Further Developing Preload Lists for the Tor Network

Vidareutveckling av preloadlistor för Tor-nätverket

Daniel Bahmiary

Faculty of Health, Science and Technology

Bachelor thesis in Computer Science

First Cycle, 15 hp (ECTS)

Supervisor: Tobias Pulls, University of Karlstad, Karlstad

Examiner: Per Hurtig, University of Karlstad, Karlstad

Karlstad, 10 January 2023

Abstract

A recently proposed defense for the anonymity network Tor uses *preload lists* of domains to determine what should be cached in the Domain Name System (DNS) caches of Tor relays. The defense protects against attacks that infer what is cached in Tor relays. By having domains continuously cached (preloaded), the cache will become independent of which websites have been visited. The current preload lists contain useless domains and have room for improvement. The objective of this project is to answer the question of ”How can we generate better preload lists?” and to provide improved methods for generating preload lists, with the ultimate goal of generating better preload lists that the Tor Project can benefit from.

We further developed existing tools to use web crawling to find more useful domains, as well as implementing filtering to remove useless domains from the preload lists. The results of this project showed promising results, as the useless domains decreased by an average of around 57% and more useful domains were found.

Keywords

Tor, DNS, Preload list, Crawling

Sammanfattning

Ett nyligen föreslaget försvar för anonymitetsnätverket Tor använder *preload listor* av domäner för att avgöra vad som ska cachelagras i domännamnssystemets (DNS) cacher för Tor reläer. Försvaret skyddar mot attacker som avgör vad som cachelagras i Tor relärer. Genom att ha domäner kontinuerligt cachade (förladdade), blir cachen oberoende av vilka websidor som har besökts. De nuvarande preload listorna innehåller värdelösa domäner och har utrymme för förbättring. Syftet med detta projekt är att svara på frågan ”Hur kan vi generera bättre preload listor?” och att bidra med förbättrade metoder för att generera preload listor, med det ultimata målet att generera bättre preload listor som Torprojektet kan dra nytta av.

Vi vidareutvecklade befintliga verktyg till att använda webbkrypning för att hitta mer användbara domäner, samt implementerade filtrering av värdelösa domäner från preload listorna. Detta projekt visade lovande resultat, då de värdelösa domänerna minskade med i genomsnitt ungefär 57% och mer användbara domäner hittades.

Nyckelord

Tor, DNS, Preload lista, Webbkrypning

Acknowledgements

I would like to thank my supervisor and project provider Tobias Pulls for his guidance throughout this project, as well as his input and help with structuring this thesis.

Acronyms

DNS	Domain Name System
IP	Internet Protocol
TLD	top-level domain
SSH	Secure Shell
VPN	Virtual Private Network
HTTPS	Hypertext Transfer Protocol Secure
URL	Uniform Resource Locator
XML	Extensible Markup Language
HTML	HyperText Markup Language
TTL	Time to live
GUI	Graphical user interface
TMPFS	Temporary File System

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Research is being conducted [8] on how the Domain Name System (DNS) should be handled by exits in the Tor network [9]. The focal point is to only cache certain domain names that are used on popular websites. This is done in order to solve the privacy issues with the current DNS cache design and to prevent possible attacks. In this thesis, we will further develop and analyze existing tools that collect popular domains which will be used to create preload lists. These preload lists will serve as the cache that the DNS will use.

1.1 Background

The DNS is a decentralized system that maps domain names to Internet Protocol (IP) addresses. IP addresses are numerical addresses and are used for identifying locations on the Internet. When requesting a webpage, e.g., `www.example.com`, a DNS query is sent to the DNS which will return the IP address of the domain `www.example.com` [3].

The Tor network is a distributed overlay network designed for anonymous communication [9]. Users of Tor have their integrity protected and are also protected against attacks such as eavesdropping [15]. In Tor, DNS requests are run on the exit relay, which is the last relay that traffic passes through before reaching its final destination [14].

Dahlberg and Pulls have motivated a preload defense [8] to prevent attackers from identifying website visits by users using Tor, since the current DNS cache design is vulnerable to timing and timeless attacks. This preload defense presents a redesign of the DNS cache in Tor, where two types of caches are used on the exit relays, i.e., shared preload cache and same-circuit cache. The shared preload cache contains domains from an allowlist, which is generated by visiting the most popular sites on the Internet. While this preload defense protects against attacks, it also improves performance: how much improved depends on the allowlist of domains. This thesis further builds on the

preload defense and explores different techniques to improve the generation of preload lists.

1.2 Problem Description

Both collection of domains and generation of preload lists can be done in a number of different ways. This thesis aims to answer the question of *"How can we generate better preload lists?"*. The definition of "better" is something that will be explored in this thesis.

1.3 Thesis Objective

The objective of this thesis is to describe how better preload lists can be created. In order to do this, the preload tools used for collecting domains and generating preload lists will be analyzed and evaluated.

1.4 Project Goals

Dahlberg and Pulls, researchers at Karlstad University, are researching [8] on how preload lists can solve the privacy issues with the current DNS design in the Tor network. Their research will protect against attacks and be contributed to the Tor Project [26]. The Tor Project is an organization that maintains the Tor network. The goal of this project is to create better preload lists that the Tor Project can use to improve the privacy and security of the Tor network. The deliverables of this project are improved preload tools that generate better preload lists. The results of this project are preload lists that are more effective and efficient, e.g, improved cache-hit ratio and fewer useless domains.

1.5 Ethics and Sustainability

No ethical issues arise during this project. The only data collected are domain names that are extracted from visiting popular websites. No personal data are therefore collected that can be a threat to privacy. There were a lot of ethical issues, caused by attacks, that were solved by Dahlberg and Pulls [8]. This thesis is a part of their solution since it further improves on their work. As for sustainability, the improvements in this thesis will bring better performance to the Tor network and improve its efficiency.

1.6 Methodology

This project was done in iterations. The first step in the iteration was to find and implement an improvement to either the preload lists directly or to the tools that collect

domain names for the preload lists, e.g., filter useless domains or find new ways of collecting more useful domains. The next step was to run the tools that visit popular websites and collect domains. After collecting domains, preload lists can be created from them. Lastly, the preload lists were evaluated to identify any improvements, e.g., number of hits or useless domains. When the evaluation was completed, the process was repeated from the beginning.

1.7 Stakeholders

Dahlberg and Pulls research on how preload lists can be used in the Tor network. Our work further builds on their research and implements new ways of creating preload lists. By improving the preload lists, the DNS cache will become more effective and efficient and will ultimately give Tor improved performance and security. The security is improved because without an effective and efficient preload-based defense, the preload-defense (or something similar) will not be deployed, and thus Tor will not be fully protected from the attacks presented by Dahlberg and Pulls [8]. This work is something that hopefully the Tor Project can use and benefit from. Tor in turn is used by millions of people daily to browse the web anonymously, circumvent censorship, and visit onion sites.

1.8 Delimitations

The focus of this thesis is to generate better preload lists for the preload cache. We will not go into how the DNS on the exits or Tor should handle the preload lists. Other defenses and lists may exists that could give different results, however this thesis will only focus on the preload lists that Pulls and Dahlberg have worked on.

1.9 Outline

The structure in this thesis is as follows. In Chapter 2, we will give a detailed description of DNS, Tor, DNS in Tor, Uniform Resource Locator (URL) and related work. Chapter 3 describes the methodologies of data collection used in this project. Chapter 4 gives a detailed description of the implementation of the methods of data collection. Chapter 5 evaluates the results. Chapter 6 describes the conclusions given from Chapter 5 and the positive effects and drawbacks from the results. Future work is also discussed.

Chapter 2

Background and Related Work

2.1 DNS

The Domain Name System (DNS) [11] is a decentralized system used for mapping domain names to Internet Protocol (IP) [10] addresses. IP addresses are numerical addresses used for communication and represent a location on the Internet for a device or a network interface. Depending on the version of the IP, addresses have the format *x.x.x.x* (IP Version 4) or *y:y:y:y:y:y:y:y* (IP Version 6). Web requests are an example where IP addresses are used. When performing a request for a webpage, the IP address of the desired webpage is used to send the request to the right destination. However, since IP addresses are long numerical addresses, it makes them difficult to read and memorize for humans. This issue is solved by the DNS, mapping domain names to IP addresses. A domain name is a unique text string that is much easier for humans to read and use compared to an IP address when requesting access to websites on the Internet, e.g, `www.example.com` rather than `93.184.216.349` [3].

The DNS architecture is also hierarchical and consists of root servers, top-level domain (TLD) servers and authoritative servers [4]. Apart from these servers, there are also recursive resolvers. The first step when requesting for a webpage is for the web client to send a DNS query to the recursive resolver. If the recursive resolver has the domain in the DNS query in its cache, a response will be made, otherwise it will send a request to a root name server. The root server will respond with the corresponding TLD server of the extension, e.g., `.com`, `.org`, `.edu` etc. After receiving response from a root server, the recursive resolver sends a request to a TLD server. The TLD server responds with the authoritative name server for the requested domain. Lastly, the resolver sends a request to a authoritative name server that responds with the IP address of the domain. When the recursive resolver receives an IP address, it will send the IP address with a respond to the web client [5].

