21, 25, 2018 and April 10-19 of 2018. The dataset is composed of more than 100 GB of network traffic generated by 50 different devices as Table 3.3 shows. The set of devices contains general purpose devices like an Ubuntu Desktop, home automation Hubs like Google home and IoT devices like TP-Link Smart Bulb. Moreover, the set of devices is composed of different vendors.

Even though this dataset permits to observe how discovery protocols are used by more vendors and device types than our testbed, as well as the interaction among these at a larger scale, we did not have control over how companion apps and IoT devices were used or interacted with each other. Additionally, Alrawi *et al.* [41] do not mention if there is a record of such interactions among companion apps and IoT devices. Likewise, there is no information about if IoT devices were always connected during the whole network traffic record nor if companion apps (if any) were up during this time.

Due to the data size, we developed a Python script to sweep the whole dataset and fetch discoveries made by devices. In this way, we gathered relevant information about the discovery process, like discovery protocols and the queries sent. We analyzed the gathered information to provide insights.

Indeed, this dataset provided us a wide overview of the use of discovery protocols in a wider range of devices. That is, we could analyze IoT devices, home automations hubs and general devices behaviour when connected together to the same network. We could identify

Table 3.3 Devices noted in Yourthings dataset

Google OnHub	nVidia Shield	Google Home Mini
Samsung SmartThings Hub	Apple TV (4th Gen)	Google Home
Philips HUE Hub	Belkin WeMo Link	Bose SoundTouch 10
Insteon Hub	Netgear Arlo Camera	Harmon Kardon Invoke
Sonos	D-Link DCS-5009L Camera	Apple HomePod
Securifi Almond	Logitech Logi Circle	Roomba
Nest Camera	Canary	Samsung SmartTV
Belkin WeMo Motion Sensor	Piper NV	Koogeek Lightbulb
LIFX Virtual Bulb	Withings Home	TP-Link Smart Bulb
Belkin WeMo Switch	WeMo Crockpot	Wink 2 Hub
Amazon Echo	MiCasaVerde VeraLite	Nest Cam IQ
Wink Hub	Chinese Webcam	Nest Guard
Belkin Netcam	August Doorbell Cam	Ubuntu Desktop
Ring Doorbell	TP-Link WiFi Plug	Android Tablet
Roku TV	Chamberlain myQ Garage Opener	iPhone
Roku 4	Logitech Harmony Hub	iPad
Amazon Fire TV	Caseta Wireless Hub	

discovery behaviours already observed in our testbed. However, we could identify behaviours not presented in our testbed.

3.4 Summary

This chapter provided the technical details of our data collection methodology. We discussed the home automation hub we selected and the examination we conducted to understand how these devices can manage many IoT devices as a central point. Furthermore, We explained the components of the testbed and the topologies that we used to gather data about the IoT devices' discovery behaviour. Moreover, we explained the public dataset we used to broaden the diagnose of IoT devices' discovery behaviour.

CHAPTER 4 EXPERIMENTAL RESULTS AND DISCUSSION

This chapter presents the results of the empirical experiments we described in the previous chapter, which helped identify the threats impacting IoT devices due to discovery protocols.

4.1 Home automation hub

When using a home automation hub, it acts as the middle point to manage IoT devices, so it needs to handle the discovery and control of devices.

We choose the well-know open source home automation hub Home Assistant (HA), because it permitted to review its code and find out how its discovery module works. Also because it can integrate more than 1500 devices and services combined together which leads to have one point that can discover many devices. This allowed us to have a first approach to understand the use and prevalence of discovery protocols in IoT devices.

HA has a module called NetDisco [21] which seems to serve as a discovery module. We inspected its source code, and we observed in general three types of discovery protocols: mDNS, UPnP and vendor-specific (proprietary) ones.

In total we identified seven discovery protocols and five out of the seven identified were proprietary discovery protocols tied to specific brands. Table 4.1 shows the discovery protocols we identified in the NetDisco module. The protocols mDNS and UPnP are not tied to any specific vendor, whereas Good Day Mate (GDM) is tied to Plex Media Servers, Logitech Media Server (LMS) is tied to streaming audio servers of Logitech, Tellstick is tied to Tellstick devices, Daikin is tied to Daikin Air Conditioners devices, and Smart Glass is tied to Xbox.

Table 4.1 Discovery protocols supported by Home Assistant

Disc protocol	Proprietary	Standard
mDNS		✓
UPnP		✓
GDM	✓	
LMS	✓	
Tellstick	✓	
Daikin	✓	
Smart glass	✓	

It is worth mentioning that the HA developers informed that if a new device wanted to be supported by the HA, the new device needed to support mDNS or UPnP because others

discovery protocols were considered deprecated, and HA developers were not longer accepting new proprietary discovery protocols [21]. This first approach set an initial trend of discovery protocols used by IoT devices.

By reviewing the code, we could understand how a home automation hub can discover IoT devices from different vendors. The HA stored *service names* and *service targets* to discover mDNS and UPnP capable IoT devices respectively. That is, the HA needs to know the service name or the service target that an IoT device answers to in order to discover the device and then taking further steps to control it.

Figure 4.1 shows a sample code snippet that shows the UPnP hardcoded service target of Sonos devices. The UPnP service target *urn:schemas-upnp-org:device:ZonePlayer:1* allows the HA to discover Sonos devices.

```
1  """Discover Sonos devices."""
2  from . import SSDPDiscoverable
3
4
5  class Discoverable(SSDPDiscoverable):
6      """Add support for discovering Sonos devices."""
7
8      def get_entries(self):
9          """Get all the Sonos device uPnP entries."""
10         return self.find_by_st("urn:schemas-upnp-org:device:ZonePlayer:1")
```

Figure 4.1 Example of Sonos service target in Home Assistant

On the other hand Figure 4.2 shows a sample code snippet that shows the mDNS hardcoded service name of Yeelight bulbs. The mDNS service name *_miio._udp.local.* allows the HA to discover Yeelight bulbs.

We set up and launched the HA to record the network traffic it generated since booting. We saw that the HA sent nineteen different discovery queries, both mDNS and UPnP. Two out of the nineteen were mDNS discovery queries, as Table 4.2 shows. Figure 4.3 shows a fragment of the discovery queries sent by the HA.

We could identify that the HA sent what we call a *root* discovery. That is, mDNS and UPnP use a service name and service target respectively which server to discover all the devices in the network. Indeed, mDNS root discovery will only discover devices supporting mDNS, and UPnP root discovery will only discover devices supporting UPnP. The two out of nineteen UPnP discovery queries sent by the HA were root discoveries queries. Table 4.2 shows these queries in red.

```
"""Discover Yeelight bulbs, based on Kodi discoverable."""
from . import MDNSDiscoverable
from ..const import ATTR_DEVICE_TYPE

DEVICE_NAME_PREFIX = 'yeelink-light-'


class Discoverable(MDNSDiscoverable):
    """Add support for discovering Yeelight."""

    def __init__(self, nd):
        """Initialize the Yeelight discovery."""
        super(Discoverable, self).__init__(nd, '_miio._udp.local.')

    def info_from_entry(self, entry):
        """Return most important info from mDNS entries."""
        info = super().info_from_entry(entry)

        # Example name: yeelink-light-ceiling4_mibt72799069._miio._udp.local.
        info[ATTR_DEVICE_TYPE] = \
            entry.name.replace(DEVICE_NAME_PREFIX, '').split('_', 1)[0]

        return info

    def get_entries(self):
        """ Return yeelight devices. """
        return self.find_by_device_name(DEVICE_NAME_PREFIX)
```

Figure 4.2 Example of Yeelight service name in Home Assistant

After that, we set up the HA together with a Wemo outlet, a Lifx bulb and a Tp-Link outlet on the same network. From the booting discovery queries sent, the HA could discover the Wemo outlet and the Lifx bulb. The Wemo outlet was discovered by the UPnP root query, and the Lifx bulb was discovered by the mDNS *_hap._tcp.local* query.

Nevertheless, we could see in the HA's web Graphical User Interface (GUI), that a Wemo device and a Lifx device were discovered, and we were asked to integrate them in the HA for controlling. Once accepting, we saw the expected UPnP process to set up the Wemo outlet. However, for the Lifx bulb, we noticed that the HA sent a broadcast request to the port 56700, and then the Lifx bulb answered in unicast to the HA through UDP using 56700 as the source port. Even though the HA discovered the Lifx bulb with a mDNS query, it used a Lifx-specific protocol to discover and control the Lifx bulb. We will discuss this protocols in section 4.10.

Table 4.2 Discovery queries sent by the Home Assistant after booting

UPnP	mDNS
ST: ssdp:all*	_axis-video._tcp.local
ST: upnp:rootdevice*	_googlecast._tcp.local
	_ipp._tcp.local
	_api._udp.local
	_elg._tcp.local
	_miio._udp.local
	_printer._tcp.local
	_esphomelib._tcp.local
	_spotify-connect._tcp.local
	_wled._tcp.local
	_daap._tcp.local
	_nut._tcp.local
	_ipps._tcp.local
	_dkapi._tcp.local
	_viziocast._tcp.local
	_hap._tcp.local
	_services._dns-sd._udp.local*

* Root discovery queries

Finally, we could control the Wemo outlet and the Lifx bulb since the HA notified that it had discovered the devices. To effectively control the devices, we only needed to turn on the HA, wait for the notification and accept it. That was all.

The HA did not discover the Tp-Link outlet automatically, however we could discover it through the Integrations section of the GUI. This section allows to find and integrate devices. So, we selected the Tp-Link integration and we got the same notification we got for the Wemo and Lifx devices. We saw that the HA sent a broadcast request to the port 9999, and then the Tp-Link outlet answered in unicast to the HA through UDP using 9999 as the source port, similar to what happened with the Lifx bulb. We accepted the integration notification and we could control the Tp-Link outlet bulb.

We concluded that the HA sent mDNS and UPnP discovery queries since they provide better compatibility as standard discovery protocols against vendor-specific ones. Although, the HA had already vendor-specific protocols stored to be compatible with the most common brands. Due to the wide variety of brands producing IoT devices and the rapid insertion of new IoT devices in the market, it is a hard task to follow the new technologies deployed on all the IoT devices. This might be the reason why the HA developers recommended to use mDNS

No.	Time	Source	Destination	Protocol	Length	Info
213	18.747190225	10.42.0.48	224.0.0.251	MDNS	82	Standard query 0x0000 PTR _axis-video._tcp.local, "QM" question
214	18.747228681	10.42.0.48	224.0.0.251	MDNS	82	Standard query 0x0000 PTR _axis-video._tcp.local, "QM" question
215	18.748131176	10.42.0.48	224.0.0.251	MDNS	82	Standard query 0x0000 PTR _axis-video._tcp.local, "QM" question
216	18.748155237	10.42.0.48	224.0.0.251	MDNS	82	Standard query 0x0000 PTR _axis-video._tcp.local, "QM" question
217	18.748173790	10.42.0.48	224.0.0.251	MDNS	82	Standard query 0x0000 PTR _axis-video._tcp.local, "QM" question
218	18.748180841	10.42.0.48	224.0.0.251	MDNS	82	Standard query 0x0000 PTR _axis-video._tcp.local, "QM" question
219	18.753924931	10.42.0.48	224.0.0.251	MDNS	82	Standard query 0x0000 PTR _googlecast._tcp.local, "QM" question
220	18.753962916	10.42.0.48	224.0.0.251	MDNS	82	Standard query 0x0000 PTR _googlecast._tcp.local, "QM" question
221	18.753988230	10.42.0.48	224.0.0.251	MDNS	82	Standard query 0x0000 PTR _googlecast._tcp.local, "QM" question
222	18.753994422	10.42.0.48	224.0.0.251	MDNS	82	Standard query 0x0000 PTR _googlecast._tcp.local, "QM" question
223	18.754001717	10.42.0.48	224.0.0.251	MDNS	82	Standard query 0x0000 PTR _googlecast._tcp.local, "QM" question
224	18.754006284	10.42.0.48	224.0.0.251	MDNS	82	Standard query 0x0000 PTR _googlecast._tcp.local, "QM" question
225	18.756505667	10.42.0.48	224.0.0.251	MDNS	75	Standard query 0x0000 PTR _ipp._tcp.local, "QM" question
226	18.756543243	10.42.0.48	224.0.0.251	MDNS	75	Standard query 0x0000 PTR _ipp._tcp.local, "QM" question
227	18.756568201	10.42.0.48	224.0.0.251	MDNS	75	Standard query 0x0000 PTR _ipp._tcp.local, "QM" question
228	18.756575565	10.42.0.48	224.0.0.251	MDNS	75	Standard query 0x0000 PTR _ipp._tcp.local, "QM" question
229	18.756583248	10.42.0.48	224.0.0.251	MDNS	75	Standard query 0x0000 PTR _ipp._tcp.local, "QM" question
230	18.756588300	10.42.0.48	224.0.0.251	MDNS	75	Standard query 0x0000 PTR _ipp._tcp.local, "QM" question
231	18.759515042	10.42.0.48	224.0.0.251	MDNS	75	Standard query 0x0000 PTR _api._udp.local, "QM" question
232	18.759552217	10.42.0.48	224.0.0.251	MDNS	75	Standard query 0x0000 PTR _api._udp.local, "QM" question
233	18.762401438	10.42.0.48	224.0.0.251	MDNS	75	Standard query 0x0000 PTR _api._udp.local, "QM" question
234	18.762443224	10.42.0.48	224.0.0.251	MDNS	75	Standard query 0x0000 PTR _api._udp.local, "QM" question
235	18.762470556	10.42.0.48	224.0.0.251	MDNS	75	Standard query 0x0000 PTR _api._udp.local, "QM" question
236	18.762480520	10.42.0.48	224.0.0.251	MDNS	75	Standard query 0x0000 PTR _api._udp.local, "QM" question
237	18.762489139	10.42.0.48	224.0.0.251	MDNS	75	Standard query 0x0000 PTR _elg._tcp.local, "QM" question
238	18.762494751	10.42.0.48	224.0.0.251	MDNS	75	Standard query 0x0000 PTR _elg._tcp.local, "QM" question
239	18.774976480	10.42.0.48	224.0.0.251	MDNS	75	Standard query 0x0000 PTR _elg._tcp.local, "QM" question
240	18.775019140	10.42.0.48	224.0.0.251	MDNS	75	Standard query 0x0000 PTR _elg._tcp.local, "QM" question
241	18.775047021	10.42.0.48	224.0.0.251	MDNS	76	Standard query 0x0000 PTR _miio._udp.local, "QM" question
242	18.775054169	10.42.0.48	224.0.0.251	MDNS	76	Standard query 0x0000 PTR _miio._udp.local, "QM" question
243	18.775063078	10.42.0.48	224.0.0.251	MDNS	79	Standard query 0x0000 PTR _printer._tcp.local, "QM" question
244	18.775068121	10.42.0.48	224.0.0.251	MDNS	79	Standard query 0x0000 PTR _printer._tcp.local, "QM" question
245	18.777561809	10.42.0.48	224.0.0.251	MDNS	82	Standard query 0x0000 PTR _esphomelib._tcp.local, "QM" question
246	18.777604611	10.42.0.48	224.0.0.251	MDNS	82	Standard query 0x0000 PTR _esphomelib._tcp.local, "QM" question
247	18.777633201	10.42.0.48	224.0.0.251	MDNS	87	Standard query 0x0000 PTR _spotify-connect._tcp.local, "QM" question
248	18.777640101	10.42.0.48	224.0.0.251	MDNS	87	Standard query 0x0000 PTR _spotify-connect._tcp.local, "QM" question
249	18.777648836	10.42.0.48	224.0.0.251	MDNS	76	Standard query 0x0000 PTR _wled._tcp.local, "QM" question

