CIRCL
Computer Incident
Response Center
Luxembourg

SECURITY
MADEIN.LU
THE CYBERSECURITY COMPETENCE CENTER

UNIVERSITÉ
DE LORRAINE

Internship Report

Information Technology Master's Degree

Network and Systems Security

Potiron

Index, Normalize and Visualize data from Honeypots

Author:
Christian STUDER

Supervisors:
Yann LANUEL
Alexandre DULAUNOY

2017 April 3 - 2017 September 30

Acknowledgements

I would like to thank first Mr Pascal STEICHEN, CEO of SMILE g.i.e for the opportunity he gave me to do my internship within his organization.

I would also like to thank Alexandre DULAUNOY as my supervisor and tutor during the whole internship period, for his welcome in CIRCL department, his time helping me, his precious pieces of advice, and his trust in my ability to have my work done.

My next acknowledgements will go to the SMILE employees in general for their kindness and CIRCL employees for their welcome, their expertise sharing, and even more precisely to:

- Gérard WAGENER who created the first version of Potiron, basis of my subject, for his knowledge about honeypots, capture files, and data storage.
- Raphaël VINOT, for his python expertise.
- Cédric BONHOMME, for the multiple tools he suggested me to use.

Finally, I would thank Mr Yann LANUEL and the University of Lorraine for all the lessons we had this year, including lectures with Alexandre DULAUNOY and Raphaël VINOT which gave me good bases about the tools and skills I used during all my internship, such as Redis and how to store data on it, tshark and some other capturing tools, looking for information from capture files, etc.

Contents

Acknowledgements **1**

1 Introduction **3**

2 SMILE g.i.e **4**
2.1 SMILE . . . 4
2.2 CASES . . . 5
2.3 CIRCL . . . 5

3 Subject Presentation **7**

4 Development **8**
4.1 Tools used and Features . . . 8
4.1.1 Focus on some features . . . 8
4.1.2 Overview of the tools used . . . 9
4.2 Initiation . . . 12
4.3 Usual functionalities . . . 15
4.3.1 Display the occurrence trend of a special field . . . 17
4.3.2 Compare in a defined timeline the most frequent occurrences of a field . . . 18
4.4 Additional functionalities . . . 20
4.4.1 Initial Sequence / Acknowledgement Numbers over time . . . 20
4.4.2 Parallel coordinates . . . 24
4.4.3 Represent Layer 2 information . . . 25
4.5 Improvements . . . 29
4.5.1 Reduce storage size and processing time . . . 29
4.5.2 Functions improvements . . . 31
4.5.3 Improve user-friendliness . . . 36
4.6 Remaining tasks . . . 38

5 Conclusion **39**

Bibliography **40**

Glossary **42**

Appendices **44**

1. Introduction

As a second year of Master's Degree student, I have the chance to conclude my scholarship with an internship for a period from April 3 2017 to September 30 2017. Hence during this period, using and improving the knowledges acquired with my University formation in Information Systems Security is the main purpose of the internship.
As a result, this is the opportunity for me to work on a real project with all the included constraints such as, in my case, data size, execution time and processing swiftness, or storage capacities, in order to have as much practice and experience as possible for the beginning of my active life.

I also wanted to find a subject I would really be interested in, so I came upon the different projects proposed by CIRCL and found the one dealing with honeypots the most attractive [1]. This is how I came at SMILE g.i.e in Luxembourg for the 26 weeks of my internship.

Apart from the legal working time which is different from what we knew so far but not a problem at all, I do not have to face any particular constraint in my work environment, I have been set in the better conditions for work from the first day I came. Indeed, I am working within CIRCL's office with the whole team, sharing their environment and including me in the conversations, reflexions sometimes, which enforces human interactions and makes my internship run in good conditions.

In the next section I will start with a description of SMILE and its features, then I will describe more precisely my internship subject with all the initial elements I started my work with. The next part of this report, also the biggest one, will deal with the presentation of my work and the functionalities provided by Potiron. Finaly, I will give a conclusion concerning all my contributions as well as the knowledge and practice acquired.

2. SMILE g.i.e

2.1 SMILE

Security Made In Lëtzebuerg is an initiative of the Grand Duchy of Luxembourg to support the economical activities inside the country. More precisely, the association of 3 ministries (Ministry of Family, Integration and the Greater Region, Ministry of Education, Childhood and Youth, and Ministry of the Economy), as well as local government federations SIGI (Syndicat Intercommunal de Gestion Informatique) and SYVICOL (Syndicat des Villes et des Communes du Luxembourg) is the origin of the foundation and management of SMILE g.i.e and its activities. Ministry of the Economy is the institution supervising SMILE's activities and finance.

Figure 2.1: Stakeholding founders

SMILE's main goal lies on the improvement of security information sharing and awareness as well as providing security solutions for private and non governmental public sectors. In order to reach its objectives, the g.i.e managed to split its activities into 2 departments which will be described later: CASES and CIRCL.
SMILE launched in February 2015 `securitymadein.lu`, as the main online source for cybersecurity in Luxembourg, providing news and relevant information for users, organizations and the ICT community, and centralizing all the information about its activities.

Figure 2.2: securitymadein.lu logo

SMILE's premises are located at 41 avenue de la Gare L-1611 Luxembourg. The group is currently composed of around 20 persons including consultants, security experts, managing employees, etc. covering a large range of skills from information security to communication, marketing, or legal and project management.

2.2 CASES

Figure 2.3: CASES logo

CASES deals with the organizational and human side of cybersecurity. Their work is to provide both risk assessment solutions and some cybersecurity knowledge.

They also develop and provide their own tool MONARC which can be used to proceed an entire risk assessment, covering different points like:

- identification of the possible threats and their criticality
- description of the consequences
- solutions that could be implemented to reduce the risks

Moreover, in order to promote cybersecurity knowledge, they give different courses for people from any professional sector, adapted for the status or position of each of them.

2.3 CIRCL

I, personally, have been welcomed by the other department, CIRCL, dealing more about the technical aspects of security like Incident Response.
All CIRCL employees work in the same office but even though there is no special unit, the expertise of each one is different, this is why I work mostly with Gérard WAGENER who created most of the version of Potiron I had at the beginning, Raphaël VINOT who is most of the time working with python, and of course Alexandre DULAUNOY who is supervising my internship.

Figure 2.4: CIRCL logo

CIRCL is the Computer Emergency Response Team (CERT) for the private sector, communes, and non-governmental entities in Luxembourg. CIRCL's main goal lies in attacks and incidents handling. As a result, their missions are to:

- provide response facility to ICT-incidents
- coordinate communication among incident response teams during security emergencies and also prevent future incidents
- support ICT users in Luxembourg in case of security incidents to recover quickly
- minimize ICT incident-based losses like information theft and disruption of services at a national level
- gather information related to incidents and security threats in order again to prevent future incidents, and to better prepare their management
- provide optimized protection for systems and data
- provide a security related alert and warning system for ICT users
- foster knowledge and awareness exchange in ICT security

In order to carry out these missions well, CIRCL also provide services like:

- Incident Coordination and Incident Handling
- Incident Handling Support Tools and Services
- Data Feeds and Early Detection Network
- Additional Request or Research Project Partnership

In parallel and in order to support this kind of activity, CIRCL is developing many different tools such as MISP (the most famous one), AIL, CIRCLean, or Potiron, providing solutions for a lot of security aspects like information sharing, malware analysis, attacks prevention etc.

3. Subject Presentation

Potiron is a tool used to index, normalize and visualize data from honeypots.
To carry out these functionalities, the modules initially implemented were able to read capture files, normalize the data into json files, and store the normalized data into a Redis database. The tools used to proceed this kind of basic read and storage operations are Ipsumdump[2], and as just mentioned, Redis[3], but also all the source code of Potiron is written in python. The problem that appeared then was the necessity, after storing data, to extract information manually in order to plot them or use them in another purpose.

As a result, the aim of my internship is to improve Potiron's functionalities, such as visualization and analysis, following some guidelines like:

- integrate another tool to process the capture files, in order to have a larger set of available options.
- automate the visualization with some plotting / graphs creation tools, instead of doing it manually like it has been done by far.
- for the plots, generate exclusively local files that will not be modified once processed.
- add some other functionalities to display more different information from the data.

Concerning usage of local files, it is the point that does not appear explicitly in the initial subject statement, and which has been redefined during the internship.
Since some functionalities or the way to implement them require additional guidelines, I will mention each of them (the guidelines) at the moment they occur, in the next section dealing with the development of the subject, and explain why they appear.

In practice, Potiron should potentially be implemented in an automated environment. Let me explain this term: as capture files are generated automatically every 5 minutes, we may have our modules used with the same process to directly parse data within the same timeline, before the next file is popping up. This feature will be mentioned later again, while speaking about processing time.

Anyway, in order to provide the functionalities defined in the subject I can use a large panel of open-source programs or libraries to extend the functionalities with well known tools that are up-to-date, and possibly maintained. For this kind of purpose, GitHub[4] is a really well provided platform.
I also mainly work with git to commit all the modifications I make, and to have the history of all the changes.
Here is the link of my GitHub account to follow the details of my work more deeply: `https://github.com/chrisr3d/potiron`

4. Development

4.1 Tools used and Features

Before diving right into the description of functionalities and how they are implemented, let us have a quick overview of some features mostly involved in the project, and their interest. We will then focus on the tools used, with a brief explanation why they are useful.

4.1.1 Focus on some features

Json format

We largely mentioned json[5] in the previous section to describe its use we make and will mention it again later for the same purpose. The interest of using such an open-standard file format relies on its human-readable characteristics consisting of attribute-value pairs and array data types.

For example:

```
[{"key 1": "value 1", "key 2": "value 2"},
 {"key 1": "value 3", "key 2": "value 4"}]
```

Having a standard typography like this is the guarantee that regardless of the parsing tool used, values with a given attribute (for instance "key 1") will always be referenced with this precise attribute.
As it is part of the functionalities as well, more precisions will be given in the next section.

Use local files

As it is mentioned in the presentation of Potiron, available on its Github page, the first versions used to deal with a Python web development open-source framework: Flask. It is not the case anymore since it has been stated that using local files would be a better solution at the moment, because static a static website is more secure (against injections for example). Also it is not planned in the subject that online server functionalities and visualization would be included in the project before the end of the internship period.
The interest here is to have local final graphics that can be shared if needed, but not modified or altered once processed.

Run processes with parallel commands

Among all the commands commonly used to execute the different modules, an important part is to run processes in parallel. The general aim of Potiron is to parse possibly big datasets, involving a lot of files and information. The first interaction with data concerns capture files reading, and as these files contain 5 minutes data, it would require a huge number of commands to process each one of them separately. Thus GNU Parallel[6] command is doing the work for us.
This shell tool provides executing jobs in parallel using one or several computers. This allows us for instance to run only one command to include multiple files at the same time, which makes easier normalization and storage or running one command for each file in not needed.
Basically to run this kind of processes, following parallelization, we use something like:

```
ls files_to_process | parallel --line-buffer --gnu module [arguments]
```

This way, 'ls' or alternatively 'find', depending on the case, is used to display the list of files to process via standard output, which is piped and given to the parallel command to execute the concerned module as many times as there are lines in the list and each line can be used as an argument of the module, using '{}' to design them. Parameters '–line-buffer' and '–gnu' are used respectively to ensure that buffer output is based on lines, and to make parallel behave like GNU Parallel.

4.1.2 Overview of the tools used

The main tool used in any case for all the functionalities to implement on this project is, very simply, Python language, more precisely in its version 3.

Figure 4.1: Python logo

Let us speak then about git[7], the free and open source distributed version control system used to manage version in any kind of project. As mentioned previously while presenting the subject, the entire work involved in the development of Potiron relies on git, and more specifically its worldwide most used web-based platform GitHub.

Figure 4.2: git & Github logos

Using distributed version control and source code management functionality is the guarantee that version updates can easily be reviewed and accepted by the repository administrators, or reviewed and modified later, etc.

Once this focus done, we will see the list of all the tools used with Potiron. This list will thus contain every tool useful to support all the functionalities and which should be installed to make Potiron work.

Parse capture files

Parsing capture files is a functionality provided by a lot of different tools. In our case, the first one supported by the initial version Potiron is Ipsumdump[2'].
Some utilities provided by Ipsumdump needed in our case are for instance:

- Read packets from network interfaces, as well as from both Tcpdump files and Ipsumdump files.
- Uncompress Tcpdump or Ipsumdump files when it is necessary in order to read them (contrary to Tcpdump which does not support compressed files).
- Sample and/or filter traffic based on its content (which is the function we need in some cases to parse our capture files)

Then, the aim is to progressively switch to Tshark[8], which is a network traffic dumping and parsing tool too but differs from Ipsumdump in some points.
Without any options set, Tshark works much like Tcpdump, but the difference relies on its ability to parse a lot more information. This is basically because Tshark is able to detect, read and write the same capture files that are supported by Wireshark[9]. Actually Tshark is simply the command-line version of Wireshark.

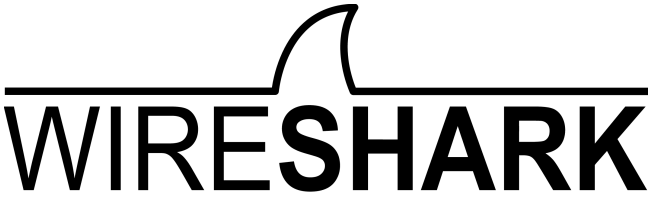

Figure 4.3: Wireshark logo

Tshark and Wireshark are also equivalent in the grammar to use in order to define filter, which is a significant information on general in our project to get help or test some configurations from the 'display filter' field on Wireshark interface.

But in some cases, Tshark could have weak points, such as its packet reading mechanics. The problem is filters applied to select samples of packets are only display filters, and even though all the fields are not displayed, they are processed in background anyway. This feature makes Tshark quite slow while processing heavy capture files.
To counter this drawback, the alternative is to use for example Tcpdump[10] and its really efficient Berkeley Packet Filter (BPF[11]) ability. With Tcpdump, it is possible to read from a capture file, apply a BPF which reduces the amount of data to process, and write what is read in another capture

file. This reduced capture file can then be processed with Tshark.
This can potentially save time and hardware resources when the sample we want from our captures is a small part of the entire data. Applying the right BPF this way is better because Tshark is processing only useful data and processes are lightened.

But after all, why are there so many capture formats ? Pcap is the main format used in Tcpdump and Wireshark. Unix-like systems implement this format in the libpcap library, and Windows uses its adapted version winpcap. While pcap continues to be used today, a new format called pcapng has been under development for some years, providing additional utilities such as multiple interfaces captures, timestamp resolutions improvement, containing additional data, or supporting expendable format, and will progressively replace pcap in the next few years.
In our situation, we use .cap capture files, the network general sniffer format, which is providing more information in the headers of the files and frames than .pcap format.

