BRNO UNIVERSITY OF TECHNOLOGY FACULTY OF INFORMATION TECHNOLOGY

Excel@FIT 2021
http://excel.fit.vutbr.cz

Gathering knowledge about devices in local networks by analyzing service discovery protocols

Ondřej Sedláček*

Abstract
Network monitoring plays a big part in network management, and considering the growing percentage of encrypted traffic, we find ourselves looking for new useful sources of data. This project practice focuses on the utility of service discovery protocols in this regard. It aims to find whether service discovery data can be extracted and used for network mapping, both alone and in combination with other measurements. We aim to collect as much relevant data as possible from a couple of selected service discovery protocols and to assign basic labels to devices. The collection is done using the NEMEA module Ipfixprobe and plugins we add to support our selected protocols. This data is then processed using an aggregating python module named SDP Analyzer. The result of our effort allows us to see what kinds of services devices query for and what kinds they advertise. We can also extract the device hostname, operating system and sometimes even get information about the specific device model. With some additional monitoring of the network, this information can be used to classify most devices in your network. This paper goes through the protocols used and details the interesting data that can be extracted. It also shows the aggregation and basic analysis of the data and it might bring use to anyone curious about what is happening on their local network.

Keywords: SSDP — DNS-SD — NetBIOS — Network monitoring — Service discovery

Supplementary Material: *N/A*

*xsedla1o@stud.fit.vutbr.cz, *Faculty of Information Technology, Brno University of Technology*

1. Introduction

Passive network monitoring improves network visibility [1], which is useful when measuring the existing traffic, troubleshooting problems, or mapping what kinds of devices are connected through our network, e.g. for reference in case of anomalies. Such is the motivation behind the ADiCT[1] project, created precisely with the goal of monitoring and classifying network actors and making further correlations and analysis over that data. The ADiCT project needs many different data sources and this project practice focuses on finding such data in service discovery protocols.

We develop a solution for the extraction of useful data from selected service discovery protocols. We choose to focus on Simple Service Discovery Protocol (SSDP) and DNS-based Service Discovery (DNS-SD), later followed by NetBIOS Name Service (NBNS). Useful in the context of network monitoring means usable to classify a device, to know what kinds of communications we should be expecting from it. We also try to extract some additional data about the device's OS or identifiers such as hostname whenever possible. We do not attempt to decipher the configuration of the device nor to create a map of which devices communicated with each other yet, even if that may be the goal of future modules.

1.1 Existing solutions

When exploring the existing solutions in the network monitoring space, we find a wide range of scale. Dis-

[1] Asset Discovery, Classification and Tagging

cerning whether a proprietary solution utilizes SD protocol data is complicated, as most of the proprietary solutions we find center their presentation around the end product, not the underlying mechanisms.

As an example of a commercial solution that extracts data from SD protocols, we selected Rumble Network Discovery [2], a network scanning tool using a larger number of protocols for its asset identification, including mDNS, SSDP, and NetBIOS.

Once we narrow our scope to the actual data collection, which is much more comparable with our work, we find that many simple open-source command-line utilities exist but actively engage with the network, which classifies them as active monitoring. These utilities also focus on the services available, providing no summary on MAC or IP address basis.

Avahi-browse [3] is a command-line utility that is part of the Avahi package. It enables us to enumerate all the services advertised through DNS-SD on our network. It does so by periodically sending a special "find all services" query and printing the responses.

The Metasploit Framework console [4] includes many tools that allow for scanning both the DNS-SD and SSDP space. Once again, the basis is similar, the tools send a query requesting all available services and display any replies.

When looking at the problem at hand from a slightly different perspective, another open-source solution presents itself. Wireshark [5] is one of the world's most widely-used network protocol analyzers. It enables the live capture and offline analysis of network traffic, inspection of hundreds of protocols including our select SDP, and does match our passive monitoring criteria. Wireshark does not have ambitions of being an SDP data extraction centered tool and does not support specific record aggregation. Despite that, it is still worth mentioning, as we use the TShark utility, which is at Wireshark's core, as the first step in our initial prototype.

1.2 Our solution using NEMEA framework

For our solution, we choose to use the NEMEA [6] framework, leveraging the existing interfaces and using the module Ipfixprobe to turn traffic in packets into flow data [7]. We create plugins for Ipfixprobe, which enrich the flow data with all the useful service discovery protocols. Enriched flow data is passed to a NEMEA module named SDP Analyzer, which performs the aggregation and basic tagging of devices, and passes the output to the ADiCT database.

