Chapter 4

Implementation

In this chapter, we specify the approach we took to fulfil the scope of the project. Section 5.1 gives an overview and describes the technologies used to carry out this project. Sections 4.2, 4.3, 4.4, 4.5 explain how the program operates, and 4.6 provides information about some additional tooling developed to get better insight into the data. Section 4.7 provides an insight into the techniques used to refactor and optimise the code. The last section talks about the challenges faced during implementation.

4.1 Overview

The program for surveying keys had various stages in which it performed different functions. Figure 3.1 depicts the flow of the program, and it can be divided into the following stages:

- **Enviroment Setup:** Downloads all the required dependencies required for the program to work, including python libraries, Maxmind databases, ZMap, ZGrab2 and setups Golang.
- **Stage 1:** This stage can be called the Maxmind stage, where the MaxMind APIs are set up and filters out IP addresses for the country we want.
- **Stage 2:** This is the scanning stage where ZMap and ZGrab are used. The IPs from Stage 1 are scanned using ZMap for open port 25. Once we have those IPs, ZGrab is used to capture data for the ports we want.
- **Stage 3:** The final stage of the program where we process the data from Stage 2 and store the information we need. Once we have the information, we perform data analysis on it.

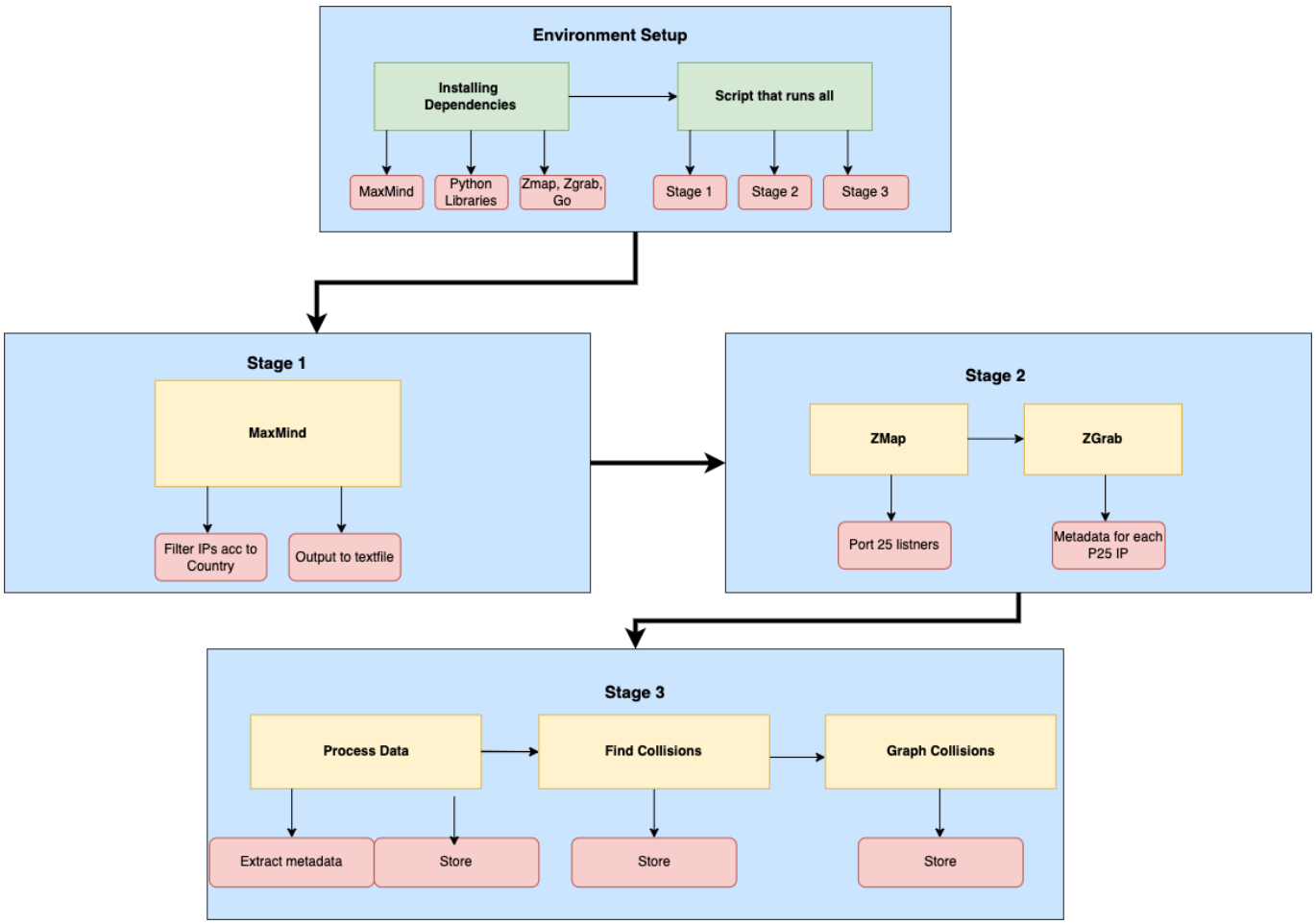

Figure 4.1: Program Strcuture

4.1.1 Tecnologies Used

The following lists the technologies that were used to carry out the project:

- **Python3+:** Majority of the program developed using Python. Used for data processing, visualisation and calling APIs and tools used.
- **Bash Script:** Used for automation of tasks and certain data extraction tasks.
- **ZMap:** Port scanner used to identity open port 25 listeners.
- **ZGrab:** Banner Grabber to obtain information about hosts in question.
- **Maxmind:** Used databases and APIs provided by MaxMind to carry out the network scans.
- **Pylint:** A code analysis tool used to measure code quality and enforce a standard coding structure.
- **cProfile and pstats:** A deterministic profiling tool used to optimise memory consumption and runtime.