Looking at a DNS query for `example.com`, as shown in Figure 2.1.1, the root server responds with the TLD server of `.com`, the TLD server responds with the authoritative server for `example.com` and the authoritative server responds with the IP address.

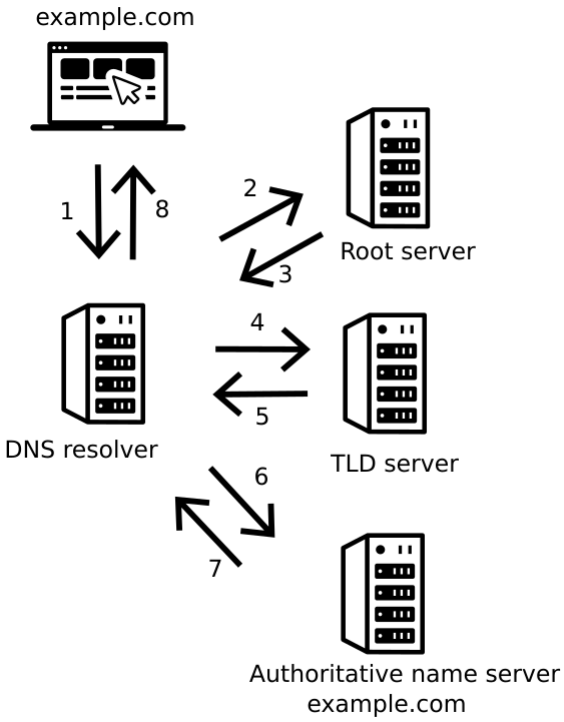


Figure 2.1.1: Example of how a DNS query is resolved.

2.2 Tor

Tor is a distributed overlay network that enables anonymous communication. The Tor network implements onion routing, which is a technique used to protect the integrity of data and users while also protecting against eavesdropping and traffic analysis [9]. When communicating, a circuit is built containing three relays that passes traffic. These relays, often called guard, middle and exit relay, only have knowledge of the relay before and after it. One benefit of this is that it makes tracking of data difficult [9, 14].

Figure 2.2.1 gives a visualisation of how circuits are created in the Tor network. Web traffic on the Tor Browser is passed through a guard, middle and exit relay before reaching the final destination.

To avoid DNS leaks, DNS requests must be sent over Tor. DNS requests in Tor are run on exit relays. The exit relay is the last relay that traffic passes through before reaching its final destination. To improve performance, the exit relays cache DNS responses [14].

2.3 URL

A Uniform Resource Locator (URL) is an unique address that identifies a resource on the Internet, e.g., the address of a website. A URL consists of a number of

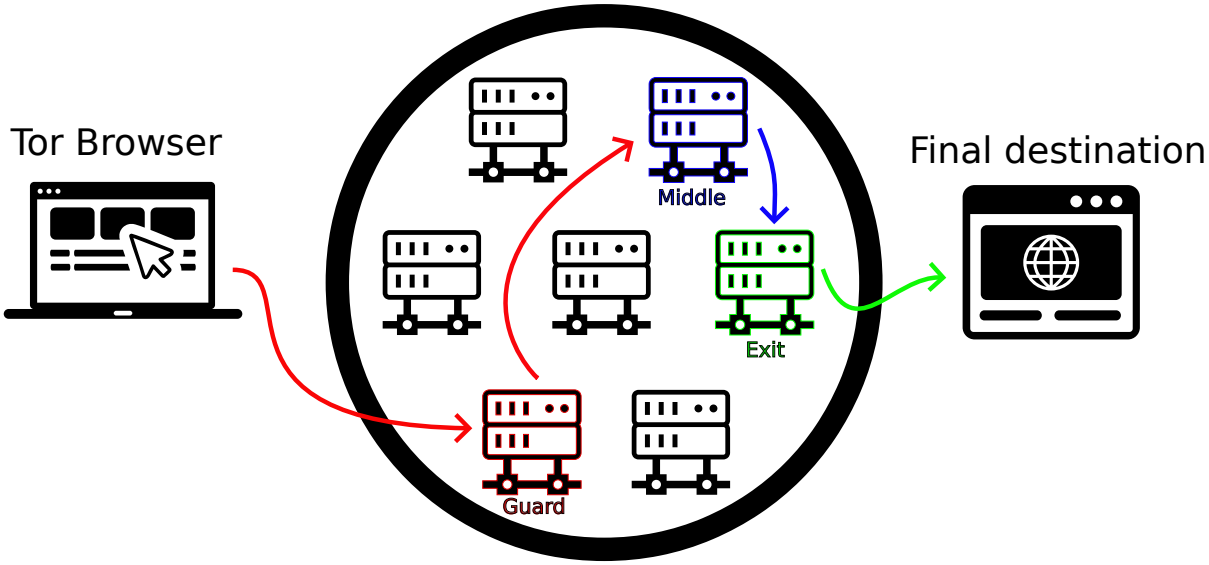


Figure 2.2.1: Visualisation of how circuits are created in Tor.

components [13], as shown in Figure 2.3.1. The most relevant components in this project are:

- Scheme: Which protocol is used, e.g., `http` or `https`.
- Subdomain: A subdomain is an optional part of a domain, e.g., `www` is a part of `example.com` in Figure 2.3.1.
- Domain name: A domain name identifies a organisation or entity within the Internet.
- Path: A path identifies the location of a resource, e.g., a web page or a file.
- Parameters: Data/elements that are being queried.
- Fragment: Specifies a location within, for example, a web page.

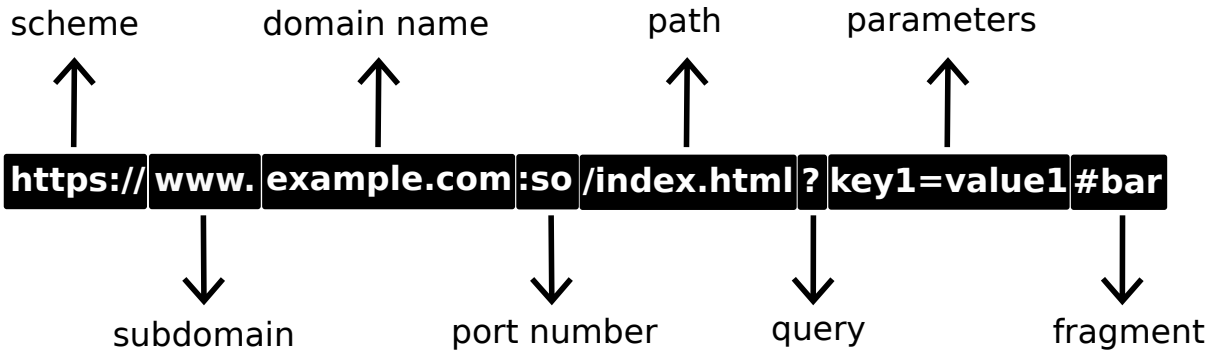


Figure 2.3.1: Components of a URL.

2.4 Related Work

Pulls and Dahlberg showed [22] how to use attacks, like the one that motivated the preload defense by Dahlberg and Pulls [8], to reliably identify website visits by users despite them using Tor. The preload defense introduces a redesign of Tor's DNS cache. The goal of this defense is to achieve false positives when attackers try to use the DNS as a tool to determine if a user has visited a certain website (timeless attack).

Before explaining the preload defense in more detail, we have to briefly explain how a DNS cache works. In Section 2.1, we discussed how a web client sent DNS queries to the resolver in order to get the IP address of a webpage. This process takes time and has to be repeated for every webpage the web client wants to visit. This is were the DNS cache is introduced. The DNS cache is a temporary storage of DNS lookups from previously visited domain names. The time these lookups are stored depends on their Time to live (TTL). When a client wants to visit a webpage, the DNS cache is checked to see if it it contains the DNS lookup for the requested webpage. If the DNS lookup exists in the DNS cache, the IP address of the domain name is returned to the client immediately. However, if the DNS lookup does not exist in the DNS cache, DNS queries have to be made to DNS servers in order to resolve the domain. In short, the DNS cache reduces the frequency of complete DNS lookups and brings improved performance since the DNS queries can be resolved much quicker [6].

The preloaded DNS cache by Pulls and Dahlberg consists of two types of caches:

- Shared preload cache. The shared preload cache consists of domains from an allowlist (a preload list) and is shared across circuits. The allowlist contains the most popular domains and are collected by visiting the most popular sites on the Internet. The exit relays receive the allowlist and resolves the domains to IPs. DNS requests on a circuit will firstly go the preload cache to see if the domain exists in the cache. The reason why this cache can safely be shared is that the DNS no longer takes previous lookups in regard, if the domain exists in the preload cache there will be a cache hit.
- Same-circuit cache. If the domain does not exists in the preload cache, the DNS request will go to the circuit's same-circuit cache. Since circuits are isolated, attackers can not gain information from caches on other circuits than their own.

Figures 2.4.1 and 2.4.2 give a visual representation of the preloaded DNS cache design. In Figure 2.4.1, a central party visits sites on a popularity list and collects domains. After the collection, an allowlist (preload list) of domains is generated. The preload list is then used in the shared preload cache, as shown in Figure 2.4.2, for every Tor relay. When a DNS lookup is made, the lookup will first go to the shared preload cache. If the DNS lookup can not be resolved in the shared preload cache, the DNS lookup goes to a dynamic same-circuit cache which is not shared across circuits. If the DNS lookup still can not be resolved, it will be sent to a DNS resolver (like in Section 2.1).

