“What is all that crap?” Analysis of DNS root server bogus queries

Daniël Sánchez daniel.sanchez@os3.nl
Joost Pijnaker joost.pijnaker@os3.nl
University of Amsterdam

February 4, 2007

Abstract

"What is all that crap?" is a question several DNS root server operators probably asked when analyzing incoming queries. This research is all about this question and about how to reduce this 'crap'.

Of all traffic arriving at the DNS root servers, a lot is considered 'bogus'. Some people say that only about 2% of the arriving queries are actually valid queries and the rest would be bogus. We believe it is interesting to do research on this subject and look for possibilities to reduce this massive amount of bogus queries.

We analyzed a two hour capture data session from the DNS K-root server. We made a categorization of the bogus queries arriving at the root server. We discovered that the largest amount of bogus is found in the category of 'repeated' queries. We also made a top 10 list of sources that generate the most queries and discovered that for one DNS root server 80.7% are bogus queries and for another DNS root server 52.31% was coming from one source. After that we described some possible causes for the bogus queries categories and for the top IP sources generating the most queries. At the end we explained what could be possible solutions to reduce the amount of bogus queries. We looked both at client side solutions and server side solutions.

Contents

Chapter 1

Preface

We, Daniël Sánchez & Joost Pijnaker, are two students at the University of Amsterdam (UvA), and study System and Network Engineering (SNE) http://www.os3.nl. Part of our course is getting acquainted with problems from the field of practice. This course is named Research Project 1 (RP1). Goal of this course will be to gain practical experience and gain collaboration skills.

Our research will be in the Domain Name System (DNS) workfield. We approached RIPE NCC and they were willing to cooperate on this project. More information about RIPE NCC can be read in chapter 2 Context.

Our supervisors will be Cees de Laat (UvA) and Daniel Karrenberg (RIPE NCC).

The following chapters are in this report: chapter 1 Preface: about the reason for writing this report, chapter 2 Context: about the environment in which the project takes place, chapter 3 Project introduction: about the research question and our approach, chapter 4 Background: about the theoretical background behind the project, chapter 5 Related work: about previous work done on this area, chapter 6 Research: about our actual work, chapter 7 Conclusion: about the results, chapter 8 Future work: some suggestions for further work.

A note about privacy: some parts, like IP addresses will be anonymized. The first two octets then will be x.y. In other parts we may be using fictitious names.

Chapter 2

Context

In this chapter we will shortly describe the context in which the project is done for a better understanding of the project. The project is executed at RIPE NCC. RIPE NCC is one of five RIRs. RIRs are getting resources from IANA, which is an operational unit of ICANN. Because of all the abbreviations, we start explaining each of them from top to bottom:

- ICANN: Internet Corporation for Assigned Names and Numbers.

  "ICANN is responsible for managing and coordinating the DNS to ensure that every address is unique and that all users of the Internet can find all valid addresses. It does this by overseeing the distribution of unique IP addresses and domain names. It also ensures that each domain name maps to the correct IP address."[8].

- IANA: Internet Assigned Numbers Authority. IANA is an operational unit of ICANN and allocates block of IP address space to RIRs [1].

- RIR: Regional Internet Registry. A RIR allocates the resources, given by IANA, within their regions to LIRs. Currently there are five RIRs: AfriNIC, APNIC, ARIN, LACNIC and RIPE NCC.

- LIR: Local Internet Registry. A LIR allocates the resources, given by a RIR, to end users. Typically a LIR is an ISP (Internet Service Provider).

- RIPE NCC: Réseau IP Européen Network Coordination Centre [2]. RIPE NCC is the RIR responsible mainly for Internet Service Providers (ISPs), telecommunication organisations and large corporations located in Europe, the Middle East and parts of Central Asia.

Chapter 3

Project introduction

In this chapter we will explain how the research question has been formed and what was the approach we used to answer this question. We will describe the following subjects: a problem definition, the research question that followed, the research scope and the exact approach.

3.1 Problem definition

Through the internet there are several so called DNS root servers. Everyone in the world, probably you included, is using one or more of these servers to access all kind of webpages, for example http://www.os3.nl. Every moment it is possible that a DNS root server is asked an IP address of, for example, a certain website. Therefore, DNS root servers always have to provide this kind of information to anyone who is asking. Maybe you can imagine, not everyone is asking these questions in the appropriate way, which causes the servers to get more questions than they should get. Actually, research on this field [16] has stated that a large part of these questions is “bogus”, or thrash. This leads us to our research question.

3.2 Research question

Though the DNS root servers have enough resources to cope with this massive amount of bogus queries, we think it would be useful to do research about the possibilities to reduce this kind of queries. In fact, there is a waste of resources. Therefore, our exact research question is: “What are the possibilities to reduce the amount of bogus queries arriving at the DNS root servers?”.

3.3 Research scope

To finish the project successfully, we have to create a scope: what we DO cover en what we DON'T cover.

What we DO cover:

- We are going to look at two DNS root servers: just one server would give a limited view and more than two servers would take more time than we have for this project.
- We categorize the bogus queries en for each category we determine causes and possible solutions.

What we DON'T cover:

- We are not going to implement the possible solutions: we do not have enough time to do this and besides we do not have authorization to access the live environment.
- We are not going to monitor the DNS root servers ourselves, because of the lack of authorization. RIPE NCC provides logs of a capture session of two hours from two DNS root servers.

3.4 Approach

To answer our research question, we split it in subquestions:

- Which categories of bogus queries exist?
- What are possible causes for bogus queries?
- What are possible solutions to decrease bogus queries?

To answer these subquestions, we start searching for related work that already has been done on this area, and basic background information. We write a short background as a base from where we can work. After that we will analyze the given logs and try to answer the first subquestion. Now we hope to know which categories of bogus queries apply to the chosen DNS root servers and we can try to find possible causes and solutions by looking at the previous background information or maybe additional information on the internet and conversations with technicians from RIPE NCC.

Chapter 4

Background

In this chapter we cover the background (theory) to have an understanding about the material which will support us in our Research chapter. First we talk about DNS and its basics, after that we go deeper into the DNS root servers and last we will discuss queries.

4.1 DNS basics

In this paragraph the basics of DNS will be explained. First we will explain its purpose and operation.

4.1.1 DNS purpose

"The purpose of DNS is to enable Internet applications and their users to name things that have to have a globally unique name." [13] This introduces two advantages:

- Users only have to remember names, instead of IP addresses, which is user friendlier. Example: if we would see the website www.os3.nl, a so called 'query' has to be sent to a DNS server. This is a question how to reach the given name. The DNS server responds with an IP address: x.y.26.20. This IP address now could be used to communicate with. It is obvious that www.os3.nl is much more simple to remember than x.y.26.20.

- Names are separated from locations: moving servers from one location to another, does not affect the name under which it can be found in the DNS. Example: we look at the same address www.os3.nl. The server running this website actually could be anywhere in the world (under the condition that it is connected to the internet). Normally this server would have the IP address shown above, but theoretically, if someone would like to replace the server to, for example another location in the Netherlands or maybe only to another place in the organisation, than the IP address could change. Now there only has to be changed one entry in a DNS server to the right IP address to fix the problem: the website is still reachable at www.os3.nl. This makes the DNS very flexible.

Now we know what the purpose of the DNS is, we can further look into its operation.

4.1.2 DNS components

Here we will talk about the components of DNS. we will take RFC 1034 [17] and RFC 1035 [18] as a base for explaining the basics of DNS. The DNS consists of three components:

- The Domain Name Space and Resource Records: these are specifications for a tree structured name space and data associated with these names. All nodes of the domain name space are a name for specific information. By a query, specific types of such information can be extracted. About resource records: read further below.

- Name Servers: these are server programs that hold information of the domain tree. In general a name server knows all information of a specific part of the domain tree. This means that a name server is authoritative for a specific zone. Name servers also contain pointers to other name servers to find other information in the domain tree.

- Resolvers: these are programs that ask information from the name servers. Clients can send requests to resolvers. Resolvers have to be able to access one or more name servers to extract the information directly or by using referrals to other name servers.

4.1.3 DNS resource records

Further information about the resource records: a domain name identifies a node and each node has a set of resource information. This set exists of resource records. Each resource record has the following aspects:

- Owner: the domain name where the resource record belongs to.

- Type: refers to the type of resource. The following types are being used:

  - A: a host address.
  - CNAME: a canonical name or an alias.
  - HINFO: the CPU and OS used by the host.

  - MX: a mail exchange for the domain.
  - NS: the authoritative name server for the domain.
  - PTR: a pointer to another part of the domain space.
  - SOA: the start of a zone of authority.

- Class: the protocol family that is being used. Two possible types: IN (the Internet system) and CH (the Chaos system).
- TTL: the time to live for the resource record.
- RDATA: describes the resource.

Example: now we shall give an example for a better understanding. A resource record could be like this:
WWW.OS3.NL. IN A x.y.26.20
In this example this information could exist in the domain name space and at least one name server holds this record. A client could be asking a resolver how to reach WWW.OS3.NL. The resolver extracts information from the name server and sends the answer to the client: the IP address x.y.26.20.

4.2 DNS root servers

In this paragraph we will explain something more about the DNS root servers and their place in DNS, about the DNS structural hierarchy and about how they work with Top-Level Domains (TLDs). Also we will explain why there are 13 root servers.

4.2.1 Purpose

"Essentially the DNS root name servers reliably publish the contents of one small file to the Internet. The file is called the root zone file. The function and content of the root zone file are what make it special and cause it to be at the focus of increased attention. The root zone file is at the apex of a hierarchical distributed database called the Domain Name System (DNS). This database is used by almost all Internet applications to translate worldwide unique names like http://www.os3.nl into other identifiers; the web, e-mail and many other services make use of the DNS. The root zone file lists the names and numeric IP addresses of the authoritative DNS servers for all TLDs" [14].

4.2.2 Hierarchy

We talked about DNS operation. Like we mentioned earlier DNS has a hierarchical tree structure. It starts with root, which is shown as a dot, and is followed by a TLD. The TLD can be followed by a subdomain and so

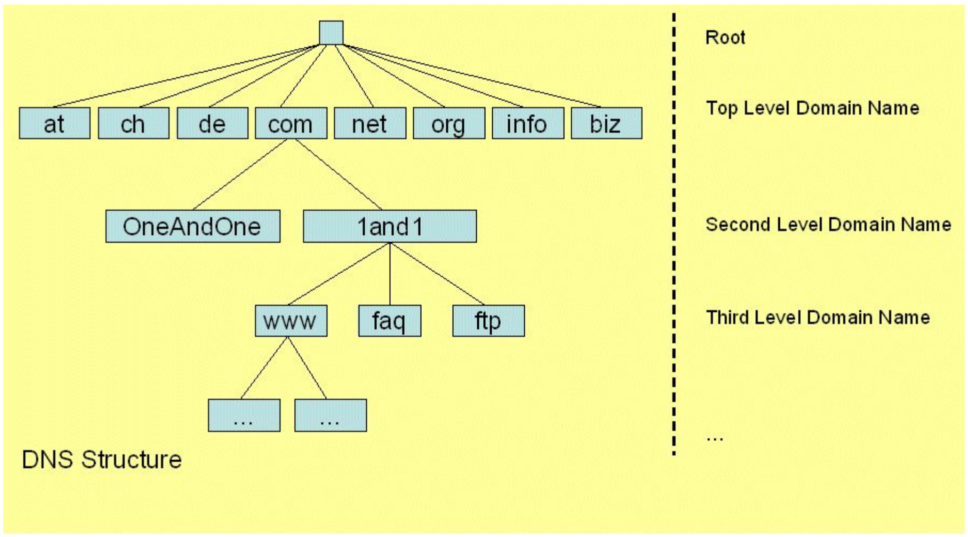

Figure 4.1: DNS structure [19]

on. There are two sorts of TLDs: ccTLD and gTLD [15]. A ccTLD [5] is a country code TLD and is used for country specific TLDs, for example: .nl for The Netherlands, .us for United States and .jp for Japan etc.

A gTLD [9] is a global TLD and is used for global domains like: .net, .org and .com etc. In addition to gTLDs and ccTLDs, there is one special TLD, .arpa, which is used for technical infrastructure purposes. For example, the domain in-addr.arpa is used for getting reverse answers (when IP address is known but domain name is unknown).

The rule says that the notation for links is formed by the labels separated by points. For example: in www.os3.nl, nl is the top level domain, os3 is the second level domain and www the third level domain or actually the link to the webserver. There is something about DNS which is often misinterpreted by administrators. The root is the only tag with a so called empty label e.g. without a name. Let's take the example of "www.os3.nl". This is often interpreted right by many applications, but should actually be "www.os3.nl.". This could lead to problems when an application does not try to finish the domain name with a dot.