4.2 Enviroment Setup

Before the program execution begins, a script called "install-deps.sh" downloads and installs all dependencies for the program to work. It starts by creating directories where the source code is available and another directory where the results for each scan are stored. After doing so, it will install all dependencies like the python libraries required, ZMap, ZGrab and the MaxMind databases. It also installs Golang and configures the gopath as ZGrab2 requires a valid gopath to function. Since this program was the last run in 2018, the Maxmind databases have changed significantly. Therefore, an additional script called "MMIPs.py" was developed to create a CSV dataset called "GeoIPCountryWhois," required for our program to run. The python script takes in two CSV files as input that contain the entire IPv4 address space network blocks with their associated Geonames. In addition, there was another CSV file that had the Geonames and the country code and name associated. The python script processes these two CSV files, map the IPv4 network blocks to their associated country code and name using the Geonames and produces the final CSV dataset needed.

4.3 Stage 1: Maxmind Stage

The program's first stage was to filter out the IP addresses for the countries we wanted to scan. For this project, the country selected was Ireland. The top script "skey-all.sh" does all the work and calls these stages sequentially, as indicated in figure 3.1. The script "IPsFromMM.py" does the first stage. It takes in the input file "GeoIPCountryWhois.csv" and filters the IPv4 CIDRs according to the selected country code. The final output from this stage is a text file that contains IPv4 CIDRs for the country chosen.

4.4 Stage 2: Port Scan and Banner Grab

This stage involved two parts. The first one was calling ZMap to check for open port 25 listeners and the next was using ZGrab on those IPs to gather the meta-data about them.

4.4.1 ZMap

After getting the list of IPs for the country selected, the program moves onto the second stage and uses ZMap to figure out which IPs listen on port 25. ZMap is called using the main script "skey-all.sh" as it requires to be called root user. The final output from ZMap is a list of IP addresses