With our solution, we achieve the proposed goals of data extraction, unifying the protocols under our format and passing the data to the rest of ADiCT's system. Our solution extracts services each device queried for and advertised, allowing us to categorize it by its capabilities. Provided the device communicates enough, we also get information about the device's OS and hostname.

2. Summary of used service discovery protocols

The main motive behind service discovery is simplicity. The goal is to simplify the configuration required for using the services available on the local network to the point that there is none, hence the associated term zero-configuration networking [8]. Both SD protocols we worked with have two basic capabilities: To query for a specific kind of service and to advertise any services the device is providing to the network. Most communication within the protocols is done using multicast. We will shortly discuss the specifics when we come to describe each protocol.

The data that has to be transferred to enable these capabilities is also similar for both protocols. Every service is discovered under a name that can be used for its identification. It also has to exist somewhere in the network, so each name must have an associated IP address and a port that it runs on. Then we can start looking for additional information. In both protocols, we can find data about operating systems. Both enable communicating service details or device models and configurations, but we decided not to focus on that data in this project practice, as our ultimate aim is to categorize these devices for further monitoring, adding the context of what the device appears like on the service discovery plain, and extracting every possible detail would not aid that goal.

2.1 Simple Service Discovery Protocol

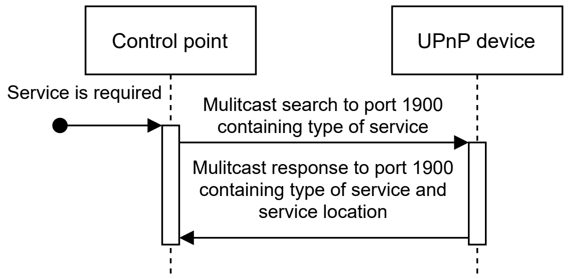

Control point
UPnP device
Service is required
Mulitcast search to port 1900 containing type of service
Mulitcast response to port 1900 containing type of service and service location

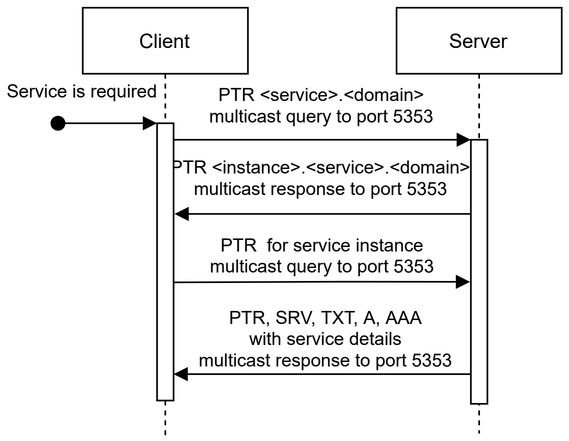
Figure 1. SSDP service discovery workflow.

SSDP [9] is a protocol managed by UPnP, created based on the structure of HTTP messages [10]. The packet structure consists of a start-line followed by message header fields. There are two main kinds of messages in SSDP: Notify and M-Search. Notify messages are used for service advertisement, M-Search for querying. Both message types include header fields (e. g. NTS or MAN) that provide greater communication context. As we do not utilize these fields in our solution, we will not go into greater detail here either.

Services are identified using URNs[2], which can be standard or vendor-defined. Standard service types are denoted by urn:schemas-upnp-org:service: followed by a unique name assigned by a UPnP forum working committee, colon, and an integer version number [9]. Vendor defined services have a custom domain in place of schemas-upnp-org.

A few examples of URNs:
schemas-upnp-org:service:ConnectionManager:1
dial-multiscreen-org:service:dial:1.

2.2 DNS-based Service Discovery



Client
Server
Service is required
PTR <service>.<domain>
multicast query to port 5353
PTR <instance>.<service>.<domain>
multicast response to port 5353
PTR for service instance
multicast query to port 5353
PTR, SRV, TXT, A, AAA
with service details
multicast response to port 5353

Figure 2. DNS-SD service discovery workflow. The PTR record serves as a level of indirection to the actual service details.

DNS-based Service Discovery [11] is a specification of mechanisms allowing clients to discover instances of services using standard DNS queries. It is one of the main pillars upon which all implementations of Zeroconf are built, such as Bonjour or Avahi [8]. It is compatible but not reliant on mDNS [12].