Store data

As stated previously, Redis[3'] is our key tool for data storing. This open source store provides an in-memory data structure and can be used as database, cache, and message broker. The interest using it relies upon the ability to run atomic operations such as appending to a string, incrementing the value in a hash, pushing an element to a list, and so on, which are precisely the kind of operations we want to run. Furthermore, every call to Redis can be proceeded from python modules.
As a result we basically have all the storage functionalities we need in our case with this single tool.

Figure 4.4: Redis logo

Generate graphics

There is a lot of different tools providing graphics generation, but the aim is to have a good representation model, and interactions if possible.
To create standard plots satisfying our few requirements, with a large panel of different possible interactions, a pretty good solution that appeared quite fast from our researches is Bokeh[12].

Figure 4.5: Bokeh logo

Once again it is possible to use it directly from python modules since it is a python library. It is also an interactive visualization library, that provides creation of graphics using html format for web browsers presentations. The interactive part is obviously allowed by javascript code, generated automatically within html by Bokeh.

Due to the quite reduced number of different models available with Bokeh, another tool is needed if we want to extend the panel of possible visualization models. Therefore we come with D3JS[13], a javascript library for graphics based on data creation.

Figure 4.6: D3JS logo

As this one is not a python library, the aim of our python modules is to build the datafiles necessary to generate the graphs, that are as bokeh graphs, html files with javascript. When Bokeh generates the output file on its own, interpreting python dedicated code, with D3JS the principle is to use an html template that will run a javascript function to generate the graphic.

Finally, an alternative for a precise case of graphics is Circos[14]. The approach is different here since it is a software package, that is executed separately. As with D3JS datafiles are needed but then Circos is building itself from the package the output file containing the graphic.

Build a virtual environment

As an additional and totally optional functionality, it is possible to build a virtual environment like it is the case for MISP images, in order to give users an updated and complete version of the program with all the requirements automatically installed. This way, users get their own version, totally functional after the installation script is executed.
On this purpose, the tools we can use is Vagrant[15]. We will come back to it later while focusing on the functionality provided by the tool, and the use we can make of it.

Let us now deal more in details with the development of the subject.

4.2 Initiation

Before I start working on any module and since Potiron has been developed couple of months ago, it is still written in python2 when I get here and Raphaël simply proceed with a basic translation to python3. My tasks here is to remind me the basics of the language, and once everything is in python3, to test the functionalities described in the README file of the project to test 2 things: is the code working in python3 ? and does the README file contain the required information to allow any newcomer user exploit all the jobs the program is supposed to do ?

At this point, the project files I have, to work on, are limited to the following functionalities:

- normalize data from capture files into a json structure
- read json files to store normalized data into Redis
- all the additional modules helping normalizing or storing data

For more details, you can find the files I just mentioned, following the link referenced as "original project" in the bibliography. It is a good way then to get back into python, and also Redis.

Once I get back into it, my first task is about writing a module reading capture files, exactly like the existing module "potiron-json-ipsumdump" does, but using tshark instead of ipsumdump. Just as we saw Redis with Alexandre during one of his lectures, we practiced tshark commands as well with him, thus again I get back into it quite fast and I am ready to write the module. The deal here is to find the matchings between ipsumdump fields and tshark fields. Then the new module is mostly similar to the ipsumdump one.

Before going any further, why using Thsark instead of Ipsumdump ? Tshark is providing a lot of options and can potentially display more information than some tools like Ipsumdump or Tcpdump do. We assume this during the whole internship when we implement additional modules dealing with some fields / data not treated by the other tools.

Storing data into Redis is then provided by the module "potiron-redis". As it takes normalized data to store it, it does not matter which tool is used to read capture files, the important part is reading data from json files. As a consequence, I do not have to change anything in this module at the moment.
By the way, there is one difference in the data stored because the details level provided by Tshark is really high and the amount of different fields you can display is big as well. Thus there is one field called "ip options" I still did not find in Tshark even after I tracked all the fields of packets in Wireshark, at least not in one single field.
Let me remind you that Tshark is the 'Wireshark for command line', it is also interesting to watch packets with Wireshark to find visually all the fields and the typography that should be used with Tshark. This is basically what I did to build the correspondence table between Ipsumdump and Tshark fields [16]:

FIELD	IPSUMDUMP	TSHARK
Timestamp	--timestamp	-e frame.time_epoch
Length	--length	-e ip.len
Protocol	--protocol	-e ip.proto
Source ip	--ip-src	-e ip.src
Destination ip	--ip-dst	-e ip.dst
Ip options	--ip-opt	(?)
Time To Live	--ip-ttl	-e ip.ttl
Type of Service	--ip-tos	-e ip.dsfield
Source port	--sport	-e tcp.srcport / udp.srcport
Destination port	--dport	-e tcp.dstport / udp.dstport
Sequence Number	--tcp-seq	-e tcp.seq
Acknowledgement Number	--tcp-ack	-e tcp.ack
Icmp Code	--icmp-code	-e icmp.code
Icmp Type	--icmp-type	-e icmp.type icmptype

As stated before, these changes do not affect our Redis structure as fields are picked up from json files, and the missing values of ip options with Tshark are not the most important ones according to my supervisors.

The Redis structure defined by Potiron's initiators / creators is set up to count the number of each different field that appears in data. The purpose of this kind of structure is to store the scores corresponding to the occurrence trend of the different fields, the relations between ports, addresses, timestamps and all the other pieces of information in each packet are not saved.
The keys (defining the location of each information to store) involved in this kind of structure are designed with the following pattern: *sensorname:timestamp:field*
The sensor name is the identifier of the honeypot involved in the data currently stored, the timestamp is actually the date with a day accuracy, and the field is the field defined in the json file. For instance if we take on packet coming from honeypot named 'chp-5890-1' at the precise time '2017-04-03 16:22:00', each field will be incremented: let's imagine in this packet destination port is 80, we will thus increment the value '80' within the key 'chp-5890-1:20170403:dport', and all the packets with destination port 80 with a timestamp corresponding to the same day will do the same. It is also the same model for each field, and each packet in each file. The structure thus looks like in the following example:

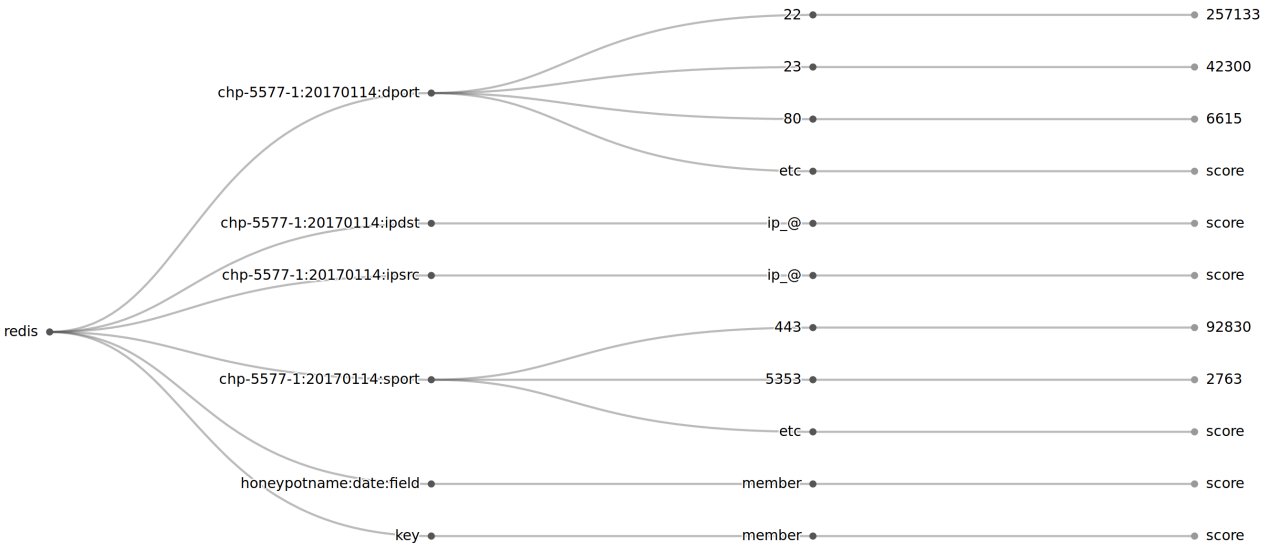

Figure 4.7: Redis initial structure

There is one python module allowing us to give arguments to a function, making easier its call from command-line: argparse[17]. The python module is employed to define what arguments are required for the program, and also in our precise case allow users to set the good variables which will be used with the functions. This kind of module is typically something we want to deploy in all of our Potiron modules because it is a really convenient tool.
We can now focus on the tools used throughout the internship period, and some features it is interesting to mention.

4.3 Usual functionalities

As normalization was already working with the basic version of Potiron (i.e with Ipsumdump), I only gave a kind of update so far, in order to support Tshark.
Normalization is crucial because of the output differences between these two different tools. Let us see the difference with an example of the last packets of the same file, parsed with both of the tools.

Ipsumdump:

```
...
1490915174.259839 52 T 94.242.208.82 68.3.57.224 . 64 0 80 59473 - -
1490915174.413705 40 T 68.3.57.224 94.242.208.82 . 55 0 59473 80 - -
1490915174.413720 694 T 68.3.57.224 94.242.208.82 . 55 0 59473 80 - -
1490915174.413740 40 T 94.242.208.82 68.3.57.224 . 64 0 80 59473 - -
1490915174.413998 264 T 94.242.208.82 68.3.57.224 . 64 0 80 59473 - -
1490915174.531670 64 U 164.77.52.19 94.242.208.83 . 118 0 53 35897 - -
1490915174.531693 92 I 94.242.208.83 164.77.52.19 . 64 192 - - 3 3
1490915174.565501 40 T 68.3.57.224 94.242.208.82 . 55 0 59473 80 - -
```

Tshark:

```
...
1490915174.259839 52 6 94.242.208.82 68.3.57.224 64 0x00000000 80 59473
1490915174.413705 40 6 68.3.57.224 94.242.208.82 55 0x00000000 59473 80
1490915174.413720 694 6 68.3.57.224 94.242.208.82 55 0x00000000 59473 80
1490915174.413740 40 6 94.242.208.82 68.3.57.224 64 0x00000000 80 59473
1490915174.413998 264 6 94.242.208.82 68.3.57.224 64 0x00000000 80 59473
1490915174.531670 64 17 164.77.52.19 94.242.208.83 118 0x00000000   53 35897
1490915174.531693 92 1 94.242.208.83 164.77.52.19 64 0x000000c0   53 35897 3 3
1490915174.565501 40 6 68.3.57.224 94.242.208.82 55 0x00000000 59473 80
```

And now to compare with our json structure, here are the two last packets within their json data file:

```
[ ...
{"timestamp": "2017-03-31 01:06:14.531693", "length": 92, "protocol": 1,
"ipsrc": "94.242.208.83", "ipdst": "164.77.52.19", "ipttl": 64, "iptos": 0,
"tcpseq": -1, "tcpack": -1, "icmpcode": 3, "icmptype": 3, "sport": "53",
"dport": "35897", "packet_id": 1211, "type": 2, "state": 0},
{"timestamp": "2017-03-31 01:06:14.565501", "length": 40, "protocol": 6,
"ipsrc": "68.3.57.224", "ipdst": "94.242.208.82", "ipttl": 55, "iptos": 0,
"tcpseq": 3604960705, "tcpack": 3084289153, "icmpcode": 255, "icmptype": 255,
"sport": "59473", "dport": "80", "packet_id": 1212, "type": 2, "state": 0}
]
```

As you can see, the formats are not the same for some fields, such as protocols (Ipsumdump gives the initial of the name, when Tshark uses a number).
Thus, the aim of this operation is to transform data from any of the 2 first outputs, parsed with our favourite parser tools, into the last model which is the normalized one, used both in json and Redis (the typography is the one defining the field in Redis keys).

Let us remind the precise process to go from capture files to Redis is the following:

- read each packet in the capture
- parse each field we extract from the packet, following the fields specified in the command
- write normalized values into a json format
- use json files to read normalized values to increment keys values in Redis

The reason why normalized data should be saved in json files is external to the scope of this internship, but it is useful by the way if we need to rebuild our Redis database, because we do not need to reprocess capture files again and just use "potiron-redis" module.

The usual pattern of processes we execute in any case to extract data from our capture files is pictured in the following diagram. The only aspect which differs from it lies in the fact that we will no longer use Ipsumdump through the next parsing modules.

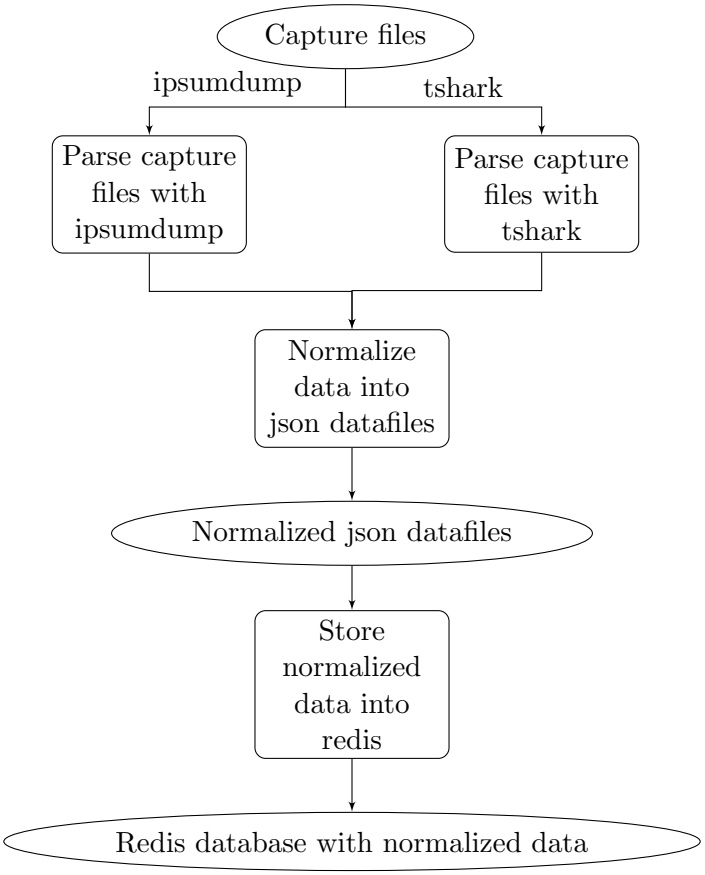

Figure 4.8: Flow diagram - From capture files to Redis database

Also one important feature to provide is about visualization. The importance of visualizing data from honeypots relies on the support it gives to detect some unusual behaviors within the network traffic. For instance watching a rising number of requests for a special ip address in a short period may be the sign of something strange.
But visualization helps following up statistics for some ports or protocols as well.

4.3.1 Display the occurrence trend of a special field

The first valuable type of representation is to display, for any field like a specific destination / source port or a specific ip address, the amount of times it appears each day, for example in a month.