that listen on port 25.

```
381 starting zmap
382 Mar 29 19:49:02.981 [INFO] zmap: output module: csv
383 Mar 29 19:49:02.981 [INFO] csv: no output file selected, will use stdout
384  0:00 0%; send: 0 0 p/s (0 p/s avg); recv: 0 0 p/s (0 p/s avg); drops: 0 p/s (0 p/s avg); hitrate: 0.00%
385  0:00 0%; send: 0 0 p/s (0 p/s avg); recv: 0 0 p/s (0 p/s avg); drops: 0 p/s (0 p/s avg); hitrate: 0.00%
386  0:01 0%; send: 147 146 p/s (142 p/s avg); recv: 0 0 p/s (0 p/s avg); drops: 0 p/s (0 p/s avg); hitrate: 0.00%
387  0:02 0%; send: 293 145 p/s (144 p/s avg); recv: 0 0 p/s (0 p/s avg); drops: 0 p/s (0 p/s avg); hitrate: 0.00%
388  0:03 0%; send: 438 144 p/s (144 p/s avg); recv: 0 0 p/s (0 p/s avg); drops: 0 p/s (0 p/s avg); hitrate: 0.00%
389  0:04 0%; send: 586 147 p/s (145 p/s avg); recv: 0 0 p/s (0 p/s avg); drops: 0 p/s (0 p/s avg); hitrate: 0.00%
390  0:05 0% (1d07h left); send: 734 147 p/s (145 p/s avg); recv: 0 0 p/s (0 p/s avg); drops: 0 p/s (0 p/s avg); hitrate: 0.00%
391  0:06 0% (1d07h left); send: 883 148 p/s (146 p/s avg); recv: 0 0 p/s (0 p/s avg); drops: 0 p/s (0 p/s avg); hitrate: 0.00%
392  0:07 0% (1d07h left); send: 1031 147 p/s (146 p/s avg); recv: 0 0 p/s (0 p/s avg); drops: 0 p/s (0 p/s avg); hitrate: 0.00%
393  0:08 0% (1d07h left); send: 1179 147 p/s (146 p/s avg); recv: 0 0 p/s (0 p/s avg); drops: 0 p/s (0 p/s avg); hitrate: 0.00%
394  0:09 0% (1d07h left); send: 1327 147 p/s (146 p/s avg); recv: 1 0 p/s (0 p/s avg); drops: 0 p/s (0 p/s avg); hitrate: 0.08%
395  0:10 0% (1d07h left); send: 1474 146 p/s (146 p/s avg); recv: 1 0 p/s (0 p/s avg); drops: 0 p/s (0 p/s avg); hitrate: 0.07%
396  0:11 0% (1d07h left); send: 1622 147 p/s (146 p/s avg); recv: 1 0 p/s (0 p/s avg); drops: 0 p/s (0 p/s avg); hitrate: 0.06%
397  0:12 0% (1d07h left); send: 1769 146 p/s (146 p/s avg); recv: 2 0 p/s (0 p/s avg); drops: 0 p/s (0 p/s avg); hitrate: 0.11%
398  0:13 0% (1d07h left); send: 1918 148 p/s (147 p/s avg); recv: 2 0 p/s (0 p/s avg); drops: 0 p/s (0 p/s avg); hitrate: 0.10%
399  0:14 0% (1d06h left); send: 2066 147 p/s (147 p/s avg); recv: 2 0 p/s (0 p/s avg); drops: 0 p/s (0 p/s avg); hitrate: 0.10%
400  0:15 0% (1d06h left); send: 2214 147 p/s (147 p/s avg); recv: 2 0 p/s (0 p/s avg); drops: 0 p/s (0 p/s avg); hitrate: 0.09%
401  0:16 0% (1d06h left); send: 2362 147 p/s (147 p/s avg); recv: 2 0 p/s (0 p/s avg); drops: 0 p/s (0 p/s avg); hitrate: 0.08%
402  0:17 0% (1d06h left); send: 2510 147 p/s (147 p/s avg); recv: 2 0 p/s (0 p/s avg); drops: 0 p/s (0 p/s avg); hitrate: 0.08%
403  0:18 0% (1d07h left); send: 2652 141 p/s (146 p/s avg); recv: 2 0 p/s (0 p/s avg); drops: 0 p/s (0 p/s avg); hitrate: 0.08%
404  0:19 0% (1d07h left); send: 2800 147 p/s (146 p/s avg); recv: 2 0 p/s (0 p/s avg); drops: 0 p/s (0 p/s avg); hitrate: 0.07%
405  0:20 0% (1d07h left); send: 2947 146 p/s (146 p/s avg); recv: 2 0 p/s (0 p/s avg); drops: 0 p/s (0 p/s avg); hitrate: 0.07%
```

Figure 4.2: ZMap Output

Figure 4.2 shows what the ZMap output looks like. It expands each IP in CIDR notation to the IP range and pings each IP in the range using the TCP SYN scan. The fields indicate the time left on the scan, the "send: 2947" shows that ZMap has pinged 2947 IPs address and the "recv:2" means that ZMap has found two port 25 listeners. The "hitrate: 0.07%" indicates that out of the total IPs pinged 0.07% are port 25 listeners.