Figure 4.3 A fragment of network traffic when HA sent discovery queries

or UPnP in new integrations for better compatibility with the HA.

4.2 Topology 1 - IoT devices individually connected to the network

In the first topology we set up, each device (refer to table 3.1) was independently connected to the AP. We saw that the Wemo outlet, whose purpose is primarily to provide a remote way to turn on and off a device connected to it, sent out four UPnP discovery queries approximately every 140 seconds.

The discoveries query sent by the smart outlet were limited to discover only IoT devices of the same vendor defined by the query *urn:Belkin:service:basicevent:1*. None of the other devices sent discovery queries, and as they were individually connected there were no queries to respond to.

4.3 Topology 2 - Only companion apps connected to the network

In this second topology, we connected each companion app independently to the AP. We used a mobile firewall to limit undesired network traffic from others apps.

The Lifx, Kasa, tpCamera and mydlink apps used proprietary discovery protocols, and the

discovery queries were sent to the broadcast IP 255.255.255.255. The UDP payload was defined by the vendor (proprietary), and it would be necessary to reverse engineer the protocol to get the details of its discovery message. We present underway efforts to reverse some of these protocols in section 4.10. We adopted the port number as a general way to identify queries for propriety discovery protocols, since payloads will always vary as per vendors decisions. Table 4.3 shows the identified discovery queries sent by each companion app used in our testbed.

Table 4.3 Discovery queries sent by companion apps

	Discovery query identified		
App	**mDNS**	**UPnP**	**Proprietary (ports)**
Lifx			56700
Wemo	_hap._tcp.local Android.local	urn:Belkin:service: basicevent:1	
Kasa			9999 20002
tpCamera			1068
Mi Home	_miio._udp.local _rc._tcp.local		
mydlink			5978

We observed that Kasa sent malformed mDNS requests, in addition to the proprietary discovery queries sent. The Figure 4.4 shows one of these malformed packets. These packets were sent to the standard mDNS IP and port, but the discovery body was empty, surrounded by the red square in the figure. The Figure 4.4 shows how Wireshark identify the packet as a mDNS Malformed Packet.

By chance, while performing our experiments, the Kasa app was updated to a newer version. This new version included a set of new discovery messages sent to the UDP port 20002. This behaviour change was not described explicitly in the app changelog, demonstrating how vendors may arbitrarily add or remove discovery functionality.

The Wemo app sent one UPnP and two mDNS discovery queries. One mDNS query was *_hap._tcp.local* which is used to discover *HomeKit Accessory Protocol* (HAP) [28] devices, even if HAP is not specifically for Wemo products. The second mDNS query was *Android.local* which aims to discover devices with Android as their hostname, like Android smartphones or perhaps Android smart TVs. The UPnP query (*urn:Belkin:service: basicevent:1*) was the one that corresponds to Wemo products.

This experience showed that companion apps can send discovery queries that are unrelated

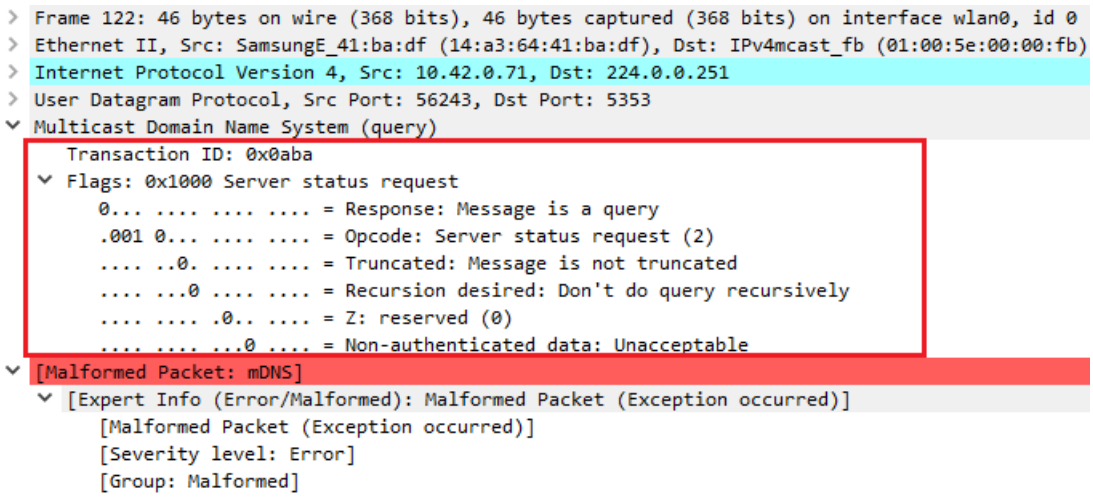

Figure 4.4 Traffic network recorded of Kasa malformed mDNS

to their IoT devices. This can lead to *cross-vendor* device discovery, as described in topology 4. IoT vendors can also use discovery protocols to partially identify other IoT devices, as Yu *et al.* [54] demonstrated, since they may be curious about what other devices a user owns, perhaps to offer purchasing incentives or loyalty discounts.

4.4 Topology 3 - All devices simultaneously connected to the network

In the third topology we connected all the IoT devices (refer to table 3.1) to the AP at the same time.

Even though the Wemo smart outlet sent UPnP discovery queries, it did not receive any replies from other devices in our set. This was likely because the discovery query *urn:Belkin:service:basicevent:1* was not understandable by other devices. However, indeed the requests were received by all devices at the network level if they are listening to the UPnP group (239.255.255.250).

IoT devices can send discovery queries regardless their purpose. Thus, this leads IoT devices to discover other devices connected to the same network.

4.5 Topology 4 - IoT devices interacting with their companion apps

In the forth topology we connected to the AP each IoT device and the smartphone with the respective device's companion app. In this topology, we also used a mobile firewall to limit undesired network traffic from others apps.

Table 4.4 only shows the discovery protocol and query used by companion apps to discover its IoT device. The Lifx, tpCamera and mydlink app sent exclusively the discovery query to discover their respective device.

Table 4.4 Discovery protocols and queries used by companion apps to discover their devices

App	mDNS	UPnP	Proprietary	Query
Lifx			✓	56700
Wemo		✓		urn:Belkin:service: basicevent:1
Kasa			✓	9999
tpCamera			✓	1068
Mi Home	✓			_miio._udp.local
mydlink			✓	5978

This topology confirmed that IoT vendors use other discovery protocols besides of the ones needed to discover their devices. Wemo, Kasa and Mi Home sent discovery queries at least not used to discover the device in the testbed. Because Kasa app sent a malformed mDNS query, and Wemo uses a mDNS query (HAP) apparently not tied to Wemo devices and even a mDNS query tied to Android systems.

4.6 Topology 5 - All devices and vendor apps connected simultaneously

The fifth topology was composed of all the IoT devices and the smartphone with all the companion apps connected to the AP simultaneously. In this topology, we also used a mobile firewall to limit undesired network traffic from others apps.

In the third topology, we observed that the Wemo app sent the mDNS query *_hap._tcp.local*. In this topology we saw that once the Wemo app sent the mDNS query, the Lifx bulb answered to it, providing its IP address, MAC address and hostname. The Wemo app apparently did not take further actions after receiving this discovery response. It showed that even though the Lifx bulb communicates with its companion app through a proprietary discovery protocol, it also replies to mDNS queries. We had this first approach when analyzing the interaction between the HA and the Lifx bulb.

Even on our small testbed, there appear to be a *cross-vendor* discovery, which points out that IoT devices from one vendor can discover other vendors' devices. Furthermore, IoT devices may answer to discovery protocols beyond those used by their companion apps.

4.7 Topology 6 - Companion apps logged in with an different account

The sixth topology was composed of each IoT device and the smartphone with the respective device's companion app connected to the AP. In this topology we used companion apps set up with a new user account to simulate guest users joining the network. We also used a mobile firewall to limit undesired network traffic from others apps.

The Lifx, Wemo and Kasa apps sent discovery messages, and immediately added the discovered IoT devices, allowing users other than their owners to control them. The other companion apps somehow matched IoT devices with their owner's account, denying others users to control the devices. However, companion apps operated from a different user account could still identify the presence of IoT devices of the same vendor in the network.

The Lifx and Wemo apps automatically added their corresponded discovered devices on the network. In contrast, the Kasa app popped up a message telling the user that some devices were found on the network and asking if they should be added to the app. This confirms the behaviour that we saw when the HA interacted with these IoT devices.

4.8 Topology 7 - IoT devices interacting with our selected discovery queries

The seventh topology was composed of each of the IoT devices individually connected to the AP. In this case the AP could send discovery queries selected by us.

Both the Lifx bulb and the TP-Link IP camera replied to the mDNS root discovery, even though their vendor mobile apps did not use mDNS to discover them. Moreover, we noticed that it was possible to know the physical location of IoT devices through mDNS and UPnP. For usability, users commonly identify their devices according to their physical location in the house like Living Room light, Entrance door light, Garage light, etc. This information is shared in the mDNS *records* reply and the UPnP description/configuration file. We were able to observe these friendly names while performing these experiments and looking at the records performed in previous topologies.

In the first topology, we saw that the Wemo outlet sent discovery queries. We set up a virtual Wemo outlet developed in Python which answered to the discovery query sent by the real one. This proved that the real Wemo outlet can discover other devices since it discovered the fake outlet. Once the discovery was performed, the real Wemo outlet started the initial stage to monitor and control the fake outlet. That is, it downloaded the description/configuration file provided by the fake Wemo outlet and asked for further information to control it, as per the UPnP process.

Table 4.5 shows a summary about what we observed in the topologies. The table organize the information in general by vendors and discovery queries. We adopted the port number to identify vendor-specific (propriety) discovery queries, since payloads will always vary as per vendors decisions. We could notice devices answering discovery queries different from the ones used by their companion apps. Furthermore, we could notice companion apps sending other discovery queries different from the ones used to discover their devices.

Table 4.5 Summary of discovery queries seen on our local testbed

		Discovery query			
		mDNS	UPnP	Port Num.	root
Lifx	Sent by app			56700	
	Sent by device				
	Device answers to	_hap._tcp		56700	mDNS
Wemo	Sent by app	_hap._tcp Android	Belkin[c]		
	Sent by device		Belkin[c]		
	Device answers to		Belkin[c]		
TP-Link	Sent by app	malformed[b]		9999 20002	
	Sent by device				
	Device answers to			9999	mDNS[a]
D-Link	Sent by app			5978	
	Sent by device				
	Device answers to			5978	UPnP
Yeelight	Sent by app	_miio._udp _rc._tcp			
	Sent by device				
	Device answers to	_miio._udp			mDNS

[a] Only Camera NC250
[b] mDNS with empty discovery query
[c] urn:Belkin:service:basicevent:1
Note: All mDNS queries end with *.local*. Proprietary queries denoted by target service port.

4.9 Public Dataset

We analyzed the Yourthings [41] dataset to identify similar patterns as the ones we observed in our testbed and further relevant information about the use of discovery protocols.

We developed a Python script which swept the dataset and gathered data related with

discovery protocols. The script filtered according to the IP and ports defined by discovery protocol. Then, it identified if there was a discovery request or an answer. Finally, it gathered information about the discovery packet such as time, source IP, source port, destination IP, destination port, discovery protocol, discovery query and payload. When the script finished it deposited the information in a CSV file for further analysis. Please refer to annex A to find the script source code.