For this king of visualization, we use a very convenient tool able to create a lot of different graphics: Bokeh.
Among the large panel of different representations provided by Bokeh, we can use the basic plotting one to meet the needs of displaying scores of some specific fields for several days:

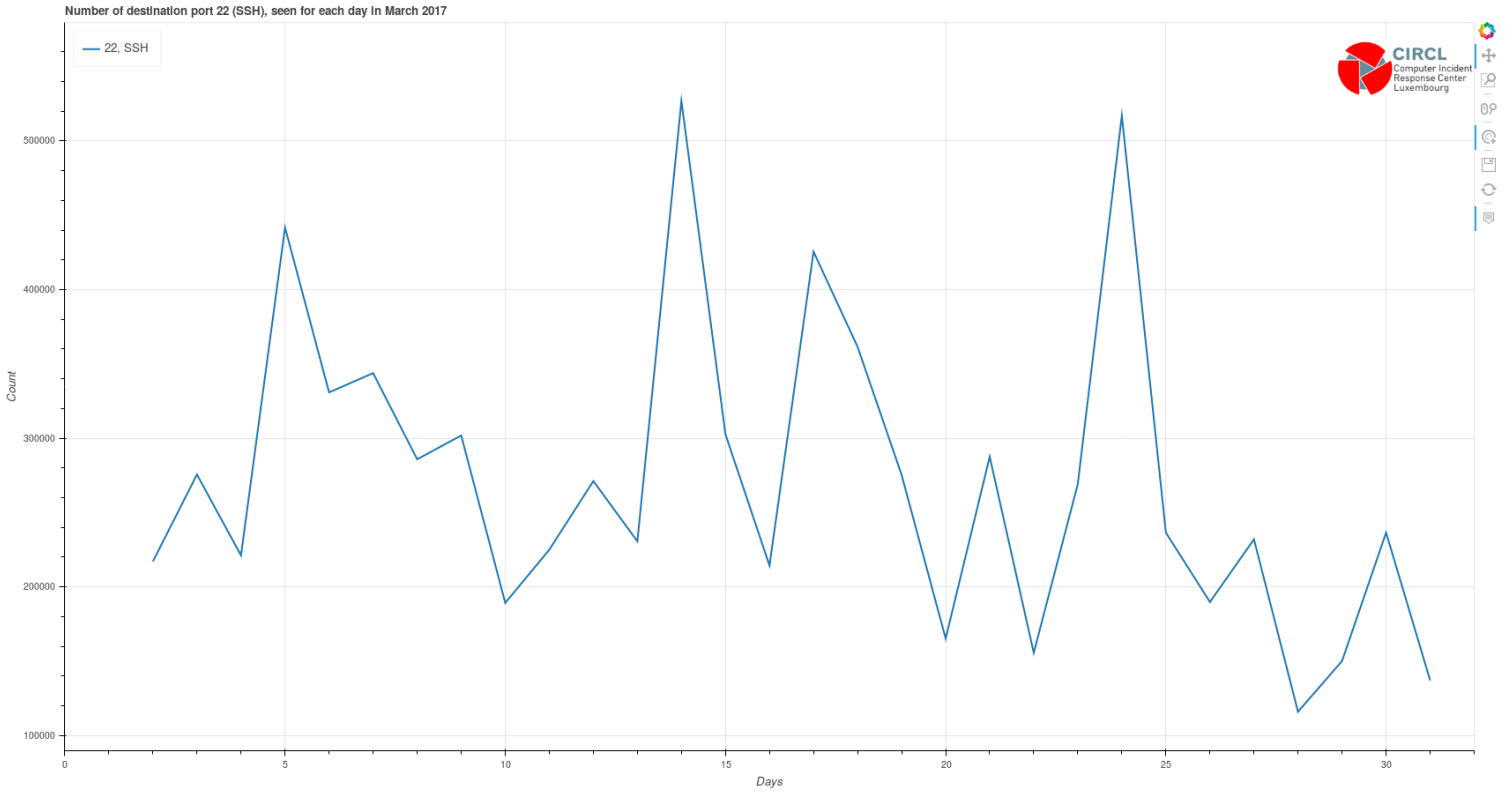

Figure 4.9: Bokeh

On this example we chose to display scores of the destination port 22. Bokeh allows zooms, selections of precise zones of the plot and several other interactions, as a consequence it is for instance possible to select only one precise part of the graph, to show up some information, and save this part as a picture with the "SaveTool".

To set up this kind of plot we want to build a python module which is able to look for the values we need, and plot them. The way to do it consists of sending requests to Redis for each day and for the precise value of the field we want to display. Since python includes a Redis library as well, everything we need to build our first type of plot can be defined and processed in a single module.

4.3.2 Compare in a defined timeline the most frequent occurrences of a field

After we saw the trend of a special field for each day in a month, we may want to compare the values observed every day for all the same type of fields (for instance after plotting scores of destination port 23, watch the scores of all the other destination ports) between each others.
From this idea, an interesting representation would be a top 5, 10, 20 or no matter how many, of the best scores for the field we want, as displaying all the existing values we have on Redis would produce too heavy graphics in terms of visualization.

To display this kind of information, we usually see circular graphics, such as pie charts, but on our case we can have disparate scores. For instance if some fields have really higher scores than any others, the smaller scores would represent only pretty small parts of the chart, making them unseen. We want to have a more exclusive and clear presentation, as a consequence we choose the 'bubble chart' to do the job. The principle of bubble charts is pretty simple, every field is represented by one bubble whose size defines the score: bigger bubbles for higher scores and vice versa.

For this kind of chart, D3JS is the tool we use. As stated previously within the description of all the tools used for this project, our goal is thus to build the data files. This task consists of writing the fields with their score, following the next model example:

```
id,value
22 - SSH,
22 - SSH,8318565
443 - HTTPS,
443 - HTTPS,1490657
80 - HTTP,
80 - HTTP,1307450
23 - Telnet,
23 - Telnet,940021
993 - imap4 protocol over TLS/SSL,
993 - imap4 protocol over TLS/SSL,207056
5358 - WS for Devices Secured,
5358 - WS for Devices Secured,147464
2323 - 3d-nfsd,
2323 - 3d-nfsd,112868
7547 - DSL Forum CWMP,
7547 - DSL Forum CWMP,92817
587 - Message Submission,
587 - Message Submission,92481
6000 - unknown destination port,
6000 - unknown destination port,37487
```

In this example only 10 different values are written out of hundreds, because a limit can be defined as an argument of the module building datafiles, with 10 as a default value. This limit is the guarantee we always have user-friendly charts with only a limited number of bubbles.

To cover multiple situations and requirements, 3 different modules are created:

- 'export_csv_month' to select the most frequent fields over a month
- 'export_csv_all_day_per_month', as implied by its name, to create for each day in the month one datafile with the daily most frequent fields
- 'export_csv_day' to create the datafile for only one single day

The last one mentioned is also useful to reprocess only one day if necessary.

With this csv datafile structure, the D3JS javascript function will watch each line and take the string before "," as the id of a new bubble, with the rest of the line as the score associated. Ids are written one first time without any score to let the function create the related bubble.
Then we use the template to generate the chart:

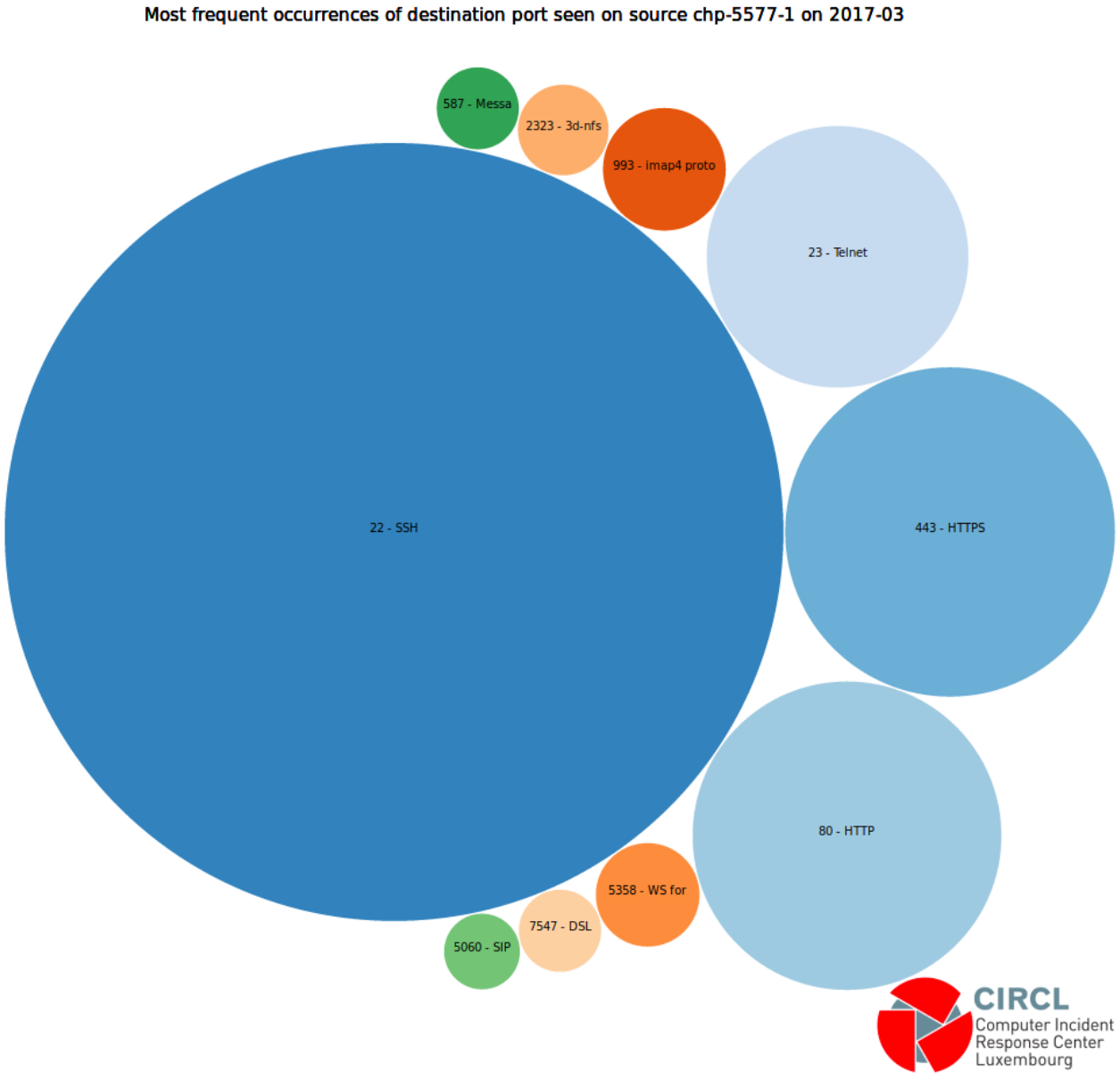

Figure 4.10: D3JS Bubble chart

Compared to the initial template[18], which only creates bubbles as we actually do, I added some content to display a logo, the title of the chart, and some variables used to define the file name, but we do not want to modify the template function itself, as it uses an online reference model as source build:

```
<script src="https://d3js.org/d3.v4.min.js"></script>
```

4.4 Additional functionalities

Besides the previous functionalities, we want to have other visualization possibilities, in order to get more information from our data. These additional visualizations require their own data structure or at least data from the usual structure to be filtered but storage operations are the same as the ones described in the previous section into the diagram. The difference of structure hence touch upon the Redis structure and storage pattern.
Let us speak about these changes is the next subsections presenting each new additional functions.

4.4.1 Initial Sequence / Acknowledgement Numbers over time

A first add-on utility is to plot Initial Sequence and Acknowledgement Numbers (respectively shortened with initials ISN and IAN) over time series in order to analyze how these numbers are distributed and potentially discover interesting patterns [19].
These patterns are defined by a succession of dots, showing lines with different directions that all have their own meaning:

- An horizontal line over the entire width of the plot, or at least not only appearing in a short period of time, and with numbers not equal to 0, is potentially representing Mirai packets.
- Horizontal lines over short periods potentially show attempts to spam a precise device with a big number of requests.
- Vertical lines, most of the time over the entire height, can be a sign of complete scans of the network.
- Diagonal line are representative of monotonic scans of the network.

Examples illustrating some of these special cases will be shown later on.

The first version of this kind of modules takes capture files as input to read each packet, extract ISN/IAN values and the related timestamp, and plot the whole content together. But fairly quickly difficulties appeared with this solution:

- Timelines are defined with the beginning and the end of each capture files, having more accuracy means testing each packet timestamp, and add more complexity
- Moreover timelines implied by capture file names are most of the time staggered with the actual timestamps in the packets.
- Recreating one graph means reprocessing one or several capture files with Tshark, while using a Redis structure consists of simple read requests contrariwise.

These kinds of problems make the function less flexible for users and harder to use with precise timelines. Modules using directly Tshark output to plot ISN/IAN values over time series still exist in the repository, but using Redis is the easiest way to have a reliable and precise timeline.

But as stated before, in introduction of this section presenting all the additional functionalities, a different Redis structure should be used. The usual structure does not save links between each packet timestamp and the ISN/IAN values, actually timestamp are not even stored in Redis. Here we need timestamps because they are mapped as the x-axis of the plot, when ISN/IAN are the y-axis.
A good solution is then to use timestamps as part of our Redis keys.
The interest of such a structure relies on the ease of reading keys: with the redis command '*keys*' it is pretty effortless to select keys corresponding to a defined pattern. And with timestamps inside the keys, it would otherwise be needed (if not impossible) to find another way to select the right keys.

By the way, in order to reduce our database memory used, it is possible to save only the required fields: Initial Sequence Number, Initial Acknowledgement Number, Protocol. The structure in this case is then different from the initial one, and looks like:

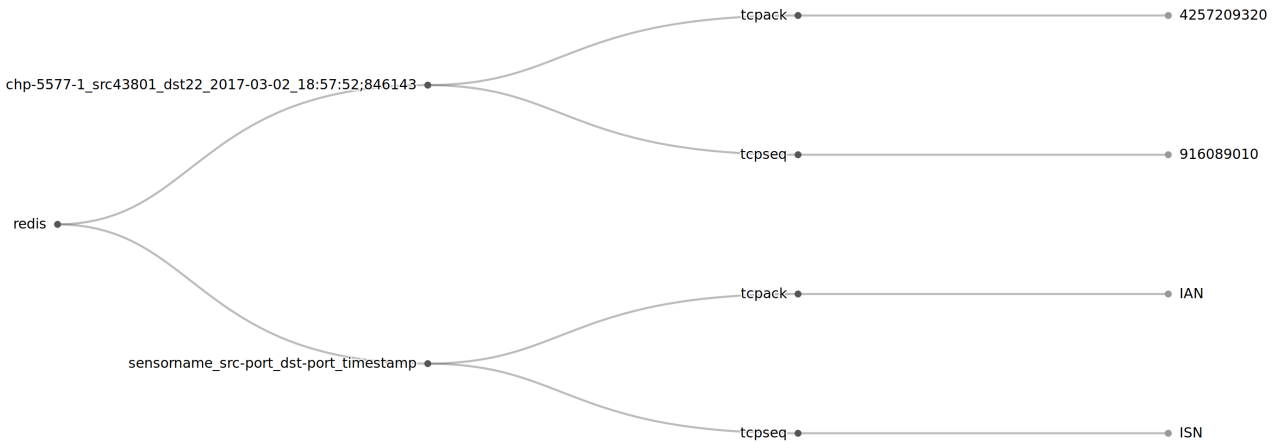

Figure 4.11: Redis structure for ISN

Ports can be used to filter data and plot only the values related to a precise destination or source port, this is the reason why they directly appear within the keys to make easier their selection, otherwise it is used to set the color of the plotted dots and make the difference between them.