This defense protects against timing and timeless attacks. These attacks are used

against the original DNS cache to identify if a user has visited a specific website. Since the DNS cache is dynamic and stores lookups based on their TTL, timing attacks can be used to identify if a lookup is stored in the DNS cache, as cached and non-cached lookups take different amount of time to resolve. The preload defense protects against these kind of attacks with the shared preload cache and same-circuit cache. The shared preload cache is continuously preloaded, which means that the domains have no TTL. This means that timing and timeless attacks can not gain any information from the shared preload cache. Timing and timeless attacks are also useless against the same-circuit cache, since attackers only can use these attacks in their own circuits.

Beside promising protection against timing and timeless attacks, this preload defense also brings improved performance. The performance is improved since the most popular domains on the Internet are continuously preloaded, resulting in quicker resolve time for DNS lookups and increased cache-hits. To measure the performance of the preload lists, the ideal would be to operate a modified Tor exit on the Tor network and measure the performance of the DNS cache. Dahlhberg and Pulls did this with significant efforts to do it safely (ethically and legally), such as contacting the Tor Research Safety Board [25] for guidance [8].

The preload defense highlights that promising future work is to improve how preload lists should be generated in order for better performance. Improvements such as filtering useless domains and crawling websites to collect more useful domains are ideas that could be implemented.

This thesis further builds on the preload defense and explores different techniques to improve the generation of preload lists.

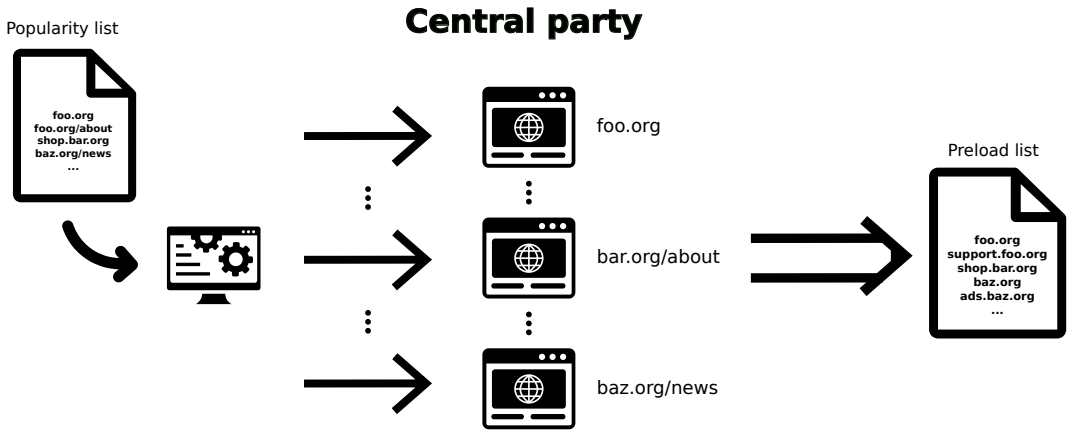


Figure 2.4.1: A central party collecting domains from popular websites and generating a preload list.

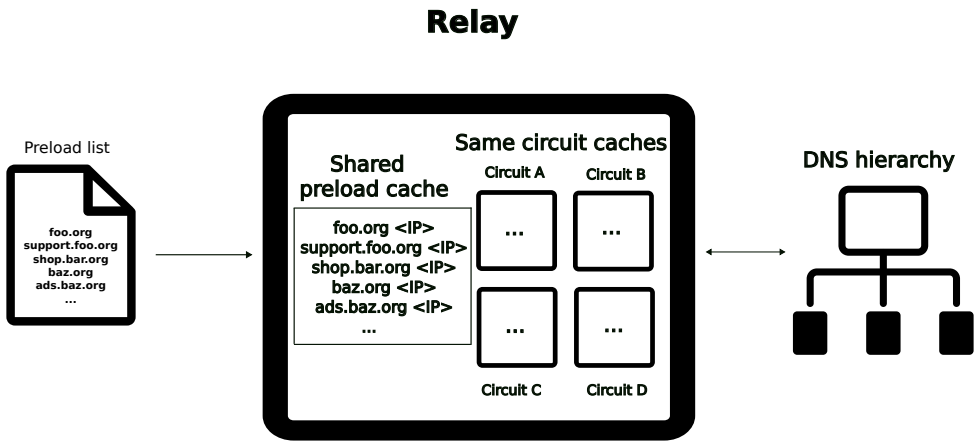


Figure 2.4.2: How Tor relays handle DNS lookups.

Chapter 3

Methodology

3.1 The Preload Design

The preload defense, proposed by Dahlberg and Pulls, uses preload lists to serve as a shared cache and contains domains from popular websites. The generation of preload lists begins with downloading a popularity list, e.g., Alexa- [2] or Tranco-list [21], from the web. These popularity lists contain the most popular websites on the Internet. The next step is to visit the popular websites and collect all the domains that are loaded. Lastly, the domains collected are used for generating preload lists.

3.2 Method

The overall structure of this project was to work in iterations. By using this method, the preload lists will gradually improve and regular evaluations can be made. The process of creating better preload lists can be divided into five smaller parts, as seen in Figure 3.2.1. The first part is to analyze the current preload lists to find a way to improve the lists. What is defined as an improvement will be discussed later in this chapter. After figuring out a way to improve the preload lists, the next step is to implement the idea by either modifying the current tools or create new tools. Now that the tools have been improved, the third step is visit the most popular websites and collect domains. After collecting the domains, the next step is to generate new preload lists from them. Lastly, the new preload lists are evaluated to see how improved tools have affected the results. After completing the evaluation the process is repeated from the beginning.

3.3 Defining Better Preload Lists

In order to create better preload lists, we must define what we mean by better. Since these preload lists will be used in the DNS cache, any improvements to the preload lists must take the DNS cache into consideration, i.e., how the DNS cache benefit from the

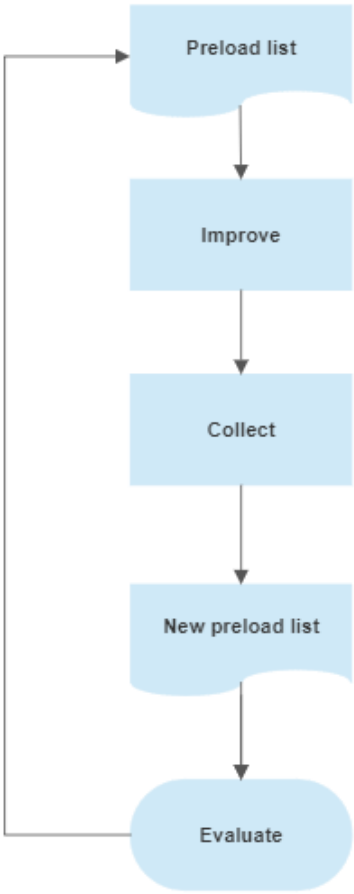


Figure 3.2.1: Visualisation of methodology used in project.

improvement. In this project, we look for two different aspects of improvements to the preload lists:

- **Filter useless domains** – The current preload lists contain useless domains. Useless domains are domains that do not get any hits in the cache. There are currently a lot of domains that are used for advertisement. Most of these domains are unique, since they are customized for a specific user. The preload lists have no use for unique domains, since the lists are shared across circuits in Tor, i.e., the unique domains most likely do not appear for other users. To improve the preload lists, we have to find a way to identify and filter useless domains.
- **More useful domains** – More useful domains would boost the performance of the shared preload cache. Useful domains are domains that get hits in the cache. Here we must explore different techniques to find more useful domains.

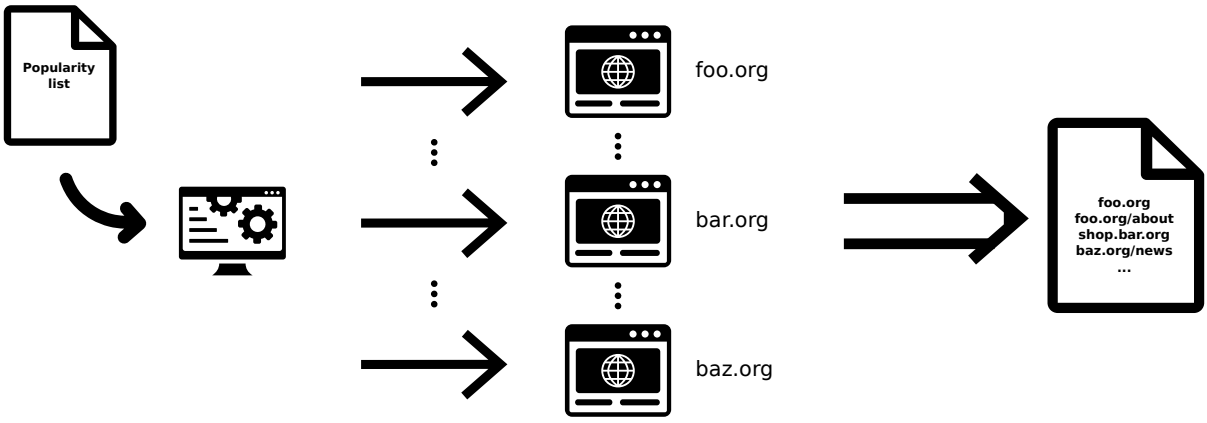
To measure the performance and any improvement to the preload lists, ideally we should operate a modified exit relay in the Tor network and measure the performance of the DNS cache, like Dahlberg and Pulls [8]. However, this is not feasible within the limited time of this thesis topic (ethical approval and contact with the Tor Research

Safety Board, etc). Instead, to measure the performance of our improved preload lists in this project, we will:

1. Visit sites, collect domains and generate preload lists.
2. Visit the same sites again and record lookups.
3. Compare the recorded lookups against the generated preload lists.