Service Instance Names used in DNS-SD have the following format: *<Instance>.<Service>.<Domain>*. The *<Domain>* portion is always "local." in services on local networks. *<Service>* consists of a pair of DNS labels, as set by convention for SRV records, describing the service and the utilized application protocol. Each label begins with an underscore. *<Instance>* then acts a user-friendly name for the service.

Examples of Service Instance Names:
LG webOS TV 7E6D._hap._tcp.local
Spotify - Withings Aura C6._spotify-connect._tcp.local

Service discovery using DNS-SD can be broken down into two parts - service browsing and service instance resolution. Service browsing is done via sending PTR requests for a particular service, which results in the enumeration of PTR records giving Service Instance Names of such services on the network. To enable the contact of a particular service instance, the client queries for SRV and TXT records of that name. SRV records contain the target host and port where the service can be found, TXT records are used for addi-tional information about the service.

[2]Uniform Resource Name

2.3 NetBIOS Name Service

NetBIOS Name Service, while not a service discovery protocol, has been included in our implementation for the simplicity of processing it using our pipeline.

NBNS implements name registration and resolution for NetBIOS names, which are required for communicating using NetBIOS. When the name is determined by a higher-level protocol [13], a convention is recommended to use the last byte of the NetBIOS name as a suffix that carries additional context within the NetBIOS namespace.

3. Extraction and labeling of data from SSDP, DNS-SD and NBNS

In this section we will explore what concrete fields were chosen for extraction from each protocol. We will also cover our approach to adding labels to devices in our module.

3.1 SSDP data extraction

Based on the gathered knowledge about SSDP through an examination of specifications [9] and network traffic samples, we propose the following steps for data extraction. We will determine the message type based on the start-line, looking either for M-Search or Notify. The message type gives us a good-enough idea of what to expect in the rest of the message.

We find that only a few fields include information relevant to our use-case, and we will go over them briefly now. Services are identified via URNs, the format is the same for both searching and advertising, and are found in fields NT[3] and ST[4]. For each service advertised, there is an IP address and port where it is running. The field LOCATION contains this information. We decide to keep only the port from the field, as we do not intend to track IP addresses. When considering advertisements, the SERVER field also proves itself valuable. The SERVER field contains the description of the device that is offering a service,

[3]Notification Type
[4]Search Type

Table 1. Summary of SSDP header fields and DNS-SD records used in data extraction for each label.

	Message type			
Label name	SSDP M-Search	SSDP Notify	DNS-SD query	DNS-SD response
SSDP service		NT, LOCATION		
SSDP query	ST			
DNS-SD service				SRV
DNS-SD query			PTR	
Hostname				SRV
OS	USER_AGENT	SERVER		TXT, HINFO
TXT				TXT

including the OS, used version of UPnP, and the product responsible for the service. It should be mentioned that we find the SERVER field to be quite irregular in its format in the captured traffic, and its parsing ends up being based more on our captured traffic data than the official specification. The last field we utilize is USER_AGENT, which is specified when searching. While the USER_AGENT field is officially optional, we observe it widely used, and since it, again, includes the OS, UPnP version, and the responsible product, we include it in our extraction.

3.2 DNS-SD data extraction

When it comes to DNS-SD, we use a pre-existing structure for parsing DNS packets, as will be mentioned in the next section, and therefore, we will not go into the details of packet parsing here. Instead, we will look at the DNS records used and the relevant information inside them.

The PTR record queries sent when searching for a service include the service name, giving us the identifier we are looking for. The responses then include SRV, TXT, and other records. An SRV record contains the service name, hostname (as srv_target), and the service port. We find working with TXT records tricky, as their content is specific to the individual services. Despite the irregularity, the information found is often substantial for us, and we will get to how we handle parsing them later. Finally, we get to the A and AAAA records that contain IP addresses. We decide not to collect IPs, so we discard them. The very last would be HINFO records, which contain the host-specific data, more precisely the OS and CPU type, which is useful, even though very rarely found in traffic.

3.3 NBNS data extraction

NBNS is added after the successful implementation of the previous protocols and does not continue the trends set by them. Its packet format is identical to DNS. The information found within is NBNS names and potentially NetBIOS suffixes. Both parts are extracted for clarity.

3.4 Device labeling

We approached labeling of devices with tags and categories in mind, but throughout the development, our view changed to be more general in the scope of what could be passed as a label. The labels that we create include services both advertised and queried for, hostnames, operating systems, and TXT labels. The term label then denotes any output field of the module. The data sources for each label are aforementioned. For a summary, see Table 1.