4.2.3 Resolving

When you want to access a webpage, http://www.os3.nl, you will ask your resolver to resolve the domain name to an IP address. A resolver is an instance you send a question to and the resolver will try to get the answer and send it back. If your resolver has cache and the name was resolved before, you will get a response right away from the resolver. Cache is a part of the

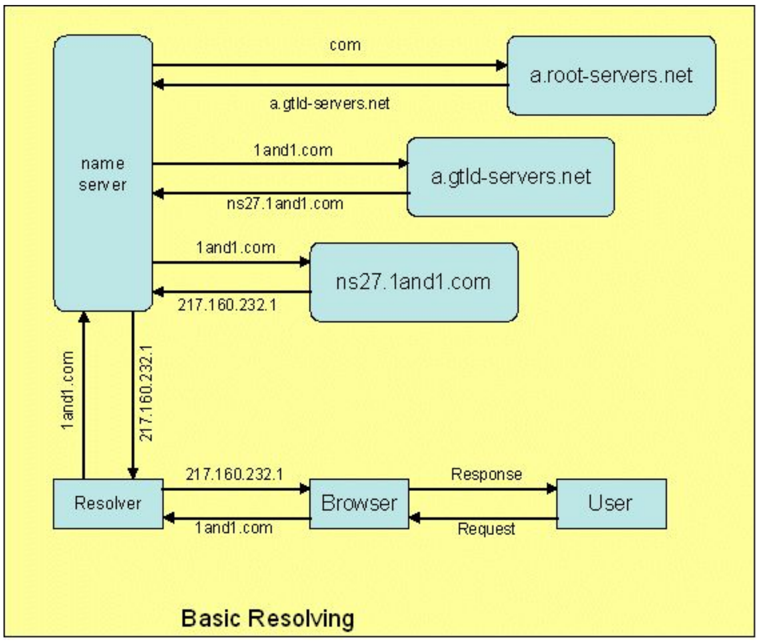

Figure 4.2: DNS resolving [19]

memory in your resolver that saves adresses so you would not have to ask DNS servers again. When the name is unknown to the resolver it will forward the request to the DNS root server. A well behaved DNS server needs to query name servers only once every 48 hours for each particular TLD. The root DNS server doesn't know the IP address of http://www.os3.nl, but it does know the the IP address of the DNS server hosting the ccTLD .nl, so it forwards this answer to the resolver. The resolver then contacts the DNS server that hosts the .nl ccTLD. The .nl DNS server does not know the IP address of http://www.os3.nl, but it does know the the IP address of the DNS server hosting the os3.nl domain and forwards this answer to the resolver. This process will keep on going until the domain name is completely resolved. The process is called resolving and the requests are called queries. In paragraph "DNS queries" we will talk more about queries and explain how they work.

4.2.4 Only 13 DNS root servers?

There is a limit for the amount of root servers the DNS design allows. This because of the UDP protocol that has a maximum amount of data it can send at once. This means only thirteen IP addresses can be sent at once. These root servers are called a to m.root-servers.net. Because of the number of requests that has to be answered by these root servers and most of them

(10) were in the United States, there was a need for more servers. This was implemented by using a model with global and local nodes which function as root servers and could be accessed by a routing mechanism called anycast. Global nodes should be public for everyone. A global node is similar to a DNS root server, it works like a mirror. Local nodes are initiated just for few network segments. This is done through the routing protocol BGP which is used to route traffic to the DNS root server. Local nodes can be advertised to only a few segments by setting the export-bgp option to “no”. This will stop BGP advertising the local node to any other network.

4.3 DNS queries

In this paragraph we will go deeper in to queries. First we will explain what a query actually is, then we will make a distinction between valid queries and bogus queries. We will state a definition of a so called bogus query which we can use in our research chapter.

4.3.1 Purpose

Queries are messages which may be sent to a name server to provoke a response. The response by the name server either answers the question posed in the query, refers the requester to another set of name servers, or signals some error condition. Normally, users send queries to a resolver, which in turn sends it to a name server. There exist different types of queries, like standard queries, inverse queries and status queries, but we will only cover standard queries: the other types are outside the scope of this project.

4.3.2 Standard queries

The majority of all DNS queries are standard queries. A standard query specifies a target domain name (QNAME), query type (QTYPE), and query class (QCLASS) and asks for resource records which match.

The QTYPE field may contain:

- <Any type>: matches just that type (like A or PTR).
- AXFR: special zone transfer QTYPE.
- MAILB: matches all mailbox related resource records.
- *: matches all resource records types.

The QCLASS field may contain:

- <Any class>: matches just that class (like IN or CH).

- *: matches all resource record classes.

So in essence the information in the domain name space discussed in the DNS operation paragraph, could be asked for with a query. It is possible that the answer of a name server to a resolver not is the asked information, but just a pointer to another DNS server which has the information.

Example: a user would like to email to info@os3.nl and could ask a resolver mail information about os3.nl. The resolver could send a query to a name server for QNAME=os3.nl, QTYPE=MX, QCLASS=IN. The name server could respond with the following resource record:
os3.nl. MX 10 mail.os3.nl.
At this moment the DNS server sends along additional information: the A record of mail.os3.nl, because it assumes the resolver would like to know this information (which is true):
mail.os3.nl. A x.y.0.10

4.3.3 Bogus queries

Now it is clear what is meant by a (standard) query and how it should be like, we can discuss about bogus queries. The amount of queries being sent to the DNS root servers is massive: several gigabytes of raw data per hour is no exception. Anyone connected to the internet could be sending these queries. Maybe you can imagine that not al queries sent are in the right or meant form: just a typing error could give a wrong result or an error from the DNS server. But these are not the only things that can go wrong. A small list of ways this can go wrong:

- The same query could be sent over and over again: normally the answer would be cached for a period of time. From the second query and above, you could be speaking of a bogus query: this is unnecessary.
- Someone is sending a query while the result should be a private IP address (like 10.0.0.1). DNS servers do not answer with private IP addresses: only with globally unique addresses. Such a query could be called a bogus query.
- A typing error: someone asking for www.os3.ml instead of www.os3.nl sends a query for something that does not exist. This also could be called a bogus query.

Beside these examples there are more, which we will discover in the research chapter. Now we can try to state a definition of a DNS bogus query, so we can select this kind of queries in the first part of our research.

We use the following definition: "A DNS bogus query is a query sent to a DNS server that should not be arriving at the DNS server.".

Note: bogus is easily mistaken by abuse. When we talk about bogus queries we mean queries that match our definition. When we look at abuse, it is always intended to be evil and initiated on purpose. Because it is difficult to separate bogus queries and abusive queries, we decide that when we search for bogus queries, we take in account all queries that should not be initiated, whether it was on purpose, generated by faulty software, or generated by mistake.

Now we understand the basics of DNS, DNS root servers and (bogus) queries, we can proceed with looking at related work.

Chapter 5

Related work

In this chapter we will discuss papers about projects that are related to our project. Any insight which we will get from reading these papers we will surely use for our project.

- The reason we started this project was because we attended a guest presentation from Erik Romijn from RIPE NCC. He was asked to give a lecture about "Determining and improving performance of anycast for DNS root servers" [21], which he was graduated on [20]. In his presentation he stated that: "Well known number is 98% bogus, however the author later said his model was overly simplistic." [21]. We were overwhelmed by this number of bogus queries and this caught our interest on the subject.

- While searching for related work on the DNS query field, we found a reasonably similar paper wrote by Duane Wessels and Marina Fomenkov [16]. The authors of the paper state that they have captured raw Tcpdump packets from the F-root server for the period of 24 hours. They only captured requests not responses. The server was authoritative for the edu, gov, arpa, in-addr.arpa and root-servers.net zones. This might have influenced the results of their research. The results showed that repeated queries are the biggest source of DNS pollution. Their answer to this problem would be informing organizations to use recursion on their name servers rather then blocking all incoming traffic, which generates repeated queries.

- We found another paper from Duane wessels and Marina Fomenkov in conjunction with Nevil Brownlee and KC Claffy [4]. In this paper they seek an answer for: "How does the choice of DNS caching software for local resolvers affect query load at the higher levels?" This could be useful information for our project, because we think quite a few problems are caused by DNS caching software. Their answer to their research question is: "We believe that resolvers should implement exponential backoff algorithms when a nameserver stops responding. We feel strongly that more caching implementations should adopt BIND9s feature that avoids forwarding duplicate queries."

- From Duane Wessels we found one last article about bogus DNS queries [24]. This paper's subject is: "DNS pollution may not be causing any perceivable performance problems. The root servers seem well equipped to handle the load. Since DNS messages are small, the pollution does not contribute significantly to the total traffic generated by most organizations. Nonetheless, this paper provides a few reasons why network operators should take the time to investigate and fix these problems." This would be useful for our project, because we will also try to find solutions to DNS pollution by bogus DNS queries. The author has stated a few solutions for his research question, his solutions involved adjusting configurations of DNS resolvers.

- We found a paper about a specific bogus query: the reverse query of the private IP range described in RFC 1918 [3]. The problem with this kind of bogus queries is that it not only consumes resources, but it also jeopardizes security and privacy of users. The researchers found that over 97% of these bogus queries where generated by Windows machines. This research did not focus on what the impact would be on the root DNS servers. But the researchers state that this would probably be almost all because, usually a DNS update is followed by a SOA query.

Note: the 2nd, 3rd en 4th item are strongly related to each other. Actually these papers could be seen as follow-ups. Duane Wessels is (co-)author of all of these works.

Chapter 6

Research

6.1 Environment

In this chapter we will describe the environment in which we will do our research. We will talk about the capture data and about the tools we used. After that we will describe our methodology.

6.1.1 Capture data

In this section we will describe the properties of the capture data which we are going to work with.

We do not have the rights or possibilities to capture the DNS root servers ourselves. So we have been given the capture data. The data is coming from the K-root server. In figure 6.1 you can see the locations of all K-root server nodes.

As you can see, the white blocks represent global nodes and the red nodes represent local nodes. The global nodes from left to right are in: Miami (US), London (GB), Amsterdam (NL), Delhi (IN) and Tokyo (JP). The data we work with is from two global nodes: AMS-IX (Amsterdam) and NAP (Miami).

The capture session has been done for approximately two hours with Tcpdump. This has been done during peak hours. Queries and also the responses are captured. In figure 6.2 you can see the properties of the capture data.

6.1.2 Tools

In this section we will describe the tools that we are going to use to analyze the capture data. We will use several tools. More information about the tools can be found on their websites. We also use self-made scripts for filtering. You can see them in Appendix A.

A list of the tools we will use:

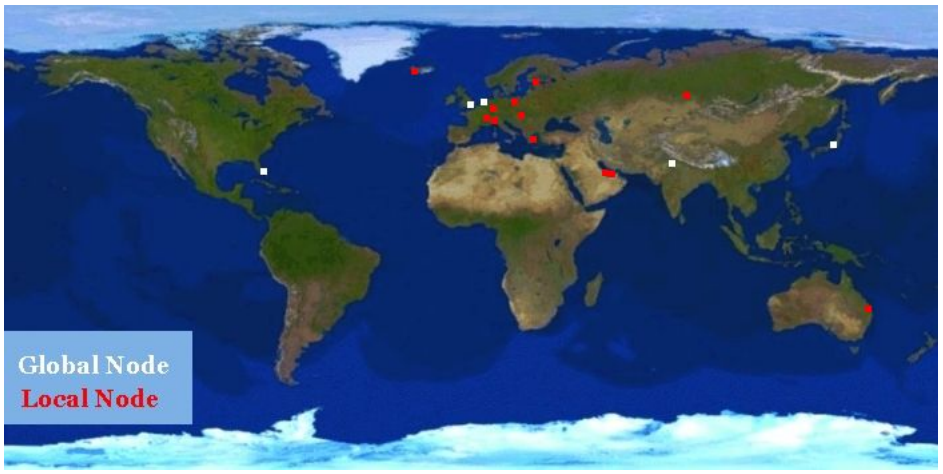

Figure 6.1: K-root locations [22]

Name	AMS-IX	NAP
Date	20-12-2006	20-12-2006
Start time	16:04:39	16:04:40
End time	18:12:52	18:12:56
Size of data	6.7 GB	8.1 GB
Size / second	0.89 MB	1.08 MB
Size of queries	2.92 GB	2.64 GB
Query entries	16805064	16195292
Query entries / second	2184	2104

Figure 6.2: Properties of the capture data

- Tcpdump [23]: was used to capture the data. We did not do this ourselves. Further we use Tcpdump to read all the capture data and save the output as textfiles.

- Ethereal [7]: to open the capture files and see the raw data. This application is used for first analysis of DNS query messages. It shows header and payload data from queries in its absolute purest form. This application was already installed on the machines we could use for this project.

- dnstop [12]: an application that generates various tables of DNS statistics. It is possible to filter a few kinds of bogus queries and see the percentages from the IP addresses that generate the most queries. It is used for determining the top sources and aid in the analysis of those sources.

- merge.rb: a self-made Ruby script that combines multiple dnstop table overviews into one table overview. The reason we made this script was that we got the data in several tcpdump files instead of one big Tcpdump file. dnstop should be run on a Tcpdump file, so we got the same amount of dnstop output files as Tcpdump files. We want all these output files to be merged to one big dnstop output file. See Appendix A for the script.