Table 4.6 shows the devices identified that sent root discovery queries. We noticed that home automation hubs such as Logitech Harmony Hub, Google Home and Google Home Mini sent root UPnP queries. In contrast, Google OnHub, Samsung SmartThings Hub, Philips HUE Hub, Insteon Hub, MiCasaVerde VeraLite and Wink 2 Hub did not use neither UPnP nor mDNS root queries. The hubs using root discovery queries can discover devices that they are not even able to control, violating the principle of least privilege and jeopardizing the user's privacy.

Moreover, we observed that the Belkin Netcam WiFi camera used a root discovery query and discovered other devices. According to the vendor website, the camera provides video streaming and uses motion sensor notification to take photos and record videos. The camera also is controlled by the Wemo mobile app. It seems unlikely that the camera needs root discovery to work properly. This causes a violation of the principle of least privilege too.

We also noticed an iPhone and an iPad using root discovery UPnP query. Additionally, the iPhone also sent the mDNS root discovery. Unfortunately, we could not determine if those queries were sent by a companion app since there is no way to match network traffic with mobile apps. Further, there was no information about the vendor apps installed in the smartphones included in the dataset. We also detected three devices using UPnP root discovery that were not identified by the dataset's device mapping file.

Table 4.6 Devices who sent root discovery queries

Device	Root discovery protocol
iPhone	mDNS
Logitech Harmony Hub	UPnP
Belkin Netcam	UPnP
iPad	UPnP
Google Home	UPnP
Securifi Almond	UPnP
Google Home Mini	UPnP
iPhone	UPnP

Table 4.7 shows the mDNS discovery queries identified that were sent by devices. We noticed that the Amazon Echo answers to a discover query meaning that the device provides a SFTP or SSH service. Both, SFTP and SSH, work in port 22 and both allow a remote access. Amazon Echo is a home automation tool, hence if the remote access is compromised, devices managed by the Amazon Echo might be compromised too. We argue that is likely that Amazon Echo remote access can have a configuration vulnerability since usually these kind of devices have default credentials and they are not changed by users and the service is announced by a discovery query.

Furthermore, we noticed that the Bose SoundTouch speakers sent a mDNS discovery query. As per the service name we can assume that the query sent is to discover Bose SounTouch speakers too. However, the speakers are a specific purpose device trying to discover.

Nevertheless, we could notice a cross-vendor discovery. The Logitech Harmony hub sent the *airplay* mDNS discovery query and the Apple HomePod answered to this query. Both devices are home automation hubs, and it is unlikely that home automation hubs have to manage other home automation hubs. Usually they control specific purpose devices. Moreover, as per the knowledge gained in the testbed observations, we can state that Wemo mobile app would can discover Caseta Wireless hub, Koogeek Lightbulb and Philips HUE Hub. As per the testbed observations, Wemo mobile app functions as the way to control Belkin Wemo devices. However, the Wemo companion app might discover a bulb and home automation hubs belonging to other vendors.

Table 4.8 shows a summary of the UPnP discovery queries identified that were sent by devices. We noticed plenty of specific purpose devices sending discovery queries. The majority from Belkin vendor, and two others were speakers, the SONOS and Base SoundTouch speakers. The Belkin Netcam was the only specific purpose device that appeared to send discovery queries that are not related with the camera function or the vendor itself.

Table 4.8 shows a summary of the UPnP discovery queries identified that were answered by devices. We noticed two devices answering to plenty of discovery queries. The Wemo Crockpot answers to at least twelve different service targets. One of these two seems to have a relation with a firmware update and the other one with a remote access to the crockpot. We declared that both functions are risky since these processes are announced by a discovery.

Table 4.7 Summary of mDNS discovery queries identified in the dataset

	Discovery query	
	Sent by device	Device answers to
Amazon Echo		_sftp-ssh._tcp
Harmon Kardon Invoke	_sleep-proxy._udp	_spotify-connect._tcp _http._tcp
Logitech Harmony Hub	_lutron._tcp _appletv._tcp _airplay._tcp	
Google Home	_googlecast._tcp _googlezone._tcp	_googlecast._tcp _googlezone._tcp
Google Home Mini	_googlecast._tcp _googlezone._tcp	_googlecast._tcp _googlezone._tcp
Apple HomePod	_sleep-proxy._udp _companion-link._tcp _homekit._tcp	_companion-link._tcp _homekit._tcp _airplay._tcp _raop._tcp
Bose SoundTouch 10	_soundtouch._tcp	_soundtouch._tcp _spotify-connect._tcp
Samsung SmartThings Hub		_smartthings._tcp
nVidia Shield	_nvstream._tcp _nvstream_dbd._tcp	_googlecast._tcp
Caseta Wireless Hub		_hap._tcp _lutron._tcp
Koogeek Lightbulb		_hap._tcp
Philips HUE Hub		_hap._tcp _hue._tcp
Piper NV		_piper._tcp
Samsung Smart-Things Hub		_smartthings._tcp

Table 4.8 Summary of UPnP discovery queries sent by devices in the dataset

	Discovery query Sent by device
Amazon Echo	urn:Belkin:device:** urn:schemas-upnp-org:device:basic:1
Wink 2 Hub	urn:smartspeaker-audio:service:SpeakerGroup:1
Belkin Wemo Switch	urn:Belkin:service:basicevent:1
Belkin Wemo Motion Sensor	urn:Belkin:service:basicevent:1
Logitech Harmony Hub	urn:Belkin:service:basicevent:1 urn:schemas-upnp-org:device:basic:1 urn:schemas-udap:service:smartText:1 urn:dial-multiscreen-org:device:dial:1
Belkin Netcam	urn:schemas-upnp-org:device:InternetGatewayDevice:2 urn:schemas-upnp-org:service:WANIPConnection:2 urn:schemas-upnp-org:device:InternetGatewayDevice:1 urn:schemas-upnp-
Wemo Crockpot	urn:Belkin:device: urn:Belkin:service:basicevent:1
SONOS	urn:schemas-upnp-org:device:ZonePlayer:1 urn:schemas-upnp-org:service:ContentDirectory:1
Samsung SmartThings Hub	urn:Belkin:device:lightswitch:1 urn:schemas-upnp-org:device:basic:1 urn:samsung.com:device:RemoteControlReceiver:1 urn:smartspeaker-audio:service:SpeakerGroup:1 urn:schemas-upnp-
Belkin Wemo Link	urn:Belkin:service:basicevent:1
Bose SoundTouch 10	urn:schemas-upnp-org:device:MediaServer:1 urn:schemas-upnp-org:device:MediaRenderer:1
Caseta Wireless Hub	urn:smartspeaker-audio:service:SpeakerGroup:1

Table 4.9 Summary of UPnP discovery queries answered by devices in the dataset

	Discovery query Device answers to
Wink 2 Hub	urn:wink-com:device:hub2:2 urn:wink-com:service:fasterLights:2
Belkin Wemo Switch	urn:Belkin:device:controllee:1 urn:Belkin:service:basicevent:1
Belkin Wemo Motion Sensor	urn:Belkin:device:sensor:1 urn:Belkin:service:basicevent:1
Logitech Harmony Hub	urn:myharmony-com:device:harmony:1
Belkin Netcam	urn:Belkin:service:basicevent:1
Wemo Crockpot	urn:Belkin:service:WiFiSetup:1 urn:Belkin:service:timesync:1 urn:Belkin:service:basicevent:1 urn:Belkin:service:crockpotevent:1 urn:Belkin:service:jardenevent:1 urn:Belkin:service:firmwareupdate:1 urn:Belkin:service:rules:1 urn:Belkin:service:metainfo:1 urn:Belkin:service:remoteaccess:1 urn:Belkin:service:deviceinfo:1 urn:Belkin:service:smartsetup:1 urn:Belkin:service:manufacture:1
Sonos	urn:schemas-upnp-org:device:MediaServer:1 urn:schemas-upnp-org:device:MediaRenderer:1 urn:schemas-upnp-org:service:AlarmClock:1 urn:schemas-upnp-org:service:MusicServices:1 urn:schemas-upnp-org:service:DeviceProperties:1 urn:schemas-upnp-org:service:SystemProperties:1

Table 4.10 gathers the proprietary discovery queries sent by devices. We map vendor-specific (propriety) discovery queries as the destination port used since payloads will always vary among vendors.

We noticed that the D-Link camera and the SONOS speakers sent a discovery query to the UDP ports 5978 and 1900 respectively. As per the knowledge gained in the testbed observations, we can state that the D-Link camera can discover D-Link cameras too. However, it is a specific purpose device trying to discover. On the other hand, we noticed that the payload sent by SONOS speakers is in fact a UPnP NOTIFY discovery query. That is, the query was sent to the wrong UPnP IP.

We went through the payload of the discovery query sent to the UDP port 1900 by the iPhone and the iPad. We noticed a UPnP search for SONOS devices, but sent to the incorrect IP. We inferred that this discovery query was sent by the companion app of the SONOS speakers. Therefore, we argued that SONOS vendor is misusing UPnP discovery.

The Lifx virtual bulb also sent a discovery query. We observed that the Lifx used in our testbed it did not send any discovery query. However, perhaps the library used to generate the Lifx virtual bulb of the dataset has this functionality. There is no information regarding how the Lifx virtual bulb was generated nor software or library used.

As per the knowledge gained in the testbed observations, we can state that Logitech Harmony Hub and the iPad can discover Lifx bulbs. Because they sent a discovery message targeting the broadcast IP address 255.255.255.255 on UDP port 56700. Moreover, we can state that the iPad can discover the D-Link camera, and the iPhone can discover Tp-Link IoT devices since they sent a discovery message targeting the broadcast IP address 255.255.255.255 on UDP port 5978 and 9999 respectively.

Table 4.10 Summary of proprietary discovery queries queries identified in the dataset

	UDP port used to send the proprietary discovery
Logitech Harmony Hub	56700 137
D-Link DCS-5009L Camera	5978
LIFX Virtual Bulb	5353
Sonos	1900
iPhone	1900 9999 10000
iPad	56700 5978 5224 1900 10000

Since there is no information about the time of use or interaction of the devices in the dataset, we could identify devices that appeared constantly through the dataset data to get the frequency of sending discovery queries. Figure 4.5 shows the frequency of sending discovery queries in minutes of ten devices. The Bose SoundTouch device sends every thirty seconds UPnP discovery queries and every hour mDNS discovery queries. The Apple HomePod and the nVidia Shield send their corresponding discovery queries every hour. On the other hand Belkin devices send queries every two minutes and a half. Google hubs give UPnP queries a minute and mDNS queries a minute and a half to be sent. The Samsung SmartThings hubs also send queries every minute. Consequently, we concluded that vendors decide the time frame to send discovery queries in their judgment.

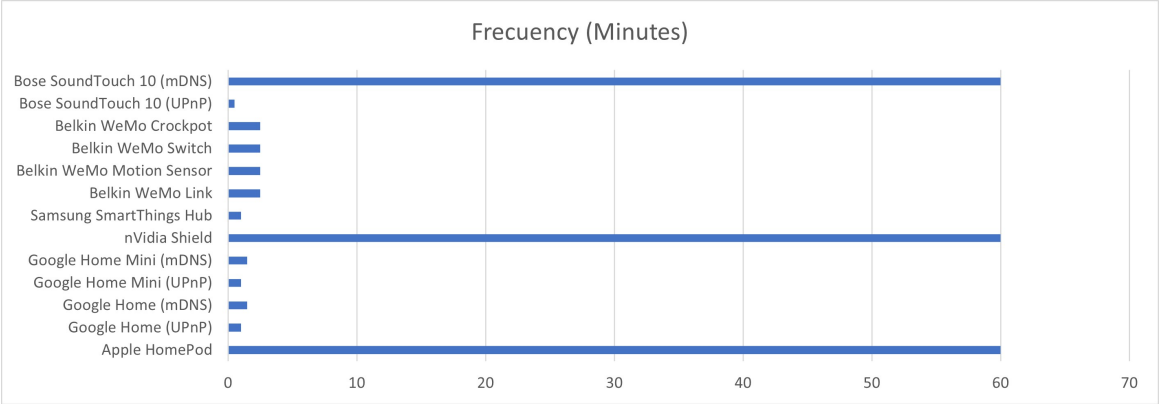

Figure 4.5 Frequency of discovery queries sent by some devices

The Philips HUE Hub was discovered by eighteen different devices, see Table 4.11. Eight out of the eighteen were also home automation hubs. However, six out of the eighteen were IoT devices (specific purpose devices), and they could discover the Philips HUE Hub.

Even though in the dataset there was no information about the existence of companion apps and the interaction between home automation hubs, companion apps (if any) and IoT devices, we could get relevant information about the use of discovery protocols. For instance, we could see more devices sending root discovery queries, both home automation hubs and specific-purpose devices. This is similar to what we saw with the analysis of the HA, but we could see that even specific purpose devices can send root discovery queries (Belkin Netcam sent it). Additionally, as we noted in our testbed, we identified specific-purpose devices sending discovery queries which seems to be unnecessary for their proper function. Moreover, we could identify cross-vendor discovery already seen in the dataset or assumed as per the knowledge gained in the testbed analysis. Therefore, we could identify situations that were presented in our previous analysis too, like Wemo devices sending discovery queries to discover other Wemo devices.

Table 4.11 Devices that discovered Philips HUE hub

Device	Type
Amazon Echo	Hub
Android Tablet	General
Belkin Netcam	IoT
Belkin Wemo Link	Hub
Belkin Wemo Motion Sensor	IoT
Belkin Wemo Switch	IoT
Bose SoundTouch 10	IoT
Caseta Wireless Hub	Hub
Google Home	Hub
Google Home Mini	Hub
iPad	General
iPhone	General
Logitech Harmony Hub	Hub
Samsung SmartThings Hub	Hub
Samsung SmartTV	General
Sonos	IoT
Wemo Crockpot	IoT
Wink 2 Hub	Hub

4.10 Inspecting proprietary discovery protocols

In our testbed, we noticed that the Lifx device and the Tp-Link devices answered to proprietary discovery protocols. There are already libraries in GitHub [55–57] which allow the use of these protocols in devices that are not tied to their vendors. We performed an analysis in the structure of these protocols to identify their features.