TXT labels, taken from DNS-SD TXT records, deserve a bit more detailed description. These key-value pair records can contain very useful information, such as OS, device models, and more, but are specific to every single service, and many contain a significant amount of encoded data or detailed configurations that aren't interesting for our use-case.

Our solution is to enable the end-user to configure the collection of these records. This way, new useful records, unknown to us at the time of completing the module, can be added to the configuration. Our module can now also be repurposed for a different use-case involving collecting different kinds of values.

4. Early prototype used by ACID module

In the very early stages of this project practice, a prototype was required to provide input for the ACID module [14], which was being developed at the time. The ACID module needs SDP data, as well as other inputs, for its activity detection algorithms. Our module receives a PCAP file as input and delivers SDP data in the format of a standard JSON file. For each MAC address, we include associated IPv4 and IPv6 addresses, hostnames, and operating systems. A simple MAC to vendor functionality is also implemented as part of the prototype. As for the SDP data, we deliver records of offered services for both SSDP and DNS-SD, mapping each service name to a port of its availability, and also records of service queries made by the device. Additionally, we have special fields for each protocol. DNS-SD includes a field containing TXT records we collect, SSDP is enriched by user-agents and server fields.

Our prototype is written in python, chosen for its flexibility. We choose to use pyshark, a TShark wrapping module, to process the input data, because it enables us to work with a pre-processed representation of packets, allowing us to skip ahead to developing our solution. We also filter the input data based on whether packets contain protocols mDNS or SSDP using the pyshark module.

Inside the prototype, we aggregate the data based on MAC addresses, as many devices have multiple IP addresses assigned, e.g. when a device supports IPv4 and IPv6. For each packet, we save the sender IP address and update our SDP records processing the packet's content on a very basic level. We use python's sets and dictionaries to ensure the uniqueness of our records. The packet processing then consists mostly of string parsing to isolate the interesting parts of data.

After aggregating all packets into our representation, the prototype goes over all the collected data, during which we run hostname and operating system detection and vendor lookup. Hostnames are detected based on DNS-SD SRV records. Detection of operating systems uses both protocols, SSDP containing OS in the SERVER field in Notify messages and DNS-SD service 'OSX Device Info' indicating MacOS X.

The prototype served as a decent input stub during the development of the ACID module, and it allowed us to explore both protocols in quick succession. Creating it before working on the real module gives us an overview of what issues we have to deal with and some basic expectations of our data extraction results.

5. Extracting data from SDP using a NEMEA pipeline

The NEMEA framework is used to transform raw network traffic to flow data and communicate between the modules. Ipfixprobe module plugins that extract necessary information from each of our selected protocols have been implemented. The protocol at hand is detected based on destination or source ports. All plugins have a data structure with extracted information, attached to each flow record, and have a callback to export it when the flow is finalized.

SSDP is detected by matching the destination port to 1900. Afterward, the packet payload is parsed, expecting SSDP structure. The parser settings are decided based on the message start-line, where we identify whether the message is a query or an advertisement. Parsing is implemented as looking for desired headers in the payload and saving their values when found. Most values are saved as a list of strings without change, the only exception being the service location, from which we parse only the service port.

Parsing DNS-SD begins by checking the destination or source ports for value 5353. We reuse the functions that handle parsing DNS packets in the existing DNS plugin. After matching the port, all of the packet's sections are parsed. From queries, we keep PTR requests unrelated to IP addresses. In case the packet is an answer, we save SRV, TXT, and HINFO records. To avoid duplicates, we only keep the records that are unique in our collection. The values are saved as lists of strings. For export, we concatenate the strings with semicolons.

As mentioned in Section 3, we implement configuration of TXT record collection. The implementation is done through whitelist-like configuration files both for the DNS-SD plugin and for SDP Analyzer. We give control of what keys, linked to which services, are to be collected, and on the SDP Analyzer level, we require naming each service-key combination that is to be extracted.