- bogusfilter.sh: this is a self-made shell script that combines multiple shell scripts and multiple Ruby scripts for filtering bogus queries out of the raw data. When this script is started it will produce several files: one file for each bogus category with only queries of the specific category and a file with remaining (non-bogus) queries which can be further analysed. See Appendix A for the script.

6.1.3 Methodology

In this section we will describe how we are going to categorize and how we are going to use the tools. The following steps are involved in our research:

- Determining valid queries: we need to determine which conditions are required for a valid (non-bogus) query. This can be done by looking at the capture data with Ethereal at queries which we think show normal behaviour. If we would be unsure whether a query is normal, we will consult the appropriate RFCs. When we know what a normal query should look like, we can analyse the data for abnormal behaviour. By studying this abnormal behaviour we should be able to categorize the bogus queries.

- Determining bogus queries: now that we know what a valid query should look like, we can try to find queries that have different properties. We can look for bogus queries by loading the captured data in Ethereal and try to find protocol fields that do not match a valid query. With the application dnstop we will look at sources that generate most queries. These top sources will be searched for in Ethereal and we will look for abnormalities in this output. The purpose of using dnstop is to get a more clear idea about the kind of bogus queries generated and dnstop will aid in determining the bogus categories. Also dnstop can filter on a few known bogus queries, so this should give some quick results.

- Determining bogus categories: in this stage we have to define the bogus categories and describe what kind of characteristics the queries should have to match that category. With the results from determining valid queries, related work and Ethereal, we can decide which bogus categories exist. For analyzing purposes we want some files to be created with only these bogus categories and a file with none of these bogus categories. This will be done by writing filters for the captured data.

- Filtering raw data: when we know the categories of bogus queries, we can filter on these categories with a self written script (bogusfilter.sh). This will leave us with a few files containing all the bogus queries of each category. Also when we have filtered out some known bogus queries there will be less queries left after each filter, kind of acting like a chute. This will leave us with another file containing remaining queries. On this reduced data that will be left, we can do some more analysis and maybe find even more bogus queries.

- Statistics: when the filters are done creating the data files containing all the bogus queries per category, we can count the amount of queries and do some statistics on this data. We will try to find statistics like: the percentage of bogus queries per category, the percentage of valid queries etc.

- Determine possible causes: by looking at the data file generated by the bogusfilter script, we can try to find some causes for each category of bogus queries. Because we now have specific files for each category, we can narrow our view to abnormalities belonging with just that category. We find causes by looking at the internet, by looking at our related work and by debating with each other and maybe our supervisor.

- Determine possible solutions: now it is clear what are the reasons for each bogus category, we can try to find solutions. Depending on the

Level 1	Level 2	Value
Transaction ID		0 <X <65535
Flags	Response	= 0
	Opcode	{0 | 1 | 2}
	Truncated	{0 | 1}
	Recursion desired	{0 | 1}
	Z	= 0
	Non-auth. data OK	= 0
Questions		= 1
Answer RRs		= 0
Authority RRs		= 0
Additional RRs		= {0 | 1}
Queries	Name	!= {local | invalid TLD}
	Type	{A | AAAA}
	Class	{IN | CH | HS | ANY }
Additional records		0 <X <65535

Figure 6.3: Values of valid queries

causes, we describe solutions per category or otherwise solutions in general.

6.2 Categorize bogus queries

In this paragraph we are going to discuss the outcome of our analysis on the captured data. We will define the categories of bogus queries and do some statistics on those categories.

6.2.1 Determining valid queries

When we determined how a valid query should look like, we looked at a few DNS queries in Ethereal and analyzed the structure of the headings and payload. We made a table with all possible header and payload information. This information can be found in figure 6.3 The values in this table are referenced from the appropiate RFCs. When any value is outside this table range, we know for sure the query is a bogus query. When a value is inside this the range of the table it would be possible to be a valid query, but this does not guarantee it is a valid query. We did not use this table as a blueprint for deciding whether a query is valid or not, but just to get an overall idea on how a valid or bogus query could look like.

6.2.2 Determining bogus queries

As mentioned before we looked at the data in Ethereal. Many queries could be recognised by invalid header fields or invalid payload information. But also we found that many sources were repeating the same queries for a period of time. When we used the application dnstop we found that some sources were sending much more queries to the AMS-IX server than others sources. On the NAP server the percentages were even higher. Also with dnstop, when filtering on a few known bogus categories, we found an astonishing number of queries generated by single sources. Sources that generate this much queries must be generating a lot of bogus queries, because a normal functioning DNS resolver should be querying the DNS root server only once every 48 hours for every TLD. When we analyzed these sources in Ethereal we found quite a few other bogus queries.

6.2.3 Determining bogus categories

Now we will discuss every bogus category and try to cover all bogus queries. All bogus categories will be explained individually.

- A for A queries: normal operation for a DNS query would be to send a request for an IP address of a certain domain name by querying an A record. We found in our analysis that sometimes a query is done by asking an IP address of an IP address. In other words, the query asks an A record for an A record. We call this kind of bogus queries A for A queries. These kind of queries should never take place, because the DNS root server could never give a suitable reply.

- Private IP reverse queries: it is possible to query a DNS root server for a reverse translation. The resolver queries the DNS root server for a domain name of a certain IP address. Our analysis showed that there are a lot of queries to translate private IP addresses stated in RFC 1918 [6]. These kind of queries should never contain questions about the private IP range, because private IP addresses should never reach the public internet.

- Reserved IP reverse queries: this category is almost similar to private IP reverse queries. The difference in this category is that these IP addresses are not private IP addresses, but reserved IP addresses. The reserved addresses can be found in [11], this is the last updated list from IANA. This list could be changed over time, but at this time it is accurate and there is no indication it will change on short notice. All reserved addresses should not be queried, because they are not in use. The same goes for reserved IPv6 addresses [10], but due to limited time for this project we will not be filtering on reserved IPv6 addresses.

Considering there will not be many queries for IPv6 addresses this should not be a big problem.

- Local domain queries: our next category we found is a query of the so called .local TLD. Local is meant for private use and should never go out on the public internet. We found a lot of these queries in several forms. The forms we found are: .local, .locale, .lokaal, .loc, .lan and .localhost.(domain, intra, invalid, workgroup). These are all refering to the same local domain in different forms. All should never reach the internet.

- Invalid TLD: like the .local domain category, this category will also look at TLD DNS queries. These TLDs include: domain, intra, invalid, workgroup, txt, html and speedport. The reason we made a new category for these kind of queries is that we think they may have different causes then the .local TLD queries. We found all sorts of wrong queries for TLDs. We take in account that a user that initiates a query could be making a typing error. The DNS system is highly fault tolerant and can handle typing errors, but we found some TLD queries that are queried so much that we want to analyse this category some more. No user makes this many typing errors and we suspect this is caused by something else.

- Identical query ID: every query has its own query ID. The query ID is a unique number, between 0 and 65535, assigned to the query at random. We found that some of the queries were sent with identical query ID's. At the rate the DNS root server received queries it could be possible to receive several requests with the same query ID. But when the requests are repeatedly asked with the same query ID from the same source, we know there is something wrong with the source of this query.

- Repeated queries: we found that some queries are repeatedly generated. Sometimes the sources of these queries generate traffic at a rate of over 1000 queries per second. It should be clear this is not normal behaviour for a resolver. Normal query rate would be one query for every TLD every 48 hours.

- No caching TLDs queries: this category we found actually after we executed some scripts. It looks a bit like the repeated queries category. The difference is that not the query is the same, but only the TLD is the same. We found a lot of queries from the same sources where continuously was asked for a certain TLD in a very short time.

6.2.4 Filtering raw data

Now that we know what properties to look for when searching for bogus categories, we will write filters and apply them to the captured data. In this paragraph we will show samples of the filters that we wrote and give a short description about how they work. All filters can be viewed exactly as we applied them on the captured data in Appendix A.

We viewed the captured data with Tcpdump and saved the data as textfiles. When the data is in plain text it is much simpler to filter on the data. We can use line editors like awk for simple filtering. For filters with more search arguments we use Ruby to do filter the data.

Because the captured data we got was divided in a few files, we have to cat the files together as one big file.

```
cat <multiple input files> > <output>
```

First, we wrote a filter for separating responses and no-responses. We will call this filter the no-responses filter. Filtering queries is needed so we can filter for bogus queries without getting responses. The input file used for this script is the catted output file as described above.

```
cat <input> | awk 'BEGIN { FS = " > " };
{ if ( $2 !~ / q: / ) print $0; }' > <output>
```

We print the input and pipe this to awk. This filter uses " >" as a delimiter, which separates every line in two parts. Every response repeats the query in its response message starting with " q: ", so when the second part of the line does not match " q: " it will be a response. Finally we print all lines that match the statement to a new output file.

All queries have a "?" for the actual question this will separate queries and no-queries. We will call this filter the query filter. The input file for this script is the output file from the no-respones filter as described above.

```
cat <input> | awk 'BEGIN { FS = " > " };
{ if ( $2 ~ /? / ) print $0; }' > <output>
```

We print the input and pipe this to awk. This filter uses " >" as a delimiter, which separates every line in two parts. We filter on "? ", so when the second part of the line matches "? " it will be a response. Finally we print all lines that match the statement to a new output file.

All of the following filters can have any input files as long as it is either the output file of the query filter or the output file of any of the following filters. So there is no particularly order in which the following filters should be run.

This script will filter all A, A6 and AAAA queries that ask for IP addresses e.g A-for-A queries.

```
cat <input> | awk --re-interval 'BEGIN { FS = " > " };
{ if ( $2 ~ / A{1,4}6{0,1}\? [0-9]{1,3}\.[0-9]{1,3}
\.[0-9]{1,3}\.[0-9]{1,3}\.
/ ) print $0; }' > <output>
```

We print the input and pipe this to awk. This filter uses " >" as a delimiter, which separates every line in two parts. The second part must match A or AAAA, with or without a 6, ending with a question mark. After that the line must match an IP address. Finally we print all lines that match the statement to a new output file.

This script will filter all PTR queries that ask for private IP addresses.

```
cat <input> | awk --re-interval 'BEGIN { FS = " > " };
{ if ( $2 ~ / PTR\?
[0-9]{1,3}\.[0-9]{1,3}\.([0-9]{1,3}\.10||[0-9]{1,3}\.127
||3[0-1]\.172||2[0-9]\.172||1[0-6]\.172
||168\.192)\.in-addr.arpa. / ) print $0; }' > <output>
```

We print the input and pipe this to awk. This filter uses " >" as a delimiter, which separates every line in two parts. The second part must match "PTR?" and match a private IP address. Finally we print all lines that match the statement to a new output file.

This script will filter all PTR queries that ask for reserved IP addresses.

```
cat <input> | awk --re-interval 'BEGIN { FS = " > " };
{ if ( $2 ~ /\.(0||233||24[0-9]||25[0-5])\.in-addr\.arpa\./ )
print $0; }' > <output>
```

We print the input and pipe this to awk. This filter uses " >" as a delimiter, which separates every line in two parts. The second part must match "PTR?" and match a reserved IP address. Finally we print all lines that match the statement to a new output file.

This script will filter .local queries.

```
cat <input> | awk --re-interval 'BEGIN { FS = " > " };
{ if ( $2 ~ /\.local\. / || $2 ~ /\.localhost\. /
|| $2 ~ /\.lan\. / || $2 ~ /\.locale\. / || $2 ~ /\.home\. /
|| $2 ~ /\.localdomain\. / || $2 ~ /\.intra\. /
|| $2 ~ /\.loc\. / || $2 ~ /\.domain\. /
|| $2 ~ /\.workgroup\. / || $2 ~ /\.lokaal\. / )
print $0; }' > <output>
```

We print the input and pipe this to awk. This filter uses " >" as a delimiter, which separates every line in two parts. The second part must match .local, .localhost, .lan, .locale, .home, .localdomain, .intra, .loc, .domain, .workgroup or .lokaal and ending with ". ". Finally we print all lines that match the statement to a new output file.

This script will filter invalid TLD queries.

```
cat <input> | awk --re-interval 'BEGIN { FS = " > " };
{ if ( $2 ~ /\.speedport_w_700v\. / || $2 ~ /\.invalid\. /
|| $2 ~ /\.site\. / || $2 ~ /\.belkin\. /
|| $2 ~ /\.speedport_w_500\. / || $2 ~ /\.txt\. /
|| $2 ~ /\.htm\. / htm) print $0; }' > <output>
```

We print the input and pipe this to awk. This filter uses " >" as a delimiter, which separates every line in two parts. The second part must match .speedport_w_700v, .invalid, .site, .belkin, .speedport_w_500, .txt or .htm and ending with ". ". Finally we print all lines that match the statement to a new output file.