3.4 Phases

Our work on improving the preload lists can be divided into four phases. The first phase, seen in Figure 3.4.1, is the crawling phase. The preload tool takes the popularity list as input and visits each domain in order to find subdomains and subpages. The collected subdomains and subpages are then added to the popularity list, creating an extended popularity list. By adding subdomains and subpages, we can hopefully find more useful domains during the next phase.



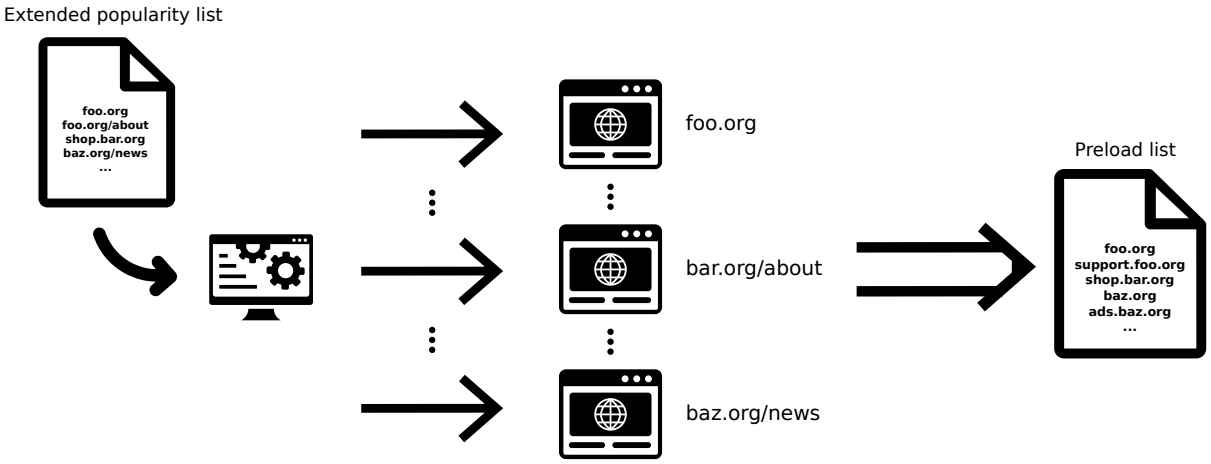
Phase 1: Crawl websites from popularity list and collect subdomains and subpages.

Figure 3.4.1: Representation of the crawling phase.

The second phase is the collection phase, seen in Figure 3.4.2. Taking the extended popularity list as input, the preload tool visits each domain, subdomain and subpage and collects all domains that are loaded. The output from the preload tool is a preload list, containing all the domains collected.

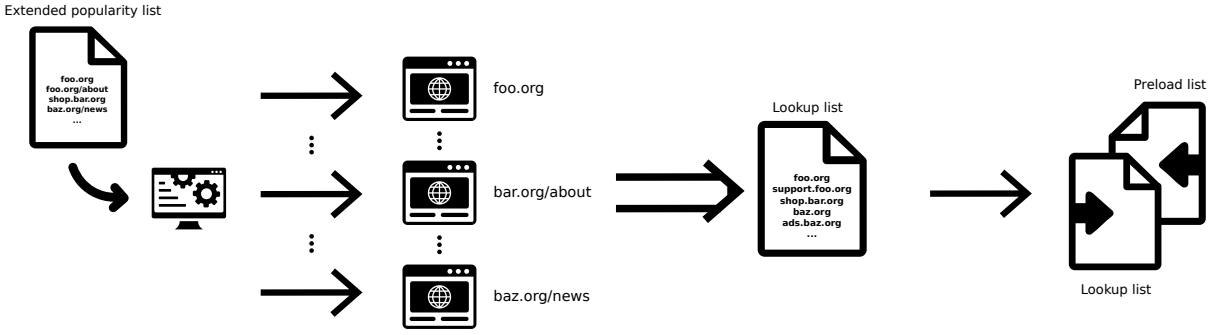
The third phase is the domain identification phase, seen in Figure 3.4.3. The purpose of this phase is to identify useless domains. To identify useless domains, an extra collection run is made on the same extended popularity list as the one in the second phase, recording domain lookups. These lookups are then compared with the collected domains from the second phase, with the purpose of identifying which domains are useless.

The fourth phase is the filter phase, see Figure 3.4.4. Now that the useless domains have been identified, we take the preload list and remove useless domains.



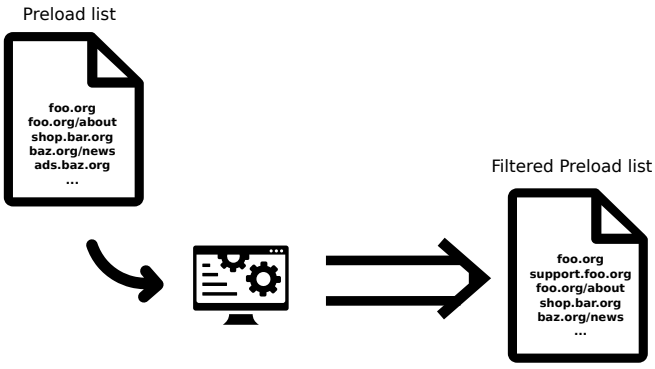
Phase 2: Visit websites from extended popularity list and collect domains found.

Figure 3.4.2: Representation of the collection phase.



Phase 3: Visit websites from extended popularity list and recording lookups. Then compare the lookups with the collected domains from phase 2.

Figure 3.4.3: Representation of domain identification phase.



Phase 4: Find and remove useless domains.

Figure 3.4.4: Representation of the filter phase.

Chapter 4

Implementation

4.1 Setup

Access to the preload tools was given by Tobias Pulls on a GitLab repository, containing Python files and a shell script. In addition, access to previously collected data was given. These are all research artifacts shared by Dahlberg and Pulls as part of their paper on the preload defense [8].

4.1.1 Remote Server

The collection of domains takes time and resources. Thus, access was given to a remote server at Karlstad University in order to run larger collection runs. OpenVPN [19] is used to access the remote network and then Secure Shell (SSH) is used to access the server.

4.1.2 Virtual Private Network

During domain collection we visit thousand of websites and make a ton of requests. This can lead to being labeled as a bot and getting our IP address blacklisted on the Internet. To prevent getting our real IP address blacklisted, we will use a Virtual Private Network (VPN) that changes our IP address. The VPN that is used in this project is Mullvad VPN [17].

An issue that occurred is that when connecting to Mullvad VPN on the remote server, we got kicked out of the SSH session and could not SSH back again. This was caused by the VPN modifying the routes on the server. To solve this, the split tunneling [18] feature offered by Mullvad VPN was used. To enable split tunneling, rules had to be added to a firewall, allowing the SSH connection being routed outside of the VPN tunnel. The rules, shown in Listing 4.1, allowed a process to listen on port 22 (SSH) while Mullvad VPN was active. An additional rule, `excludeOutgoing`, was also added, which allowed traffic to the default gateway 10.91.0.1. The following lines describe the code in Listing 4.1:

- Line 1 declares a new table called `excludeTraffic`.
- Lines 2, 7 and 11 create the input chains `allowIncoming`, `allowOutgoing` and `excludeOutgoing` respectively.
- Line 3 sets chain type to filter (for filtering packets) and links the `allowIncoming` chain to the input hook. The priority is set to -100 and the default policy is set to accept.
- Line 4 allows incoming SSH traffic on port 22.
- Line 8 sets chain type to route (for rerouting packets) and links the `allowOutgoing` chain to the output. The priority is set to -100 and the default policy is set to accept.
- Line 9 allows outgoing SSH traffic on port 22.
- Line 12 sets chain type to filter and links the `excludeOutgoing` chain to the output hook. The priority is set to -10 and the default policy is set to accept.
- Line 13 allows SSH and Hypertext Transfer Protocol Secure (HTTPS) traffic to the default gateway 10.91.0.1.

The rules were saved to file as `excludeTraffic.rules` and then with the command `sudo nft -f excludeTraffic.rules` a new firewall table was set up.

```
1  table inet excludeTraffic {
2    chain allowIncoming
3      type filter hook input priority -100; policy accept;
4      tcp dport 22 ct mark set 0x00000f41 meta mark set 0x6d6f6c65;
5    }
6
7    chain allowOutgoing {
8      type route hook output priority -100; policy accept;
9      tcp sport 22 ct mark set 0x00000f41 meta mark set 0x6d6f6c65;
10   }
11   chain excludeOutgoing {
12     type filter hook output priority -10; policy accept;
13     ip daddr 10.91.0.1 tcp dport {22, 443} ct mark set 0x00000f41;
14   }
15 }
```

Listing 4.1: Input chains of excludeTraffic.rules

4.2 Preload List Generation

In order to discuss how the preload lists can be improved, the generation of preload lists has to be explained in detail. Preload lists were also generated at the start of the project to make sure everything worked and to understand the tools better.