Let us watch some of the possible examples:

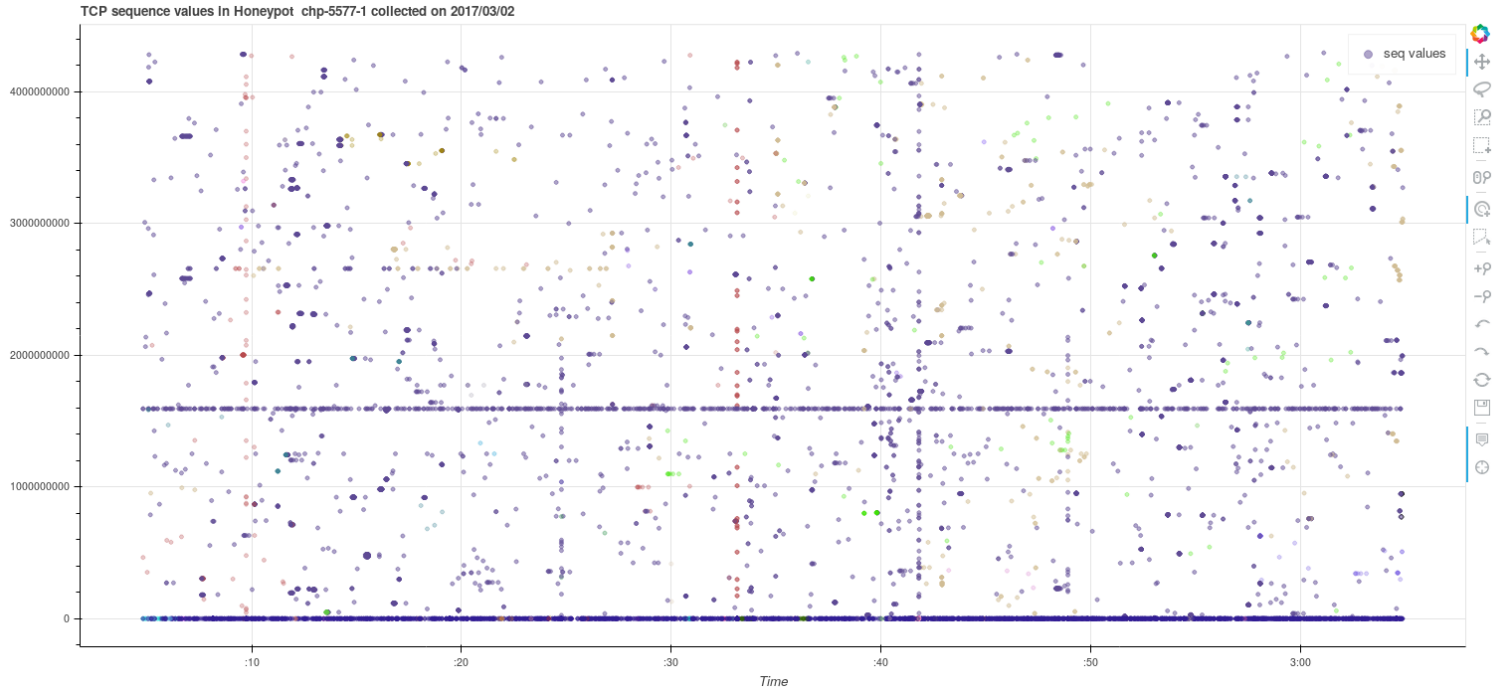

Figure 4.12: Basic example with Mirai - 1 hour timeline - directly from capture files

This example is the result of a module we built to generate a graphic with data picked directly from capture files, but in the general use case we want to use the modules storing data on Redis.

Figure 4.13: Example with horizontal and vertical lines - 5 minutes timeline

As timelines are precise in both previous and next examples, it means they have been generated with data from Redis.

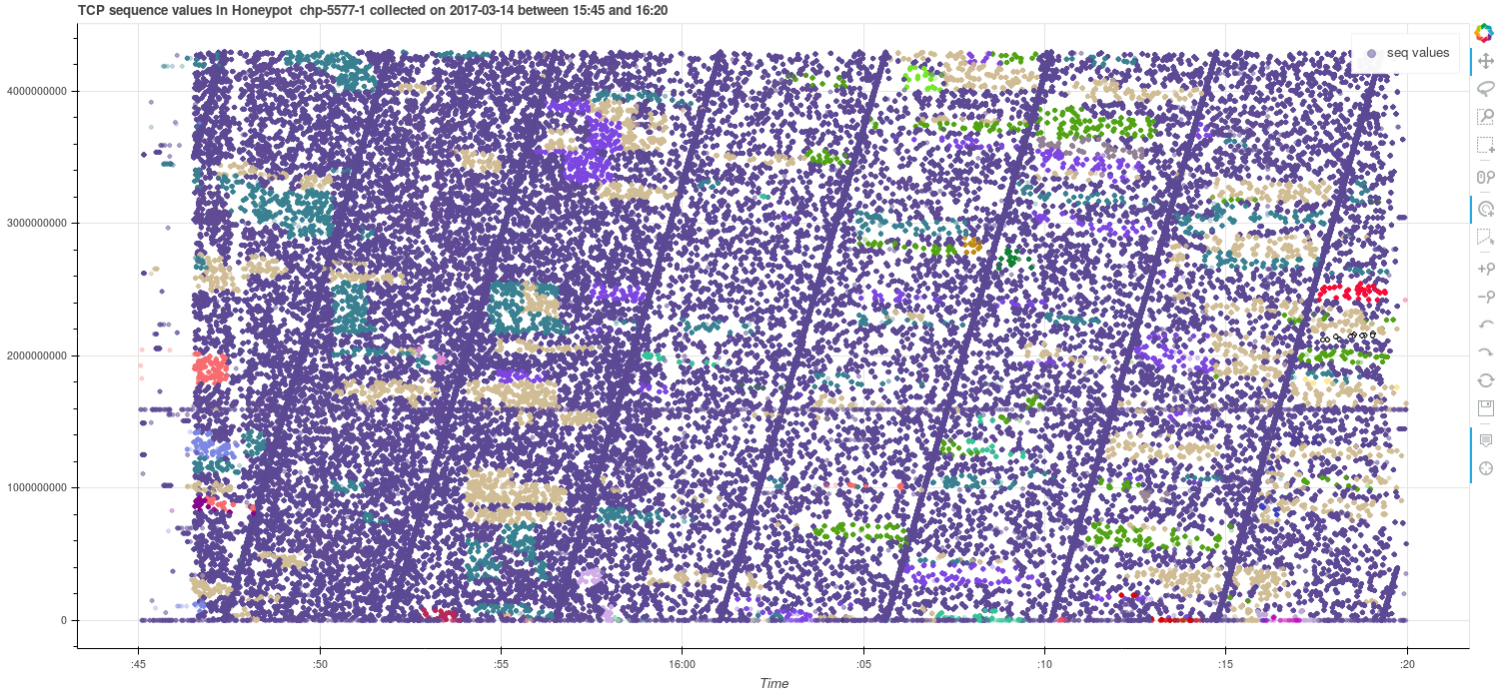

Figure 4.14: Example with monotonic scans - 5 minutes duration

On this last case, the amount of data displayed is really important, which illustrates the need to reduce datasets, whenever it is possible, but this part will be covered later within the next chapter dedicated to improvements. But before we reach this part, there is one utility we can mention which is kind of improvement and actually a trick to have a preview of each ISN graphic. As we just saw, some graphics are too heavy, with so much data that the plot is crowded with thousands or millions of dots, and in this particular case, web browsers have difficulties to load the graphic file.
It is then possible to generate a preview file of the plot with Phantomjs[20]. When we start implementing this function, it is not supported by Bokeh yet, and we need to call Phantomjs as a subprocess in our module, but after some updates, we do not even need to install it by ourselves with its dependencies as it is part of Bokeh utilities as an 'export' function.

As a result on picture is now generated every time an ISN graph is created, and for even more efficiency, we decide to build an index of all the plot preview, so users can have a general overview per month instead of going to each directory to see plots or graphs independently:

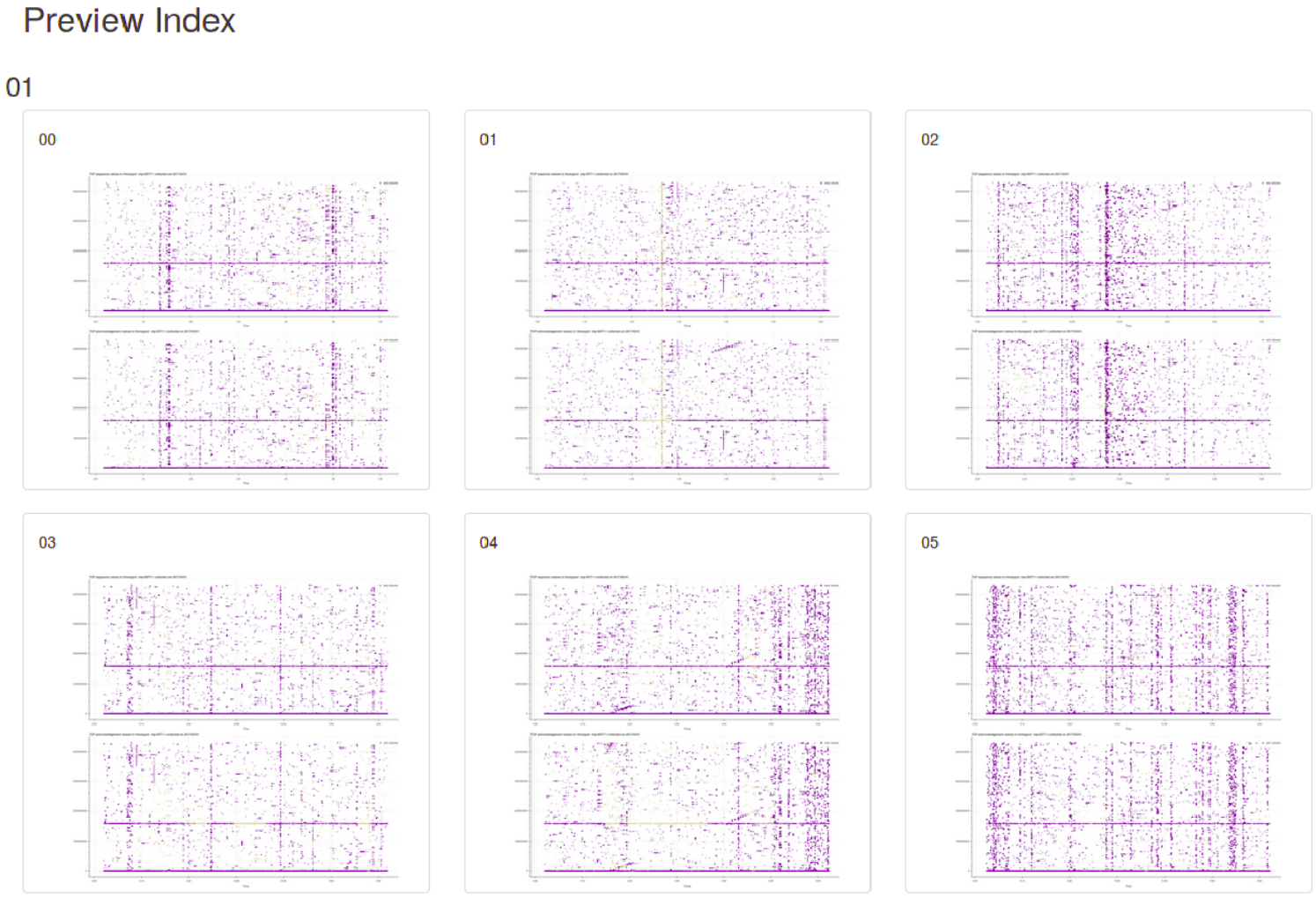

Figure 4.15: ISN graphs overview index

From this file, generated with a python module and using a Bootstrap[21] template, clicking in one of the previews redirects to the actual graphic. The entire file contains all the previews available for the corresponding month, on this example, one per hour every day.

4.4.2 Parallel coordinates

Basically, parallel coordinates are a common way of visualizing high-dimensional geometry and analyzing multivariate data. The aim is to show a set of point in an n-dimensional space, drawing n parallel lines, typically vertical and equally spaced. Each line is actually a polyline with vertices on the parallel axes. The positions of vertices on parallel axis correspond to the coordinates of the points.

With this kind of visualization [22], the aim is to see the repartition of, for instance, ip addresses over the days. There is by the way a drawback with this kind of visualization: it is not possible to find the value corresponding to each line once the plot is generated. Vertical axis are also designed to have their own scale, which means in our case we cannot compare days between each others.

These characteristics, specific to this representation, and because of our disparate type of data, make the plot look like the following:

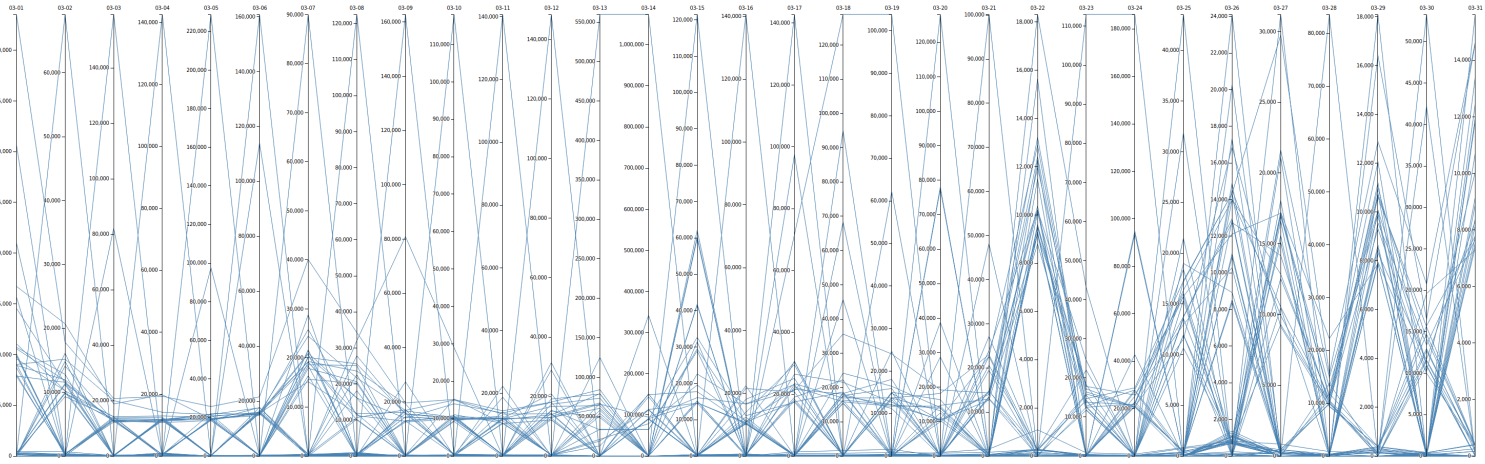

Figure 4.16: Parallel Coordinate - Example with destination ip addresses over March 2017

Each vertical axis is representing one day, with its own score for each line representing here scores of ip addresses. Even though a majority of vertices have most of the time low values, making visualization more difficult, we can anyway see the days with a lot of high scores, characterizing days with traffic from or towards multiple addresses.