This script will filter all repeated queries. The code below is strongly simplified. For the complete code see Appendix A. First we will make one directory for each letter in the alphabet and 0-9 plus one folder for all other signs.

```
   @out_a = File.new("#{@@folder}/a", "w")
   @out_b = File.new("#{@@folder}/b", "w")
   ...
```

Then the script will write every entry to a file which matches the first sign of the query. The reason why we do this is because we do not have enough memory to load every entry into one big array. We did this before and noticed because of swapping the script would take too long to finish (actually it did not finish at all). So we have to chop the input file in chunks.

```
      case x[0,1]
         when /[aA]/
            @out_a.write(line)
         when /[bB]/
            @out_b.write(line)
         ...
```

The x in the code above stands for the asked data in the query, for example www.os3.nl. Now we can sort all queries based on this field.

```
@@sorted_by_query = @@file2.sort_by { |line|
   line.split(" > ")[1].split("? ")[1].split[0]
}
```

After that we can save all second (and above) same queries to a new array. After we have the new array, we can again sort, but now based on source IP address.

```
   if ( @@new_query == @@old_query ) and
      ( @@new_source == @@old_source )
   then
      @@temp << line
   else
      @@nonbogus_temp << line
...
```

After that we sort on time and do some other things (again: look at Appendix A if you are interested). Last thing we need to do is cat all files together.

We also made a script that filters the same query IDs. This filter looks a lot like the repeat filter. The most important change (but there are more) is that the file now is sorted by line.split[7].

After we executed all filters, we looked again at the file that remained with all queries that should not be bogus. We discovered that there were still many bogus queries and therefore we adjusted some filters a little bit (see Appendix A): we added some more terms to the local domain and invalid TLD filters and we added the IGNORECASE option because a lot of bogus queries with capitals were found in the "no bogus" file. Also we added the split method in the repeat filter because the memory was overloaded. We also made a new script and found another category: no caching of TLDs bogus queries.

A note about the reserved IP addresses script: after we executed the script, we discovered the IANA webpage with reserved IP addresses [10]. So we did not fully filter the reserved IP addresses but the most important are filtered.

The script for the no caching of TLDs again is very similar to the repeated script. Now everything is sorted by line.split(" >")[1].split("? ")[1]. split[0].split(".")[-1].

A note for this script: later we discovered that we made a mistake: the K-root server is authoritative for in-addr.arpa, which means that someone who is asking for 1.in-addr.arpa. and 2.in-addr.arpa. is NOT sending bogus queries! We adjusted the script in a way that this is taken into account. Later again, we discovered that K-root is also authoritative for .arpa., so the same story applies to this TLD. We did not have the time to adjust the script for this. However, we also think that the majority of .arpa is also

IP source	AMS-IX count	AMS-IX %	NAP count	NAP %
Highest	447393	2.65	757420	4.66
2	195687	1.17	723621	4.46
3	168478	1.01	721755	4.45
4	168239	1.01	697172	4.31
5	165652	0.99	694569	4.29
6	163302	0.97	693983	4.29
7	157556	0.93	686758	4.26
8	131023	0.78	680628	4.20
9	126478	0.75	651120	4.01
10	84523	0.49	562632	3.47
Total	1808331	10.75	6869658	42.40

Figure 6.4: Top 10 speakers

in-addr.arpa, so the script would actually barely have problems with this issue. It should however be good to correct it for future work.

Here you can see the output of a command which proves that k-root is authoritative for .arpa. and in-addr.arpa.:

```
dig @k.root-servers.net arpa. axfr | awk
'{ if ( $5 ~ /k\.root-servers\.net\./ ) print $0; }'

arpa.             518400  IN     NS      k.root-servers.net.
in-addr.arpa.     86400   IN     NS      k.root-servers.net.
```

6.2.5 Statistics

In this section we will analyze how much bogus queries are actually received by the DNS root servers.

When we applied the filters on the captured data it leaves us with a few files containing the bogus queries. By counting the number of entries in those files and counting the overall number of queries, we can calculate percentages of how many queries are bogus. Note that one query could be of several bogus categories. Summing up all bogus percentages will not be the same as calculating all bogus queries. For calculating bogus queries, we take 100 percent and substract the amount of non-bogus queries.

When we analyzed the queries in dnstop we can see the IP sources that generate most queries. In figure 6.4 and 6.5 we have listed the results from dnstop. Note that all sources are anonymized for privacy reasons. In figure 6.6 are all percentages of each category listed.

When we want to run all scripts, we noticed some failures occur and we had to repeatedly adjust the scripts. Because it takes a very long time (a

IP source	AMS-IX count	AMS-IX %	NAP count	NAP %
Highest	681727	4.04	8478012	52.31
2	577292	3.47	2992605	18.46
3	447393	2.65	139708	0.86
4	294325	1.75	135134	0.84
5	227768	1.38	75047	0.45
6	195687	1.17	73931	0.43
7	157556	0.93	58221	0.42
8	136035	0.81	53062	0.33
9	130583	0.78	49092	0.31
10	116550	0.69	47559	0.30
Total	2964916	17.67	12102371	74.71

Figure 6.5: Top 10 speakers based on 3 octets

Category	AMS-IX count	AMS-IX %	NAP count	NAP %
A-for-A	1287476	7.66	280584	1.73
Private reverse	113483	0.68	27261	0.17
Reserved IP	34705	0.20	5353	0.03
Local domain	2119363	12.61	408912	2.52
Invalid TLD	574026	3.42	95023	0.59
Repeated queries	4779070	28.44	1914330	11.82
Total no-bogus	3243987	19.30	13822295	85.35
Total bogus	13561077	80.70	2372997	14.65
Total	16805064	100.00	16195292	100.00

Figure 6.6: Percentages of bogus queries

few days) to run all scripts, and our time was limited, we did not have the possibility to run the same query ID script and the no caching of TLDs script for all data files. We did however filter the no caching of TLDs bogus queries on the AMS-IX file that had been filtered already by the other awk scripts. In other words: we know how many queries for the AMS-IX are bogus and non bogus, but we do not know how many queries for the AMS-IX are no caching of TLDs bogus queries.

In the figures a few remarkable results can be found:

- For the AMS-IX server 80.7% is bogus and for the NAP server 14.65% bogus.
- The repeated queries occur by far the most of all bogus categories: 28.44% for the AMS-IX server and 11.82% for the NAP server.
- For the AMS-IX server the top 10 IP addresses are responsible for 10.75% of all queries. For the NAP server the top 10 IP addresses are responsible for 42.40%.
- For the AMS-IX server the top 10 IP addresses if we only look at the first 3 octets, are responsible for 17.67% of all queries. For the NAP server the top 10 IP addresses are responsible for 74.71%.
- Our filters show that the AMS-IX server gets more bogus queries than the NAP server. We have an explanation for this (but cannot prove it because of lack of time): the statistics show that for AMS-IX the largest categories are repeated queries and no caching of TLDs queries. These numbers are lower for the NAP server. But the case is that more than 52.31% of all queries for the NAP server is coming from an x.y.112 (so only looking at the first 3 octets). Our scripts however are checking for the complete source address (4 octets). When we talk about octets we do not mean class a, b or c addresses but range of addresses. With 3 octets this could be an other prefix than /24.

6.3 Determine possible causes

Now it is clear how many bogus queries we found in each category, we describe some possible causes for each category. There may be more possible causes, but we describe the most important ones. After that we will look at the top IP sources and determine in which bogus category they fall in.

Possible causes for each bogus category:

- A for A queries: this kind of bogus query is often caused by an old Microsoft Windows NT bug in the stub resolver. This bug was fixed in Windows NT Service Pack 2, so it does not occur as often as other bugs.

- Private IP reverse queries: these bogus queries could be caused by different kinds of software, like SMTP or HTTP servers, that are sending reverse queries for validation or other purposes.

- Invalid IP reverse queries: these ones could be caused by abuse. Especially queries to the 0- and 255-range. Another reason could be testing purposes if one has asked, for example, 250.0.0.1.

- Local domain queries: normally names like "local" and "localhost" should be hardcoded in the operating system to the loopback interface. This may be missing on some systems which causes them to ask the DNS server where names like this could be found.

- Invalid TLD queries: very often these are caused by faults in software or devices, like modems. A lot of queries can be seen for speedport, which is a certain modem. Internally something probably has been coded wrong, so that a query is sent for the speedport TLD, which does not exist.

- Identical query IDs queries: often these were caused by applications that do not conform to the appropriate standards.

- Repeated queries: these kind of bogus queries are often caused by (misconfigured) firewalls: they do not let through the answers of DNS servers. The result is that the resolver, sending the query, send the same query over and over again because it gets no answer.

- No caching TLDs queries: a big part of these kind of queries are obviously overlapping with the previous category, but another big part is caused by incorrect Time To Live (TTL) values in the DNS configuration of resolvers.

Now we will look at the top IP sources for the first 3 octets (anonymized) and try to explain why they are sending so much queries. First for the AMS-IX server: x.y.150: this IP is responsible for 4.04% of all incoming AMS-IX queries. Most of the queries are repeated and no cached TLD bogus queries.

And now for the NAP server: x.y.112: this IP is responsible for 52.31% of all incoming AMS-IX queries. Most of the queries are also repeated and no cached TLD bogus queries. This number is remarkably high, so we decided to look up the IP address in the RIPE NCC Whois database [25]. We found a source that indeed covered the whole range of IP addresses we found in the dnstop output. The owner of the address was a hosting company (we do not mention names) in Ukraine. We searched for this company and found several references to some spamming business. We think the hosting company probably has a spamming company as a client.

We clearly can see that the most bogus queries fall in the categories repeated queries and no cached TLD queries.

6.4 Determine possible solutions

Now we have seen some causes for the bogus queries, we will talk about possible solutions. First we will describe some possible solutions:

- Install and use stable applications and update them when (security) bugs have been resolved. This will decrease A for A queries, private IP reverse queries and identical query IDs queries.
- Use stable hardware and update firmware when (security) bugs are resolved. This will reduce invalid TLD queries.
- Configure your software appropriately. Local names should be added to for example the hostsfile and firewalls should be configured to let through query responses. This will reduce local domain queries and repeated queries. DNS servers should also be configured correctly. An appropriate TTL value causes a reduction of the “no caching of TLDs” queries.
- In addition to the previous items: soft- en hardware vendors could be contacted to fix bugs that are causing problems. This could be a good solution.
- Another mechanism, which could be implemented on root servers itself, could be access lists. IP addresses that cause a lot of bogus queries, simply could be blocked. After the system administrator has corrected the failures, the IP address can be given access again. This solution may sound simple, but in fact it probably is not a very good one (but we feel we should mention it): a lot of queries are indeed bogus, but there also are a lot of valid queries from top IP sources. These valid queries would not get an answer, which is ethically not right: a DNS root server always should respond to a valid query.
- Another solution, which also could be implemented at the server side, is the so called (u)RPF, or (Unicast) Reverse Path Forwarding. In short: this is a technique that is used on routers. It checks the source IP address from where the traffic (like a DNS query) is coming. Then the router checks its routing table for the IP range where the traffic is coming from: if the route back to the source is not going to the same interface as the incoming traffic, then it is assumed the traffic is not valid (or could be a bogus query). Besides the same argument from the previous solution that it probably not is ethical, because of blocking a few valid queries, there is another argument for not implementing this solution: Each incoming packet has to be checked and therefore the load on the DNS server actually would increase. This of course is the opposite we want to accomplish in the first place: reducing the load at the DNS server.

- There also is a solution not directly technically: simply contact the owner of a source IP address that is generating a lot of queries and ask him/her to take measures. This probably would be a good solution, which is kind of ironical, seen the more technical viewpoint in this report. We have talked about this solution with some DNS operators from RIPE NCC: if there would be a lot of traffic that is causing problems at the DNS root server, then this is a logical solution. Note: because the DNS root servers can cope with massive amount of queries, problems do not occur much. Often, this solution would only be used if the traffic is really causing problems.

- Another very pragmatic solution for the server side which at the moment often is used: just add more power. If a DNS server is having problems because of a large amount of queries, a simple solution is to add more machines and load balance. This solution is the most useful, because it is cheap and fast and therefore preferred.

We can see that there are a few solutions for the "client side" and a few for the "server side", but a few of them are hard or impossible or not ethical to implement. The best solutions are: contacting software vendors about bugs, contacting owners of top sources and simply placing additional servers.

Chapter 7

Conclusion

We discovered some interesting things about the captured data:

- For the AMS-IX server 80.7% is bogus and for the NAP server 14.65% bogus.
- For the AMS-IX server the top 10 IP addresses are responsible for 10.75% of all queries. For the NAP server the top 10 IP addresses are responsible for 42.40%.
- Our filters show that the AMS-IX server gets more bogus queries than the NAP server. We have an explanation for this (but cannot prove it because of lack of time): our scripts are checking complete source IP addresses, while 52.31% of all queries for the NAP server is coming from a 3 octet source (x.y.112).

So in the end we can answer our research question: "What are the possibilities to reduce the amount of bogus queries arriving at the DNS root servers?". Answer: we described a few possible solutions. Solutions for the "client" and "server" side. The most useful solutions:

- Contact software vendors about software bugs: if certain bugs that cause a lot of bogus queries would be fixed, that would decrease the amount of bogus queries.
- Contact owners of IP addresses that are causing massive amounts of bogus queries and tell them to take measures.
- What however in most cases is the most useful solution, because it is cheap and fast: just place additional servers to cope with the amount of bogus queries.