4.2.1 Generate Visit List

The process begins with downloading a fresh popularity list from the web. Since Dahlberg and Pulls made it clear [8] that only Tranco lists are of interest, this project will only use Tranco lists. Tranco lists contain the most popular websites on the Internet, ranked in order of popularity. The reason Tranco is chosen instead of Alexa, is that Tranco is a more recent source of website popularity [27]. The next step is to generate a list of unique domains for sites that will be visited. The list is generated by the Python program *preload-unique-sites.py*. *Preload-unique-sites.py* takes, as shown in Listing 4.2, two arguments as input. The arguments are:

- A popularity list to parse, i.e., a Tranco list.
- Number of sites to parse. This argument is optional and the default value is set to 10,000.

The downloaded Tranco zip file is then opened and parsed, adding the sites to a list. Lastly, the list of sites is shuffled (to not make parallel visits when visiting websites later on) and written to file as *unique-domains-10k*.

```
ap = argparse.ArgumentParser()
ap.add_argument("-l", nargs="+", required=True,
    help="popularity lists to parse")

ap.add_argument("-n", required=False, type=int, default=10000,
    help="number of sites")

args = vars(ap.parse_args())

def main():
    unique = {} # dict for faster lookup
    for l in args["l"]:
        print(l)
        with zipfile.ZipFile(l) as z:
            with z.open('top-1m.csv') as c:
                for line in c:
                    p = line.decode('utf-8').strip().split(",")
                    if int(p[0]) <= args["n"] and p[1] not in unique:
                        unique[p[1]] = True

    domains = list(unique.keys())
    random.shuffle(domains)
    fname = f"unique-domains-10k"

    with open(fname, "w") as f:
        for d in domains:
            f.write(f"{d}\n")
```

Listing 4.2: How the file unique-domains-10k is generated by preload-unique-sites.py

4.2.2 Domain Collection

After generating a visit list, the next step is to visit all the sites in the list and collect domains that are loaded. The collection was done by the program *preload-collect-domains.py*. *Preload-collect-domains.py* takes the list *unique-domains-10k* as input and initiates scripts for domain collection. For each website visit, a headless Chromium is ”fired up” and from each visit loaded domains are collected. Headless Chromium is a web browser without a Graphical user interface (GUI). Headless browsers are used for machines that do not have a GUI and for the performance they bring. Compared with a regular browser, a headless browser is much faster. When finished with all the visits, the collected domains are written to file as *results.csv*.

To find even more domains, Mullvad VPN is used in order to make visits from Europe (EU), United States (US) and Hong Kong (HK). Collection from EU is repeated three times, while US and HK have two repetitions each. The reason for repetitions is that the collection runs do not always succeed and by making repetitions, the measurements become more reliable. To make collections from different regions, the shell script *collect-domains.sh* is used. As shown in Listing 4.3, the shell script begins with connecting to a server in Stockholm, Sweden (EU) with the command `mullvad relay set location se sto`. Then the collection tool *preload-collect-domains.py* is run with the command `time ./preload-collect-domains.py -l $LIST -n 3 -b 60 -t 20`. The arguments needed for *preload-collect-domains.py* are the following:

- `-l $LIST`: List of unique sites to visit. In our case this is the list unique-domains-10k.
- `-n`: Number of repetitions.
- `-b`: Batch size. Optional, default is set to 10.
- `-t`: Timeout for each attempted visit. Optional, default is set to 10.

Since we want three repetitions of domain collections, `-n` is set to three. The command `time` is used for determine how long the collection tool takes to run. The collected domains stored in *results.csv* is renamed *results-eu-3.csv* and copied to the ∼/Downloads/ folder. These commands are then repeated for domain collection from US and HK. For US, we connect to New York, United States, set `-n` to two repetitions and store the collected domains as *results-us-2.csv*. Lastly, for HK, we connect to Hong Kong, set `-n` to two repetitions and store the collected domains as *results-hk-2.csv*.

```
# unique list of sites
LIST=unique-domains-10k

# 3x from Europe
mullvad relay set location se sto
echo "set location to Stockholm"
sleep 5
time ./preload-collect-domains.py -l $LIST -n 3 -b 60 -t 20
sleep 10
```

```
mv results.csv results-eu-3.csv
cp results-eu-3.csv ~/Downloads/

# 2x from US
sleep 5
mullvad relay set location us nyc
echo "set location to New York"
sleep 10
time ./preload-collect-domains.py -l $LIST -n 2 -b 60 -t 20
mv results.csv results-us-2.csv
cp results-us-2.csv ~/Downloads/

# 2x from HK
sleep 5
mullvad relay set location hk
echo "set location to Hong Kong"
sleep 10
time ./preload-collect-domains.py -l $LIST -n 2 -b 60 -t 20
mv results.csv results-hk-2.csv
cp results-hk-2.csv ~/Downloads/

mullvad relay set location se got
echo "set location to Göteborg"
```

Listing 4.3: How the shell script collect-domains.sh runs domain collection.

4.2.3 Collection Details

Collection of domains take a lot of time, since there a great number of Chromium browser logs (parsed to extract domains) that are written to disk. To speed up the process, we run the collection tool on a ramdisk on the server. A ramdisk is created with the command `sudo mount -t tmpfs -o rw,size=2G tmpfs /mnt/ramdisk`. This command creates a ramdisk using a Temporary File System (TMPFS) on the mounting point `/mnt/ramdisk`. Using the ramdisk for collecting domains, it takes around nine hours to complete a collection run.

4.2.4 Preload List Generation

When the collection of domains is done, the last step is to generate preload lists out of them. The preload lists are generated by the tool *preload-gen-list.py*. *Preload-gen-list.py* takes the following arguments:

- -t: The tranco zip file earlier downloaded. There were also the arguments -a and -u that took an Alexa and Umbrella zip file. However, since only Tranco lists are of interest, every piece of code that handled the popularity lists Alexa and Umbrella were removed from this project.
- -l: Unique sites list. This is the *unique-domains-10k* list.

- -c: Collected domains. These are the collected domains stored in *results-eu-3.csv*, *results-us-2.csv* and *results-hk-2.csv*.
- -s: Folder which the generated lists will be stored to.
- -n: Maximum site/domain popularity to consider. This argument is optional and set to 10,000 as default.

With these arguments, *Preload-gen-list.py* generates preload lists of two different types. The first type is preload lists containing domains from only the Tranco popularity list. The second type is extended preload lists containing the domains collected when visiting Tranco sites, e.g., a top 10 extended preload list contains all the domains collected when visiting top 10 most popular sites on the Tranco popularity lists. For both list types, top 10,200,..1000,...10,000 lists are created.

4.3 Evaluation Tool

To evaluate a preload list, a tool for evaluation had to be implemented. The metrics used for evaluating a preload list are as follows:

- Number of domains in the list.
- Time to create the list.
- Number of hits.
- Number of useless entries (no hits).

To count the number of hits and useless entries, we reuse the collection tools. After having generated new preload lists, the top X sites are visited in the same way again and lookups (collected domains) are recorded. The lookups are then run against the generated preload lists.

The evaluation tool *preload-eval.py* was implemented that took the following arguments:

- `-l`: Lookup list. This is the collected domains stored in *results-eu-3.csv*, *results-us-2.csv* and *results-hk-2.csv*.
- `-e`: Extended preload list. This is the extended Tranco list that is to be evaluated.

Preload-eval.py begins with storing all the collected domains in a list named *lookups*. Thereafter, all the domains in the extended preload list are stored in a dictionary named *preload*, as shown in Listing 4.4. The *key:value* pairs are set as *domain:hit* in order to count the number of hits a domain in the preload list has.

```
with open(args["e"], "r") as f:
    for line in f:
        preload[line.strip()] = 0
```

Listing 4.4: How preload-eval.py stores collected domains.

The next step is to iterate through each domain in the *lookups* list and check if the domain exists in the dictionary *preload*, as shown in 4.5. If a lookup exists in the dictionary, the amount of hits is increased by one as well as the value for the domain in the dictionary. However, if a lookup does not exist it is counted as a miss.

```
hit = 0
miss = 0
for domain in lookups:
    if domain in preload:
        hit += 1
        preload[domain] += 1
    else:
        miss += 1
```

Listing 4.5: Code for checking domain hits.

To count the number of useless domains, the dictionary *preload* is checked for domains with zero hits, as shown in Listing 4.6 .

```
for domain in preload:
    if preload[domain] == 0:
        useless_entries += 1
```

Listing 4.6: Code for counting useless domains.

Another good metric is the cache hit ratio. The cache hit ratio gives a measurement of how many requests a cache successfully fulfills, compared to the total amount of requests. In our case, requests are domains in the *lookup* list. The cache hit ratio is calculated as shown in Listing 4.7

```
hit_ratio = round(hit/(hit + miss)*100)
```

Listing 4.7: Code for calculating hit ratio.