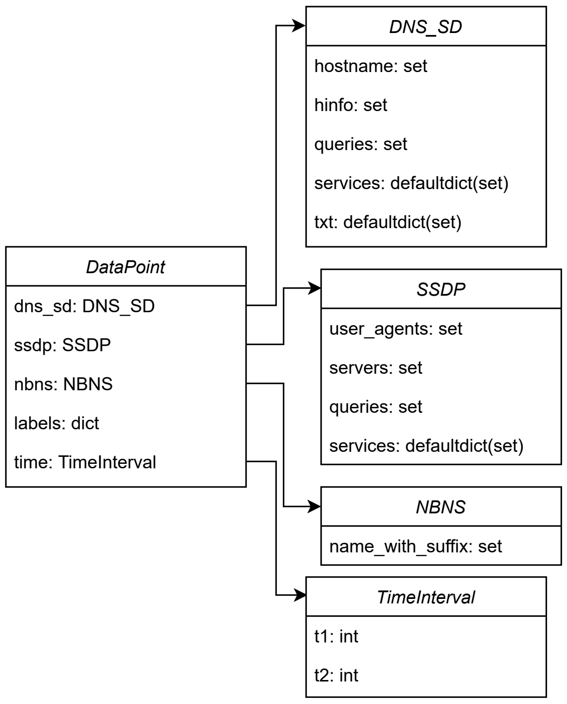


Figure 3. Class diagram for *DataPoint* object

NBNS is parsed after matching source or destination port to 137. NBNS names must be compressed because each byte of the original name is split into two bytes in the packet. We only save one name per flow.

After processing the traffic using Ipfixprobe with the above-described plugins, we get the output on three

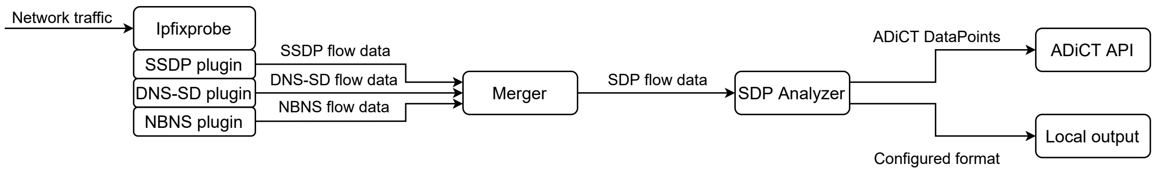


Figure 4. Final pipeline diagram showing the transformation and flow of data.

UNIX sockets, one for each plugin's flow, communicated in the NEMEA UniRec format. We use another module called Merger to join these streams of flow data into one. As a part of merging the flows, we also remove fields that will not be aggregated later, such as the number of bytes transferred, protocol, or TCP flags. The merged flow data is then available on a single socket, the name of which we pass to SDP Analyzer as an argument. To communicate with the NEMEA interface we utilize the pytrap module, which handles converting the incoming flow records to python objects.

A dictionary with MAC addresses as keys and *DataPoint* class objects as values is used to store the data. One *DataPoint* object stores data about one device. The data is separated into child objects, one for each protocol, a time interval of the data collection, and a dictionary of additional labels. For the class diagram of the final solution see Figure 3. With each incoming flow record, the *DataPoint* stored for that particular MAC address is updated.

After all flow records are processed, labels are assigned based on collected data. The hostname is propagated without adjustments from the DNS-SD layer, the OS label is added as a combination of extracted OS in DNS-SD and SSDP. TXT record labels are based on SDP Analyzer's configuration.

Finally, the aggregated and labeled data is passed out from the module. The format of the local output is configurable with the module's launch arguments. As SDP Analyzer is meant to be an input module for the ADiCT system, an API endpoint can also be configured. Then the output will be sent to that endpoint in the ADiCT format using POST requests.

SDP Analyzer has been deployed in a VM setup used for testing of the NEMEA framework. Analyzed traffic data comes from few selected offices at the faculty and is saved to files based on date and time. SDP Analyzer is then run periodically every hour, using the newest traffic file as input. A testing API endpoint is set up on ADiCT's development server, and it is used for the module's output so the platform can use it for further analysis. The final pipeline is visualized in Figure 4.

The network traffic in the form of PCAP files or captured directly from the network is fed into the Ipfixprobe module. Ipfixprobe opens a UNIX socket for each plugin, where it then outputs the processed flow data. The Merger module is then used to merge the flow steams into one of our specifications, which is passed to the SDP Analyzer. The final output is printed to standard output in one of the selected formats (ADiCT DataPoints, JSON, plain text) and sent to ADiCT's database.

5.1 Pipeline evaluation

Comparing the current pipeline to the prototype, we can see the total run-time shorten about twenty-fold, measured on multiple larger PCAP inputs. Based on the improvement of run-time we can confirm the positive effect of using the NEMEA framework for pre-processing the raw traffic data before aggregation.

One negative side effect of the pipeline change is the process management that is now required to run the analysis. For easier use of SDP Analyzer, we provide a shell script that handles running the NEMEA pipeline. Using the script is a solution for a one-time use of the module. It should be noted that for use on live local traffic, setting up a network-capturing pipeline would be required to run the prototype, and the NEMEA framework takes care of it nicely.