4.4.2 ZGrab

The following process in this stage was to gather metadata about each port for IP addresses we obtained from ZMap. This is where the ZGrab2 tool was used, and the process was done using the script "FreshGrab.py" that specifies the ZGrab parameters and calls the ZGrab command. The script requires an input file that is a list of IPs (ZMap output) and first verifies whether the IP belongs to the specified country using some functions defined that make use of the Maxmind APIs. If the IP does not match the country according to Maxmind, they are marked as "out of country" and are not processed further. The remaining IP addresses are passed to the ZGrab2 tool, and we obtained information about each port we wanted for the particular IP. Since the output from ZGrab is in JSON, we stored the data in a file called "records.fresh" that contains one JSON structure per line. This part of Stage 2 can take anywhere from a few hours to a day to run since the scan rate of ZGrab2 was limited by adding a 100 ms wait between each IP scanned so as not to cause any congestion in the network.

```
ports=['22', '25', '110', '143', '443', '587', '993']
ztimeout=' -t 2'
pparms={
    '22': 'ssh -p 22',
    '25': 'smtp -p 25 --starttls',
    '110': 'pop3 -p 110 --starttls',
    '143': 'imap -p 143 --starttls',
    '443': 'http -p 443 --use-https',
    '587': 'smtp -p 587 --smtps',
    '993': 'imap -p 993 --imaps',
    }
for port in ports:
    cmd=zgrab_path + " " + pparms[port] + ztimeout
    proc=subprocess.Popen(cmd.split(),stdin=subprocess.PIPE,stdout=subprocess.PIPE)
    pc=proc.communicate(input=ip.encode())
```

Listing 4.1: ZGrab2 Parameters

Listing 4.1 shows how Zgrab2 is called from the "FreshGrab.py" script. The parameters specify the protocols and associated ports to Zgrab2 to scan for. To capture the TLS metadata for the ports the *STARTTLS* command was used for standard ports that provide no encryption and implicit TLS for the other ports.

4.5 Stage 3: Data Processing and Visualisation

Stage 3 of the program involved parsing out the metadata from the JSON structures the project required, performing DNS and reverse DNS lookups, storing the data, and finding where the same key is being used for each IP irrespective of the ports. Stage 3 can be broken down as follows:

4.5.1 Data Processing

The script that handles the data processing is "SameKeys.py" and starts by iterating through data for each IP address in file "records.fresh". Since we capture a lot of metadata for each IP address, the data for each IP is stored using a class instance with multiple attributes using Python. Then, the program starts by performing reverse DNS lookups for the IP and storing the reverse DNS names in the same class instance for each IP. After that, the Fully Qualified Domain Names, TLS certificates and fingerprint information are parsed and stored for every port. The FQDNs are stored to assist

with verifying the asset owners that are operating the said service. We also get information about if it is an autonomous system and the name and number associated with it using Maxmind databases. The methods to parse and store these fields are defined in the "SurveyFuncs.py" file. Before moving onto the analysis stage, the program verifies names associated with the IPs by doing a DNS lookups with the SANs associated with the IP. If IP addresses from SAN matches the IP (when a DNS lookup is performed) in "records.fresh" it is recorded as good otherwise, it is recorded as bad. The good IPs with a key are stored in a JSON file for further analysis. The same is done with the bad IPs, and another file contains both IPs marked as both "good" and "bad" with their associated information. Appendix A1 provides the output of ZGrab2 and shows the information collected for all seven ports for each IP address. The appendix only provides data for one IP address in JSON format but we store these outputs from ZGrab2 as one JSON structure per line.

4.5.2 Data Analysis

The data analysis process entails checking for the duplicate keys for different services within and across IPs. The clustering is based on Fingerprint SHA-256 for each service. If a specific key is shared by two IPs irrespective of protocol, they are put in the same cluster. Cross protocol checks are included in the code as it is not uncommon to see key reuse for different services, as proved in [8]. When key reuse is identified, clusters are cross-checked with other clusters and merged if the same key is being used there. Once, this is complete three files are produced as follows:

- "collisions.json": Contains collision information.
- "dogies.json": Contains information about IP address that fail DNS lookups.
- "all-key-fingerprints.json": Contains information that is included in the first two files.

4.5.3 Data Visualisation

After finding out the collisions for all the IPs, they are stored in the files above. To produce graphs for all the collisions found, the Graphviz library in python is used. The file used to graph the collisions is "collisons.json". After that, each IP is iterated through again, and the IPs with the same cluster number are merged and finally graphed using custom methods defined using Graphviz.