Tp-Link's protocol is encrypted and the encryption algorithm is an autokey cipher. The algorithm performs an XOR between the plaintext and an initial secret key (decimal value of 171) [58]. Tp-Link's protocol uses JSON text format in the whole communication. Lifx's protocol transmits in plain text, but uses its own communication format. Despite this, both protocols send an initial query to discover their devices that follows their protocol structure. These initial queries are meant only to discover their devices and then start a control communication.

Lifx's protocol as well as its discovery queries have mainly the following structure:

- Packet size
- Protocol
- Communication ID
- MAC Address
- Lifx version
- Command ID
- Time stamp
- Packet type

To identify such query, we must search for the following information:: *Packet size* defines the payload's number of bytes, and the discovery query has the value 0x0024. *Protocol* has the value of 0x3400 when a discovery query is sent. It changes to 0x1400 when the control communication starts. Finally, *Packet type* has the value of 0x0002 in the discovery query, and it changes to 0x0003 when a device answers to this query.

The Tp-Link discovery relies on JSON text format, and it uses a variable called *system* which value is a nested JSON. This nested JSON has a variable called *get_sysinfo* whose value is *null*. The message shown below is the one used by Tp-Link to discovery its IoT devices.

{"system":{"get_sysinfo":null}}

Thus, we argue that mDNS or UPnP have the same features than those of proprietary protocols. mDNS can be used as the discovery query to find the devices. Devices will answer back with useful information for the control communication. However, perhaps vendors think that it is better to rely on only one protocol to discover and then controlling the IoT device. UPnP provides discovery and control stages so, it is suitable when requiring both stages working in the same protocol. Both mDNS and UPnP are not encrypted and vendors can argue that a proprietary protocol is necessary to aggregate encryption. However, Tp-Link proprietary protocol has shown that vendors implement bad encryption in their devices [58].

We conclude that proprietary protocols do not provide any new features to the already existing standard discovery protocols used by some IoT devices.

4.11 Summary

This chapter presented the results of the empirical experiments we did to observe the discovery behaviour of IoT devices. The empirical experiments include, a code review of a well-know open-source home automation hub and an inspection of the network traffic generated by the hub. Moreover, the empirical experiments include an examination in detail of the network traffic and interaction patterns of discovery protocols of both, a small testbed and a public dataset.

The empirical experiments helped identify the threats impacting IoT devices due to discovery protocols. We have highlighted current real world security concerns in IoT networks due to the use of discovery protocols. The most relevant security concerns found during the empirical analysis can be summarized as follows:

IoT devices can discover other devices regardless of their purpose through the use of root discovery queries. We observed several instances of this including from apps, cameras, and home automation hubs.

Companion apps send discovery queries beyond those their devices understand. We observed some vendor apps using discovery protocols different from the ones that their devices use. This situation enables cross-vendor discovery, i.e., a vendor app discovering an IoT device from another vendor on the same local network.

Proprietary discovery protocols seem not to provide improvements against the standard ones. We inspected two proprietary protocols and they provide both stages, discovery and control, together. However, UPnP provides the same feature. Moreover, we discussed that proprietary protocols showed bad encryption implementations.

Vendor apps may allow anyone connected to the network to control IoT devices. We noticed that some IoT devices allowed any user on the same network to control them from any smartphone running their vendor mobile app, even though the user account logged in was different from the device owner's account.

Discovery protocols share information about IoT devices' location on the network. To better identify the same type of IoT devices in a network, users can assign friendly names related to their physical location in the house. As discovery protocols share this information, it is possible to know the devices' location through a discovery query.

CHAPTER 5 TOWARDS PROTECTING IOT NETWORKS FROM DISCOVERY PROTOCOL THREATS

This chapter describes security recommendations and proofs of concept implementations that address the security concerns identified in previous chapters. We developed several tools that can send custom mDNS and UPnP queries, as spoof a Lifx bulbs and a Wemo outlets, as well as a comprehensive SDN-based discovery protocol filtering mechanism. We present the design and use of these tools, demonstrating that they reduce the risk of using discovery protocols on home networks.

5.1 Example attacks exploiting IoT discovery protocols

Based on our empirical experiments, we identified threats that impact directly in the use of discovery protocols in IoT devices which brings security concerns. In this section we describe some possible attack scenarios in IoT networks:

- An individual without any IT knowledge trying to control IoT devices without authorization. As we observed during the empirical analysis we conducted, home automation hubs and some companion apps allow an automatic control of IoT devices once they are found in the network. An individual without IT knowledge, with either authorized or unauthorized access to the network, and having a home automation hub or the companion app of the targeted IoT device, could control such IoT device. The access to the network may even be authorized since many users usually share the home network's password with guests and relatives during social meetings. Consequently, more people can access the home network even if it is protected by a strong password.
- An IoT device has been compromised and is being used as a means of obtaining information of the IoT devices connected to the same network. An attacker who wants to break-in the place where there are IoT devices installed can use this information to likely know the physical location of cameras, bulbs or smart locks for instance. Since some IoT devices are placed outdoors, the IoT device can be infected by a physical action such as a memory swap or a Joint Test Action Group (JTAG) attack. JTAG is designed for debugging and testing a device after its manufacture, and can be used as software defacing.
- An individual with advanced computer security skills can leverage discovery protocols to perform an unnoticed passive reconnaissance. Discovery protocols provide a much

less invasive reconnaissance than tools like Nmap [59] do. Nmap reconnaissances are easily detected by Intrusion Detection System (IDS) or similar tools, whereas discovery protocols are considered benign.

5.2 Improving the security of IoT networks while using discovery protocols

Up to now we have highlighted real-world discovery protocol security issues present on modern IoT devices. However, these protocols appear to be necessary to facilitate onboarding and day-to-day use of IoT devices. In this section we provide some security recommendations on how to improve security while allowing devices and apps to continue using discovery protocols.

We suggest to implement visibility to be aware of the use of discovery protocols on the network by showing which devices are being discovered by which others, showing the discovery exposure of a device and notifying when a device sends a root discovery. Moreover, we limited the discovery on the network since discovery is denied by default and any root discovery is denied too. The discovery control is conducted as granular restriction, since it is defined by service (mDNS), label (UPnP) or port (proprietary) and not by the protocol. Nevertheless, the discovery behaviour of IoT devices is monitored to detect changes in the future and notify about such changes.

5.2.1 Limit/control of discovery queries

Conventional ways to limit/control the network traffic, such as firewalls, are not able to limit the use of discovery protocols on local networks. Discovery queries do not cross the gateway/firewall because they usually do not reach any external network. Unless a firewall is placed in front of each IoT device, and the devices running the companion apps, as Yul *et al.* did [48], discovery queries cannot be easily limited.

Virtual LANs (VLANs) are used to isolate networks. A VLAN could be used to isolate IoT devices from non-IoT devices by placing them all into one network. However, cross-vendor discovery would still persist in the IoT VLAN. Thus, ideally devices should be placed in a vendor-specific VLAN, such that cross-vendor discovery would be prevented. But in practice this approach presents several challenges, like how to use companion apps to control the devices in each VLAN. No simple approach exists yet for mobile devices to easily switch VLANs or to join several VLANs at a time.

Software Defined Networking (SDN) provides the facility to monitor and control local network traffic. SDN separates the control plane from the data plane, enabling policy-based

management of network resources. SDN has been used in network security research proposals to implement access control [44, 45]. It also has been used in IoT security research, both in ad-hoc [47, 60, 61] and WiFi [49, 62] networks. We view SDN as a plausible technology to enable the deployment of discovery protocol filtering in IoT networks. The use of SDN requires only a controller and a data plane device. There are many options for SDN controllers, from graphical interfaces such as OpenDayLight [63] to programmatic ones such as Ryu [64]. The controller communicates with the data plane using the OpenFlow [65] protocol. Some commercial routers (such as Mikrotik [66]) are distributed in the market with software options to turn them into SDN-capable devices.

In next sections we will provide security recommendations and their proof of concept to improve the security concerns we found in the use of discovery protocols by IoT devices.

5.2.2 Discovery protocol mapping

We named discovery mapping the trace of discovery between two devices on the network. To better understand how discovery protocols are used in IoT networks, we suggest using a direct graph to display the traces of recent discovery queries and responses (if any). Such a map can quickly show, even to non-experts, what devices are interacting with other devices through discovery queries. Devices can be mapped as nodes, and the discovery query sent out can be displayed as edges. The edge's direction determines the requester and responder. The nodes give information to identify the device like IP, MAC address and vendor. The vendor is obtained based on the device's MAC address from a Wireshark data base [67]. Please refer to Figure 5.3 to see an example of a map generated when devices discover other devices.

The discovery mapping provides visibility and transparency about the discovery performed by devices. Discovery mapping helps identify devices that are being discovered by undesired requesters. For instance, it is possible to visualize if a specific purpose device like a smart switch is discovering other devices, or if a device/vendor is discovering certain devices that they should not discover, as we observed in our testbed.

5.2.3 Discovery exposure

As we discussed in chapter 4, IoT devices may transmit numerous discovery queries while they are online. However, not all the queries will receive responses, depending on the diversity and configuration of devices connected to the network. We borrowed the "exposure map" concept from Whyte *et al.* [68] to define a "discovery exposure", which identifies the set of discovery queries that may receive a response, even if the query do not receive any. If there

is no response, it is not possible to get the mapping discovery, yet the device sending the queries has that range to discover other devices.

The discovery exposure provides visibility and transparency about discovery queries sent by the devices. This helps identify the range of forthcoming devices that may be discovered in the future. This complements the discovery mapping since the whole portion of the discovery queries used by a device can be visualized.

We suggest to use a drop-down list to present the discovery queries sent by each device, due to the possible size of the discovery queries sent. For instance, the discovery exposure representation could follow the following structure: the discovery protocol is placed at the top of the drop-down list, then a drop-down list of devices by IP is presented, and finally the queries sent are shown.

5.2.4 Notification about undesired discovery messages

Our analysis revealed the use of unnecessary root discovery queries, as well as undisclosed discovery protocol changes to the behavior of a companion app. We suggest logging and classifying discovery queries per device, and alerting the user when the type, destination port, or frequency of discovery messages changes, as this can help detect potential risks. These notifications can be sent, for example, via email to the user. Such alerts should always be sent out for root discovery queries, since these queries will successfully elicit responses from all devices.

Such notification system would have detected the change of behavior of the Kasa app, when the new destination port/protocol was added. Moreover, this notification system would detect any changes in a device's behaviour. The notification should contain enough information about the device that sent the query, the discovery protocol used and the discovery query. In annex B there is a sample of a Python script which such functionalities to detect and notify via email about a root discovery seen in the network.

5.3 Deploying our recommendations

This section describes how we implemented a proof-of-concept to evaluate the recommendations we gave in the previous section. We implemented a Python-based system to provide visibility and transparency of IoT discovery protocols, as well as to enforce policies regarding the type of discovery queries allowed. Our system is depicted in Figure 5.1.

We considered controlling discovery protocols in a way that a discovery query can be denied

or allowed. Since devices are discovered by a query using a protocol and not by the protocol itself, if the whole protocol is denied or allowed the discovery can be limited. That is the reason why we considered to implement a granular control, that is, instead of denying or allowing mDNS, UPnP or proprietaries protocols, we deny or allow discovery queries.

The proposed solution analyzes the queries sent by a device, and it determines if such queries are allowed or not for that device. Hence, the device can still use the discovery protocols but only the allowed queries will be forwarded.

In our implementation, deny is set as default to all the discovery queries. However, it is possible to select the ones to be allowed. We rely on the discovery exposure structure mentioned in section 5.2.3, and once queries are allowed, the mapping will start being drawn.

Conventional access control tools like firewalls cannot control local network traffic because they cannot see this traffic. As discovery requests are sent in the local area network, we need a technology that allows us to control local network traffic. Software-Defined Networking (SDN) provides the flexibility to control and visualize local network traffic.

5.3.1 Software-Defined Networking

SDN is an emerging network architecture which decouple the switch's data plane with control plane, moving the control plane in a centralized place called controller. SDN allows to add programmability on the control plane which is where the SDN uniqueness resides. Thus, SDN offers a centralized network's control through programming, which can be based on network status and/or user defined policies [69].

5.3.2 OpenFlow

In order to communicate the data plane with the control plane, there is a protocol that makes that task possible. OpenFlow is a mature solution and a standardized protocol that allows data and control plane communication. It is considered as the implementation of SDN [69].

5.3.3 SDN controllers

The controller needs to understand OpenFlow and be able to send commands to the data plane. For such capability, there are commercial and open source controllers. Open source controllers provide a wide range of options since some of them support Java or Python programming, and others provides a graphical interface for controlling. Well know open source controllers are POX [70], Ryu [64], Floodlight [71] and OpenDaylight [63].

5.3.4 Proposal deployment

Based on what we saw when analyzing the discovery protocols in IoT devices, we propose an initial attempt to counteract the security issues found in our analysis. This initial attempt is one possible architecture toward solving the problem. Due to our proposal is an initial attempt, it has only been preliminary evaluated rather than fully analyzed/evaluated. Our proposal is based on SDN since it allows to have a control on the local network traffic. With SDN the control plane is separated from the network switch, while the data plane is kept in the switch.