As with bubble charts, the main task is to build the datafiles containing the scores of each lines displayed, by sending simple read requests to Redis in order to get all the needed values. Then, the template, even modified a bit, is still the same as the model provided within the creator's page. As scores of the fields are needed here, the initial Redis structure can be used.

This function is not one of the most important and relevant of the project, however source files will stay in the repository and can be used as a starting point for possible future improvements or modifications.

4.4.3 Represent Layer 2 information

From the beginning of the project, everything is processed within Layer 3 perspective, we are able to deal with ip addresses, protocols, ports, etc. But Layer 2 can make an interesting analysis subject too, such as parsing ARP information.
With this kind of visualization, it is possible to detect differences between the number of ARP requests and replies. As a reminder, when a machine does not know which other machine has a precise IP address, it send on the network a request to ask who has this address. Then the machine actually identified with the IP address in question sends a reply with its MAC address, so the machine which sent the request can note the association MAC address - IP address in its ARP table. This association makes direct redirection possible whenever the first machine receives a packet with the second machine IP address as destination address.

Two different kinds of information are here important in the Redis structure: having precise values

of some packets (ARP data on our example) like we have for ISN, and scores as the initial structure does. Hence in that case, Redis is organized like:

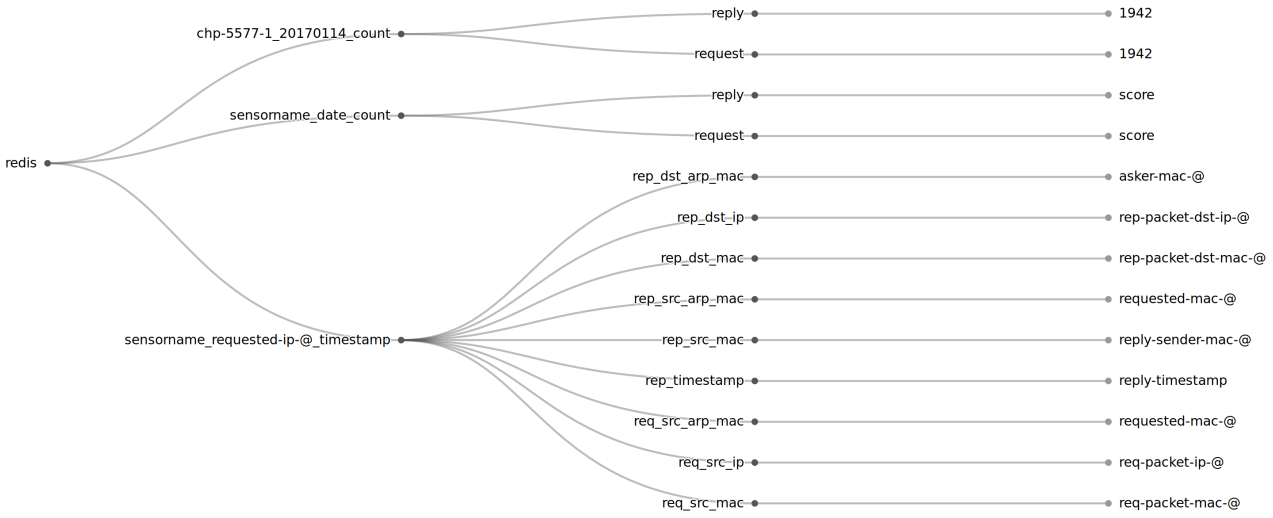

Figure 4.17: Redis structure for ARP

The first plot displaying scores thus looks like the following:

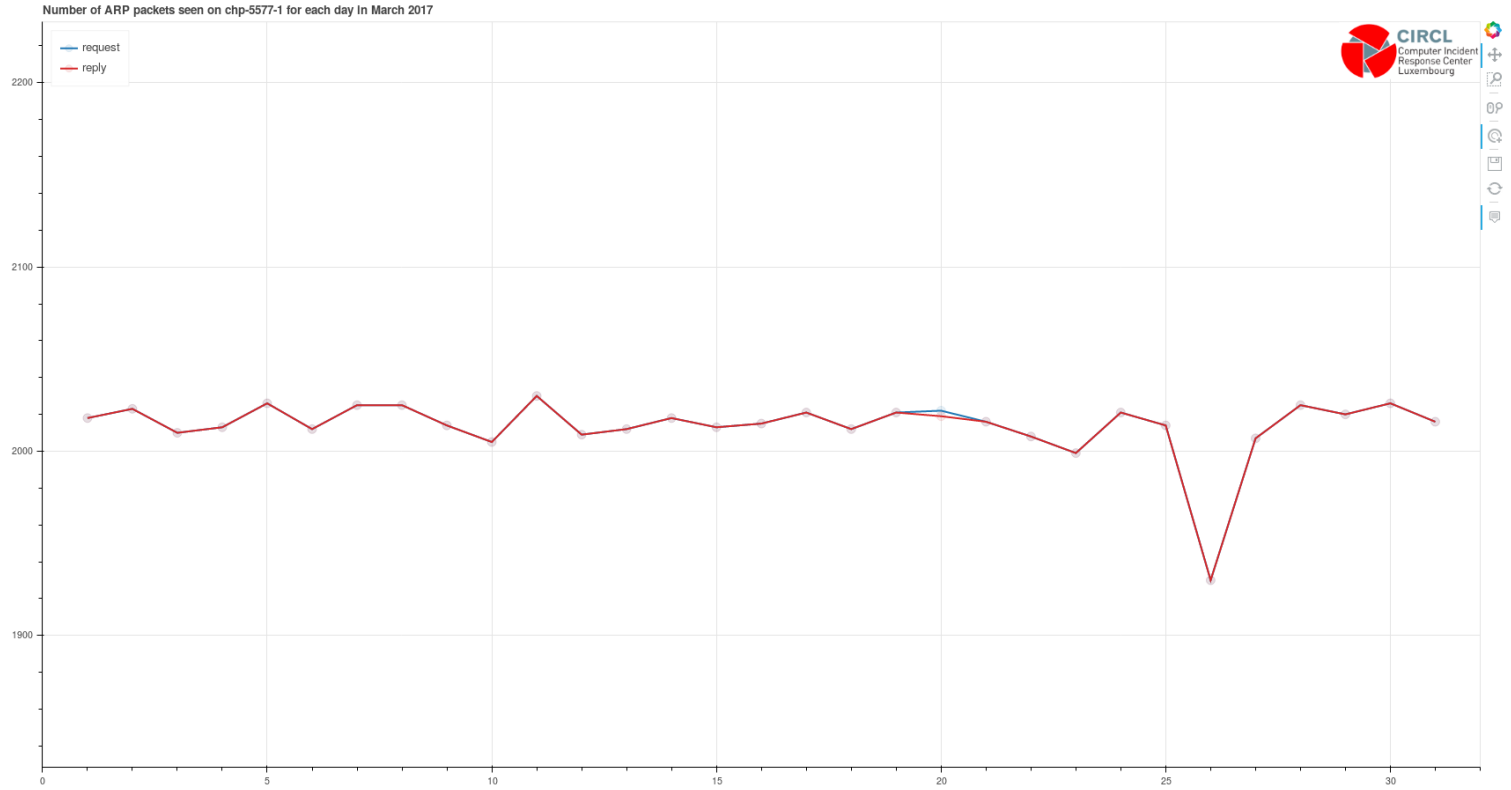

Figure 4.18: Count of ARP packets, requests and replies

With this example we do not observe suspicious cases from this point as both requests and replies numbers of packets are similar almost all the time in all the data sample we have to test our modules, except for March 20th 2017 as displayed just above, with a little difference of 3 more request packets. Even though this example shows that most of the time everything is in order regarding the number of ARP packets, visualization is however useful to detect when strange behaviors occur.
Then coming from the plot, it is possible to find the precise values stored in Redis, since visualize the difference directly on the plot is not so easy with only the eyes:

```
$ redis-cli -p 6378 zrevrange chp-5577-1_20170320_count 0 -1 withscores
1) "request"
2) "2022"
3) "reply"
4) "2019"
```

If we want to go further, using for instance Tshark commands in parallel, like in the example mentioned previously in the description of Tools used and Features, in order to count directly in the terminal the number of ARP packets, from the full month period, to find quickly the right file(s) involved in the difference:

```
$ ls Documents/home/circl/blackhole/archive/2017/03/20/chp-5577-1-2017032011*
  | parallel tshark -Tfields -e arp.opcode -r {} | grep 1 | wc -l
85
```

```
$ ls Documents/home/circl/blackhole/archive/2017/03/20/chp-5577-1-2017032011*
  | parallel tshark -Tfields -e arp.opcode -r {} | grep 2 | wc -l
82
```

'1' (right after 'grep') is the value of the field arp.opcode designating requests, when '2' are the replies. At this point we know the difference is within files between approximately 11:00 and 12:00, since 'chp-5577-1-2017032011*' identifies all the files with the name starting with this string, which means having as timestamp '11' as hour value. Then, we find our 3 additional request packets in one single file:

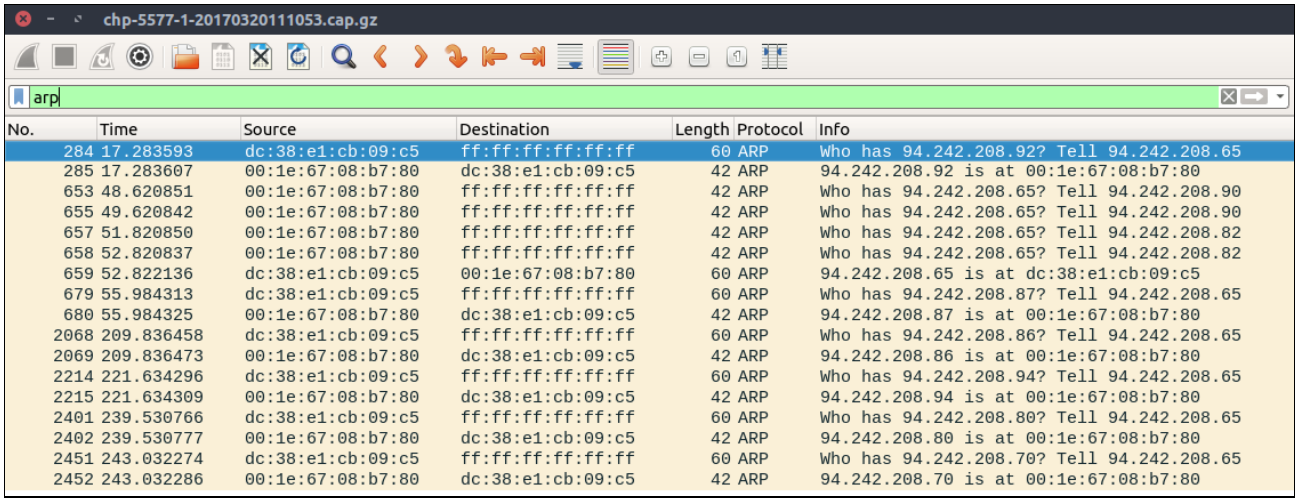

Figure 4.19: Visualization of arp packets with Wireshark

But these pieces of information provided by only an ARP packet count are quite limited.
Another type of information which is interesting about ARP packets is the repartition between IP addresses questioned and MAC addresses of the machines replying to the requests. Thus it is possible to detect when one single machine is replying with its MAC address to multiple different IP address requests. For this function, Circos can be used.
This software providing visualization over circular layout is usually used to show relationships between genotype and other medical stuffs. This is the functionality we need here to point out relations between MAC and IP addresses, like in this example:

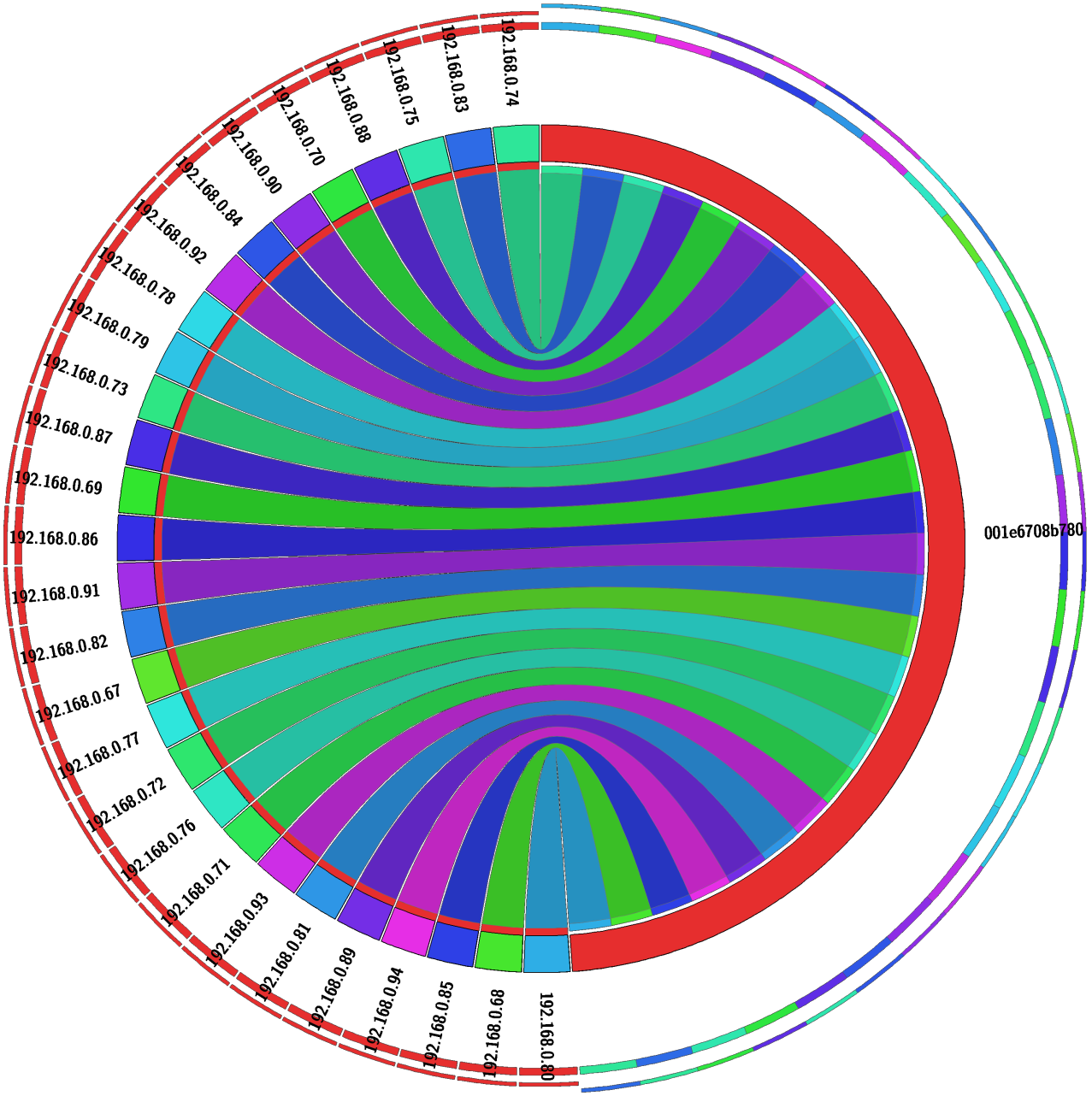

Figure 4.20: Circos circular layout

Both Alexandre DULAUNOY and Cédric BONHOMME suggested me Circos, since Cédric has already used it in the past. The same model is available with D3JS as well.
We can see from this data sample that only one machine replied with its MAC address to all the requests for IP addresses, which is usual on the kind of honeypot (black hole) where data comes from. But more generally it is a powerful tool to see the distribution of MAC addresses.