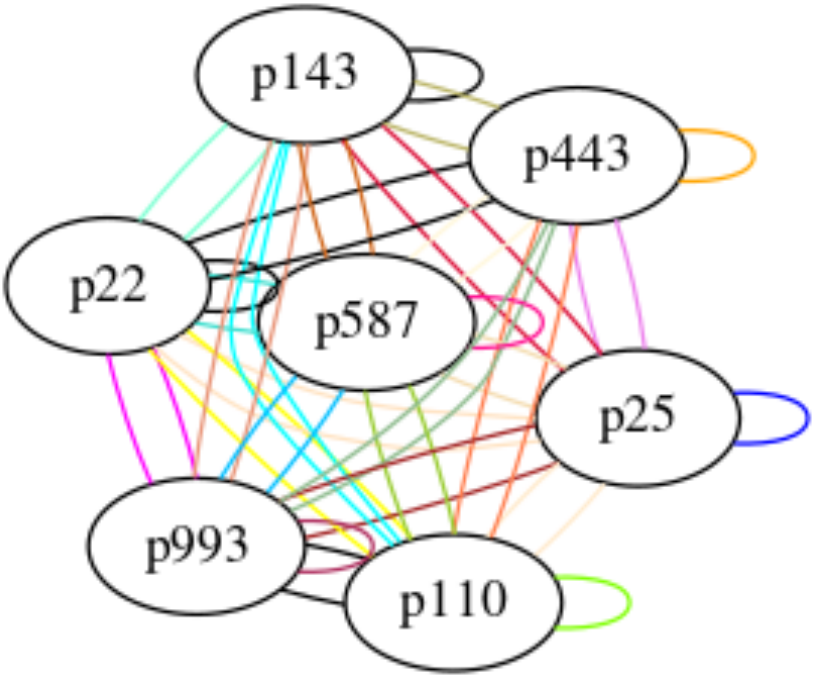

Figure 4.3: Graph Legend

Figure 4.3 represents the graph legend and the graphs can be visualised as follows:

- The nodes are represented as IP addresses.
- The colour of the nodes represent entities.
- The edges represent key reuse across ports.
- The colour represents the pair port combination.

4.6 Additional Tools

Some additional scripts are available that provide in-depth analysis of the data we collect. For example, the script "ProtocolVersions.py" provides an insight into what TLS/SSL versions were seen throughout the scans and provides a count for them. It also provides details of the SSH versions seen on port 22. More tooling offers information on how many IP addresses were mapped by ZMap, how many host port combinations are there, how many unique fingerprints are seen, and many more.

4.7 Code Migration, Refactoring and Optimisation

One of the primary goals of this project was to migrate the code to Python3+ and refactor it to increase readability, improve the structure, and optimise the run time and memory usage. This section gives details about the migration and the refactoring process. It also provides information about the techniques used to increase code performance.

4.7.1 Code Refactoring

To carry out the refactoring process, the entire program was analysed to check for duplicate code and complex methods and involved heavy nesting. In the "SameKeys.py" script where the data processing takes place, all information parsing was done manually according to the JSON headers for each port. Due to this, the code was not so readable and looked complex. Therefore, the techniques outlined in section 2.6 were used to carry out the refactoring. The following code snippets give an example of this.

```
try:
    p25=j_content['p25']
        if thisone.writer=="FreshGrab.py":
            banner=p25['data']['banner']
        else:
            banner=p25['smtp']['starttls']['banner']
        ts=banner.split()
        if ts[0]=="220":
            banner_fqdn=ts[1]
            nameset['banner']=banner_fqdn
        elif ts[0].startswith("220-"):
            banner_fqdn=ts[0][4:]
            nameset['banner']=banner_fqdn
except Exception as e:
    print >> sys.stderr, "FQDN banner exception " + str(e) + " for record:" +
        str(overallcount) + " ip:" + thisone.ip
    nameset['banner']=''
try:
    if thisone.writer=="FreshGrab.py":
        tls=j_content['p25']['data']['tls']
        cert=tls['server_certificates']['certificate']
    else:
        tls=j_content['p25']['smtp']['starttls']['tls']
```

```
            cert=tls['certificate']
            fp=cert['parsed']['subject_key_info']['fingerprint_sha256']
            get_tls(thisone.writer,'p25',tls,j_content['ip'],thisone.analysis['p25'],scandate)
            get_certnames('p25',cert,nameset)
            thisone.fprints['p25']=fp
            somekey=True
    except Exception as e:
        print >> sys.stderr, "p25 exception for:" + thisone.ip + ":" + str(e)
        pass
```