Lastly, the calculated metrics are written to file as *eval*, as shown in Listing 4.8. The size of the preload list is calculated with the `len()` function. The only metric that is not calculated in *preload-eval.py* is the time to create lists. The time to create lists is instead calculated during collection in the shell script *collect-domains.sh* with the `time` command.

```
with open("eval", "w") as w:
    w.write(f"Preload list: size = {len(preload)}\nLookup list: size = {len
    (lookups)}\n")
    w.write(f"\nHits = {hit}\nHit ratio  {hit_ratio}%\nMiss (not in
    preload cache) = {miss}\n")
    w.write (f"\nUseless entries = {useless_entries}")
```

Listing 4.8: Code for writing calculated metrics to file.

4.4 Filtering

This section will discuss how useless domains were identified and how the filtering was implemented.

4.4.1 Identifying Useless Domains

A useless domain was as earlier mentioned a domain that was unique and did not get any hits in the cache. A lot of the useless domains in the preload lists were unique tracking domains used for advertisement. A good example of useless domains are *.safeframe.googlesyndication, which appears a lot in the preload lists.

To get a better understanding of how to identify useless domains, *preload-eval.py* is extended to write the domains and their corresponding hits to file as *sorted_hits*. As shown in Listing 4.9, the domains are sorted in order of hits (low to high) and the written to file.

```
20 with open("sorted_hits", "w") as w:
21     newlist = sorted(preload.items(), key=lambda item: item[1])
22     for item in newlist:
23         w.write(f"{item}\n")
```

Listing 4.9: How the domains in the dictionary are sorted and written to file.

Firstly, we explored how the most popular browser extensions, uBlock [20] and AdBlock [1], used filter lists to block ads. The findings were as follows:

- AdBlock - Within the AdBlock settings, the only readable list available to the user is the EasyList [12]. EasyList is a filter list with the purpose of removing content such as adverts and trackers. This list was however not of any use, since EasyList blocks almost all adverts. The reason this is bad for us, is that we do not want to filter all adverts, we only want to filter adverts that are useless (no hits). As an example, `ad.doubleclick.net` is an advertising service that EasyList blocks. However, when looking at the *sorted-hits* list, it was seen that `ad.doubleclick.net` gets a lot of hits. Domains that get hits in a cache are worth storing, since time and resources are saved by not having to do DNS requests.
- uBlock - For blocking adverts and trackers, uBlock uses EasyList and Peter Lowe's Ad and tracking server list [16]. EasyList was as earlier mentioned disregarded in this project. Peter Lowe's Ad and tracking server list did, similar to EasyList, block all adverts. This type of generalisation was not of any use when looking for a way to filter useless domains.

Moving on from the browser extensions, we ran the lookup lists against preload lists with the purpose of gaining information on useless domains. *Preload-eval.py* was extended to give a visualisation of the domains in both lookup and preload lists. The following additions were made:

- If a domain from the lookup list does not exists in the preload list, add the domain to the list *miss_list_lookups*. The list is then written to file as *miss_list*.
- If a domain in the preload list do not have any hits, add the domain to the list *useless_list_preload*. The list is then written to file as *useless_list_preload*.
- The domains in the preload list are sorted by order of hits (low to high) and

written to file as *sorted_hits*. An example of how the domains are sorted are shown in Figure 4.4.1

Figure 4.4.1: Example of how domains are sorted according to amount of hits in the file *sorted_hits*.

By comparing and analysing the previous mentioned files together, information about the domains the preload lists could be given. The file *miss_list_lookups* contained domains that could either be worth storing or useless for the preload lists. The file *useless_list_preload* gives as earlier mentioned all the domains with zero hits. A domain with zero hits could indicate that the domain is useless. However, it is not enough information to certainly call the domain useless. By combining both *useless_list_preload* and *sorted_hits*, we could tell if a certain type of domain was useless or not. Examples of how useless domains were determined are as follows:

- In *useless_list_preload* there were around 100 unique domains with the format `*.fls.doubleclick.net`. When looking at *sorted_hits*, around 600 domains of the same format were found that got hits. Since the format of the domains are the same, the domains with zero hits can not be separated. In addition, the amount of domains with zero hits was a lot less than those with hits. As a result, `*.fls.doubleclick.net` is a domain type that is worth storing.

- In *useless_list_preload* there were around 2000 unique domains with the format `*safeframe.googlesyndicaton.com`. In *sorted_hits*, there were no domains of the same type that had any hits. Since all `*.safeframe.googlesyndication.com` domains had zero hits, this type of domain was considered useless.

4.4.2 Filtering Implementation

After identifying useless domains, the next step was to implement domain filtering. To filter domains, a function called `filterDomain` was added to the preload list generating program *preload-gen-list.py*. The function, shown in Listing 4.10, took a domain from a filter list as a parameter and checked if the domain was to be filtered. The check was done by using regular expressions that checks for a match anywhere in a string. The parameters in the `re.match()` function was:

- `pattern`: A pattern that is checked against a domain.
- `domain`: The domain in which the pattern is looked for.

```
25 def filterDomain(domain):
26     for filter in filterList:
27         match = re.search(filter,domain)
28         found = bool(match)
29         if(found):
30             return True
31 
32     return False
```

Listing 4.10: Function that checks if a domain should be filtered.

A filter list, shown in Listing 4.11, was created which contained domain patterns that represented useless domains. These domain patterns were the pattern parameters used in the `filterDomain()` function. For every pattern in the filter list, the function checked if the pattern was found in a collected domain.

```
33 filterList = [
34     "safeframe.googlesyndication",
35     "[a-z0-9]+report.wc.yahoodns",
36     "v-[a-z0-9]+.wc.yahoodns",
37     "sombrero.yahoo",
38     ".am.dotnxdomain",
39     ".ap.dotnxdomain",
40     ".eu.dotnxdomain",
41     ".tbap.dotnxdomain",
42     ".tbam.dotnxdomain",
43     ".tbeu.dotnxdomain",
44     "[a-z0-9]+.tmptrk.sensic",
45     "[a-z0-9]+.trk.sensic",
46     "ts-[a-z0-9]+-clienttons-s.akamaihd.net",
47     "[a-z0-9]+-[a-z0-9]+-[a-z0-9]+-clientnsv4-s.akamaihd",
48     "init.cedexis-radar",
49     "nuid.imrworldwide",
50     "darnuid.imrworldwide",
51     "metric.gstatic",
52     "sync.upravel",
53     "l4.adsco",
54     "n4.adsco",
55     "s4.adsco"
```

```
]
```

Listing 4.11: A list containing domain patterns.

The function will return a Match object, which will be True if a match is found. If a match is found, the function will return True, otherwise it returns False.

An optional argument, `"-f"`, was also added to *preload-gen-list.py* for the purpose of filtering. This argument was a bool with default set as False. If this argument was used, the program was to enter a filter mode. The purpose of this was to choose if filtering was to be made. The filtering was used before creating preload lists out of the collected domains, as shown in Listing 4.12. For every domain in list, a check was done if we were in filter mode. If True, the function `filterDomain()` was called and if a match was not returned, the domain was written to file. If the program was not in filter mode, the domain would always be written to file.

```
for domain in list:
            if args["f"]== True:
                if not filterDomain(domain):
                    w.write(f"{domain}\n")
            else:
                w.write(f"{domain}\n")
```

Listing 4.12: Code for checking if the program should filter domains or not.

4.5 Crawling

The second improvement that could be done to the preload lists is to add more useful domains to the preload lists. At the moment, we only visit the frontpages of the sites to collect domains. One idea to find more domains is to crawl a website to find subpages and then visit the subpages to (hopefully) find more useful domains. This section will discuss how crawling was implemented and how more useful domains were added to the preload lists.

4.5.1 Scrapy

At first we looked into how Scrapy [24] was used to crawl websites. Scrapy is one of the most popular web scraping frameworks used to extract data from websites. It is a powerful tool that offers, in addition to data extraction, redirect handling, maintaining sessions, asynchronous request handling and more. When researching the advanced features that Scrapy provided, we believed that this framework might be a bit overkill for this project, since we only want to do simple data extractions. Our understanding was that Scrapy would be more suitable for bigger and more complex projects that had a need for a more advanced web-crawling framework.

4.5.2 Beautiful Soup

Beautiful Soup [7] is a Python parsing library for extracting data from HyperText Markup Language (HTML) and Extensible Markup Language (XML) documents. Beautiful Soup is not in it self a scraping tool, but offers simple and direct ways to extract specific data during web crawling. Complex HTML documents are converted into parse trees that are easy to navigate through. Because of Beautiful Soups simplicity and great documentation, Beautiful Soup was chosen for this project.

4.5.3 Implementation

A crawling tool called *web-crawl.py* is created with the purpose of crawling websites and collecting subpages. The only argument required for the program is a list of sites to visit. This is the *unique-domains-10k* list that was generated by *preload-unique-sites.py*. However, one change is that the list is in order of popularity. The reason this order of popularity is desired is to be able to choose top X sites to crawl and collect from. To create a ordered popularity list, the code in *preload-unique-sites.py* was modified as shown in Listing 4.13. `"s"` is an optional argument for the program that makes it possible to choose if the domains should be shuffled or not and the default value is set to false.

```
63 if args["s"]:
64         random.shuffle(domains)
65         fname = f"unique-domains-10k"
66 else:
67     fname = f"unique-domains-10k-ordered"
```

Listing 4.13: Implementation of choosing to shuffle or not.