In order to build this kind of graphic, the aim is to generate the data matrix used then by Circos software to build the layout. This matrix only has to follow the model:

```
              | MAC address 1 | MAC address 2 | ...
IP address 1  |     n 1_1     |     n 1_2     | ...
IP address 2  |     n 2_1     |     n 2_2     | ...
    ...             ...             ...
```

Lines and columns can be switched, as long as scores are referenced for each combination of 1 IP address with 1 MAC address. Circos is then managing every visual aspects, described in configuration files, as well as generation of the output layout.

4.5 Improvements

One of the most recurrent problem encountered during tests is the size of data. It is common to have captures files with a really high amount of packets, which makes the files size grow up really fast.
This is directly impacting our processing time as well as the memory used to store data on our Redis database, it is hence up to us to find solutions to reduce as much as possible the impact of our modules, since capture files size can only be reduced in one precise case, which will be presented later in this section. Let us start with this first kind of problem to fix, while other improvement approaches will be explained afterwards.

4.5.1 Reduce storage size and processing time

As just briefly stated in introduction to improvements, the higher captures files size is, the slower our modules are, because of the increased amount of data to deal with. And this feature is actually directly impacting both memory size and time needed to process capture files.
To handle this kind of constraint, it is unmanageable to act directly on the files since we cannot take the risk of losing information. The first proposition was to find something working like Redis, with a better memory management, such as Ardb[23]. The problem with this tool is its really slowness, compared to Redis, the memory size management is more efficient and it could be interesting as having an active Redis instance consumes RAM, but the processing time is pretty longer. Processing time is important as well because in a real situation, we need our modules to be able to work in the environment briefly described in the section 'Subject Presentation', and to not exceed the next capture file generation timeline.
The other use case in practice is to process or reprocess manually some dataset if needed, regardless of the previous usage. As a result, the alternative is to find how to filter what is really needed, and information that can currently be left behind.
The idea here is not to exclude data, but to select what should be parsed and what does not need to be processed at the moment. Then only parsed and normalized data that has been filtered is stored.

This is a good way to reduce json datafiles size, as well as Redis memory size.

Since Redis database is the keystone of our storage model, reducing as much as possible its size is an important objective. Flushing one version with a special dataset, in order to store another data sample is an option considered as part of a regular use case, it is hence possible to apply filters as we just stated, to have not only one final database but temporary data samples used to generate the graphics, and replaced then by other samples, over and over again if needed.

Filtering fields to store

The first possibility is to select only some fields to be stored. Basically around 13 or 14 (depending if ipsumdump or tshark is used) fields are used without any filter in the usual structure storing scores, but in most of the cases, the fields commonly watched are source & destination ports, consequently there is a lot of data we do not need to process in that case.
For this purpose and aside from all the code changes which are involved, the aim is to let users define if they want to filter fields or not. As mentioned at the beginning of this part focusing on the development of my subject, Argparse is the tool in question to manage arguments we let users define. Then only these fields data are stored both in json files and Redis, the unique exception is in the case of separation by protocol, another improvement which will be presented later on.

Filtering data with a Berkeley-like Packet Filter

In addition to a filter on fields, we may want to select data as we could do with a BPF. The aim is to retain only the packets satisfying the condition defined with the filter. Here we speak about BPF-like because as stated previously Tshark provides a filter utility, but contrary to a real BPF, it only is a display filter. Even though the principle is the same, precise Wireshark/Tshark fields syntax should be used by users to exploit this convenience. To avoid any confusion, this function argument is defined as a "Tshark Filter" rather than "BPF" or anything suggesting it is a real Berkeley Packet Filter.

As a result, it is for instance required to specify 'tcp.dstport' or 'udp.dstport' to reach the values of destination ports, dealing with the confusion that these 2 fields cover all the different protocols: with some unusual protocols (i.e that are not some of the most common ones like tcp, udp, icmp), both fields can display values, depending on the case. This is the difference with a real BPF, where fields are defined with its own syntax and implementation.

Reduce tshark processing

Within the presentation of capture files parsers, 3 different tools have been presented, 2 of them have already been presented, and now we pay more attention to the last one: Tcpdump. As previously said, the interest is to have a really efficient and fast packets selection with a BPF, which is something Tcpdump is really performing at. This utility can be used to restrict the number of packets in capture files to the ones responding to the filter.
More generally here the aim is to read an input capture file, which contains all the data, apply a BPF, and write the result in a new capture file. This resulting file is then easier to parse with Tshark, which will, as always, read all the packets, which are all corresponding to the BPF just applied. Thus all

data processed is part of the useful selected data we want: nothing is processed to be left out.
This operation is done pretty easily with:

```
tcpdump -r input_capture_file arp -w output_capture_file
```

One of the most relevant examples in favor of using Tcpdump and its efficiency with BPF is the case of the ARP packets. It always depends on the case, but usually the amount of ARP packets is insignificant compared to the thousands ones in total, hence it is way much faster and more efficient read one first time a really light data volume, to write it in another file, to then read it a second time with Tshark, than processing more than 90% of data that will not be stored or even display by Tshark.

By the way, the reason why Tcpdump is not used directly to parse this data in order to store it relies in the output typography provided by this parser:

```
12:11:09.981690 ARP, Request who-has ip-static-94-242-208-92.server.lu
        tell ip-static-94-242-208-65.server.lu, length 46
12:11:09.981704 ARP, Reply ip-static-94-242-208-92.server.lu is-at
        00:1e:67:08:b7:80 (oui Unknown), length 28
```

This kind of output is not as accurate and parsable as a Tshark output and all its possible fields, like:

```
frame.time_epoch eth.src eth.dst arp.src.proto_ipv4 arp.dst.proto_ipv4 arp.src.hw_mac arp.dst.hw_mac arp.opcode
1490004669.981690000 dc:38:e1:cb:09:c5 ff:ff:ff:ff:ff:ff 94.242.208.65 94.242.208.92 dc:38:e1:cb:09:c5 00:00:00:00:00:00 1
1490004669.981704000 00:1e:67:08:b7:80 dc:38:e1:cb:09:c5 94.242.208.92 94.242.208.65 00:1e:67:08:b7:80 dc:38:e1:cb:09:c5 2
1490004701.318948000 00:1e:67:08:b7:80 ff:ff:ff:ff:ff:ff 94.242.208.90 94.242.208.65 00:1e:67:08:b7:80 00:00:00:00:00:00 1
1490004702.318939000 00:1e:67:08:b7:80 ff:ff:ff:ff:ff:ff 94.242.208.90 94.242.208.65 00:1e:67:08:b7:80 00:00:00:00:00:00 1
1490004704.518947000 00:1e:67:08:b7:80 ff:ff:ff:ff:ff:ff 94.242.208.82 94.242.208.65 00:1e:67:08:b7:80 00:00:00:00:00:00 1
1490004705.518934000 00:1e:67:08:b7:80 ff:ff:ff:ff:ff:ff 94.242.208.82 94.242.208.65 00:1e:67:08:b7:80 00:00:00:00:00:00 1
1490004705.520233000 dc:38:e1:cb:09:c5 00:1e:67:08:b7:80 94.242.208.65 94.242.208.82 dc:38:e1:cb:09:c5 00:1e:67:08:b7:80 2
1490004708.682410000 dc:38:e1:cb:09:c5 ff:ff:ff:ff:ff:ff 94.242.208.65 94.242.208.87 dc:38:e1:cb:09:c5 00:00:00:00:00:00 1
1490004708.682422000 00:1e:67:08:b7:80 dc:38:e1:cb:09:c5 94.242.208.87 94.242.208.65 00:1e:67:08:b7:80 dc:38:e1:cb:09:c5 2
1490004862.534555000 dc:38:e1:cb:09:c5 ff:ff:ff:ff:ff:ff 94.242.208.65 94.242.208.86 dc:38:e1:cb:09:c5 00:00:00:00:00:00 1
1490004862.534570000 00:1e:67:08:b7:80 dc:38:e1:cb:09:c5 94.242.208.86 94.242.208.65 00:1e:67:08:b7:80 dc:38:e1:cb:09:c5 2
1490004874.332393000 dc:38:e1:cb:09:c5 ff:ff:ff:ff:ff:ff 94.242.208.65 94.242.208.94 dc:38:e1:cb:09:c5 00:00:00:00:00:00 1
1490004874.332406000 00:1e:67:08:b7:80 dc:38:e1:cb:09:c5 94.242.208.94 94.242.208.65 00:1e:67:08:b7:80 dc:38:e1:cb:09:c5 2
1490004892.228863000 dc:38:e1:cb:09:c5 ff:ff:ff:ff:ff:ff 94.242.208.65 94.242.208.80 dc:38:e1:cb:09:c5 00:00:00:00:00:00 1
1490004892.228874000 00:1e:67:08:b7:80 dc:38:e1:cb:09:c5 94.242.208.80 94.242.208.65 00:1e:67:08:b7:80 dc:38:e1:cb:09:c5 2
1490004895.730371000 dc:38:e1:cb:09:c5 ff:ff:ff:ff:ff:ff 94.242.208.65 94.242.208.70 dc:38:e1:cb:09:c5 00:00:00:00:00:00 1
1490004895.730383000 00:1e:67:08:b7:80 dc:38:e1:cb:09:c5 94.242.208.70 94.242.208.65 00:1e:67:08:b7:80 dc:38:e1:cb:09:c5 2
```

Figure 4.21: Arp packets with Tshark - From left to right: timestamp, physical source address, physical destination address, sender IP address, receiver IP address, sender MAC address, receiver MAC address, operation code (1 if request, 2 if reply)

This kind of trick to reduce capture files size and also the tshark workload is a piece of advise given by Gérard WAGENER, who usually works with heavy datasets and uses it to consequently reduce his processes timelines.

4.5.2 Functions improvements

Once processes workload has been lightened, another way of improvement is to provide an additional functionality over all the modules to have a more relevant data classification: separate each protocol

so every one of them can be compared with each others.
On the other hand readability can be improved as well by using classes for some modules calling functions from other modules: this is a way to avoid having functions with too many arguments.

Separation with protocols

With no distinction between each protocol, data from packets is stored but there is no way to know about ports for instance is they are tcp ports or udp ports, since they are incremented the same way regardless of the protocols. Here the aim is then to keep links between data of each packet and the protocol concerned, to make separated data samples concerning each one only one protocol.

This kind of modification has no impact on json files, which makes them flexible and reusable no matter if we want protocols separation or not. The goal is to change partially Redis structure to include information concerning protocols, in order to make the separation. Therefore, from the initial structure, we simply add the protocol value to the keys pattern: *sensorname:protocol:timestamp:field*

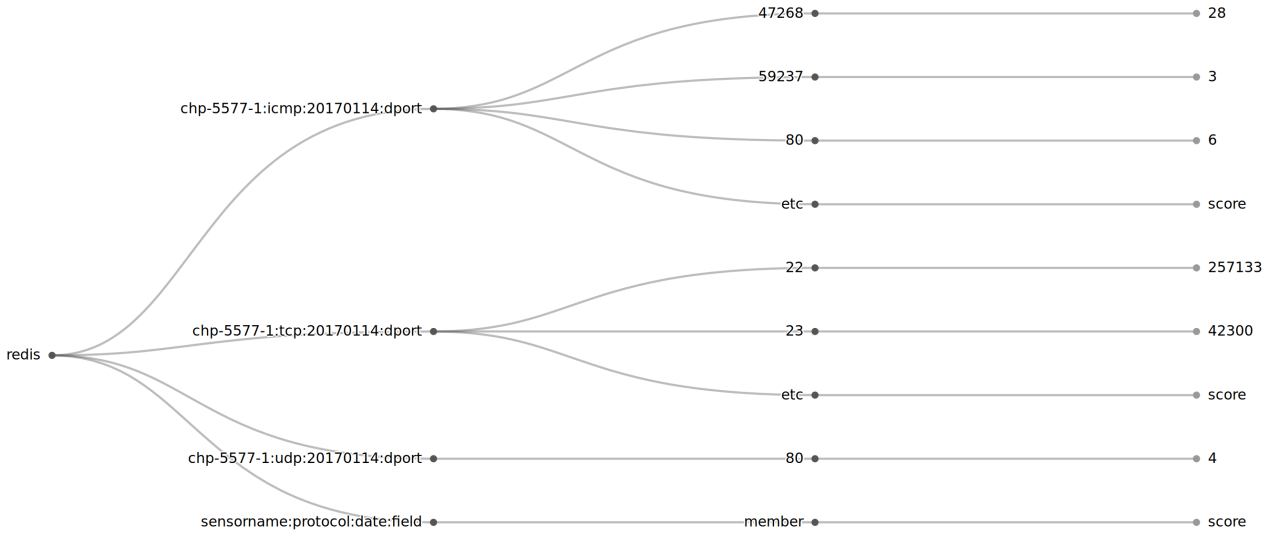

Figure 4.22: Redis structure with separated protocols

As a result, for each protocol sample it is possible to compare values and plot each one independently:

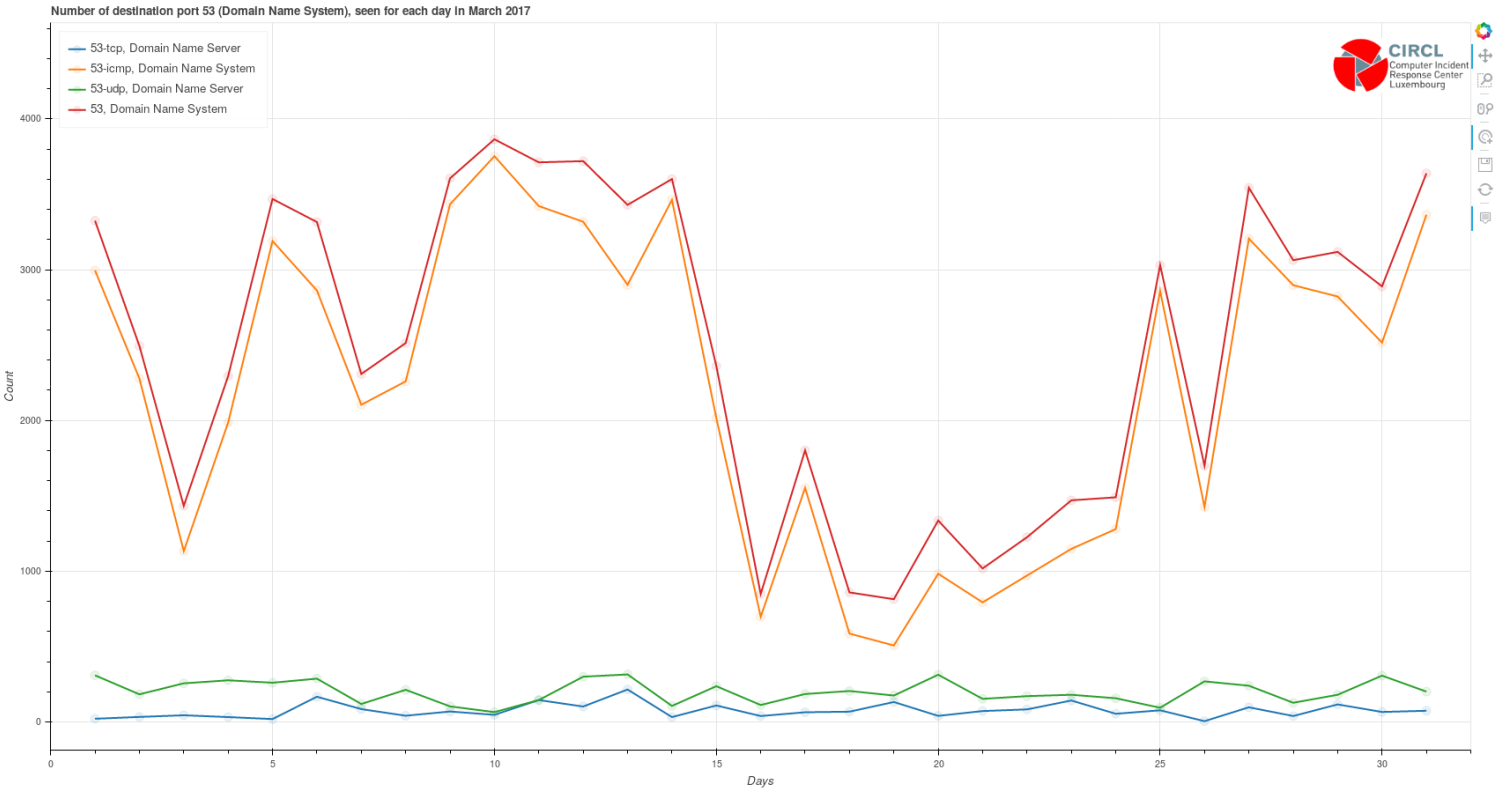

Figure 4.23: Bokeh - Example with separated protocols

Here is plotted the usual case: the amount of different possible graphics is potentially infinite since from this plot we could make 4 different ones, actually each one containing each line, thus generating plots with all the lines together is a better solution to keep having a reasonable amount of different files. One plot with 4 lines is better for files management than 4 files containing each single line, and it is possible to generate these 4 plots too anyway, but this kind of feature or specification will be explained more in details in the next part about user-friendliness.

More readability with classes

Readability is not insignificant, thus functions with a lot of arguments could be pretty hard to understand at first sight, and also not really convenient to be called from other modules, as well as reused in potential new modules of functions. Consequently readability improvement is important, first versions of the new modules created to provide functionalities included in the internship subject have been written with simple function calls first, which is efficient when there is only few arguments and the functions are only called within their own module. At the moment calls are coming from other modules, every argument should be verified to ensure its value is the one expected and with the expected format and type, hence it adds complexity for developers to create new content using these functions, as well as to understand the arguments.

The solution to make external calls and re-usability smarter is then to use the features directly provided by python language: classes. Transforming our modules with simple procedures into modules

with classes increases visual readability, and also re-usability as well as flexibility. By the way using classes provides approximately the same level of performance than procedures, or at least even though classes declaration looks like there is more code and more complexity, python language is built to ensure complexity is equivalent.
Consequently, the difference with some functions are considerable, and from a function which looks for instance like:

```
def process_all_files(red, source, date, field, limit, skip, outputdir,
                      links, gen, logofile, ck, lentwo):
    ...
```

we make it mode readable, more easy to call from outside the module:

```
def process_all_files(self):
    ...
```

All the previous arguments are actually on this example class arguments thus there is no need to specify them with the function declaration, they are included by implication with the only argument '*self*'. In fact, the only thing to do is to replace all the class arguments by '*self*' whenever one of them is needed in one procedure, plus potential external arguments, and to not use it as long as a procedure is only dealing with non-class arguments.

Code profiling

Another good practice to improve efficiency of a program is to profile the different functions. A profile is a set of statistics that describes how often and for how long various parts of the program are executed. To do so, the Python Profilers[24] 'profile' and 'cProfile' are python standard library tools providing a profiling interface to proceed such kind of operations. Then with the 'pstats' module, it is possible to analyze the statistics resulting from the profiling.

To illustrate this utility, we use it on one process calling multiple times some of our modules, that seems unusually slow right after nodules transformation from procedures to class:

```
In [5]: p.sort_stats('cumulative').print_stats(50)
Fri Jul 28 15:48:13 2017    /home/student/test_class.txt

         1274204231 function calls (1247725528 primitive calls) in 826.496 seconds

   Ordered by: cumulative time
   List reduced from 5419 to 50 due to restriction <50>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
    829/1    0.013    0.000  826.498  826.498 {built-in method builtins.exec}
        1    0.001    0.001  826.498  826.498 ./export_csv_all_days_per_month.py:3(<module>)
        1    0.080    0.080  825.484  825.484 ./export_csv_all_days_per_month.py:115(process_all_files)
     1550    5.720    0.004  728.105    0.470 ./export_csv_all_days_per_month.py:99(generate_links)
     1550    8.072    0.005  722.308    0.466 ./bokeh_month.py:59(process_file)
  6971831   18.611    0.000  340.500    0.000 /usr/local/lib/python3.6/dist-packages/redis/client.py:566(execute_command)
    32552    0.199    0.000  311.492    0.010 /usr/local/lib/python3.6/dist-packages/bokeh/model.py:435(references)
    32552    2.446    0.000  311.293    0.010 /usr/local/lib/python3.6/dist-packages/bokeh/model.py:26(collect_models)
```

Figure 4.24: Profiling module with classes

We can see the module makes almost 7 million Redis calls, which seems to be the reason of the slowness, since the majority of other function calls that are as plentiful as these ones concern functions we are

not really able to manage, like Bokeh calls. Let us compare with the same process before module transformation into classes:

```
In [4]: p.sort_stats('cumulative').print_stats(50)
Fri Jul 28 15:43:50 2017    /home/student/test_without_class.txt

         463655804 function calls (454667002 primitive calls) in 314.712 seconds

   Ordered by: cumulative time
   List reduced from 5404 to 50 due to restriction <50>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
    829/1    0.015    0.000  314.714  314.714 {built-in method builtins.exec}
        1    0.001    0.001  314.714  314.714 ./export_csv_all_days_per_month.py:3(<module>)
        1    0.065    0.065  313.617  313.617 ./export_csv_all_days_per_month.py:97(process_all_files)
     1550    1.616    0.001  206.404    0.133 ./export_csv_all_days_per_month.py:82(generate_links)
      431    1.705    0.004  204.756    0.475 ./bokeh_month.py:40(process_file)
  2769001    7.829    0.000  167.042    0.000 /usr/local/lib/python3.6/dist-packages/redis/client.py:566(execute_command)
      124    1.707    0.014   89.310    0.720 ./export_csv_all_days_per_month.py:24(process_score)
     8646    0.058    0.000   88.327    0.010 /usr/local/lib/python3.6/dist-packages/bokeh/model.py:435(references)
```

Figure 4.25: Profiling module with procedures

Profiling statistics here indicate the same process with procedures in modules and no classes, is 2.5 times faster, as well as there is 2.5 times less Redis calls, which consolidates the idea that Redis calls are the reason of the slowness.
With 2.5 times more calls within something supposed to do the same operations is the sign that one or several functions are called unnecessarily. Moreover, looking to the line just before the one concerning Redis, there is another difference of function call number: one of our module is called 431 times usually, when it is called 1550 times once classes are used.

This analyze helps use finding the precise part of the program that is responsible for the extra functions calls, in order to correct the performance issue. It also gives the indication that Redis calls could be reduced as well. Following that guess, the idea is to change the way we access to information in Redis:

```
if red.exists(redisKey): ...
```

Here is the initial way to know if some values should be included in a plot, depending if the key defined actually exists. This test should be done for every key that can possibly define data stored in our current dataset. But there is another way to select directly all the keys identifying the concerned data:

```
for keys in red.keys(pattern): ...
```

With the first version, we test if a precise key is on our Redis dataset, whereas the second command is actually selecting all the keys corresponding to the pattern. Redis receives on request every time the first command is executed, with potential negative calls. With a 'keys' command, Redis is only requested one time, and it returns all the keys at the same time.
Then operations to read scores and save them in order to generate plots remain identical, but we save a huge time here with this review. Adding everything together, we finally reduce consequently the number of requests, dividing it actually almost by 2, and the processing time too since those two characteristics are depending one from each other.

```
In [5]: p.sort_stats('cumulative').print_stats(20)
Mon Jul 31 15:28:11 2017    /home/student/export_csv_all_days_per_month.profile

         359176333 function calls (350030119 primitive calls) in 231.649 seconds

   Ordered by: cumulative time
   List reduced from 5419 to 20 due to restriction <20>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
    829/1    0.014    0.000  231.650  231.650 {built-in method builtins.exec}
        1    0.001    0.001  231.650  231.650 export_csv_all_days_per_month.py:3(<module>)
        1    0.081    0.081  230.627  230.627 export_csv_all_days_per_month.py:111(process_all_files)
     1550    1.771    0.001  135.863    0.088 export_csv_all_days_per_month.py:96(generate_links)
      406    0.344    0.001  134.060    0.330 /home/student/git/test_potiron/bin/bokeh_month.py:56(process_file)
  1414417    3.839    0.000   92.972    0.000 /usr/local/lib/python3.6/dist-packages/redis/client.py:566(execute_command)
     8740    0.061    0.000   86.252    0.010 /usr/local/lib/python3.6/dist-packages/bokeh/model.py:435(references)
     8740    0.692    0.000   86.190    0.010 /usr/local/lib/python3.6/dist-packages/bokeh/model.py:26(collect_models)
```

Figure 4.26: Profiling module with classes, after code review

4.5.3 Improve user-friendliness

Finally with improvements, users should be considered as well. Utilities are functional and even though modules fulfill their role of storing data, displaying graphics and so on, users experience is not really satisfying whether they have to call manually for instance one command for every graphic they want to create, or open by hand every graphic searching them independently in their directory. In order to simplify these tasks, we want to complement our modules with user-friendly utilities, supporting even more interactivity on graphics, and adding some automation in processes.

Links between graphics

Linking graphics together is a first step to users support. As a quite large number of plots and charts are possibly created to cover the a possibly even larger panel of values, information or specific data to display concerning the usual functionalities presented earlier, when someone is observing a plot showing the scores of a specific value over a month for instance, we do not want them to search in the potential big amount of files to have a look at the daily top scores for the same kind of values, the aim is to allow them to click on the score they want, and the bubble chart concerning the day and the field involved appears.
For this purpose, the idea is to put links on graphics. Consequently, transparent dots are added into our Bokeh plots, allowing a visualization of these links. In order to make visualization of the links easier, we display fulfilled dots here, but transparency is implemented in the real case.

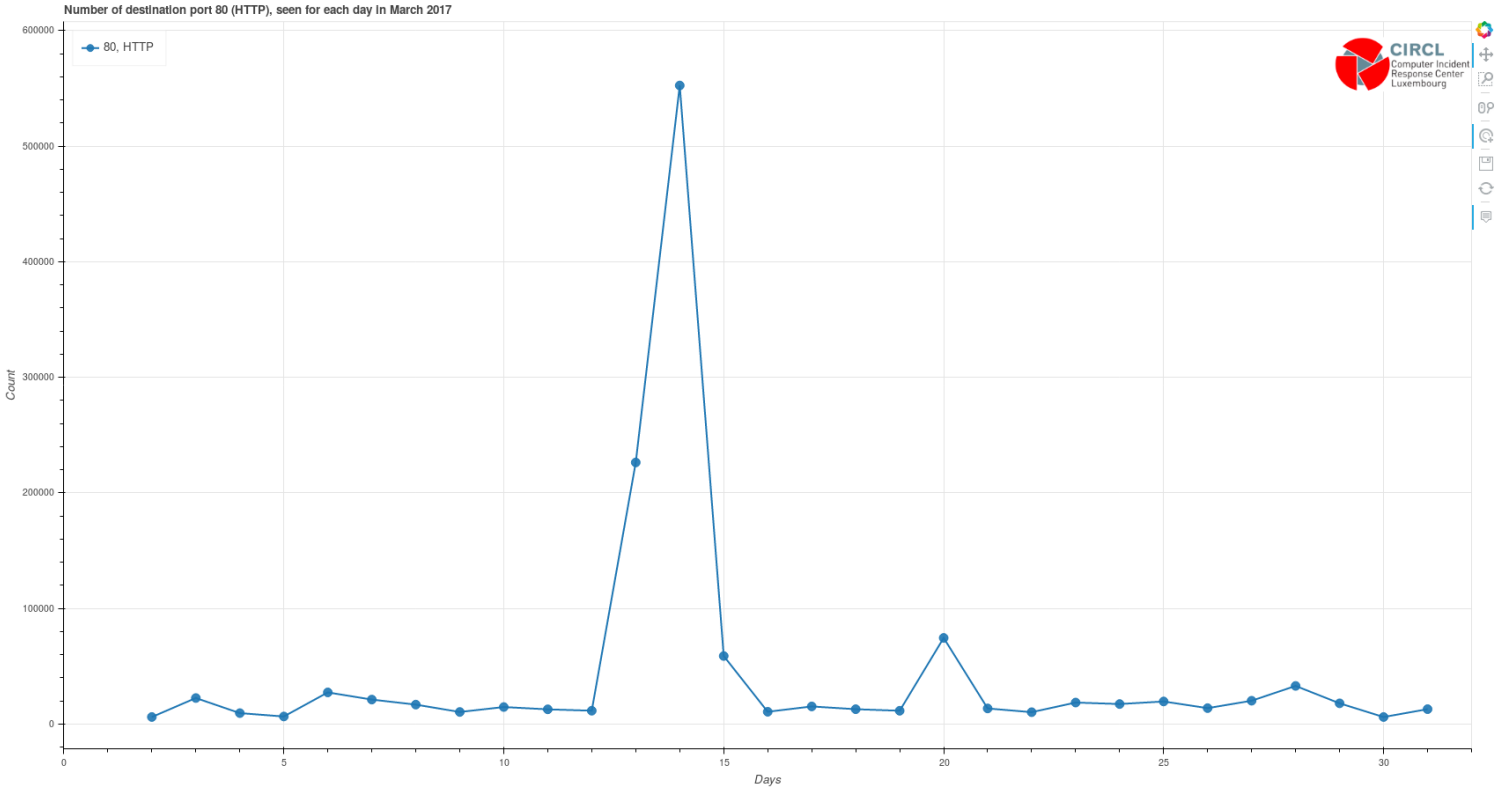

Figure 4.27: Bokeh plot with links

Then clicking on one of the dots opens the chart corresponding to the highest scores of the same field (in this example destination ports) for the corresponding day.
But redirection from bubble charts to a Bokeh plot should therefore be implemented too, as a consequence we simply add into every bubble a link as well. Since bubble, as dots or every round object, are more likely to make users click intuitively on them, our D3JS bubble charts do not receive visual modification to support links, they only do support it with an additional attribute within their definition making the redirection. Thus we have links working from the two sides.