Listing 4.2: Code before Refactoring

```
    try:
        p25 = j_content['p25']
        bn = "y"
        if thisone.writer == "FreshGrab.py":
            banner_fqdn = get_mail_data(p25, bn)
        else:
            banner = p25['smtp']['starttls']['banner']
            nameset['banner'] = banner_fqdn
    except Exception as e:
        print(sys.stderr, "FQDN banner exception " + str(e) + " for record:" +
            str(overallcount) + " ip:" + thisone.ip)
        nameset['banner'] = ''
    try:
        if thisone.writer == "FreshGrab.py":
            data = j_content['p25']['data']['smtp']['result']['tls']
            cert, fp = get_mail_data(data, None)
        else:
            tls = j_content['p25']['smtp']['starttls']['tls']
            cert = tls['certificate']
        get_tls(thisone.writer, 'p25', data, j_content['ip'], thisone.analysis['p25'],
            scandate)
        get_certnames('p25', cert, nameset)
        thisone.fprints['p25'] = fp
        somekey = True
    except Exception as e:
        print (sys.stderr, "p25 exception for:" + thisone.ip + ":" + str(e))
        pass
```

Listing 4.3: Code After Refactoring

Listing 4.3 shows the code before refactoring and Listing 2.3 shows the code after refactoring. A method was introduced called `get_mail_data(data,banner)` that was used to extract the banner information for port 25, but also to gather the metadata for all mail protocols instead of doing this manually. Other methods were created as well and the existing ones were modified for simplicity everytime data processing occurs for each port. This was done to decrease the number of lines of code in the "SameKeys.py" script and to avoid writing duplicate code. This also has another benefit in that in case there are changesto the ZGrab output in the future, one will have to make changes in the program at one point instead of doing it manually for all the seven ports.

Another instance where refactoring was carried out was in the "SurveyFuncs.py" script that contains utility functions that help us during the surveying process. While it is still somewhat acceptable to have manual code in this file, and we do not use it directly, refactoring was carried out here to improve performance. For example, existing methods that were being used to call the Maxmind API had a lot of nesting of if statements and for loops. To decrease nesting, the abstraction technique was used to break down the nested statements. Different methods were developed to perform the same function overall.

4.7.2 Code Optimisation

To optimise this program, memory analysis had to be carried out that gave information about the memory overheads of the program and what was taking the most amount of time to complete the whole process. This analysis was done using the cProfile and pstats libraries available in open source. A profile here is defined as a "set of statistics that describes how often and for how various long parts of the program are being executed" [1]. When used with the scripts, this library gives a detailed report about how the program behaves. It gives us information about the number of monitored calls and how many of those were primitive or recursive. In addition, it provides the total time spent in each function and the cumulative time spent in the method and all sub-methods called. Once this data was available, it was investigated, and areas of the program were identified where bottlenecks were being created. Finally, alternative solutions could be looked at to increase run time or decrease memory usage.

```
rs@rs:~/surveys/rsstuff$ ./memory_profile.py
Wed Apr 13 14:32:03 2022    /home/rs/survey_results/old_mem_stats

         248238208 function calls (198949372 primitive calls) in 73816.116 seconds

   Ordered by: cumulative time
   List reduced from 2574 to 15 due to restriction <15>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
    363/1    0.018    0.000 73816.116 73816.116 {built-in method builtins.exec}
        1  222.382  222.382 73816.116 73816.116 SameKeys.py:22(<module>)
    43941    0.975    0.000 70991.462    1.616 /home/rahulseth/surveys/SurveyFuncs.py:765(get_dns)
    43941 70985.715    1.615 70989.834    1.616 {built-in method _socket.gethostbyname}
    17647    0.483    0.000 1816.256    0.103 /home/rahulseth/surveys/SurveyFuncs.py:753(get_rdns)
    17647 1813.977    0.103 1815.126    0.103 {built-in method _socket.gethostbyaddr}
    17647    0.442    0.000  402.074    0.023 /home/rahulseth/surveys/SurveyFuncs.py:710(mm_info)
    17647    0.552    0.000  401.633    0.023 /home/rahulseth/surveys/SurveyFuncs.py:726(extract_from_mm)
    52941    0.575    0.000  385.286    0.007 /home/rahulseth/.local/lib/python3.6/site-packages/geoip2/database.py:232(_get)
    52941    1.809    0.000  384.711    0.007 /home/rahulseth/.local/lib/python3.6/site-packages/maxminddb/reader.py:123(get_with_prefix_len)
    35294    0.779    0.000  330.803    0.009 /home/rahulseth/.local/lib/python3.6/site-packages/geoip2/database.py:245(_model_for)
6407352/105882   98.081    0.000  247.852    0.002 /home/rahulseth/.local/lib/python3.6/site-packages/maxminddb/decoder.py:141(decode)
467844/105882   31.748    0.000  232.761    0.002 /home/rahulseth/.local/lib/python3.6/site-packages/maxminddb/decoder.py:85(_decode_map)
    17647    0.245    0.000  218.798    0.012 /home/rahulseth/.local/lib/python3.6/site-packages/geoip2/database.py:142(city)
    52941    4.989    0.000  186.128    0.004 /home/rahulseth/.local/lib/python3.6/site-packages/maxminddb/reader.py:154(_find_address_in_tree)
```