We first wanted to provide visibility for the user to be aware of the use of discovery protocols in the network, by showing which devices are being discovered by which others. Thus, our solution provides a graphical mapping of the devices that are being discovered by others devices. In addition, we show the device's discovery exposure. That is, number of discovery queries that a device can send, even if none device on the network can answer to it, Such queries show the device's range of discovery.

In addition, we wanted to notify the user when a device sends a *root* discovery and deny these queries by default, since we consider it is harmful to the user's privacy.

We implemented all that with the help of SDN and an IDS. We used SDN to send all the WiFi local traffic to the IDS. The IDS provides the new three functionalities we proposed: the discovery mapping, the discovery exposure and the *root* discovery notification. Moreover, it provides a security policy to define who can discover what.

We developed the IDS in python, which analyse the network traffic to identify discovery protocols. It identifies the type of discovery protocols being used (mDNS, UPnP or proprietary). It records the discovery query used by devices and generates the mapping about which device answered (was discovered) if any. Moreover, the IDS triggers an alert when a device uses a root discovery query. This alert sends an email to the user with information about the device sending the root discovery query and the devices being discovered by such query.

We want to prevent the use of root discovery queries because we argue that this violates the least privilege concept. However, as we noticed in the public data set, the home automation hub Google Home is a device which seems to rely on a root discovery query to work properly since it is the only UPnP query sent by the device. Thus, we need to provide the network administrator the ability to select which discovery queries are allow by devices. We also wanted to provide a way to identify known discovery queries being tied to specific vendors being used by devices from other vendors. For instance, we saw UPnP discovery queries being tied to Belkin since they start with the label *urn:Belkin:device:*. Thence the network

administrator can identify if a device from other vendor is using a discovery query known to be tied for other vendor.

Hence, the network administrator is the one in charge of set the allow/deny rules through the discovery mapping and discovery exposure interfaces. When a selection is performed, the IDS generates a security policy which stands for which discovery queries are allowed or denied for device. Unfortunately, this initial attempt to counteract the security issues when using discovery protocols needs a person who has networking technical knowledge.

The security policy provided by the IDS is used by the module who enforces the rules. This limits the discovery of devices on the network. This control is granular since it is defined by service (mDNS), label (UPnP) or port (proprietary) and not by protocol, since if we deny a device to discover with mDNS it wont be able to discover anything using mDNS. With the granular restriction, we can select exactly what queries we want to deny for a specific device. So, a device discovering can be restricted to discover a specific smart bulb for example.

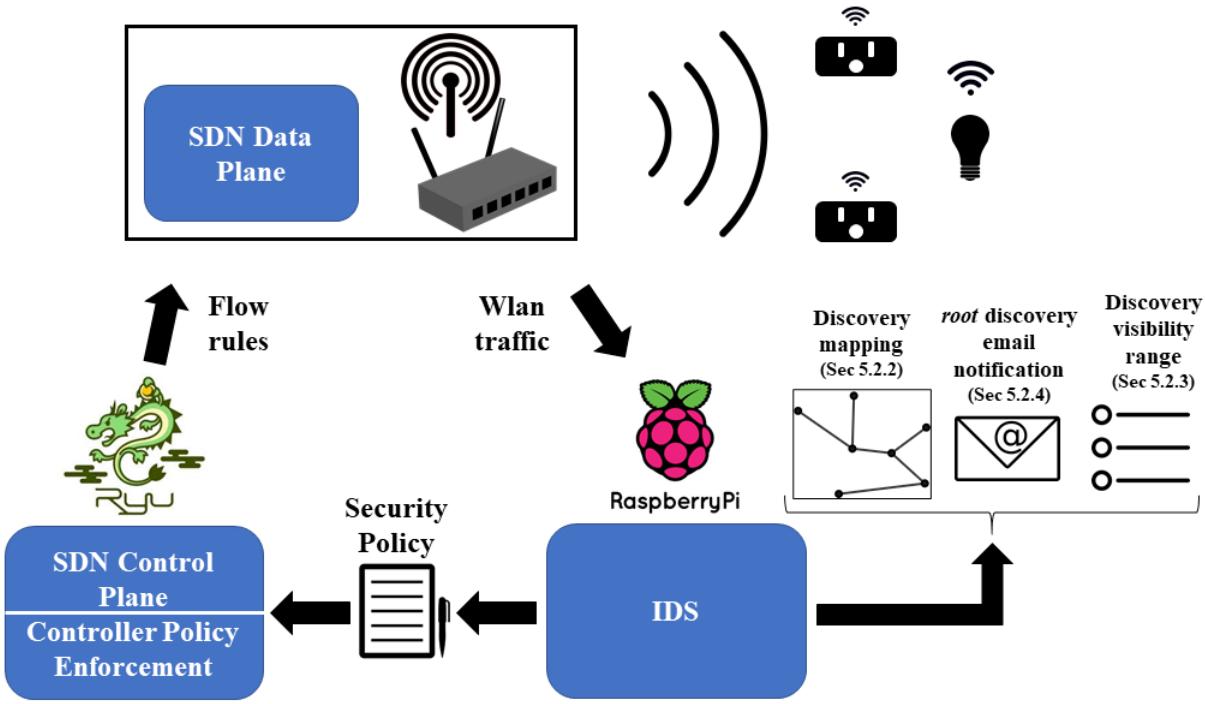

Figure 5.1 Diagram of the proposed solution to control discovery protocols

We consider acceptable discovery queries the ones that are sent by a device and can discover devices from the same vendor as long as the sender is not a general purpose device or a home automation hub. We consider root discovery queries as unacceptable, however they seem to be necessary for some device, reason we suggest a policy management in chapter 5.4.

Additionally, the IDS logs the time when discovery queries are sent by the devices on the network. With this record the IDS generates an initial discovery behaviour for each device. The initial discovery behaviour is generated through the discovery queries sent by a device during a day. As per our analysis in the public data set we argue that a day is enough for a device to send its discovery queries, as well as enough time for users to interact with their devices in order to trigger discovery queries due to this interaction. Hence, IoT devices are monitored to detect any change in the use of discovery protocols in the future, since they can change their behaviour and start sending queries different from the ones they usually send.

Figure 5.2 simplifies the idea of our proposal. We have a discovery control module that can communicate with the switch through Open Flow and such switch is able to control the WiFi traffic. Hence, devices can be restricted to what they can discover and they are monitored.

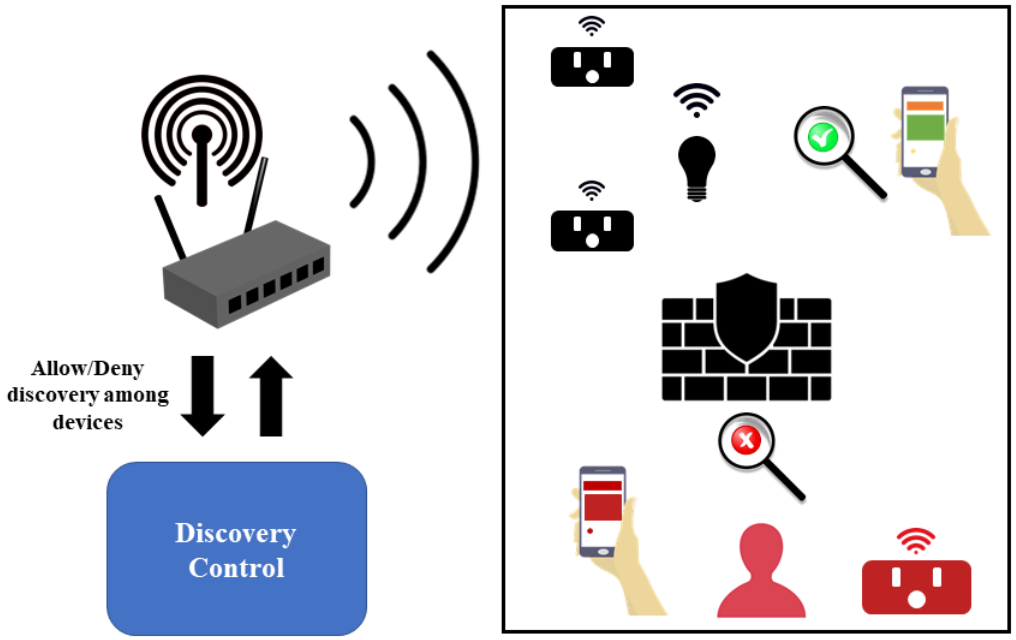

Figure 5.2 Diagram of the proposed solution working, where only specific devices can discover others on the network

5.3.5 Proposal testing

We performed small scale experiments with real IoT devices, their companion apps, a home automation hub and our custom tool to analyze the response of our proposal and the impact it can have in the environment.

5.3.6 Implementation notes

The proof of concept was developed mainly in Python for all the network analysis and for sending the notifications when needed. The directed graph was developed in node.js. This directed graph permits interaction to allow or deny present discovery queries in the network. Moreover, we used the Ryu framework for the SDN controller which is also programmed in Python. Either Python and node.js are cross platforms and open source. Last but not least, we used a Mikrotik router board model Rb951g-2hnd which supports SDN, thus, we could communicate our controller with the router to control the flows.

5.3.7 Transparency experiments

To evaluate the transparency about the use of discovery protocols, we connected all the IoT devices together in order to visualize the discovery performed in the network. A common interaction with the devices was performed. That is, launching mobile apps and interaction with the devices, turn on/off. The directed graph started drawing the discovery. A principal node was got which could discover three other nodes with three different queries.

On the other hand, once the home automation hub was launched, the graph allowed us to observe that the devices were discovered as first instance by root discovery queries instead of their specific queries. Once we looked at to the home automation hub GUI, a message was popped up asking if the devices found should be added. However, if a manual discovery was performed by cancelling this message, or even accepting this message, the graph showed how the discovery was performed by the original discovery query.

Figure 5.3 shows an example of the graph generated while performing the mentioned above. It is possible to observe nodes with their information, as well as edges.

The other part of transparency relied on the discover range, because the graph only showed successful discoveries but not the queries already sent by a device. Using the same logic when doing the discovery transparency, we could get the queries sent by the companion apps and the home automation hub, and we presented in a structured list. Figure 5.3 shows that the node Xiaomi discovered two devices since it sent a discovery message targeting the UDP ports 9999 and 56700. However, the discovery exposure showed that the same node also sent a discovery message targeting the UDP port 20002. See the discovery exposure in Figure 5.4.

On the other side, the home automation hub sent many mDNS queries which could not discover anything. As they were sent, the device is capable to discover if a device understand one.

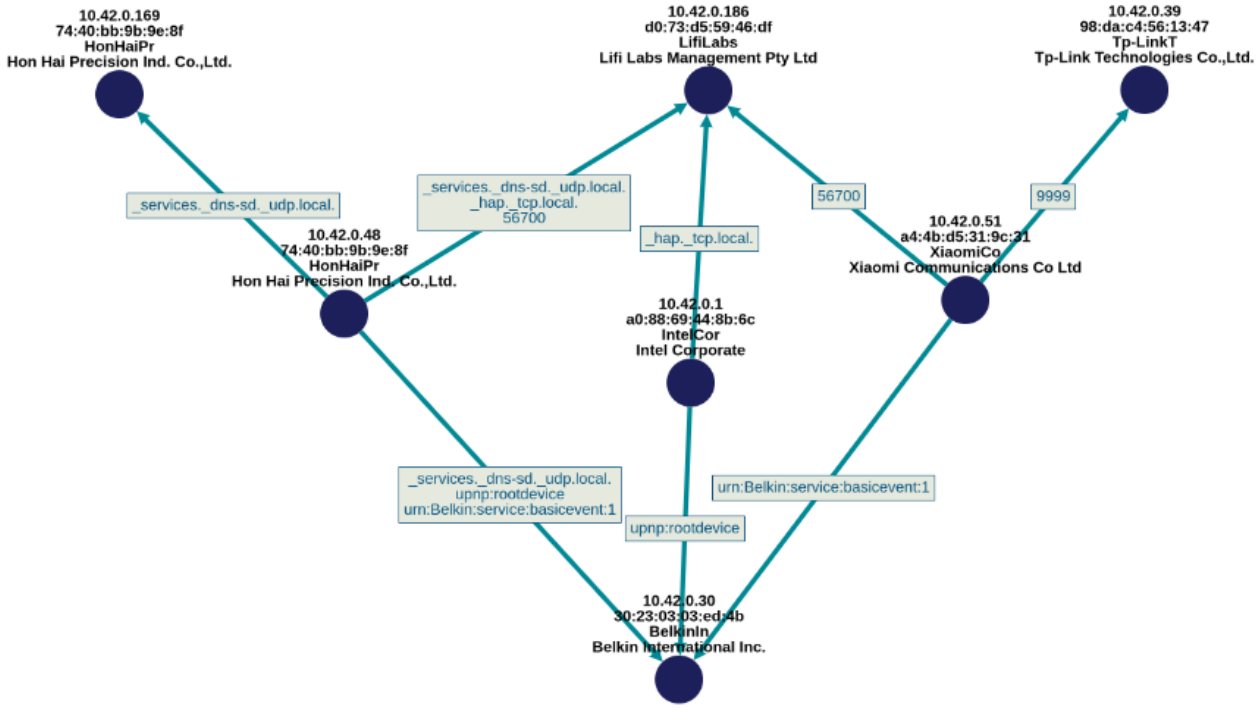

Figure 5.3 A discovery mapping generated doing day-to-day interaction with IoT devices

5.3.8 Alerts experiments

To evaluate the alerts generation, we used our custom tool to send discovery queries.

We sent root discovery queries and, regardless if they discovered something or not, our solution notified about it by sending information that included the source, discovery protocol used and the devices discovered (if any).