Web-crawl.py begins with reading all the sites in the *unique-domains-10k-ordered* and appending them to the list *crawl_list*. For every domain in *crawl_list*, the function *crawl()* is called, as shown in Listing 4.14.

```
68 scheme = "https://"
69 for idx, domain in enumerate(crawl_list):
70     if idx < 2000:
71         url = scheme + domain + "/"
72         crawl(url,domain)
73     if idx >= 2000:
74         domain_to_link[domain] = [domain]
```

Listing 4.14: Code for iterating domains and choosing which to crawl.

The function *crawl()* takes the following two arguments:

- `url`: URL for the site. When doing requests, the URL for the site is needed. To create a URL, the protocol HTTPS and the domain are concatenated.
- `domain`: The domain that will be visited and crawled.

To control how many domains that are crawled, the index `idx` is used. In Listing 4.14, `Idx` is used to crawl the first 2000 sites. After 2000 sites, the rest of the sites will not be crawled and instead be added to the dictionary *domain_to_link*. The dictionary *domain_to_link* consists of the *key:value* pairs *domain:links*, where domain is the visited domain and links are the collected links for the domain.

4.5.4 Requests

The *crawl* function begins with extracting the domain name with the function call *urlparse(url).netloc*. This domain name will later be used for identifying internal and external links. The next step is to create a request with a fake user agent. A user agent is a software that represents a real user. In our case, the user agent tells servers which web browser is being used and makes the server believe we are a real user. A request is created with the function `set_request` as shown in Listing 4.15.

```
75 def set_request(url):
76     req = Request(
77         url,
78         data = None,
79         headers = {
80              "User-Agent": "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7)
     AppleWebKit/537.36 (KHTML, like Gecko) Chrome/107.0.0.0 Safari/537.36"
81         }
82     )
83     return req
```

Listing 4.15: Function that creates and returns a request.

The request contains the following objects:

- `url`: A string containing a valid URL.
- `data`: Additional data to send to server. Data is set to None since it is not needed.
- `headers`: A dictionary containing a fake user agent.

The next step after creating a request is to download the HTML content and create a BeautifulSoup object, as shown in Listing 4.16.

```
84 try:
85     req = set_request(url)
86     soup = BeautifulSoup(urlopen(req,timeout=10).read(), "html.parser")
```

Listing 4.16: Code for sending a request and creating a BeautifulSoup object.

The code in line 86 works as follows:

- The function call *urlopen(req, timeout=10).read()* sends the created request and reads the HTML document of the site. A timeout is set to 10 seconds to avoid a 408 Request Timeout response status code.
- *"html.parser"* is the parser that *BeautifulSoup()* will use.

- The function call *soup = BeautifulSoup(urlopen(req,timeout=10).read(), "html.parser")* downloads the HTML content of the site, parses it and creates a BeautifulSoup object called soup.

The reason the code in Listing 4.16 is in a try block is that the request is not guaranteed to succeed. Exceptions like for example 400 Bad Request, 308 Permanent Redirect and 403 Forbidden often appear. An except block is used to catch and log expressions, as shown in Listing 4.17. Inside the except block, the current thread is then suspended for two seconds in order to prevent errors that arrive when doing to many requests in a short interval. Then the request is retried, however this time with the subdomain `"www."`. The reason for using `"www."` is that it seemed to handle some of the exceptions like 308 Permanent Redirect.

```
87 except Exception as e:
88     logger.error(repr(e))
89     print(f"HTTPError: {url}\n")
90     try:
91         time.sleep(2)
92         www_url = "https://www." + domain_name
93         req = set_request(www_url)
94         soup = BeautifulSoup(urlopen(req,timeout=10).read(), "html.parser")
```

Listing 4.17: Code for catching exceptions and retrying request.

Another except block is then used to catch exceptions. Inside a new try block, the thread will sleep for four seconds to prevent servers learning sleep patterns. Then one last attempt to make a request is made. This time the request is done with a cloudscraper [23]. Cloudscraper is a Python module that is used to bypass sites that are protected by Cloudflare. Cloudflare is a network that improves the security and performance of websites. What cloudscraper does is that it bypasses Cloudflare's anti-bot page, a page that tries to detect bots. As shown in Listing 4.18, a cloudscraper instance is created and then with *BeautifulSoup()* a request is made. One could ask why this program does not use from the beginning. The reason behind this is that *urlopen()* retrieves more links from websites when compared with cloudscraper.

```
95 scraper = cloudscraper.create_scraper()
96 soup = BeautifulSoup(scraper.get(url,timeout=10).text, "html.parser")
```

Listing 4.18: Request done with cloudscraper.

Lastly, one last exception block is used, as shown in Listing 4.19. Errors are logged and the domain that could not be crawled is printed to console. Since the domain could not be crawled, the only collected link will be the original domain. The domain is then added to the lists:

- *internal_urls_preload*: A list containing internal links. An internal link is a hyperlink to a webpage that is on the same website/domain.
- *domain_to_link*: A dictionary with the *key:value* pairs *domain:links*, where domain is the visited domain and links are the links collected from the website.

```
except Exception as g :
    logger.error(f"{repr(g)}: {www_url}")
    print(f"********** Could not crawl {www_url} **********")
    internal_urls_preload.append(domain_name)
    domain_to_link[domain_name] = [domain]
    return
```

Listing 4.19: Code for catching exceptions and adding domain to list and dictionary.

The exception block ends with a return statement, since the domain could not be crawled there is no point to continue.

4.5.5 Link Collection

The next step after crawling a site is to collect links from the site. The BeautifulSoup object soup that was created in the Listing 4.17 represents the HTML document as a nested data structure and is now used for navigating the HTML document. A for loop is created, as shown in Listing 4.20, that finds all <a> tags in the HTML document.

```
for a_tag in soup.findAll("a"):
```

Listing 4.20: Iteration of all <a> tags in the HTML document.

The <a> tag is used in HTML to create a hyperlink to on the webpage. A hyperlink links a webpage to other webpages, email addresses, sections in the same page and many more. The <a> tag has an attribute href, which is the hyperlinks destination. To find more links, the program retrieves the href attributes of <a> tags, as shown in Listing 4.21.

```
href = a_tag.attrs.get("href")
```

Listing 4.21: Code for retrieving the href attribute of the <a> tag.

Some hrefs only contain the relative path and not the absolute path. Examples of relative and absolute paths are:

- Relative path: `index.html`, `/about/contact/`.
- Absolute path: `https://www.example.com/index.html`.

Do identify if the found link is internal or external, the absolute path is needed. The absolute path is created by calling the function `urljoin()`, as shown in Listing 4.22. The function `urljoin()` constructs an absolute path by combining the url with the href. The newly created href is then parsed into six different components. The components of interest are scheme, netloc and path. By concatenating the scheme, netloc and path, an absolute path is created without URL GET parameters and URL fragments.

```
href = urljoin(url,href)
parsed_href = urlparse(href)
href = parsed_href.scheme + "://" + parsed_href.netloc + parsed_href.path
```

Listing 4.22: Code for creating an absolute path.

After retrieving the href, the following checks are made:

- If href is empty and if true, skip the current iteration and move on with the next <a> tag.
- If href is in the set *internal_urls*, this to avoid duplicates. Internal links are added to this set and if the check says true, the link has already exists in the set. If the link already exists in the set, skip the current iteration.
- Is the link internal or external? This is done by checking the netloc (network location) part of the URL, as shown in Listing 4.23. The netloc includes the domain or subdomain. If the domain of the visited website exists in the netloc of the absolute path, it is an internal link. If not, the absolute path is an external link and the current iteration is skipped.
- Does the string ”mailto” exists in the href. If true, skip the current iteration since the <a> tag is a Mailto link used for redirecting to email addresses instead of web page URLs.

```
108 if domain_name not in parsed_href.netloc:
```

Listing 4.23: Code for checking if a link is internal or external.

If all checks are passed, the collected link is added to the set *internal_urls*. This set is as earlier mentioned used to keep a record of collected links. The link is then simplified to only contain netloc and path, as shown in Listing 4.24, since the scheme (”https”) is not of interest.

```
109 href_filtered = parsed_href.netloc + parsed_href.path
```

Listing 4.24: Code for simplifying the absolute path.

A check is then done to see if the link starts with `"www."`. If true, the string is sliced to return a string without `"www."`. Lastly, the link is added to the list *domain_list*, which contains all links found on the website.

After finding all links on a website, it is time to add the links to list. First, a check is made to see if the crawled domain is in the list of collected links, as shown in Listing 4.25. If the domain does not exist in the list, insert the domain at the beginning of list.

```
110 if not domain in domain_list:
111         domain_list.insert(0, domain)
```

Listing 4.25: Code for checking if domain exists in list and if True, inserting the domain to list.

The collected links for the website are then added to the dictionary *domain_to_link*, as shown in Listing 4.26. If no links were found during crawling, only add the domain of the site as found link in *domain_to_link*, as shown in Listing 4.27.

```
112 domain_to_link[domain_name] = domain_list
```

Listing 4.26: Code for adding collected links to dictionary.

```
113 if len(domain_to_link[domain_name]) == 0:
114     domain_to_link[domain_name] = [domain]
```

Listing 4.27: Code for only adding domain of the site to dictionary if no links were found.