Figure 4.4: Memory Profile before Refactoring

Figure 4.4 depicts the output upon profiling the code before any optimising or major refactoring was carried out. The above profiling was done on the same data that was used to carry out the data analysis and in turn produce the results. Referring to the figure, we can see the total execution time of the program was about 73816 seconds, but most of the time was consumed by the function `gethostbyname()` that is used to perform DNS lookups on the names we parse out from the metadata. The names include banner information and subject alternative names for each port. Once this bottleneck was identified, alternative DNS resolution solutions were looked at (discussed in the section below) and a benchmark test was created in order to see the timing of each and what meets the needs.

4.7.3 DNS Resolving

After memory profiling the program, it was found that the DNS resolution part of the code was consuming much time. A couple of alternate solutions were looked at for faster resolution to decrease the run time. Reasons why DNS resolution was taking so much time were explored. Some alternate solutions were looked at to mitigate this problem, like DNSpython, MassDNS Resolver, Berserker Resolver, and stubby plus Unbound. A benchmark test was created for the four, and it was found that using a combination of stubby and unbound was the fastest among them. The problem with the first three options were as follows:

- MassDNS is used to make queries in the range of millions to billions and was not fitting the scope of the project.
- DNSpython had the same performance as `gethostbyname()` function in the socket library unless a timeout value was set for lookups which was not the most efficient way to solve this issue.
- Berserker Resolver had an upgrade in performance but used DNSpython in the backend.

A more permanent solution was needed and used a combination of two open-source tools called Unbound and Stubby. Unbound is a validation, recursive, caching DNS resolver that is designed to be fast and lean and provides modern services like DNS over TLS and DNS over HTTPS, which allows encryption while making name resolutions [14]. Stubby is an application that acts as a local DNS stub resolver and uses DNS-over-TLS for resolutions. The combination of Unbound and Stubby was used to speed up the DNS resolution for our program. Unbound was used behind stubby, and all queries made using Unbound were forwarded to Stubby. Since Stubby uses DNS over TLS, which is assigned to port 853, the configuration had to be set up. Before testing our code with this combination, Wireshark was used to analyse the traffic on port 853 while doing some DNS lookups to ensure it was functioning as expected. Wireshark is a network packet analyser that captures in-depth detail about the captured packet [34].

4.8 Challenges

OR This project came across a few expected challenges. One of the significant challenges was configuring and visualising the ZMap and ZGrab tools. Although it was relatively easy to understand how ZMap works and how the output would look, the ZGrab2 posed a real challenge since the output from the tool was in JSON and involved heavy nesting. Development for this program was done using various Linux systems, and the table below shows the Virtual Machines that were set up and their purpose during the course of this proejct.

Ubuntu Version	Purppose
22.04	Development
21.04	Target VM for testing ZGrab
18.04	Testing

Table 4.1: VM Setup

Service	Protocol
Apache Server	HTTPS,SSH
Dovecot	IMAP and POP3
Postfix	SMTP

Table 4.2: Target VM Setup

The target VM was set up with different servers available in open-source that offered other protocols. Table 4.2 shows the server setup on the target VM and the protocols provided by each. These servers were set up using minimal configurations to test the ZGrab2 tool over localhost. Since the program used a Python Script to use the ZGrab tool and performed IP checks according to the MaxMind databases, the reserved IP addresses could not be used as MaxMind does not contain recognise them as it operates the same blacklist as ZMap. Due to this, the ZGrab tool was tested manually using the command line interface for each protocol.