We can see that the final result maybe is a bit unexpected. The amount of bogus queries actually is not reduced at all: the problems that could come with the bogus queries are simply avoided by placing additional servers.

Chapter 8

Future work

There is still a lot of work that could be done on the field of bogus queries. We will now mention a few areas for future research:

- In our research we made some scripts that filter queries for each category of bogus queries we made earlier. It could be useful to make scripts that filter queries based on specific causes, like software failures and maybe others.

- Maybe it would be a good idea to do research on which exact applications produce the highest amount of queries. This should be done in an organised matter where the sources are researched in a structural manner. It would be interesting to know which exact applications are causing bogus queries.

- We can imagine there is an interest for RIPE NCC to do continuous monitoring of DNS bogus queries. The scripts and tools we made for this project could be helpful in accomplishing this. With such a project there should be made a design for an automated system that checks for bogus queries every day.

- Our scripts could be fine-tuned. Especially the script for TLDs that are not cached: in this script it should be taken into account for which (arpa-)domains the specific DNS root server is authoritative. The repeated script, no caching of TLDs script and same query ID script could be adjusted so that the source IP addresses could also be checked on only 3 octets.

Bibliography

[1] Internet Assigned Numbers Authority. `http://www.iana.org`.

[2] Réseau IP Européen Network Coordination Centre. `http://www.ripe.net`.

[3] Andre Broido Hao Shang Marina Fomenkov Young Hyun KC Claffy. The windows of private dns updates. *CAIDA, SDSC*, 2006. `http://www.caida.org/publications/papers/2006/private_dns_updates/private_dns_updates.pdf`.

[4] Duane Wessels Marina Fomenkov Nevil Brownlee KC Claffy. Measurements and laboratory simulations of the upper dns hierarchy. *CAIDA, San Diego Supercomputer Center, University of California*, 2004. `http://dns.measurement-factory.com/writings/wessels-pam2004-paper.pdf`.

[5] IANA country code top level domains. `http://www.iana.org/root-whois`.

[6] Y. Rekhter B. Moskowitz D. Karrenberg G. J. de Groot E. Lear. Address allocation for private internets. *Cisco Systems Chrysler Corp. RIPE NCC Silicon Graphics, Inc.*, 1996. `http://www.ietf.org/rfc/rfc1918.txt`.

[7] Ethereal. `http://www.ethereal.com`.

[8] Internet Corporation for Assigned Names and Numbers. `http://www.icann.org`.

[9] IANA global top level domains. `http://www.iana.org/gtld/gtld.htm`.

[10] IANA. Internet protocol v6 address space. *IANA*, 2006. `http://www.iana.org/assignments/ipv6-address-space`.

[11] IANA. Internet protocol v4 address space. *IANA*, 2007. `http://www.iana.org/assignments/ipv4-address-space`.

[12] The Measurement Factory Inc. http://dns.measurement-factory.com.

[13] Daniel Karrenberg. The internet domain name system explained for non-experts. *ISOC*, 2004. http://www.isoc.org/briefings/016/briefing16.pdf.

[14] Daniel Karrenberg. Dns root name servers explained for non-experts. *ISOC*, 2005. http://www.isoc.org/briefings/019/briefing19.pdf.

[15] Dr. John C. Klensin. Internationalizing top-level domain names: Another look. *ISOC*, 2004. http://www.isoc.org/briefings/018/briefing18.pdf.

[16] Wessels Duane Fomenkov Marina. Wow, that's a lot of packets. *CAIDA, San Diego Supercomputer Center, University of California*, 2003. http://dns.measurement-factory.com/writings/wessels-pam2003-paper.pdf.

[17] P. Mockapetris. Domain names - concepts and facilities. *USC/Information Scienses Institute*, 1987. http://www.ietf.org/rfc/rfc1034.txt.

[18] P. Mockapetris. Domain names - implementation and specification. *USC/Information Scienses Institute*, 1987. http://www.ietf.org/rfc/rfc1035.txt.

[19] Oneandone. http://faq.oneandone.co.uk.

[20] Erik Romijn. Determining and improving performance of anycast for dns root servers. *Hogeschool van Amsterdam*, 2006. http://www.uname.nl/wp-content/uploads/other/thesis/thesis.pdf.

[21] Erik Romijn. Dns root servers. *Hogeschool van Amsterdam*, 2006. http://homepages.staff.os3.nl/~karst/web/2006-2007/CIA/DNS_root_Romijn.pdf.

[22] DNS K root server. http://k.root-servers.org.

[23] Tcpdump. http://www.tcpdump.org/.

[24] Duane Wessels. Is your caching resolver polluting the internet? *CAIDA, The Measurement Factory Inc.*, 2004. http://dns.measurement-factory.com/writings/wessels-netts2004-paper.pdf.

[25] RIPE NCC whois database. http://www.ripe.net/whois.

Appendix A

Scripts

A.1 Merge.rb

```
#!/usr/bin/ruby -w

def create
   print "Name of the file (without dot and number): "
   @@name = gets.chomp
   print "Number of files to merge (always starting with 01): "
   @@n = gets.chomp.to_i
   print "Number of octets in IP address {3|4}: "
   @@octets = gets.chomp.to_i
   i = 1
   nr = ""
   @@sources = []
   @@destinations = []
   @@query = []
   @@opcode = []
   @@tld = []
   while i <= @@n
      if i < 10 then
         nr = "0" + i.to_s
      else
         nr = i.to_s
      end
      file = File.open("#{@@name}.#{nr}", "r")
      @section = ""
      file.each { |line|
         next if not line
         if ( line.split[0] == "Query" ) and
            ( line.split[1] == "Type" )
```

```
then
   x = "Query Type"
else
   x = line.split[0]
end
next if not x
next if line["-----"] == "-----"
next if line["You must"] == "You must"
next if line["collect"] == "collect"
if ( x == "Sources" ) or
   ( x == "Destinations" ) or
   ( x == "Query Type" ) or
   ( x == "Opcode" ) or
   ( x == "TLD" )
then
   @section = x
end
case @section
   when "Sources"
      if @@octets == 3 then
         if not x == "Sources" then
            ip = x.split(".")
            line2 = [ ip[0].to_i,
                      ip[1].to_i,
                      ip[2].to_i,
                      line.split[1],
                      line.split[2]
                    ]
            @@sources << line2
         end
      else
         if not x == "Sources" then
            ip = x.split(".")
            line2 = [ ip[0].to_i,
                      ip[1].to_i,
                      ip[2].to_i,
                      ip[3].to_i,
                      line.split[1],
                      line.split[2]
                    ]
            @@sources << line2
         end
      end
   when "Destinations"
```

```
            if @@octets == 3 then
               if not x == "Destinations" then
                  ip = x.split(".")
                  line2 = [ ip[0].to_i,
                            ip[1].to_i,
                            ip[2].to_i,
                            line.split[1],
                            line.split[2]
                          ]
                  @@destinations << line2
               end
            else
               if not x == "Destinations" then
                  ip = x.split(".")
                  line2 = [ ip[0].to_i,
                            ip[1].to_i,
                            ip[2].to_i,
                            ip[3].to_i,
                            line.split[1],
                            line.split[2]
                          ]
                  @@destinations << line2
               end
            end
         when "Query Type"
            if not x == "Query Type" then
               @@query << line.split
            end
         when "Opcode"
            if not x == "Opcode" then
               @@opcode << line.split
            end
         when "TLD"
            if not x == "TLD" then
               @@tld << line.split
            end
         else
            puts "No valid sections found in file!"
      end
   }
   i += 1
  end
end
```

```
def merge(section, name)
   all = section
   all.sort!
   @all2 = []
   @all3 = []
   @all4 = []
   @old_ip = -1
   @old_count = 0
   @old_perc = -1
   @new_ip = -1
   @new_count = -1
   @new_perc = -1
   if ( name == "Source" ) or ( name == "Destination" ) then
      if @@octets == 3 then
         all.each { |line|
            ip = line[0,3].join(".")
            @all4 << [ip, line[3], line[4]]
         }
      else
         all.each { |line|
            ip = line[0,4].join(".")
            @all4 << [ip, line[4], line[5]]
         }
      end
   else
      @all4 = all
   end
   @all4.each { |line|
      @new_ip = line[0]
      @new_count = line[1].to_i
      @new_perc = line[2].to_f
      if @new_ip == @old_ip then
         @old_count += @new_count
         if @old_perc == -1 then
            @old_perc = @new_perc
         else
            @old_perc += @new_perc
         end
      else
         @old_perc /= @@n
         @old_perc = (100 * @old_perc).round / 100.0
         if @old_ip == -1 then
            @old_ip = @new_ip
            @old_count = @new_count
            @old_perc = @new_perc
         end
         @all2 << [@old_ip, @old_count, @old_perc]
         @old_ip = @new_ip
         @old_count = @new_count
         @old_perc = @new_perc
      end
   }
   @@total << [name.center(17), "Count".center(17), "%".center(17)]
   @@total << ["--------------------------------------------"]
   @all3 = @all2.sort_by { |line| 1 - line[1] }
   @all3.each { |line| @@total << line }
   @@total << ["", "", ""]
end


@@total = []
create
merge(@@sources, "Source")
merge(@@destinations, "Destination")
merge(@@query, "Query Type")
merge(@@opcode, "Opcode")
merge(@@tld, "TLD")

if not @@octets == 3 then
   @@octets = 4
end

@@out = File.new("#{@@name}#{@@octets}.txt", "w")
@@total.each { |line|
   @@out.write("#{line[0].center(17)}
                #{line[1].to_s.center(17)}
                #{line[2].to_s.center(17)}\n")
}
puts "Result written to #{@@name}#{@@octets}.txt"
```