Moreover, we launched the Home Assistant which was installed in a VM in a Windows host to catch the root discovery queries it sends, and our solution also notified about the queries and the discovery made because of it.

To avoid spam, once the first notification was send, further queries sent by the same source were kept on track. After one hour, a new notification was sent with the number of times this source has sent the discovery query during the hour and the discoveries made.

To evaluate the behaviour's change notification, we sent predefined queries from our custom tool to establish the device's initial behaviour. Then, once the experimental five minutes threshold has been raised, we sent a new discovery query. Our solution detected such changes and sent the notification with information about the source, the discovery protocol and the

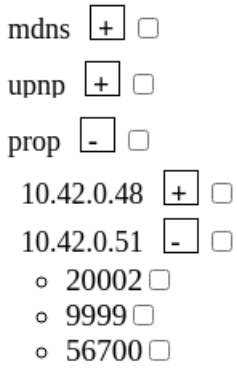

Figure 5.4 Discovery exposure presented in a structured list

new queries detected.

Through the time of this thesis, we observed that the Kasa app from TP-Link got updated, and a new query was sent. However, none notification from the companion app was sent about such new discovery performed by the app. The behaviour monitor we implemented is able to detect this type of changes and so, notify about it.

The behaviour's change is measured by getting a hash of the queries sent by a device through a full day and comparing it with the previous record, which is also a hash. If a change is detected, a notification is sent.

5.3.9 Security enforcement

By default, the discovery is denied and it is possible to allow discovery through the discovery mapping or the discovery exposure interfaces.

The home automation hub was launched and it could not discover anything. We observed the same with the companion apps and our custom tool. However, the queries were caught in the discovery exposure. The discovery exposure is the central point to allow or deny discovery queries. Discovery protocols were not denied as a whole element, instead the queries were the ones which were controlled. Such control was enabled by checking the boxes corresponding to the queries to allow, as Figure 5.5 shows. Thus, through the discovery exposure it is possible to allow a specific query or, if necessary, all the queries a device is sending.

We selected the necessary discovery queries needed by the home automation hub and the companion apps to discover the IoT devices. The home automation hub and the companion apps did not have problem to find the IoT devices. Moreover, there was no impact in the user experience since the deployment and control of devices in the apps or GUI was seamless,

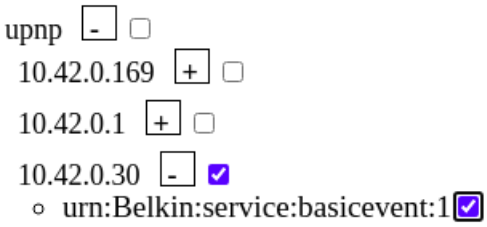

Figure 5.5 Discovery exposure to select discovery queries to being allowed

just as when the security enforcement was not up.

Once a discovery query is allowed, the discovery mapping can be drawn if the query discovers something. Through the discovery mapping it is also possible to going back to deny the discovery query by clicking on the corresponding edge, as Figure 5.6 shows.

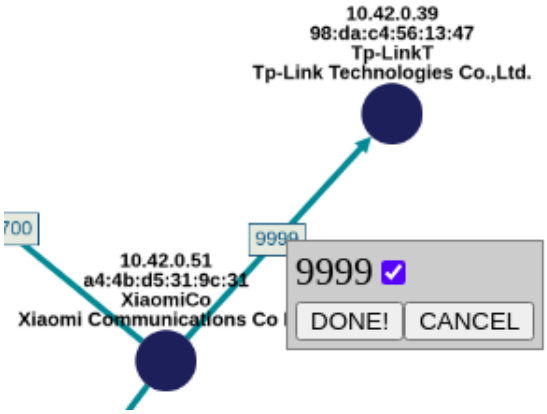

Figure 5.6 Allowing Tp-Link device to be discovered from the discovery mapping

As soon as a query was put back as denied, the home automation hub and the companion apps thought that the devices were not anymore on the network and showed them as unavailable.

Thus, our solution provided a discovery control, denying to discover IoT devices. Since the companion apps and the home automation hub allow an automatic control once they discover IoT devices, denying the discovery also denies the control through the companion app or the home automation hub. This is similar for a malicious IoT device which wants to discover. Since discovery queries need to be allowed, a passive reconnaissance would not be effective, neither if they send root discovery queries like metasploit module does, nor sending many specific queries.

Besides of providing transparency and be aware of the use of discovery protocols, through the security enforcement we are stopping some of the attacks being used in traditional IT

environments. For instance, we are denying any kind of passive reconnaissance being used by an attacker. Due to attackers will gather only information about the devices, they are allowed to discover, if any, instead of having freedom of discover everything on the network.

Moreover, devices are protected to being used for a DoS since normally this technique requires to spoof the source IP when sending the discovery query, so the device discovered will answer to the spoofed IP. However, there is a restriction about which devices, IPs, can discover what.

It has been shown that an attacker can perform a buffer overflow using discovery protocols. This is due to a lack of input validation by devices when receiving the discovery query. Since our proposal do not modify discovery protocols, we cannot stop this attacks.

5.4 Policy management

Cisco launched the Manufacturer User Description (MUD) [72] as a representation of IoT devices' network behaviour. This representation allows manufacturers to provide visibility of IoT behaviour in terms of networking. That is, it can be seen as an Access List Control (ACL) since it represents the connections the device is expected to perform. This helps identify possible attacks when an IoT device's behaviour is out of this MUD policy. IoT vendors are crucial part in MUD, due to they need to settle the expected network behaviour their devices need.

In general, MUD relies on three principal entities: the MUD file, which is supposed to be created by the IoT vendors, a URL where this file can be retrieved, and a network mechanism to get such file [73].

A MUD policy contains mainly two root sections: the MUD container and the ACLs container. The MUD container mainly has useful information for retrieving the MUD file and data to validate the file itself. The ACLs container has information about the from-device and to-device connections allowed. The ACLs container can be seen as a policy of a packet firewall, since it defines TCP and UDP connections and their respective components such as IP or DNS, and ports.

Hazma *et al.* [74] defined behavioural profiles for IoT devices and produced MUD policies. They provided MUD profiles for 28 IoT devices. These profiles include the communication components that discovery protocols require as expected since IoT devices rely on them. However, it is not possible to define the discovery query used by a device to determine if this connection should be allowed or not, either from or to [75].

MUD policies do not consider the payload which in discovery protocols is translated as discovery query. We propose to include discovery queries in MUD policies. In addition to

packet control, this will allow to control in the discovery of a device.

The discovery query can be inserted as an element of the section defining the discovery protocol communication in the MUD policy. The discovery query should be settled with the query itself and a justification on the use of it. This will provide visibility to the network administrator in the queries used by device and their purpose. In listing 5.1, a MUD policy is presented which contains a UPnP communication generated by a Belkin Camera [75]. The line 9 defines the UPnP IP, and line 14 the UPnP port. Line 19 says that this communication is accepted. However, if the Belkin Camera sends a root discovery it would not be blocked or seen as a suspected action.

```
1  {
2    "name" : "from-ipv4-belkincamera-4",
3    "matches" : {
4      "ietf-mud:mud" : {
5        "local-networks" : [ null ]
6      },
7      "ipv4" : {
8        "protocol" : 17,
9        "destination-ipv4-network" : "239.255.255.250/32"
10     },
11     "udp" : {
12       "destination-port" : {
13         "operator" : "eq",
14         "port" : 1900
15       }
16     }
17   },
18   "actions" : {
19     "forwarding" : "accept"
20   }
21 }
```

Listing 5.1 MUD policy for UPnP communication

Since MUD policies are JSON based, a variable *discovery-queries* can be added. This variable can define the discovery queries the device needs for its proper function and their justification. Listing B.1 represents the UPnP communication requested by a Belkin Camera but with our proposal of adding discovery queries.

In line 15 we added our *discovery-queries* variable which contains description and query fields. Line 17 gives information regarding the purpose of the use of the queries. This will provide

visibility to the network administrator about the purposes of such discoveries. Moreover, as a device might need more than one query for a proper performance, an array is used for such purposes as line 18 presents.

```
{
  "name" : "from-ipv4-belkincamera-4",
  "matches" : {
    "ietf-mud:mud" : {
      "local-networks" : [ null ]
    },
    "ipv4" : {
      "protocol" : 17,
      "destination-ipv4-network" : "239.255.255.250/32"
    },
    "udp" : {
      "destination-port" : {
        "operator" : "eq",
        "port" : 1900,
        "   discovery-queries   " : [
            {
                " description " : "In order to provide a good experience to
                    the users, Belkin devices needs to be discovered among
                    them for proper function and integrity" ,
                "query" : ["urn:Belkin: service:basicevent:1"]
            }
        ]
      }
    }
  },
  "actions" : {
    "forwarding" : "accept"
  }
}
```

Listing 5.2 Our proposal for inclusion of UPnP communication to MUD policies

MUD policies have been applied in SDN environments as Feraudo *et al.* [73] and Hamza *et al.* [76] have shown. In fact there are already applications proposed by well know security organizations [77]. Thus, MUD policies can be used in our security recommendations discussed before.

5.5 Summary

In this chapter, we gave recommendations to remedy the security concerns that exist while using discovery protocols in IoT networks. We presented proof of concept that provided

transparency and control over discovery protocols, but without losing their benefits.

CHAPTER 6 CONCLUSION

This chapter gives a recapitulation of the research objective which assess the impact in terms of security concerns of discovery protocols being used in consumer IoT devices. The chapter also addresses a discussion on the methodology used and its results to approach this objective.

The chapter contains a summary of the work and research contributions, the limitations of the work and the research approach, future work and recommendations.

6.1 Summary of Works

O.1 Identify the use of discovery protocols in IoT devices and assess devices' behaviour

- O.1.1 Perform a static and dynamic analysis of an open source home automation hub
- O.1.2 Perform an empirical analysis on both a small local IoT device network as well as a large public dataset

O.2 Identify the security concerns on using discovery protocols by IoT devices

O.3 Provide security recommendations and proof-of-concept tools to enhance security while keeping discovery protocols usability benefits

O.1 We could identify the use of discovery protocols in IoT devices and assess devices' behaviour through a static and dynamic analysis of an open source home automation hub and an empirical analysis on both a small local IoT device network as well as a large public dataset.

O.1.1 Our first research contribution is the analysis of a home automation hub. We performed both a static and a dynamic analysis of how home automation hubs can discover IoT devices. Since home automation hubs can function as the main point to manage IoT devices, we decided to perform a code review and a network behaviour analysis.

We took a well-know open source home automation and reviewed its code. This let us to get a first glimpse of the discovery process in IoT networks. Then, we set the home automation hub and the IoT devices to examine the interaction between them.

We gained understanding on how discovery is performed by a home automation hub which capabilities are to manage IoT devices of different category and vendors.

Moreover, we could identify the types of protocols used for discovery, like standard and vendor-specific. We identify that there are two stages in the management of IoT devices: the discovery stage (which is the subject of this thesis), and the control stage, which comes always after the discovery stage.

O.1.2 We built a small testbed to connect IoT devices and their companion mobile apps. We selected IoT devices from well-know IoT vendors. The companion apps were the ones recommended by the IoT vendors respectively. Then, we set seven topologies to visualize and analyze the behaviour of IoT devices and companion apps, as well as the interaction among IoT devices and between IoT devices and companion apps. Furthermore, we launched some custom discovery queries and spoofed alleged services to inspect the behaviour of IoT devices.

Moreover, we analyzed a dataset containing network traffic recorded during 13 full days that included more than 50 devices (among IoT devices, home automation hubs and general purpose devices) and around 100 GB of data. We gathered information about discovery use and behaviour of a wider range of devices than our small testbed. However, we could not control the devices interaction, and we had not any information about neither the interaction nor then companion apps used, if any.

We could gather relevant information about the discovery protocols used by IoT devices and home automation hubs from different brands. We gained an understanding of how the discovery process in an IoT networks works and the protocols involved. Moreover, we could identify similar situations between what we observed in our small testbed and the information included in the dataset, as well as discovery queries used to discover devices.

O.2 We could identify security concerns of using discovery protocols by IoT devices since the methodology used to gather relevant information about the discovery process in IoT networks.

Discovery protocols facilitate onboarding and service discovery in IoT networks. However, their unrestricted use assumes that all devices on the network are trusted, which is not often the case. There is a misuse of discovery protocols in IoT networks, and we found that many of the issues related to discovery protocols in regular computer networks also exist on IoT networks.

We identified a violation of the principle of Least Privilege, since discoveries were performed by specific-purpose IoT devices. It exposes an overuse of root discovery (which stands for discovering everything on the network). Moreover, we identified cross-vendor

discoveries performed by both companion apps and IoT devices. Nevertheless, we observed several instances of service misconfiguration, such as some companion apps using discovery protocols not necessary to control their IoT devices, or IoT devices answering to discovery protocols not used by their companion apps. We also noticied that the user's privacy is also impacted, as IoT devices' names are generally related to the devices' physical place in houses, and we observed that this information is shared through discovery queries. Furthermore, we observed that there is a lack of transparency in the device's behaviour because there is no notification about who is discovering what, or who is abusing of the discovery itself.

O.3 We could provide security recommendations and proof-of-concept tools to enhance security while keeping discovery protocols usability benefits.

In light of our findings, we proposed some readily-deployable security recommendations for IoT network administrators to reduce the impact of discovery protocol abuse by IoT vendors, untrusted users, and attackers. Our results highlighted the need for better tools to protect users from discovery protocols on legacy IoT devices, as well as the need for new protocols to protect future devices.