Auto-generation

With links, problems concerning the need to search manually each graphic in its own directory is fixed, but having plots linking to tens of bubble charts, each one of them with 10, 20 or more bubbles, each one representing one value of a field with a link to the plot showing this value, there is potentially a huge amount of graphic files that should be generated to support all the links.
To counter this problem, an optional argument is created within modules generating Bokeh plots or D3JS bubble charts, in order to allow users to define if they want the module to generate automatically all the needed graphics in order to avoid links redirecting to nonexistent files. This utility is one of the main reasons why classes are better than procedures on the involved modules: generating additional graphics is done with calls to the corresponding modules, within the current module. This is basically the reason of, for instance, the 1550 calls to 'export_cvs_all_day_per_month' module and 406 calls to 'bokeh_month' module, you can find in Figure 4.22 we saw earlier, showing profiling report statistics

Provide a virtual environment with Potiron and its dependencies installed

Lastly, with the idea of a free distribution, including every requirement satisfied and every dependency installed, it is possible to proceed as it is done for MISP [25], with Vagrant, to automatically deploy potiron within a virtual environment, directly accessible through SSH connection with Vagrant utilities, as well as from VirtualBox.
Here we get helped by Cédric BONHOMME again, who created the structure of the same utility as he did for MISP [26], where we only need to modify the installation script to match with potiron requirements. This function will probably need further tests regarding physical resources needed to run potiron modules, and updates, but it currently provides a functional and up-to-date program.

4.6 Remaining tasks

Considering functionalities currently implemented, there are still few modules needing some updates, which could actually be updates on current utilities, for instance in order to improve them, or adding new ones. A non exhaustive list of remaining tasks would thus be:

- Allow users to define different timelines on bokeh graphics, as for example choosing to plot data from the last 6 months, or instead of a full month, only few days.
- Find a way to open files through links without opening a new browser tab (it does not already not from bokeh to D3JS)
- Modify parallel coordinates graphics in order to display which pieces of information are plotted
- Separate ISN values in Redis by IP addresses, and potentially select some IP to include in the database while the others will not be stored (as it is done with filters for the usual Redis structure)
- Try to build circular layout as Circos does, with a D3JS template to compare to versions and their flexibility regarding configuration
- Support other types of Layer 2 data than only ARP packets
- Try to reduce ISN graphics size, for instance by using a tool like 'Datashader', that shows good performances with big datasets.

Some of these tasks and potential other ones can actually be implemented during the period remaining before the end of the internship, which is obviously reserved for final tests and debugging as well.

5. Conclusion

The internship is the first I made concerning cybersecurity problematics such as forensic. As we saw, more specifically we dealt with programming, at the same time we saw data mining, normalization and visualization. By the way, having such a long period gives the chance to work on longer projects, with professional dimension and requirements.

I am pretty happy the way this internship took since it was an occasion to look back on a lot of different knowledge, as well as discovering several tools and methods helping me on my work. As seen with the presentation of all the functionalities, the variety of tasks to perform was as wide as interesting to improve different types of skills and knowledge.

In a more human dimension, I found a friendly environment from the first day, with teamwork oriented colleagues, taking care of each others and being always available to give any piece of advice if needed. I had pleasure working on this kind of context, being involved in a lot of conversations about many different security subjects including some parts of my subject, and acquiring knowledge for general purpose but also to get guidelines about my tasks.

Concerning the correlation with the educational background I got on Master's Degree, a lot of knowledge from the entire panel of courses is always useful as it gives some general approach of information security. But mode precisely, I used almost the entire content of the lessons we had with Alexandre DULAUNOY and Raphaël VINOT.
Many of my tasks use tools such as Redis, Tshark and command lines to process capture files, just like we saw during these lessons. In this case, I got back into using the tools, reading the lessons which give pretty good bases of classification, honeypots, as well as Redis and Tshark.

To put everything in a nutshell, I showed a real interest concerning my internship, the subject and the different tasks I worked on. Both work environment and everything I did throughout my internship period were really interesting.
On the other hand, I hope the tasks I carried out gave satisfaction to my supervisors, and were helpful to improve data classification and visualization.

Bibliography

[original project] Initial files, basis of the internship - https://github.com/chrisr3d/potiron/tree/913eab2a02f41eb5f264974974c8870c110a0c66

[1] MSc Student Internship Position (HONEYPOT-01) - Available at https://www.circl.lu/projects/internships/honeypot01/

[2] IPSUMDUMP, open source packet analyzer - Information available at http://read.seas.harvard.edu/~kohler/ipsumdump/

[3] Redis - Information available at https://redis.io

[4] Github, a web-based git platform - Manual available at https://guides.github.com/

[5] JSON data format - Information available at http://www.json.org/

[6] GNU Parallel, a shell tool or executing jobs in parallel - Information available at https://www.gnu.org/software/parallel/

[7] git, distributed version control system - Information available at https://git-scm.com/

[8] Tshark, command-line Wireshark version - Information available at https://www.wireshark.org/docs/man-pages/tshark.html

[9] Wireshark, open source packet analyzer - Information available at https://www.wireshark.org/

[10] Tcpdump, open source packet analyzer - Manual available at http://www.tcpdump.org/tcpdump_man.html

[11] Berckeley Packet Filter - Information available at https://biot.com/capstats/bpf.html

[12] Bokeh, a python library providing interactive graphics generation - Information available at http://bokeh.pydata.org/en/latest/docs/gallery.html and source available at https://github.com/bokeh/bokeh

[13] D3JS, Data-Driven Documents - Information available at https://d3js.org/ or https://github.com/d3/d3/wiki, and source available at https://github.com/d3/d3

[14] Circos, Software package for data visualization - Information available at http://circos.ca/

[15] Vagrant, open source software for building virtual environments - Informations available at https://www.vagrantup.com/

[16] Correspondence table between fields typo in Ipsumdump, Tshark, and json / Redis - https://github.com/chrisr3d/potiron/blob/internship/doc/correspondence_table_fields.txt

[17] Argparse, a parser for command-line options, arguments and sub-commands - Information available at https://docs.python.org/3/library/argparse.html

[18] D3JS Bubble chart - Initial template available at https://bl.ocks.org/mbostock/4063269

[19] ISN graph, initial example used as a model to build our own ISN plots - www.foo.be/isn

[20] Phantomjs, a scriptable Webscript - Information available at http://phantomjs.org/

[21] Bootstrap, Web front-end component library - Information available at http://getbootstrap.com/

[22] Parallel Coordinates - Template available at https://bl.ocks.org/mbostock/1341021

[23] Ardb, a Redis-protocol compatible NoSql database - Available at https://github.com/yinqiwen/ardb

[24] Python Profilers - Information and examples available at https://docs.python.org/3/library/profile.html

[25] MISP-Vagrant, Automatic deployment of MISP through virtual environment with Vagrant - Source and information available at https://github.com/MISP/misp-vagrant

[26] Potiron-Vagrant, Automatic deployment of potiron through virtual environment with Vagrant - Source and information available at https://github.com/cedricbonhomme/potiron-vagrant

Glossary

AIL: Modular framework to analyze potential information leak from unstructured data source like pastes from Pastebin or similar services.
See also: AIL repository - Available at `https://github.com/CIRCL/AIL-framework`

API: Application Program Interface. It is a set of routines, protocols, and tools for building software applications.

ARP: Address Resolution Protocol, a communication protocol used for resolution of Internet layer addresses into link layer addresses.

Botnet: collection of internet-connected devices, including PCs, servers, mobile devices and internet of things devices that are infected and controlled by a common type of malware.

Berkeley Packet Filter: language providing a packet filter.

Capture files or packet capture: file generated with an API (Application Programming Interface) for capturing network capture.

CASES: Cyberworld Awareness and Security Enhancement Services.

CERT: Computer Emergency Response Center, an expert group that deals with computer security incidents.

CIRCL: Computer Incident Response Center Luxembourg, the national CERT for private and non governmental public sectors.
See also: CIRCL projects - Available at `https://github.com/circl`

CIRCLean: USB key sanitizer.
See also: CIRCLean repository - Available at `https://github.com/CIRCL/Circlean`

Flask: Python-based web development framework.
See also: Flask - Information available at `http://flask.pocoo.org/`

Honeypots: Computer security mechanism used to detect unauthorized requests coming to information systems, by isolating and monitoring data that appears to be legitimate but seems to contain suspicious information that is then blocked.

ICT: Information and Communication Technologies.

ISN/IAN: Initial Sequence/Acknowledgement Number, referring to the unique 32-bit sequence number assigned to each new connection on a Transmission Control Protocol (TCP)-based data communication.

JSON: JavaScript Object Notation, a language-independent data format.

Layer 2: Second layer of the seven-layer OSI model of computer networking. This layer is the protocol layer that transfers data between adjacent network nodes in a wide area network (WAN) or between nodes on the same local area network (LAN) segment.

Layer 3: Third layer of the seven-layer OSI model of computer networking. This layer called network layer is responsible for packet forwarding including routing through intermediate routers.

Mirai: Malware that turns networked devices into remotely controlled "bots" that can be used as part of a botnet in large-scale network attacks.
See also: Mirai source code - Available at `https://github.com/jgamblin/Mirai-Source-Code`

MISP: Malware Information Sharing Platform.
See also: MISP project - Available at `http://www.misp-project.org/` (official web site) or `https://github.com/MISP/MISP`

MONARC: Optimized Risk Analysis Method, a tools provided by CASES to promote risk analysis.

Packet Fields (mentioned as "Field(s)": The fields part of different headers within each network packet.

SMILE: Security Made In Lëtzebuerg.

Appendices

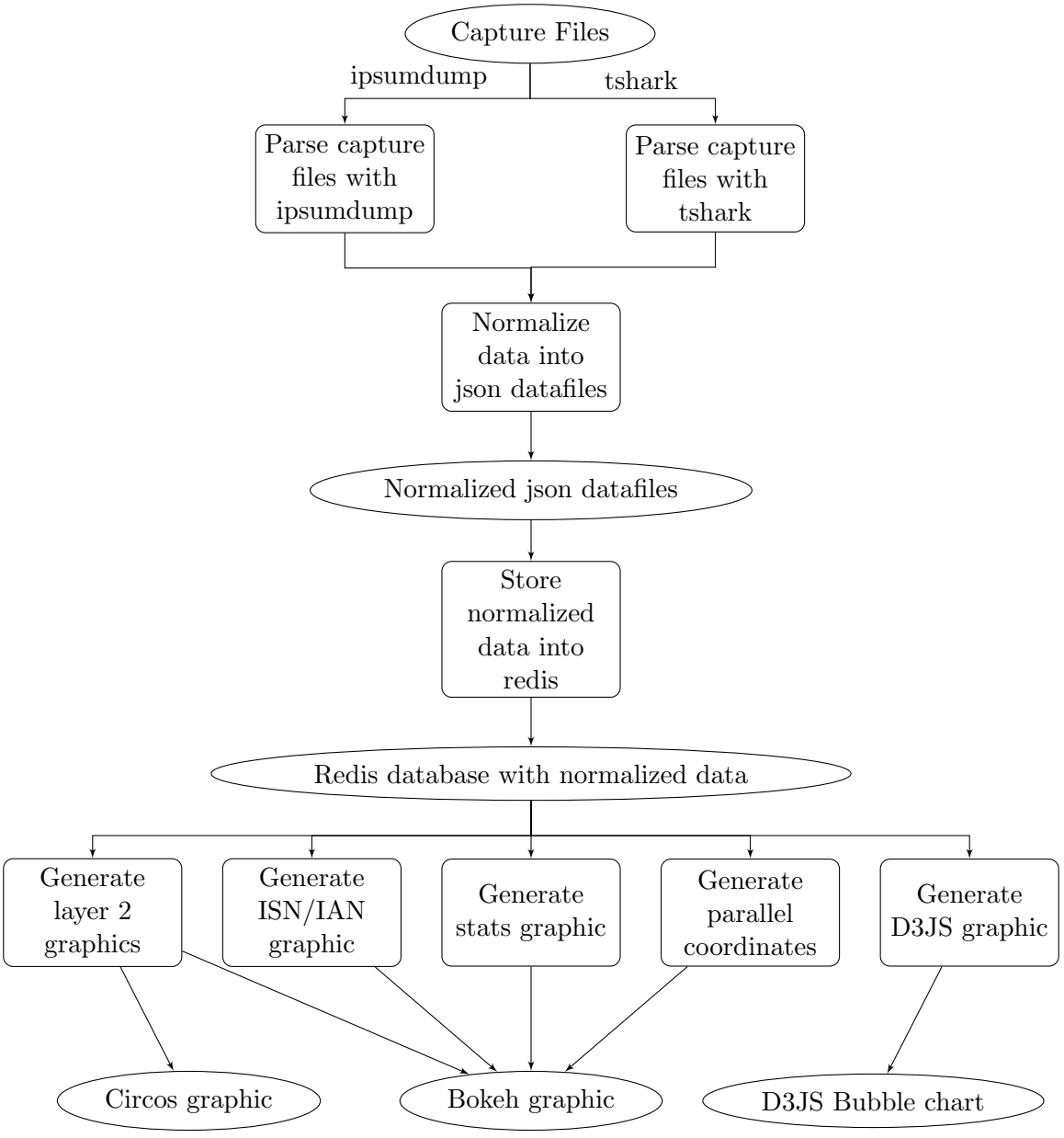

Figure 5.1: General flow diagram