A.2 Bogusfilter.sh

```
#!/bin/sh

# rename files below 10
```

```
for (( i = 1; i<=9; i++ ))
do
   mv amsix.0$i amsix.$i
   mv nap.0$i nap.$i
done


# bz2 to txt

for (( i = 1; i<=9; i++ ))
do
   bzcat tcpdump.k.ams-ix.0$i.bz2 | tcpdump -n -r - -vv > amsix.0$i
done

for (( i = 10; i<=15; i++ ))
do
   bzcat tcpdump.k.ams-ix.$i.bz2 | tcpdump -n -r - -vv > amsix.$i
done

for (( i = 1; i<=9; i++ ))
do
   bzcat tcpdump.k.nap.0$i.bz2 | tcpdump -n -r - -vv > nap.0$i
done

for (( i = 10; i<=18; i++ ))
do
   bzcat tcpdump.k.nap.$i.bz2 | tcpdump -n -r - -vv > nap.$i
done


echo -e "seperating responses and no-responses...\n"

# separate respones and no-responses

for (( i = 1; i<=15; i++ ))
do
   cat amsix.$i | awk 'BEGIN { FS = " > " };
   { if ( $2 !~ / q: / ) print $0; }' > amsix.no-responses.$i
done

#for (( i = 1; i<=15; i++ ))
#do
#   cat amsix.$i | awk 'BEGIN { FS = " > " };
#   { if ( $2 ~ / q: / ) print $0; }' > amsix.responses.$i
```

```
#done

for (( i = 1; i<=18; i++ ))
do
   cat nap.$i | awk 'BEGIN { FS = " > " };
   { if ( $2 !~ / q: / ) print $0; }' > nap.no-responses.$i
done

#for (( i = 1; i<=18; i++ ))
#do
#   cat nap.$i | awk 'BEGIN { FS = " > " };
#   { if ( $2 ~ / q: / ) print $0; }' > nap.responses.$i
#done


echo -e "catting files...\n"

# cat

cat amsix.no-responses.* > amsix.no-responses.all
for (( i = 1; i<=15; i++ ))
do
   rm amsix.no-responses.$i
done

cat nap.no-responses.* > nap.no-responses.all
for (( i = 1; i<=18; i++ ))
do
   rm nap.no-responses.$i
done


echo -e "filtering queries...\n"

# queries

cat amsix.no-responses.all | awk 'BEGIN { FS = " > " };
{ if ( $2 ~ /? / ) print $0; }' > amsix.queries.all
cat nap.no-responses.all | awk 'BEGIN { FS = " > " };
{ if ( $2 ~ /? / ) print $0; }' > nap.queries.all

rm amsix.no-responses.all
rm nap.no-responses.all
```

```
echo -e "filtering a for a queries...\n"

# A-A filter

cat amsix.queries.all | awk --re-interval 'BEGIN { FS = " > " };
{ if ( $2 ~ / A{1,4}6{0,1}\? [0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}
\.[0-9]{1,3}\. / ) print $0; }' > amsix.queries.afora.all
cat amsix.queries.all | awk --re-interval 'BEGIN { FS = " > " };
{ if ( $2 !~ / A{1,4}6{0,1}\? [0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}
\.[0-9]{1,3}\. / ) print $0; }' > amsix.queries.no-bogus.all

cat nap.queries.all | awk --re-interval 'BEGIN { FS = " > " };
{ if ( $2 ~ / A{1,4}6{0,1}\? [0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}
\.[0-9]{1,3}\. / ) print $0; }' > nap.queries.afora.all
cat nap.queries.all | awk --re-interval 'BEGIN { FS = " > " };
{ if ( $2 !~ / A{1,4}6{0,1}\? [0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}
\.[0-9]{1,3}\. / ) print $0; }' > nap.queries.no-bogus.all


echo -e "filtering private ip reverse queries...\n"

# private ip reverse filter

cat amsix.queries.all | awk --re-interval 'BEGIN { FS = " > " };
{ IGNORECASE = 1 }; { if ( $2 ~ / PTR\? [0-9]{1,3}\.[0-9]{1,3}\.
([0-9]{1,3}\.10||[0-9]{1,3}\.127||3[0-1]\.172||2[0-9]\.172||1[0-6]
\.172||168\.192)\.in-addr.arpa. / ) print $0; }'
> amsix.queries.private.all

#cat amsix.queries.all | awk --re-interval 'BEGIN { FS = " > " };
{ IGNORECASE = 1 }; { if ( $2 !~ / PTR\? [0-9]{1,3}\.[0-9]{1,3}\.
([0-9]{1,3}\.10||[0-9]{1,3}\.127||3[0-1]\.172||2[0-9]\.172||1[0-6]
\.172||168\.192)\.in-addr.arpa. / ) print $0; }'
> amsix.queries.no-private.all

cat nap.queries.all | awk --re-interval 'BEGIN { FS = " > " };
{ IGNORECASE = 1 }; { if ( $2 ~ / PTR\? [0-9]{1,3}\.[0-9]{1,3}\.
([0-9]{1,3}\.10||[0-9]{1,3}\.127||3[0-1]\.172||2[0-9]\.172||1[0-6]
\.172||168\.192)\.in-addr.arpa. / ) print $0; }'
> nap.queries.private.all

#cat nap.no-responses.all | awk --re-interval 'BEGIN { FS = " > " };
{ IGNORECASE = 1 }; { if ( $2 !~ / PTR\? [0-9]{1,3}\.[0-9]{1,3}\.
([0-9]{1,3}\.10||[0-9]{1,3}\.127||3[0-1]\.172||2[0-9]\.172||1[0-6]
\.172||168\.192)\.in-addr.arpa. / ) print $0; }'
> nap.no-responses.no-private.all

# extra for filtering

cat amsix.queries.no-bogus.all | awk --re-interval 'BEGIN
{ FS = " > " }; { IGNORECASE = 1 }; { if ( $2 !~
/ PTR\? [0-9]{1,3}\.[0-9]{1,3}\.([0-9]{1,3}\.10||[0-9]{1,3}
\.127||3[0-1]\.172||2[0-9]\.172||1[0-6]\.172||168\.192)
\.in-addr.arpa. / ) print $0; }' > amsix.queries.no-bogus2.all

rm amsix.queries.no-bogus.all
mv amsix.queries.no-bogus2.all amsix.queries.no-bogus.all

cat nap.queries.no-bogus.all | awk --re-interval 'BEGIN
{ FS = " > " }; { IGNORECASE = 1 }; { if ( $2 !~
/ PTR\? [0-9]{1,3}\.[0-9]{1,3}\.([0-9]{1,3}\.10||[0-9]{1,3}
\.127||3[0-1]\.172||2[0-9]\.172||1[0-6]\.172||168\.192)
\.in-addr.arpa. / ) print $0; }' > nap.queries.no-bogus2.all

rm nap.queries.no-bogus.all
mv nap.queries.no-bogus2.all nap.queries.no-bogus.all


echo -e "filtering local domain queries...\n"

# local domain filter

cat amsix.queries.all | awk --re-interval 'BEGIN { FS = " > " };
{ IGNORECASE = 1 }; { if ( $2 ~ /\.local\. / || $2 ~ /\.localhost\. /
|| $2 ~ /\.lan\. / || $2 ~ /\.locale\. / || $2 ~ /\.home\. /
|| $2 ~ /\.localdomain\. / || $2 ~ /\.intra\. / || $2 ~ /\.loc\. /
|| $2 ~ /\.domain\. / || $2 ~ /\.workgroup\. / || $2 ~ /\.lokaal\. / )
print $0; }' > amsix.queries.local.all

#cat amsix.queries.all | awk --re-interval 'BEGIN { FS = " > " };
{ IGNORECASE = 1 }; { if ( $2 !~ /\.local\. / && $2 !~ /\.localhost\. /
&& $2 !~ /\.lan\. / && $2 !~ /\.locale\. / && $2 !~ /\.home\. /
&& $2 !~ /\.localdomain\. / && $2 !~ /\.intra\. / && $2 !~ /\.loc\. /
&& $2 !~ /\.domain\. / && $2 !~ /\.workgroup\. / && $2 !~ /\.lokaal\. / )
print $0; }' > amsix.queries.no-local.all

cat nap.queries.all | awk --re-interval 'BEGIN { FS = " > " };
```

```
{ IGNORECASE = 1 }; { if ( $2 ~ /\.local\. / || $2 ~ /\.localhost\. /
|| $2 ~ /\.lan\. / || $2 ~ /\.locale\. / || $2 ~ /\.home\. /
|| $2 ~ /\.localdomain\. / || $2 ~ /\.intra\. / || $2 ~ /\.loc\. /
|| $2 ~ /\.domain\. / || $2 ~ /\.workgroup\. / || $2 ~ /\.lokaal\. / )
print $0; }' > nap.queries.local.all

#cat nap.no-responses.all | awk --re-interval 'BEGIN { FS = " > " };
{ IGNORECASE = 1 }; { if ( $2 !~ /\.local\. / && $2 !~ /\.localhost\. /
&& $2 !~ /\.lan\. / && $2 !~ /\.locale\. / && $2 !~ /\.home\. /
&& $2 !~ /\.localdomain\. / && $2 !~ /\.intra\. / && $2 !~ /\.loc\. /
&& $2 !~ /\.domain\. / && $2 !~ /\.workgroup\. / && $2 !~ /\.lokaal\. / )
print $0; }' > nap.queries.no-local.all

# extra for filtering

cat amsix.queries.no-bogus.all | awk --re-interval 'BEGIN { FS = " > " };
{ IGNORECASE = 1 }; { if ( $2 !~ /\.local\. / && $2 !~ /\.localhost\. /
&& $2 !~ /\.lan\. / && $2 !~ /\.locale\. / && $2 !~ /\.home\. /
&& $2 !~ /\.localdomain\. / && $2 !~ /\.intra\. / && $2 !~ /\.loc\. /
&& $2 !~ /\.domain\. / && $2 !~ /\.workgroup\. / && $2 !~ /\.lokaal\. / )
print $0; }' > amsix.queries.no-bogus2.all

rm amsix.queries.no-bogus.all
mv amsix.queries.no-bogus2.all amsix.queries.no-bogus.all

cat nap.queries.no-bogus.all | awk --re-interval 'BEGIN { FS = " > " };
{ IGNORECASE = 1 }; { if ( $2 !~ /\.local\. / && $2 !~ /\.localhost\. /
&& $2 !~ /\.lan\. / && $2 !~ /\.locale\. / && $2 !~ /\.home\. /
&& $2 !~ /\.localdomain\. / && $2 !~ /\.intra\. / && $2 !~ /\.loc\. /
&& $2 !~ /\.domain\. / && $2 !~ /\.workgroup\. / && $2 !~ /\.lokaal\. / )
print $0; }' > nap.queries.no-bogus2.all

rm nap.queries.no-bogus.all
mv nap.queries.no-bogus2.all nap.queries.no-bogus.all


echo -e "filtering invalid domain queries...\n"

# invalid domain filter

cat amsix.queries.all | awk --re-interval 'BEGIN { FS = " > " };
{ IGNORECASE = 1 }; { if ( $2 ~ /\.speedport_w_700v\. /
|| $2 ~ /\.invalid\. / || $2 ~ /\.site\. / || $2 ~ /\.belkin\. /
|| $2 ~ /\.speedport_w_500\. / || $2 ~ /\.txt\. / || $2 ~ /\.htm\. / )
print $0; }' > amsix.queries.invalid.all

#cat amsix.queries.all | awk --re-interval 'BEGIN { FS = " > " };
{ IGNORECASE = 1 }; { if ( $2 !~ /\.speedport_w_700v\. /
&& $2 !~ /\.invalid\. / && $2 !~ /\.site\. / && $2 !~ /\.belkin\. /
&& $2 !~ /\.speedport_w_500\. / && $2 !~ /\.txt\. / && $2 !~ /\.htm\. / )
print $0; }' > amsix.queries.no-invalid.all

cat nap.queries.all | awk --re-interval 'BEGIN { FS = " > " };
{ IGNORECASE = 1 }; { if ( $2 ~ /\.speedport_w_700v\. /
|| $2 ~ /\.invalid\. / || $2 ~ /\.site\. / || $2 ~ /\.belkin\. /
|| $2 ~ /\.speedport_w_500\. / || $2 ~ /\.txt\. / || $2 ~ /\.htm\. / )
print $0; }' > nap.queries.invalid.all

#cat nap.no-responses.all | awk --re-interval 'BEGIN { FS = " > " };
{ IGNORECASE = 1 }; { if ( $2 !~ /\.speedport_w_700v\. /
&& $2 !~ /\.invalid\. / && $2 !~ /\.site\. / && $2 !~ /\.belkin\. /
&& $2 !~ /\.speedport_w_500\. / && $2 !~ /\.txt\. / && $2 !~ /\.htm\. / )
print $0; }' > nap.queries.no-invalid.all

# extra for filtering

cat amsix.queries.no-bogus.all | awk --re-interval 'BEGIN { FS = " > " };
{ IGNORECASE = 1 }; { if ( $2 !~ /\.speedport_w_700v\. /
&& $2 !~ /\.invalid\. / && $2 !~ /\.site\. / && $2 !~ /\.belkin\. /
&& $2 !~ /\.speedport_w_500\. / && $2 !~ /\.txt\. / && $2 !~ /\.htm\. / )
print $0; }' > amsix.queries.no-bogus2.all

rm amsix.queries.no-bogus.all
mv amsix.queries.no-bogus2.all amsix.queries.no-bogus.all

cat nap.queries.no-bogus.all | awk --re-interval 'BEGIN { FS = " > " };
{ IGNORECASE = 1 }; { if ( $2 !~ /\.speedport_w_700v\. /
&& $2 !~ /\.invalid\. / && $2 !~ /\.site\. / && $2 !~ /\.belkin\. /
&& $2 !~ /\.speedport_w_500\. / && $2 !~ /\.txt\. / && $2 !~ /\.htm\. / )
print $0; }' > nap.queries.no-bogus2.all

rm nap.queries.no-bogus.all
mv nap.queries.no-bogus2.all nap.queries.no-bogus.all


echo -e "filtering invalid ip queries...\n"

cat amsix.queries.all | awk --re-interval 'BEGIN { FS = " > " };
```

```
{ IGNORECASE = 1 }; { if ( $2 ~ /\.(0||233||24[0-9]||25[0-5])
\.in-addr\.arpa\./ ) print $0; }' > amsix.queries.invalidip.all

cat nap.queries.all | awk --re-interval 'BEGIN { FS = " > " };
{ IGNORECASE = 1 }; { if ( $2 ~ /\.(0||233||24[0-9]||25[0-5])
\.in-addr\.arpa\./ ) print $0; }' > nap.queries.invalidip.all

cat amsix.queries.no-bogus.all | awk --re-interval 'BEGIN { FS = " > " };
{ IGNORECASE = 1 }; { if ( $2 !~ /\.(0||233||24[0-9]||25[0-5])
\.in-addr\.arpa\./ ) print $0; }' > amsix.queries.no-bogus2.all

rm amsix.queries.no-bogus.all
mv amsix.queries.no-bogus2.all amsix.queries.no-bogus.all

cat nap.queries.no-bogus.all | awk --re-interval 'BEGIN { FS = " > " };
{ IGNORECASE = 1 }; { if ( $2 !~ /\.(0||233||24[0-9]||25[0-5])
\.in-addr\.arpa\./ ) print $0; }' > nap.queries.no-bogus2.all

rm nap.queries.no-bogus.all
mv nap.queries.no-bogus2.all nap.queries.no-bogus.all


echo -e "filtering repeated queries...\n"

# repeated filter

./repeat.rb amsix.queries.all
rm amsix.queries.all.no-bogus
mv amsix.queries.all.repeated amsix.queries.repeated.all

./repeat.rb nap.queries.all
rm nap.queries.all.no-bogus
mv nap.queries.all.repeated nap.queries.repeated.all

./repeat.rb amsix.queries.no-bogus.all
rm amsix.queries.no-bogus.all.repeated
mv amsix.queries.no-bogus.all.no-bogus amsix.queries.no-bogus.all

./repeat.rb nap.queries.no-bogus.all
rm nap.queries.no-bogus.all.repeated
mv nap.queries.no-bogus.all.no-bogus nap.queries.no-bogus.all


echo -e "filtering same query ids queries...\n"
```

```
# id filter

./id.rb amsix.queries.all
rm amsix.queries.all.no-bogus
mv amsix.queries.all.id amsix.queries.id.all

 ./id.rb nap.queries.all
rm nap.queries.all.no-bogus
mv nap.queries.all.id nap.queries.id.all

./id.rb amsix.queries.no-bogus.all
rm amsix.queries.no-bogus.all.id
mv amsix.queries.no-bogus.all.no-bogus amsix.queries.no-bogus.all

./id.rb nap.queries.no-bogus.all
rm nap.queries.no-bogus.all.id
mv nap.queries.no-bogus.all.no-bogus nap.queries.no-bogus.all


echo -e "filtering no-cache queries...\n"

# id filter

./nocache.rb amsix.queries.all
rm amsix.queries.all.no-bogus
mv amsix.queries.all.notcached amsix.queries.notcached.all

 ./nocache.rb nap.queries.all
rm nap.queries.all.no-bogus
mv nap.queries.all.notcached nap.queries.notcached.all

./nocache.rb amsix.queries.no-bogus.all
rm amsix.queries.no-bogus.all.notcached
mv amsix.queries.no-bogus.all.no-bogus amsix.queries.no-bogus.all

./nocache.rb nap.queries.no-bogus.all
rm nap.queries.no-bogus.all.notcached
mv nap.queries.no-bogus.all.no-bogus nap.queries.no-bogus.all

echo -e "done\n\nlook in server.queries.no-bogus.all"
```