To provide visibility and transparency on the use of discovery protocols, we proposed a discovery mapping based on a directed graph that allows to visualize which devices are discovering others. We also introduced the notion of discovery exposure, that stands for the discovery capability that a device has. That is, a device sending a discovery query cannot always get an answer back if any device on the network is not able to understand such query, however the sender is capable to discover future devices arriving to the network capable to understand such discovery query. We also proposed a notification functionality to alert users about undesired discoveries and root discoveries.

Besides providing visibility and transparency with the previous mentioned recommendations, we proposed a way to control the discovery in IoT networks. Using Software Defined Networking (SDN) we could control the discovery process in the IoT network while keeping the discovery protocols' usability benefits. This enhanced the security in the IoT network. Finally, we gave examples of how it is possible to include discovery protocols in MUD policies, the IoT network behaviour policies proposed by Cisco. Such policies will determine and explain why a discovery is necessary for an specific IoT device.

6.2 Limitations

We identified two type of discovery protocols used in IoT devices: standard and vendor-specific. Vendor-specific discovery protocols are closed and it is necessary to perform a reverse engineer in order to know what the discovery queries means. Even though our proposal considered the technical aspects of how vendor-specific discovery works, it could not identify or translate the discovery query (payload) of them.

Companion apps are usually installed in smartphones and they run together with system apps and other necessary apps, as per user likes and necessities. Each of these app installed in the smartphones can generate network traffic. If a companion app generates network traffic, it is not possible to identify that traffic from others apps but the smartphone itself. Hence, in normal conditions (without the control we had in our testbed topologies) it is not possible to match discovery queries with companion apps. Even though our proposal shows a node sending discovery queries, it is not capable to label or identify the companion app that sent such discovery query.

The recommended MUD policy management considers discovery protocols as part of IoT network behaviour. However, if this policy management is put in place, it is not possible to detect when an IoT device is released using discovery protocols for marketing purposes, or if an update adds this functionality to the IoT device.

6.3 Future Research

It is possible to consolidate further the security concerns when using discovery protocols if a dataset of companion apps for IoT devices is analyzed. We could identify IoT devices and home automation hubs sending root discovery queries, but we did not identify if any companion apps used them. However, companion apps are not excluded to use root discovery or a set of discovery queries. Since companion apps are essentials when using IoT devices, a similar proposal like MUD policy that states the companion apps network behaviour can be suggested.

A good practice can be to observe the behaviour of IoT devices when sending crafted discovery queries. For instance, create a list of technically correct and used discovery queries, send the queries and observe if IoT devices answer to more discovery queries than the expected ones. In addition, it will be worth to observe what happen when incorrect discovery queries are sent, like a type of fuzzing. This might crash IoT devices and hence impact the physical environment.

The presented proof of concept to control discovery in the network did not include a MUD policy management. Such functionality would help better control the devices in the network and absolve the network administrator total responsibility in the control of discovery protocols and queries allowed or denied by devices.

REFERENCES

[1] CISCO, "Cisco annual internet report (2018–2023)," March 2020. [Online]. Available: https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.pdf

[2] "Digital impact playbook," https://www.cisco.com/c/dam/assets/csr/pdf/Digital-Impact-Playbook.pdf, online.

[3] A. K. Simpson, F. Roesner, and T. Kohno, "Securing vulnerable home IoT devices with an in-hub security manager," in *2017 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*. IEEE, 2017, pp. 551–556.

[4] T. Aura *et al.*, "Chattering laptops," in *International Symposium on Privacy Enhancing Technologies Symposium*. Springer, 2008, pp. 167–186.

[5] B. Könings *et al.*, "Device names in the wild: Investigating privacy risks of zero configuration networking," in *2013 IEEE 14th International Conference on Mobile Data Management*, vol. 2. IEEE, 2013, pp. 51–56.

[6] D. Kaiser and M. Waldvogel, "Efficient privacy preserving multicast DNS service discovery," in *2014 IEEE International Conference on High Performance Computing and Communications, 2014 IEEE 6th International Symposium on Cyberspace Safety and Security, 2014 IEEE 11th International Conference on Embedded Software and Systems (HPCC, CSS, ICESS)*. IEEE, 2014, pp. 1229–1236.

[7] A. Atlasis, "An attack in depth analysis of multicast DNS and DNS Service Discovery," Hack In The Box Security Conference, 2017.

[8] "Name (mDNS) poisoning attacks inside the LAN," https://www.gnucitizen.org/blog/name-mdns-poisoning-attacks-inside-the-lan/, online.

[9] X. Bai *et al.*, "Staying secure and unprepared: Understanding and mitigating the security risks of apple zeroconf," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 655–674.

[10] A. Al Hasib and M. Mottalib, "Vulnerability analysis and protection schemes of universal plug and play protocol," in *2010 13th IEEE International Conference on Computational Science and Engineering*. IEEE, 2010, pp. 222–228.

[11] A. Haque, "UPnP networking: Architecture and security issues," in *Proc. TKK Seminar on Network Security*. Citeseer, 2007.

[12] D. Kaiser and M. Waldvogel, "Adding privacy to multicast DNS service discovery," in *2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications*. IEEE, 2014, pp. 809–816.

[13] J. Pang *et al.*, "Tryst: The case for confidential service discovery." in *HotNets*, vol. 2, 2007, p. 1.

[14] D. J. Wu *et al.*, "Privacy, discovery, and authentication for the internet of things," in *European Symposium on Research in Computer Security*. Springer, 2016, pp. 301–319.

[15] J. Liebeherr and M. E. Zarki, *Mastering Networks: An Internet Lab Manual*. Addison-Wesley Longman Publishing Co., Inc., 2003.

[16] C. Systems, "IP Application Services Configuration Guide, Cisco IOS Release 15.1 M&T," 2011.

[17] J. Loveless, R. Blair, and A. Durai, *IP Multicast, Volume I: Cisco IP Multicast Networking*. Cisco press, 2016.

[18] "IPv4 multicast address space registry," https://www.iana.org/assignments/multicast-addresses/multicast-addresses.xhtml, online.

[19] "IPv4 multicast address space registry - 224.0.0/24," https://www.iana.org/assignments/multicast-addresses/multicast-addresses.xhtml#multicast-addresses-3, online.

[20] "IPv4 multicast address space registry - scoped multicast ranges," https://www.iana.org/assignments/multicast-addresses/multicast-addresses.xhtml#multicast-addresses-12, online.

[21] "Home assistant discovery source codes," https://github.com/home-assistant/netdisco, online.

[22] "Home assistant discovery," https://www.home-assistant.io/components/discovery/, online.

[23] D. Andrew *et al.*, "UPnP device architecture 2.0," UPnP Forum, Tech. Rep., February 2015. [Online]. Available: http://upnp.org/specs/arch/UPnP-arch-DeviceArchitecture-v2.0.pdf

[24] C. Cabrera, A. Palade, and S. Clarke, "An evaluation of service discovery protocols in the internet of things," in *Proceedings of the Symposium on Applied Computing*. ACM, 2017, pp. 469–476.

[25] S. Cheshire, M. Krochmal, and I. Apple, "DNS-Based Service Discovery," Internet Requests for Comments, RFC Editor, RFC 6763, February 2013. [Online]. Available: https://tools.ietf.org/html/rfc6763

[26] D. H. Steinberg and S. Cheshire, *Zero Configuration Networking: The Definitive Guide*. " O'Reilly Media, Inc.", 2005.

[27] S. Cheshire, M. Krochmal, and I. Apple, "Multicast DNS," Internet Requests for Comments, RFC Editor, RFC 6762, February 2013. [Online]. Available: https://www.rfc-editor.org/rfc/rfc6762.txt

[28] "Using the homekit accessory protocol specification," https://developer.apple.com/homekit/faq/, online.

[29] "Zero configuration networking (zeroconf)," http://www.zeroconf.org/, online.

[30] A. I. Ashley Butterworth, Matthew Xavier Mora, "Device discovery with mDNS and DNS-SD," Apple, Tech. Rep., October 2009. [Online]. Available: http://grouper.ieee.org/groups/1722/contributions/2009/Bonjour%20Device%20Discovery.pdf

[31] "Avahi," https://www.avahi.org/, online.

[32] "Bonjour," https://developer.apple.com/bonjour/, online.

[33] "Lightweight UPnP client library for python," https://github.com/5kyc0d3r/upnpy, online.

[34] M. Antonakakis *et al.*, "Understanding the mirai botnet," in *26th USENIX security symposium (USENIX Security 17)*, 2017, pp. 1093–1110.

[35] D. Kumar *et al.*, "All things considered: an analysis of IoT devices on home networks," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 1169–1185.

[36] R. Williams *et al.*, "Identifying vulnerabilities of consumer internet of things (IoT) devices: A scalable approach," in *2017 IEEE International Conference on Intelligence and Security Informatics (ISI)*. IEEE, 2017, pp. 179–181.

[37] H. Liu, T. Spink, and P. Patras, "Uncovering security vulnerabilities in the Belkin WeMo home automation ecosystem," in *2019 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*. IEEE, 2019, pp. 894–899.

[38] Z. Ling *et al.*, "Security vulnerabilities of internet of things: A case study of the smart plug system," *IEEE Internet of Things Journal*, vol. 4, no. 6, pp. 1899–1909, 2017.

[39] G. Chu, N. Apthorpe, and N. Feamster, "Security and privacy analyses of internet of things children's toys," *IEEE Internet of Things Journal*, vol. 6, no. 1, pp. 978–985, 2018.

[40] S. Shasha *et al.*, "Playing with danger: A taxonomy and evaluation of threats to smart toys," *IEEE Internet of Things Journal*, vol. 6, no. 2, pp. 2986–3002, 2019.

[41] O. Alrawi *et al.*, "SoK: Security evaluation of home-based IoT deployments," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 1362–1380.

[42] X. Wang *et al.*, "Looking from the mirror: evaluating IoT device security through mobile companion apps," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 1151–1167.

[43] K. Scarfone *et al.*, "Technical guide to information security testing and assessment," *NIST Special Publication*, vol. 800, no. 115, pp. 2–25, 2008.

[44] P. Krongbaramee and Y. Somchit, "Implementation of SDN stateful firewall on data plane using open vswitch," in *2018 15th International Joint Conference on Computer Science and Software Engineering (JCSSE)*. IEEE, 2018, pp. 1–5.

[45] W. M. Othman *et al.*, "Implementation and performance analysis of SDN firewall on POX controller," in *2017 IEEE 9th International Conference on Communication Software and Networks (ICCSN)*. IEEE, 2017, pp. 1461–1466.

[46] C. Vandana, "Security improvement in IoT based on software defined networking (SDN)," *International Journal of Science, Engineering and Technology Research (IJSETR)*, vol. 5, no. 1, pp. 2327–4662, 2016.

[47] O. Flauzac *et al.*, "SDN based architecture for IoT and improvement of the security," in *2015 IEEE 29th International Conference on Advanced Information Networking and Applications Workshops*. IEEE, 2015, pp. 688–693.

[48] T. Yu *et al.*, "Handling a trillion (unfixable) flaws on a billion devices: Rethinking network security for the internet of things," in *Proceedings of the 14th ACM Workshop on Hot Topics in Networks*. ACM, 2015, p. 5.

[49] S. Goutam, W. Enck, and B. Reaves, "Hestia: simple least privilege network policies for smart homes," in *Proceedings of the 12th Conference on Security and Privacy in Wireless and Mobile Networks*. ACM, 2019, pp. 215–220.

[50] "Home assistant," https://www.home-assistant.io/, online.

[51] "Home assistant github," https://github.com/home-assistant, online.

[52] "Wireshark," https://www.wireshark.org/, online.

[53] "Home assistant discovery," https://analytics.home-assistant.io/#integrations, online.

[54] L. Yu *et al.*, "You are what you broadcast: Identification of mobile and IoT devices from (public) wifi," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 55–72.

[55] "pyhs100," https://github.com/GadgetReactor/pyHS100, online.

[56] "python-kasa," https://github.com/python-kasa/python-kasa/blob/master/kasa/protocol.py, online.

[57] "Arduino lifx bulb," https://github.com/kayno/arduinolifx, online.

[58] "Tp-link wifi smartplug client and wireshark dissector," https://github.com/softScheck/tplink-smartplug, online.

[59] "Network mapper," https://nmap.org/, online.

[60] Z. Qin *et al.*, "A software defined networking architecture for the internet of things," in *2014 IEEE Network Operations and Management Symposium (NOMS)*. IEEE, 2014, pp. 1–9.

[61] R. C. Alves and C. B. Margi, "Discovery protocols for SDN-based wireless sensor networks with unidirectional links," *XXXV SBrT, São Pedro, Brazil*, 2017.

[62] T. OConnor *et al.*, "Homesnitch: behavior transparency and control for smart home IoT devices," in *Proceedings of the 12th Conference on Security and Privacy in Wireless and Mobile Networks*. ACM, 2019, pp. 128–138.

[63] "Opendaylight," https://www.opendaylight.org/, online.

[64] "Ryu," https://github.com/faucetsdn/ryu, online.

[65] "Openflow," https://opennetworking.org/sdn-resources/customer-case-studies/openflow/, online.

[66] "Mikrotik," https://mikrotik.com/, online.

[67] "Wireshark manufacturer database," https://gitlab.com/wireshark/wireshark/-/raw/master/manuf, online.

[68] D. Whyte, P. van Oorschot, and E. Kranakis, "Exposure maps: Removing reliance on attribution during scan detection," in *USENIX Workshop on Hot Topics in Security (HotSec)*, 2006.

[69] W. Xia *et al.*, "A survey on software-defined networking," *IEEE Communications Surveys & Tutorials*, vol. 17, no. 1, pp. 27–51, 2014.

[70] "Pox," https://noxrepo.github.io/pox-doc/html/, online.