Thanks to the module deployment for testing, we have been able to extract data from service discovery protocols on the selected local networks. The ADiCT system, to which we send the aggregated results, is currently under development. Due to limitations of the system's functionality, the collected data is currently hardly accessible and it is difficult to judge its ultimate usefulness. Nevertheless, use-cases for the collected data are still being explored, and it will be interesting to see its analysis when combined with other data sources.

6. Conclusions

This project practice was focused on data extraction from SDP. We have studied the structure of our selected service discovery protocols, showing what data is possible to extract from them. A simple prototype was implemented to prove our findings and used during the development of the ACID module. The prototype was followed by a more advanced implementation using the NEMEA framework. Plugins for Ipfixprobe were created to extract data from SSDP, DNS-SD, and NetBIOS. A NEMEA module called SDP Analyzer that aggregates and processes the SDP data from Ipfix-probe was created. A basis for a more advanced SDP data analysis was built by creating an output interface for the ADiCT system database API. The solution was deployed on development servers for NEMEA and ADiCT to prove its functionality on data from real network traffic.

Our solution relied only on passive monitoring, allowing us to run our module remotely, using only recorded traffic captures. Our solution extracted the services advertised and queried for by each device, the device's hostnames, operating systems, and other data found in DNS-SD's TXT records.

As was mentioned before, SDP data extraction was done to provide data for further analysis, some of which is already taking part while writing this project practice paper. Our solution provided a new data category to the ADiCT system, widening the possibilities for analysis. The overall utility of SDP data showed promising, and we will be building on top of it inside the ADiCT system, looking to find relationships between service discovery data that could be used to categorize the devices, and possibly comparing our findings with those based on other data sources.

Acknowledgements

I would like to thank my supervisor Martin Žádník and my technical supervisor Václav Bartoš for their advice and other helpful input during the project practice and writing of this paper.

References

[1] Cisco. Using passive monitors, 2020. https://www.cisco.com/c/en/us/td/docs/net_mgmt/cisco_netmanager/1-1_voice/user/guide/CnMuguide/10_pmonitors.pdf.

[2] Inc Rumble. Rumble network discovery, 2020. https://www.rumble.run/.

[3] Lennart Poettering and Trent Lloyd. Avahi, 2015. https://www.avahi.org/.

[4] Rapid7. Metasploit framework, 2020. https://www.metasploit.com/.

[5] Gerald Combs and contributors. Wireshark, 2020. https://www.wireshark.org/.

[6] Tomas Cejka, Vaclav Bartos, Marek Svepes, Zdenek Rosa, and Hana Kubatova. Nemea: A framework for network traffic analysis. In *12th International Conference on Network and Service Management (CNSM 2016)*, 2016.

[7] B. Claise, B. Trammell, and P. Aitken. Specification of the ip flow information export (ipfix) protocol for the exchange of flow information. STD 77, RFC Editor, September 2013. http://www.rfc-editor.org/rfc/rfc7011.txt.

[8] Stuart Cheshire and Daniel H. Steinberg. *Zero Configuration Networking: The Definitive Guide*. OREILLY MEDIA, 2005.

[9] Andrew Donoho, Bryan Roe, Maarten Bodlaender, John Gildred, Alan Messer, YoonSoo Kim, Bruce Fairman, and Jonathan Tourzan. Upnp device architecture 2.0. Technical report, Open Connectivity Foundation, Inc., 2015.

[10] Roy T. Fielding, James Gettys, Jeffrey C. Mogul, Henrik Frystyk Nielsen, Larry Masinter, Paul J. Leach, and Tim Berners-Lee. Hypertext transfer protocol – http/1.1. RFC 2616, RFC Editor, June 1999. http://www.rfc-editor.org/rfc/rfc2616.txt.

[11] S. Cheshire and M. Krochmal. Dns-based service discovery. RFC 6763, RFC Editor, February 2013. http://www.rfc-editor.org/rfc/rfc6763.txt.

[12] S. Cheshire and M. Krochmal. Multicast dns. RFC 6762, RFC Editor, February 2013. http://www.rfc-editor.org/rfc/rfc6762.txt.

[13] Microsoft. [ms-brws]: Common internet file system (cifs) browser protocol. Technical report, Microsoft Corporation, 2018.

[14] Jan Neužil. Network devices and services identification using passive monitoring. Master's thesis, Factulty of Information Technology CTU in Prague, 2020.