A.3 Repeat.rb

```
#!/usr/bin/ruby -w

def split
   @@input = ARGV[0]
   file = File.open("#{@@input}", "r")
   @@folder = "temp.#{@@input}"
   Dir.mkdir(@@folder)
   @out_a = File.new("#{@@folder}/a", "w")
   @out_b = File.new("#{@@folder}/b", "w")
   @out_c = File.new("#{@@folder}/c", "w")
   @out_d = File.new("#{@@folder}/d", "w")
   @out_e = File.new("#{@@folder}/e", "w")
   @out_f = File.new("#{@@folder}/f", "w")
   @out_g = File.new("#{@@folder}/g", "w")
   @out_h = File.new("#{@@folder}/h", "w")
   @out_i = File.new("#{@@folder}/i", "w")
   @out_j = File.new("#{@@folder}/j", "w")
   @out_k = File.new("#{@@folder}/k", "w")
   @out_l = File.new("#{@@folder}/l", "w")
   @out_m = File.new("#{@@folder}/m", "w")
   @out_ns1 = File.new("#{@@folder}/ns1", "w")
   @out_ns2 = File.new("#{@@folder}/ns2", "w")
   @out_ns3 = File.new("#{@@folder}/ns3", "w")
   @out_ns4 = File.new("#{@@folder}/ns4", "w")
   @out_nz = File.new("#{@@folder}/nz", "w")
   @out_o = File.new("#{@@folder}/o", "w")
   @out_p = File.new("#{@@folder}/p", "w")
   @out_q = File.new("#{@@folder}/q", "w")
   @out_r = File.new("#{@@folder}/r", "w")
   @out_s = File.new("#{@@folder}/s", "w")
   @out_t = File.new("#{@@folder}/t", "w")
   @out_u = File.new("#{@@folder}/u", "w")
   @out_v = File.new("#{@@folder}/v", "w")
   @out_w = File.new("#{@@folder}/w", "w")
   @out_x = File.new("#{@@folder}/x", "w")
   @out_y = File.new("#{@@folder}/y", "w")
   @out_z = File.new("#{@@folder}/z", "w")
   @out_0 = File.new("#{@@folder}/0", "w")
   @out_1 = File.new("#{@@folder}/1", "w")
   @out_2 = File.new("#{@@folder}/2", "w")
   @out_3 = File.new("#{@@folder}/3", "w")
   @out_4 = File.new("#{@@folder}/4", "w")
```

```
@out_5 = File.new("#{@@folder}/5", "w")
@out_6 = File.new("#{@@folder}/6", "w")
@out_7 = File.new("#{@@folder}/7", "w")
@out_8 = File.new("#{@@folder}/8", "w")
@out_9 = File.new("#{@@folder}/9", "w")
@out_others = File.new("#{@@folder}/zz", "w")
file.each { |line|
   if line.split(" > ")[1].split("? ")[1] != nil then
      if line.split(" > ")[1].split("? ")[1].split[0] != nil then
         x = line.split(" > ")[1].split("? ")[1].split[0]
         case x[0,1]
            when /[aA]/
               @out_a.write(line)
            when /[bB]/
               @out_b.write(line)
            when /[cC]/
               @out_c.write(line)
            when /[dD]/
               @out_d.write(line)
            when /[eE]/
               @out_e.write(line)
            when /[fF]/
               @out_f.write(line)
            when /[gG]/
               @out_g.write(line)
            when /[hH]/
               @out_h.write(line)
            when /[iI]/
               @out_i.write(line)
            when /[jJ]/
               @out_j.write(line)
            when /[kK]/
               @out_k.write(line)
            when /[lL]/
               @out_l.write(line)
            when /[mM]/
               @out_m.write(line)
            when /[nN]/
               if x[1,2] == "s1" then
                  @out_ns1.write(line)
               end
               if x[1,2] == "s2" then
                  @out_ns2.write(line)
               end
               if x[1,2] == "s3" then
                  @out_ns3.write(line)
               end
               if x[1,2] == "s." then
                  @out_ns4.write(line)
               end
               if ( x[1,2] != "s1" ) and
                  ( x[1,2] != "s2" ) and
                  ( x[1,2] != "s3" ) and
                  ( x[1,2] != "s." )
               then
                  @out_nz.write(line)
               end
            when /[oO]/
               @out_o.write(line)
            when /[pP]/
               @out_p.write(line)
            when /[qQ]/
               @out_q.write(line)
            when /[rR]/
               @out_r.write(line)
            when /[sS]/
               @out_s.write(line)
            when /[tT]/
               @out_t.write(line)
            when /[uU]/
               @out_u.write(line)
            when /[vV]/
               @out_v.write(line)
            when /[wW]/
               @out_w.write(line)
            when /[xX]/
               @out_x.write(line)
            when /[yY]/
               @out_y.write(line)
            when /[zZ]/
               @out_z.write(line)
            when "0"
               @out_0.write(line)
            when "1"
               @out_1.write(line)
            when "2"
               @out_2.write(line)
            when "3"
```

```
                  @out_3.write(line)
               when "4"
                  @out_4.write(line)
               when "5"
                  @out_5.write(line)
               when "6"
                  @out_6.write(line)
               when "7"
                  @out_7.write(line)
               when "8"
                  @out_8.write(line)
               when "9"
                  @out_9.write(line)
               else
                  @out_others.write(line)
            end
         end
      end
   }
end


def repeat(input)

   file = File.open("#{@@folder}/#{input}", "r")

   @@file2 = []

   file.each { |line|
      if line.split(" > ")[1] != nil then
         if line.split(" > ")[1].split("? ")[1] != nil then
            if line.split(" > ")[1].split("? ")[1].split[0] != nil then
               @@file2 << line
            end
         end
      end
   }

   @@sorted_by_query = []

   @@sorted_by_query = @@file2.sort_by { |line|
      if line.split(" > ")[1] != nil then
         if line.split(" > ")[1].split("? ")[1] != nil then
            if line.split(" > ")[1].split("? ")[1].split[0] != nil then
               line.split(" > ")[1].split("? ")[1].split[0]
            end
         end
      end
   }

   @@old_query = ""
   @@new_query = ""
   @@temp = []
   @@sorted_by_source = []
   @@nonbogus = []
   @@nonbogus_temp = []

   @@sorted_by_query.each { |line|
      @@new_query = line.split(" > ")[1].split("? ")[1].split[0]
      if @@new_query == @@old_query then
         @@temp << line
         @@old_query = @@new_query
      else
         @@old_query = @@new_query
         @@nonbogus_temp << line
         if ( @@temp != [] ) and ( @@old_query != "" ) then
            subsorted_by_source = []
            subsorted_by_source = @@temp.sort_by { |line|
               if line.split(" > ")[0].split[-1] != nil then
                  line.split(" > ")[0].split[-1]
               end
            }
            subsorted_by_source.each { |line|
               @@sorted_by_source << line
            }
            @@temp = []
         end
      end
   }

   @@old_query = ""
   @@new_query = ""
   @@old_source = ""
   @@new_source = ""
   @@temp = []
   @@sorted_by_time = []

   @@sorted_by_source.each { |line|
      @@new_query = line.split(" > ")[1].split("? ")[1].split[0]
      @@new_source = line.split(" > ")[0].split[-1]
      if ( @@new_query == @@old_query ) and
         ( @@new_source == @@old_source )
      then
         @@temp << line
         @@old_query = @@new_query
         @@old_source = @@new_source
      else
         @@old_query = @@new_query
         @@old_source = @@new_source
         @@nonbogus_temp << line
         if ( @@temp != [] ) and
            ( @@old_query != "" ) and
            ( @@old_source != "" )
         then
            subsorted_by_time = []
            subsorted_by_time = @@temp.sort_by { |line|
               line.split[0]
            }
            subsorted_by_time.each { |line|
               @@sorted_by_time << line
            }
            @@temp = []
         end
      end
   }

   @@nonbogus = @@nonbogus_temp.sort_by { |line|
      line.split[0]
   }

   @@file_bogus = File.new("#{@@input}.bogusfolder/
                            #{input}.repeated", "w")
   @@file_nonbogus = File.new("#{@@input}.nonbogusfolder/
                               #{input}.no-bogus", "w")

   @@sorted_by_time.each { |line|
      @@file_bogus.write(line)
   }

   @@nonbogus.each { |line|
      @@file_nonbogus.write(line)
   }
```

```
end


split

Dir.mkdir("#{@@input}.bogusfolder")
Dir.mkdir("#{@@input}.nonbogusfolder")

Dir.foreach(@@folder) { |file|
   if ( file != "." ) and ( file != ".." ) then
      repeat(file)
      system("rm #{@@folder}/#{file}")
   end
}

system("rm -r #{@@folder}")

system("touch #{@@input}.repeated")
system("touch #{@@input}.no-bogus")

Dir.foreach("#{@@input}.bogusfolder") { |file|
   if ( file != "." ) and ( file != ".." ) then
      system("cat #{@@input}.repeated
                  #{@@input}.bogusfolder/#{file} >
                  #{@@input}.repeated.temp")
      system("rm #{@@input}.repeated")
      system("mv #{@@input}.repeated.temp #{@@input}.repeated")
   end
}

system("rm -r #{@@input}.bogusfolder")

Dir.foreach("#{@@input}.nonbogusfolder") { |file|
   if ( file != "." ) and ( file != ".." ) then
      system("cat #{@@input}.no-bogus
                  #{@@input}.nonbogusfolder/#{file}
                  > #{@@input}.no-bogus.temp")
      system("rm #{@@input}.no-bogus")
      system("mv #{@@input}.no-bogus.temp #{@@input}.no-bogus")
   end
}

system("rm -r #{@@input}.nonbogusfolder")
```