[71] "Floodlight," https://github.com/floodlight/floodlight, online.

[72] E. Lear, R. Droms, and D. Romascanu, "Manufacturer usage description specification," Internet Requests for Comments, RFC Editor, RFC 8520, March 2019. [Online]. Available: https://tools.ietf.org/html/rfc8520

[73] A. Feraudo *et al.*, "SoK: Beyond IoT MUD deployments–challenges and future directions," *arXiv preprint arXiv:2004.08003*, 2020.

[74] A. Hamza *et al.*, "Clear as MUD: Generating, validating and applying IoT behavioral profiles," in *Proceedings of the 2018 Workshop on IoT Security and Privacy*, 2018, pp. 8–14.

[75] "Belkin camera MUD policy," https://iotanalytics.unsw.edu.au/mud/belkincameraMud.json, online.

[76] A. Hamza, H. H. Gharakheili, and V. Sivaraman, "Combining MUD policies with SDN for IoT intrusion detection," in *Proceedings of the 2018 Workshop on IoT Security and Privacy*, 2018, pp. 1–7.

[77] M. N.-B. A. U. Manufacturer, "Securing small-business and home internet of things (IoT) devices," *NIST SPECIAL PUBLICATION*, p. 15A, 1800.

APPENDIX A SOURCE CODE TO SWEEP YOURTHINGS DATASET

Python code used to sweep the Yourthings [41] dataset to gather discovery information of the devices in the dataset.

```
from scapy.all import *
from netaddr import *
import os
import csv
import time
import re

filename = "resultAnalysis.csv"

data_file = open(filename, 'w')
csv_writer = csv.writer(data_file)
csv_writer.writerow(["Epoch","Time", "IP src", "device", "IP dst", "device",
    "src port", "dst port", "protocol", "disc protocol", "query", "Q&A", "
    payload size", "payload", "file"])

with open ('device_mapping.csv', mode='r') as file:
        csvFile = csv.reader(file)
        devicesName = dict((rows[1],rows[0]) for rows in csvFile)

proprietaryPorts = {}

def analyzingProprietaryQ(pkt):
        proprietaryPorts[pkt[IP].src] = pkt[UDP].dport
        return "proprietary", str(pkt[UDP].dport), "Q", pkt[UDP].payload

def analyzingProprietaryA(pkt):
        if  pkt[UDP].dport > 1023:
                PORT = str(pkt[UDP].sport)
                if PORT in proprietaryPorts: #If the port used by the sender
                    (IoT device) is in the db
                        #Add the mapping, the IP src is discoverable by the
                            IP dst
                        return "proprietary", str(pkt[UDP].sport), "A", pkt[
                            UDP].payload
        return "", "", "", ""

def analyzingUPnPpacketQ(pkt):
```

```
33          data = str(pkt[UDP].payload)
34          try:
35                  if "M-SEARCH" in data and "ssdp:discover" in data and "ST:"
                        in data:
36                          try:
37                                  ST = re.findall("\r\nST: (.+)\r\n", data)[0]
                                        #Since findall returns a list, which
                                        will be of size one, the only element is
                                        extracted to have a string instead of a
                                        list
38                          except Exception as e:
39                                  return "", "", "", ""
40                          return "upnp", ST, "Q", pkt[UDP].payload
41                  return "", "", "", ""
42          except Exception as e:
43                  return "", "", "", ""
44
45  def analyzingUPnPpacketA(pkt):
46          data = str(pkt[UDP].payload)
47          try:
48                  if "ST:" in data and pkt[UDP].dport > 1023 and "M-SEARCH"
                        not in data and "ssdp:discover" not in data:
49                          try:
50                                  ST = re.findall("\r\nST: (.+)\r\n", data)[0]
                                        #Since findall returns a list, which
                                        will be of size one, the only element is
                                        extracted to have a string instead of a
                                        list
51                          except Exception as e:
52                                  return "", "", "", ""
53                          return "upnp", ST, "A", pkt[UDP].payload
54                  return "", "", "", ""
55          except Exception as e:
56                  return "", "", "", ""
57
58  def analyzingmDNSpacket(pkt, ip_version):
59          try:
60                  if (valid_ipv4(pkt[ip_version].src)):
61                          #On this if statement, it's analized the Map
62                          if DNSRR in pkt and not DNSQR in pkt: #If the mDNS
                                packet has only answer(s)
63                                  for x in range(pkt[DNS].ancount):
64                                          if pkt[DNSRR][x].type == 12: #PTR
65                                                  return "mdns", (pkt[DNSRR][x
```

```
                                                        ].rrname).decode('UTF-8
                                                        '), "A", pkt[UDP].
                                                        payload
66
67                          #On this if statement, it's analized the
                                whatDiscovers
68                          elif DNSQR in pkt and not DNSRR in pkt: #If the mDNS
                                 packet has only question(s)
69                                  for x in range(pkt[DNS].qdcount): #Looping
                                        in the number of questions
70                                      if pkt[DNSQR][x].qtype == 12: #PTR
71                                              return "mdns", (pkt[DNSQR][x
                                                ].qname).decode('UTF-8')
                                                , "Q", pkt[UDP].payload
72                  else:
73                          return "", "", "", ""
74                  return "", "", "", ""
75          except Exception as e:
76                  print("mdns: " + str(e))
77                  return "", "", "", ""
78
79  def received_pkt(pkt):
80          #pkt.show()
81          #input("Continue:")
82          global totalPackets
83
84          disc_protocol = ""
85          query = ""
86          QA = ""
87          payload = ""
88
89          try:
90                  if IP in pkt:
91                          if UDP in pkt:
92                                  #mDNS
93                                  if pkt[IP].dst == "224.0.0.251" and pkt[UDP]
                                        .dport == 5353:
94                                          disc_protocol, query, QA, payload =
                                                analyzingmDNSpacket(pkt, "IP")
95                                  #UPnP
96                                  elif pkt[IP].dst == "239.255.255.250" and
                                        pkt[UDP].dport == 1900:
97                                          disc_protocol, query, QA, payload =
                                                analyzingUPnPpacketQ(pkt)
```

```
98                                  #Proprietary
99                                  elif pkt[IP].dst == "255.255.255.255":
100                                         disc_protocol, query, QA, payload =
                                                analyzingProprietaryQ(pkt)
101                                 else:
102                                         disc_protocol, query, QA, payload =
                                                analyzingUPnPpacketA(pkt)
103
104                                 if disc_protocol == "" and query == "" and
                                        QA == "" and payload == "":
105                                         disc_protocol, query, QA, payload =
                                                analyzingProprietaryA(pkt)
106                         if disc_protocol != "":
107                                 if IP in pkt:
108                                         ip_src = pkt[IP].src
109                                         ip_dst = pkt[IP].dst
110                                 else:
111                                         ip_src = ""
112                                         ip_dst = ""
113
114                                 if UDP in pkt:
115                                         protocol = "UDP"
116                                         payLoad = pkt[UDP].payload
117                                 elif TCP in pkt:
118                                         protocol = "TCP"
119                                         payLoad = pkt[TCP].payload
120                                 else:
121                                         protocol = ""
122                                         payLoad = ""
123
124                                 local_time = time.localtime(pkt.time)
125                                 human_time = time.strftime('%m/%d/%y-%H:%M:%
                                        S', local_time)
126
127                                 csv_writer.writerow([pkt.time, human_time,
                                        ip_src, devicesName.get(ip_src, ""),
                                        ip_dst, devicesName.get(ip_dst, ""), pkt
                                        [protocol].sport, pkt[protocol].dport,
                                        protocol, disc_protocol, query, QA, len(
                                        payLoad), payload, file])
128                         totalPackets = totalPackets + 1
129                 else:
130                         totalPackets = totalPackets + 1
131                         #local_time = time.localtime(pkt.time)
```

```
132                         #human_time = time.strftime('%m/%d/%y-%H:%M:%S',
                                local_time)
133                         #csv_writer.writerow([pkt.time, human_time, "", "",
                                "", "", "", "", "", "", "", "", "", file])
134         except Exception as e:
135                 totalPackets = totalPackets + 1
136                 pass
137                 #local_time = time.localtime(pkt.time)
138                 #human_time = time.strftime('%m/%d/%y-%H:%M:%S', local_time)
139                 #csv_writer.writerow([pkt.time, human_time, "", "", "", "",
                        "", "", "", "", "", "", "", file])
140
141 counter = 0
142 totalPackets = 0
143 for root, dirs, files in os.walk('.', topdown=True):
144         for file in files:
145                 #if file == "yourThings.py" or file == "YourThings.p" or
                        file == "device_mapping.csv":
146                 #       continue
147                 pcapFile = root + '/' + file
148                 #if pcapFile.lower().endswith(('.pcap', '.pcapng')):
149                 print(str(counter) + "/" + str(len(files)) + " " + pcapFile
                        )
150                 try:
151                         #sniff(offline=pcapFile, filter="(dst host 224.0.0.2
                                51 and port 5353) or (ip6 dst host ff02::fb and
                                port 5353) or (dst host 239.255.255.250 and port
                                 1900) or (dst host 255.255.255.255 and port not
                                 67)", prn = received_pkt)
152                         sniff(offline=pcapFile, prn = received_pkt)
153                 except Exception as e:
154                         #print("Sniff: " + str(e))
155                         totalPackets = totalPackets + 1
156                         pass
157                 counter = counter + 1
158         print(totalPackets)
159
160 data_file.close()
```

Listing A.1 Python script to gather discovery information from a pcap

APPENDIX B SOURCE CODE TO NOTIFY ABOUT ROOT DISCOVERY QUERIES BEING SENT

Python code used to notify when a root discovery query is seen in the network. That is, if a device send a UPnP or mDNS root discovery query, the script will send an email about it.

```
from scapy.all import *
import os
import re
import argparse
import smtplib

def sendNotification(pkt, discProt):
        user = 'poly.iot.notification@gmail.com'
        password = 'polyiot2020'

        sent_from = "poly.iot.notification@gmail.com"
        to = [email]
        subject = 'A device is discovering as root'
        body = "The device with IP " + pkt[IP].src + " and MAC " + pkt[Ether
            ].src + " is discovering as root with " + discProt
        print(body)

        email_text = """\
From: %s
To: %s
Subject: %s

%s
""" % (sent_from, ", ".join(to), subject, body)

        try:
            server = smtplib.SMTP_SSL('smtp.gmail.com', 465)
            server.ehlo()
            server.login(user, password)
            server.sendmail(sent_from, to, email_text)
            server.close()

            print 'Email sent'
        except:
            print 'Something went wrong...'
        return
```

```
36
37 def analyzingUPnPpacket(pkt):
38         try:
39                 data = str(pkt[UDP].payload)
40                 if "M-SEARCH" in data and "ssdp:discover" in data:
41                         ST = re.findall("\r\nST: (.+)\r\n", data)[0] #Since
                               findall returns a list, which will be of size
                               one, the single element is extracted and became
                               it in a string
42                         if ST == "ssdp:all" or ST == "upnp:rootdevice":
43                                 return True
44         except:
45                 print ("Error in UPnP packet")
46                 pkt.show()
47                 return False
48         return False
49
50 def analyzingmDNSpacket(pkt):
51         try:
52                 if DNSQR in pkt: #If the mDNS packet has question(s)
53                         for x in range(pkt[DNS].qdcount): #Looping in the
                               number of questions
54                                 if pkt[DNSQR][x].qtype == 12: #PTR
55                                         if pkt[DNSQR][x].qname == "_services
                                               ._dns-sd._udp.local.":
56                                                 return True
57         except:
58                 print ("Error in mDNS packet")
59                 pkt.show()
60                 return False
61         return False
62
63 def received_pkt(pkt):
64         #pkt.show()
65         if IP in pkt:
66                 if UDP in pkt:
67                         #mDNS
68                         if pkt[IP].dst == "224.0.0.251" and pkt[UDP].dport
                               == 5353:
69                                 print("mDNS received")
70                                 if analyzingmDNSpacket(pkt):
71                                         sendNotification(pkt,"mDNS")
72                                 else:
73                                         return
```

```
74                          #UPnP
75                          elif pkt[IP].dst == "239.255.255.250" and pkt[UDP].
                                dport == 1900:
76                                  if analyzingUPnPpacket(pkt):
77                                          sendNotification(pkt,"UPnP")
78                                  else:
79                                          return
80          else:
81                  return
82
83  parser = argparse.ArgumentParser( description ='This tool notify via email if
        a device on the network using mDNS or UPnP discovery protocols is
        discovering with a \'root\' query .')
84  requiredArgument = parser.add_argument_group('required arguments')
85  requiredArgument.add_argument("-i", "--interface", required=True, type=str,
        help="The interface where the tool will sniff the network.")
86  requiredArgument.add_argument("-e", "--email", required=True, type=str, help
        ="The email where the notifications will be sent to.")
87  args = parser.parse_args()
88  i = args.interface
89  email = args.email
90
91  sniff(iface=i, filter="(dst host 224.0.0.251 and port 5353) or (dst host 239
        .255.255.250 and port 1900)", prn = received_pkt)
92
93  """for root, dirs, files in os.walk('.', topdown=True):
94          if root == ".":
95                  continue
96          for file in files:
97                  pcapFile = root + '/' + file
98                  #if pcapFile.lower().endswith(('.pcap', '.pcapng')):
99                  #print (pcapFile)
100                 try:
101                         sniff(offline=pcapFile, filter="(dst host 224.0.0.25
                                1 and port 5353) or (dst host 239.255.255.250
                                and port 1900)", prn = received_pkt)
102                 except:
103                         print ("Error in file " + pcapFile)"""
```

Listing B.1 Python script to gather discovery information from a pcap