A.4 ID.rb

```
#!/usr/bin/ruby -w

def split
   @@input = ARGV[0]
   file = File.open("#{@@input}", "r")
   @@folder = "temp.#{@@input}"
   Dir.mkdir(@@folder)
   @out_0 = File.new("#{@@folder}/0", "w")
   @out_1 = File.new("#{@@folder}/1", "w")
   @out_2 = File.new("#{@@folder}/2", "w")
   @out_3 = File.new("#{@@folder}/3", "w")
   @out_4 = File.new("#{@@folder}/4", "w")
   @out_5 = File.new("#{@@folder}/5", "w")
   @out_6 = File.new("#{@@folder}/6", "w")
   @out_7 = File.new("#{@@folder}/7", "w")
   @out_8 = File.new("#{@@folder}/8", "w")
   @out_9 = File.new("#{@@folder}/9", "w")
   file.each { |line|
      if line.split[7] != nil then
         x = line.split[7]
         case x[0,1]
            when "0"
               @out_0.write(line)
            when "1"
               @out_1.write(line)
            when "2"
               @out_2.write(line)
            when "3"
               @out_3.write(line)
            when "4"
               @out_4.write(line)
            when "5"
               @out_5.write(line)
            when "6"
               @out_6.write(line)
            when "7"
               @out_7.write(line)
            when "8"
               @out_8.write(line)
            when "9"
               @out_9.write(line)
         end
      end
   }
end


def sameid(input)

   file = File.open("#{@@folder}/#{input}", "r")

   @@file2 = []

   file.each { |line|
      if line.split[7] != nil then
         @@file2 << line
      end
   }

   @@sorted_by_id = []

   @@sorted_by_id = @@file2.sort_by { |line|
      if line.split[7]!= nil then
         line.split[7]
      end
   }

   @@old_id = ""
   @@new_id = ""
   @@temp = []
   @@sorted_by_source = []
   @@nonbogus = []
   @@nonbogus_temp = []

   @@sorted_by_id.each { |line|
      @@new_id = line.split[7]
      if @@new_id == @@old_id then
         @@temp << line
         @@old_id = @@new_id
      else
         @@old_id = @@new_id
         @@nonbogus_temp << line
         if ( @@temp != [] ) and ( @@old_id != "" ) then
            subsorted_by_source = []
            subsorted_by_source = @@temp.sort_by { |line|
               if line.split(" > ")[0].split[-1] != nil then
```

```
                line.split(" > ")[0].split[-1]
             end
          }
          subsorted_by_source.each { |line|
             @@sorted_by_source << line
          }
          @@temp = []
       end
    end
}

@@old_id = ""
@@new_id = ""
@@old_source = ""
@@new_source = ""
@@temp = []
@@sorted_by_time = []

@@sorted_by_source.each { |line|
   @@new_id = line.split[7]
   @@new_source = line.split(" > ")[0].split[-1]
   if ( @@new_id == @@old_id ) and
      ( @@new_source == @@old_source )
   then
      @@temp << line
      @@old_id = @@new_id
      @@old_source = @@new_source
   else
      @@old_id = @@new_id
      @@old_source = @@new_source
      @@nonbogus_temp << line
      if ( @@temp != [] ) and
         ( @@old_id != "" ) and
         ( @@old_source != "" )
      then
         subsorted_by_time = []
         subsorted_by_time = @@temp.sort_by { |line|
            line.split[0]
         }
         subsorted_by_time.each { |line|
            @@sorted_by_time << line
         }
         @@temp = []
      end
      end
   }

   @@nonbogus = @@nonbogus_temp.sort_by { |line|
      line.split[0]
   }

   @@file_bogus = File.new("#{@@input}.bogusfolder/
                           #{input}.id", "w")
   @@file_nonbogus = File.new("#{@@input}.nonbogusfolder/
                              #{input}.no-bogus", "w")

   @@sorted_by_time.each { |line|
      @@file_bogus.write(line)
   }

   @@nonbogus.each { |line|
      @@file_nonbogus.write(line)
   }

end


split

Dir.mkdir("#{@@input}.bogusfolder")
Dir.mkdir("#{@@input}.nonbogusfolder")

Dir.foreach(@@folder) { |file|
   if ( file != "." ) and ( file != ".." ) then
      sameid(file)
      system("rm #{@@folder}/#{file}")
   end
}

system("rm -r #{@@folder}")

system("touch #{@@input}.id")
system("touch #{@@input}.no-bogus")

Dir.foreach("#{@@input}.bogusfolder") { |file|
   if ( file != "." ) and ( file != ".." ) then
      system("cat #{@@input}.id
                  #{@@input}.bogusfolder/#{file}
```

```
                > #{@@input}.id.temp")
      system("rm #{@@input}.id")
      system("mv #{@@input}.id.temp #{@@input}.id")
   end
}

system("rm -r #{@@input}.bogusfolder")

Dir.foreach("#{@@input}.nonbogusfolder") { |file|
   if ( file != "." ) and ( file != ".." ) then
      system("cat #{@@input}.no-bogus
               #{@@input}.nonbogusfolder/#{file}
               > #{@@input}.no-bogus.temp")
      system("rm #{@@input}.no-bogus")
      system("mv #{@@input}.no-bogus.temp #{@@input}.no-bogus")
   end
}

system("rm -r #{@@input}.nonbogusfolder")
```

A.5 Nocache.rb

```
#!/usr/bin/ruby -w

def split
   @@input = ARGV[0]
   file = File.open("#{@@input}", "r")
   @@folder = "temp.#{@@input}"
   Dir.mkdir(@@folder)
   @out_a = File.new("#{@@folder}/a", "w")
   @out_b = File.new("#{@@folder}/b", "w")
   @out_c = File.new("#{@@folder}/c", "w")
   @out_d = File.new("#{@@folder}/d", "w")
   @out_e = File.new("#{@@folder}/e", "w")
   @out_f = File.new("#{@@folder}/f", "w")
   @out_g = File.new("#{@@folder}/g", "w")
   @out_h = File.new("#{@@folder}/h", "w")
   @out_i = File.new("#{@@folder}/i", "w")
   @out_j = File.new("#{@@folder}/j", "w")
   @out_k = File.new("#{@@folder}/k", "w")
   @out_l = File.new("#{@@folder}/l", "w")
   @out_m = File.new("#{@@folder}/m", "w")
   @out_n = File.new("#{@@folder}/n", "w")
```

```
@out_o = File.new("#{@@folder}/o", "w")
@out_p = File.new("#{@@folder}/p", "w")
@out_q = File.new("#{@@folder}/q", "w")
@out_r = File.new("#{@@folder}/r", "w")
@out_s = File.new("#{@@folder}/s", "w")
@out_t = File.new("#{@@folder}/t", "w")
@out_u = File.new("#{@@folder}/u", "w")
@out_v = File.new("#{@@folder}/v", "w")
@out_w = File.new("#{@@folder}/w", "w")
@out_x = File.new("#{@@folder}/x", "w")
@out_y = File.new("#{@@folder}/y", "w")
@out_z = File.new("#{@@folder}/z", "w")
@out_0 = File.new("#{@@folder}/0", "w")
@out_1 = File.new("#{@@folder}/1", "w")
@out_2 = File.new("#{@@folder}/2", "w")
@out_3 = File.new("#{@@folder}/3", "w")
@out_4 = File.new("#{@@folder}/4", "w")
@out_5 = File.new("#{@@folder}/5", "w")
@out_6 = File.new("#{@@folder}/6", "w")
@out_7 = File.new("#{@@folder}/7", "w")
@out_8 = File.new("#{@@folder}/8", "w")
@out_9 = File.new("#{@@folder}/9", "w")
@out_others = File.new("#{@@folder}/zz", "w")
file.each { |line|
   if line.split(" > ")[1].split("? ")[1] != nil then
      if line.split(" > ")[1].split("? ")[1].split[0] != nil then
         if line.split(" > ")[1].split("? ")
            [1].split[0].split(".")[-1] != nil
         then
         x = line.split(" > ")[1].split("? ")[1].split[0].split(".")[-1]
         case x[0,1]
            when /[aA]/
               @out_a.write(line)
            when /[bB]/
               @out_b.write(line)
            when /[cC]/
               @out_c.write(line)
            when /[dD]/
               @out_d.write(line)
            when /[eE]/
               @out_e.write(line)
            when /[fF]/
               @out_f.write(line)
            when /[gG]/
               @out_g.write(line)
            when /[hH]/
               @out_h.write(line)
            when /[iI]/
               @out_i.write(line)
            when /[jJ]/
               @out_j.write(line)
            when /[kK]/
               @out_k.write(line)
            when /[lL]/
               @out_l.write(line)
            when /[mM]/
               @out_m.write(line)
            when /[nN]/
               @out_n.write(line)
            when /[oO]/
               @out_o.write(line)
            when /[pP]/
               @out_p.write(line)
            when /[qQ]/
               @out_q.write(line)
            when /[rR]/
               @out_r.write(line)
            when /[sS]/
               @out_s.write(line)
            when /[tT]/
               @out_t.write(line)
            when /[uU]/
               @out_u.write(line)
            when /[vV]/
               @out_v.write(line)
            when /[wW]/
               @out_w.write(line)
            when /[xX]/
               @out_x.write(line)
            when /[yY]/
               @out_y.write(line)
            when /[zZ]/
               @out_z.write(line)
            when "0"
               @out_0.write(line)
            when "1"
               @out_1.write(line)
            when "2"
```

```
               @out_2.write(line)
            when "3"
               @out_3.write(line)
            when "4"
               @out_4.write(line)
            when "5"
               @out_5.write(line)
            when "6"
               @out_6.write(line)
            when "7"
               @out_7.write(line)
            when "8"
               @out_8.write(line)
            when "9"
               @out_9.write(line)
            else
               @out_others.write(line)
         end
         end
      end
   end
  }
end


def repeat(input)

  file = File.open("#{@@folder}/#{input}", "r")

  @@file2 = []

  file.each { |line|
     if line.split(" > ")[1] != nil then
        if line.split(" > ")[1].split("? ")[1] != nil then
           if line.split(" > ")[1].split("? ")[1].split[0] != nil then
              if line.split(" > ")[1].split("? ")
                 [1].split[0].split(".")[-1] != nil
              then
                 @@file2 << line
              end
           end
        end
     end
  }
```

```
@@sorted_by_tld = []

@@sorted_by_tld = @@file2.sort_by { |line|
   if line.split(" > ")[1] != nil then
      if line.split(" > ")[1].split("? ")[1] != nil then
         if line.split(" > ")[1].split("? ")[1].split[0] != nil then
            if line.split(" > ")[1].split("? ")
               [1].split[0].split(".")[-1] != nil
            then
               line.split(" > ")[1].split("? ")[1].split[0].split(".")[-1]
            end
         end
      end
   end
}

@@old_tld = ""
@@new_tld = ""
@@temp = []
@@sorted_by_source = []
@@nonbogus = []
@@nonbogus_temp = []

@@sorted_by_tld.each { |line|
   x = line.split(" > ")[1].split("? ")[1].split[0].split(".")
   if ( ( x[-1] =~ /arpa/i ) == 0 ) and
      ( ( x[-2] =~ /in-addr/i ) == 0 ) and
      ( x[-3] != nil )
   then
      @@new_tld = x[-3,3].join(".")
   else
      @@new_tld = x[-1]
   end
   if @@new_tld == @@old_tld then
      @@temp << line
      @@old_tld = @@new_tld
   else
      @@old_tld = @@new_tld
      @@nonbogus_temp << line
      if ( @@temp != [] ) and ( @@old_tld != "" ) then
         subsorted_by_source = []
         subsorted_by_source = @@temp.sort_by { |line|
            if line.split(" > ")[0].split[-1] != nil then
```

```
                line.split(" > ")[0].split[-1]
             end
          }
          subsorted_by_source.each { |line|
             @@sorted_by_source << line
          }
          @@temp = []
       end
    end
 }

 @@old_tld = ""
 @@new_tld = ""
 @@old_source = ""
 @@new_source = ""
 @@temp = []
 @@sorted_by_time = []

 @@sorted_by_source.each { |line|
    x = line.split(" > ")[1].split("? ")[1].split[0].split(".")
    if ( ( x[-1] =~ /arpa/i ) == 0 ) and
       ( ( x[-2] =~ /in-addr/i ) == 0 ) and
       ( x[-3] != nil )
    then
       @@new_tld = x[-3,3].join(".")
    else
       @@new_tld = x[-1]
    end
    @@new_source = line.split(" > ")[0].split[-1]
    if ( @@new_tld == @@old_tld ) and
       ( @@new_source == @@old_source )
    then
       @@temp << line
       @@old_tld = @@new_tld
       @@old_source = @@new_source
    else
       @@old_tld = @@new_tld
       @@old_source = @@new_source
       @@nonbogus_temp << line
       if ( @@temp != [] ) and
          ( @@old_tld != "" ) and
          ( @@old_source != "" )
       then
          subsorted_by_time = []
```

```
            subsorted_by_time = @@temp.sort_by { |line|
               line.split[0]
            }
            subsorted_by_time.each { |line|
               @@sorted_by_time << line
            }
            @@temp = []
         end
      end
   }

   @@nonbogus = @@nonbogus_temp.sort_by { |line|
      line.split[0]
   }

   @@file_bogus = File.new("#{@@input}.bogusfolder/
                           #{input}.notcached", "w")
   @@file_nonbogus = File.new("#{@@input}.nonbogusfolder/
                              #{input}.no-bogus", "w")

   @@sorted_by_time.each { |line|
      @@file_bogus.write(line)
   }

   @@nonbogus.each { |line|
      @@file_nonbogus.write(line)
   }

end


split

Dir.mkdir("#{@@input}.bogusfolder")
Dir.mkdir("#{@@input}.nonbogusfolder")

Dir.foreach(@@folder) { |file|
   if ( file != "." ) and ( file != ".." ) then
      repeat(file)
      system("rm #{@@folder}/#{file}")
   end
}

system("rm -r #{@@folder}")
system("touch #{@@input}.notcached")
system("touch #{@@input}.no-bogus")

Dir.foreach("#{@@input}.bogusfolder") { |file|
   if ( file != "." ) and ( file != ".." ) then
      system("cat #{@@input}.notcached
                  #{@@input}.bogusfolder/#{file}
                  > #{@@input}.notcached.temp")
      system("rm #{@@input}.notcached")
      system("mv #{@@input}.notcached.temp #{@@input}.notcached")
   end
}

system("rm -r #{@@input}.bogusfolder")

Dir.foreach("#{@@input}.nonbogusfolder") { |file|
   if ( file != "." ) and ( file != ".." ) then
      system("cat #{@@input}.no-bogus
                  #{@@input}.nonbogusfolder/#{file}
                  > #{@@input}.no-bogus.temp")
      system("rm #{@@input}.no-bogus")
      system("mv #{@@input}.no-bogus.temp #{@@input}.no-bogus")
   end
}

system("rm -r #{@@input}.nonbogusfolder")
```