The process of collecting links is then repeated for every site that is going to be crawled.

4.5.6 Storing Collected Links

The next step after collecting links is to store them. However, before saving to file, the links have to be filtered. During link collection a lot of duplicates were found and added to the dictionary *domain_to_link*. An example of duplicate links are example.com/images and example.com/images/. By iterating the key:value pairs in the *domain_to_link*, duplicates can be identified and removed, as shown in Listing 4.28. If a value belonging to a key is a duplicate, that specific value is removed.

```
115 for key, values in domain_to_link.items():
116     for value in values:
117         dup_domain = key + "/"
118         dup_val = value + "/"
119         for val in values:   # check duplicate value
120             if val == dup_val:
121                 domain_to_link[key].remove(val)
```

Listing 4.28: Code for finding and removing duplicate domains in the dictionary.

Lastly, the collected links are written to file as *crawl_list_full*, as shown in Listing 4.29. For every line in the file, a key (domain) and its values (links) are written.

```
122  with open("crawl_list_full", "w") as w:
123     for k, v in domain_to_link.items():
124         w.write(f"{k},")
125         for val in v:
126             w.write(f"{val.strip()},")
127 
128         w.write("\n")
```

Listing 4.29: Code for writing collected domains and links to file.

An additional file called *domain_data* is created, as shown in Listing 4.30. *Domain_data* contains information about how many links were found for each site. This file will later be used when generating preload lists.

```
129 with open("domain_data", "w") as w:
130     for k, v in domain_to_link.items():
```

```
131         w.write(f"{k}, {len(domain_to_link[k])}\n")
```

Listing 4.30: Code for writing amount of found links for each site to file.

4.5.7 Domain Collection Changes

During domain collection, sites that were visited came from the file *unique_domains_10k*. Since crawling was implemented, *preload-collect-domains.py* had to read sites from *crawl_list_full* instead. As earlier mentioned, *crawl_list_full* contained a site and collected links on every line. The code for reading *crawl_list_full* was modified, as shown in Listing 4.31. Since around 700,000 links were found during crawling, a decision was made to only use around 1% of the links when collecting domains. Thus, the following variables were declared:

- max_hit: Boolean that tells if 1% of the links have been added to list.
- max: Number of extra links to add to list.
- n_links: Number of links that have been added to list.

The code in Listing 4.31 read every line in *crawl_list_full* and did the following:

1. Add all domains and links to list *p*.
2. Continue if maximum numbers of links have not been added to the list *sites*, otherwise skip to line four.
3. For every domain in list *p*: If domain is not a duplicate and the number of added domains are less than the maximum allowed, add the domain to the list *sites*. Then increment the number of added links by one.
4. Since the maximum amount of (extra) domains have been added, only add the first domain of every line to the list *sites*.

```
132 with open(args["l"], "r") as f:
133 
134     max_hit = False
135     max = 7000 # maximum links (extra) to visit
136     n_links = 0
137 
138     for i,line in enumerate(f):
139         if "," in line:
140             p = line.split(",")
141             if not max_hit: # If not hi
142                 for domain in p:
143                     if domain == "\n":
144                         continue
145                     if not domain in sites: # no duplicates
146                         if n_links >= max:
147                             max_hit = True
148                             break
149 
```

```
150                         sites.append(domain.strip())
151                         n_links += 1
152             else:
153                 if domain == "\n":
154                     continue
155                 sites.append(p[0].strip()) # Add OG domain to list
```

Listing 4.31: Code for reading domains and links from file and adding them to list.

4.5.8 Preload List Generation Changes

The crawling implementation also changed how *Preload-gen-list.py* should generate preload lists. Before the implementation, it was easy to know which domains were collected for each site and generate preload lists from them. After the implementation, the number of collected domain for each site has to be known. An argument `"-d"` was added to the program, which was the *domain_data* file that contained the number of domains visited for each site. The *domain_data* file was then read and the number of domains for each site were added to the list *domain_data_info*, as shown in Listing 4.32.

```
157 with open(args["d"], "r") as d:
158     for i, line in enumerate(d):
159         if "," in line:
160             p = line.split(",")
161             domain_data_info.append(p[1])
```

Listing 4.32: Code for reading number of domains and adding them to list.

The list *domain_data_info* was then used in the function *loadcollected()*, which read the collected domains, as shown in Listing 4.33. The variables used are explained as follows:

- index: Index used for retrieving value from domain_data_info.
- link_per_domain: Number of domains/rows read in total.

The function *loadcollected()* began with reading a line from file containing domains (results-eu-3.csv, results-us-2.csv or results-hk-2.csv). For every line, the code did the following:

- In line 167, the first value in *domain_data_info* is assigned to the variable *num_domain*. If, for example, *num_domain* has the value 14, it means that there are 14 rows of domains collected for the domain at index X in the ordered popularity list.
- In line 169, a check is made to see if all rows for the domain at index X have been read. If false, all rows have not been read and we continue to add domains for the current domain at index X in the ordered popularity list. If true, increment *index* by one and add *num_domain* to the total rows read (*link_per_domain*). *Index* is incremented by one to get the next value of the list *domain_data_info*. Then

continue with adding domains to the domain at index X in the ordered popularity list.

```
def loadcollected(l):
    with open(l, "r") as f:
        index = 0
        link_per_domain = 0
        for i, line in enumerate(f,1):
            num_domain = int(domain_data_info[index])

            if i > link_per_domain + num_domain:
                index += 1
                link_per_domain += num_domain

            site = uniquesites[index]

            if site not in site2domains:
                site2domains[site] = []

            if "," in line:
                domains = site2domains[site]
                p = line.split(",")

                for j in range(len(p)-1):
                    d = p[j+1].strip()
                    if d not in domains:
                        domains.append(d)

                site2domains[site] = domains
```

Listing 4.33: Code for reading collected domains and links and then adding them dictionary.

4.6 Collection Runs and Preload List Generation

At the start of the project and when improvements were implemented, collection runs were made and preload lists were generated from them. For every collection run, an additional collection run was made in order to make evaluations.

4.6.1 Original Preload Lists

At the start of this project, a collection run was done with the original unchanged preload tools. The preload lists generated from this run were the base point in this project and every improvement would be compared with these preload lists.

4.6.2 Crawling - All Links

A collection run was made in combination with the implemented crawling tools. The crawling run stored every link found from the first 2000 domains in the popularity

list.

4.6.3 Crawling - Top 10,000 List

Two collection runs were made in combination with the implemented crawling tools. The crawling runs stored five and six links respectively from the first 2000 domains. The collected links were however inconsistent, some domains had a few rows missing. This made it impossible to generate extended top 10-9000 preload lists, since the program could not tell which domains belonged to a certain site. However, an extended top 10,000 preload list could be generated. This was because the effect a few missing rows has on the extended top 10,000 preload lists was not as noticeable, since there are a lot of duplicate links for each domain visited. To prevent collection that results in missing rows, the crawling tool was modified. More about the modification is discussed in Section 4.6.4.

4.6.4 Crawling - Five Links

A collection run was made in combination with the implemented crawling tools. The crawling run stored five links from each of the first 2000 domains in the popularity list. To only store five links, the code that created the file *crawl_list_full* in *web-crawl.py* was modified, as shown in Listing 4.34. The following variables were used:

- `max_links`: Maximum number of links to store.
- `link_counter`: Counter for the number of links stored for the current domain.

When creating the file *crawl_list_full*, the variables `max_links` and `link_counter` were used to only store five links per domain. For each added link, the counter `link_counter` was incremented by one. If `link_counter` reached `max_links`, the for loop would be terminated and started storing links for the next domain.

```
188 with open("crawl_list_full", "w") as w:
189     max_links = 5
190     link_counter =0
191     for k, v in domain_to_link.items():
192         w.write(f"{k},")
193         link_counter = 0
194         for val in v:
195             if link_counter == max_links:
196                 link_counter = 0
197                 break
198             w.write(f"{val.strip()},")
199             link_counter += 1
200 
201         w.write("\n")
```

Listing 4.34: Code to only store five links per domain.

An additional part of *web-crawl.py* that had to be changed was the creation of the file *domain_data*, as shown in Listing 4.35. *Domain_data* contained a domain and a value

(number of links collected) at every line. Since five links were stored per domain, if a value was greater than five, the value was set to five. For example, if a domain had 20 collected links, the number of links would be stored as five. The purpose of this change was that *preload-gen-list.py* used this file to generate preload lists.

```
203 with open("domain_data", "w") as w:
204     for k, v in domain_to_link.items():
205         if len(domain_to_link[k]) > 5:
206             w.write(f"{k}, {5}\n")
207         else:
208             w.write(f"{k}, {len(domain_to_link[k])}\n")
```

Listing 4.35: Code for writing a maximum of five links per domain to file.

One thing to note with the collected links were that when crawling the site `epicgames.com`, six links were collected for some unknown reason. Since this was the only issue with the collected links, one row belonging to epicgames.com that contained duplicate links was removed (for each `*.csv` file)

4.6.5 Filtering

For every preload list generated, a filtered version of the preload list was generated by using the filter functions implemented earlier in this section.

Chapter 5

Results and Evaluation

This chapter begins with presenting the results of this project. Thereafter, the evaluation of the results is made.

5.1 Results

The results of this project, as shown in Table 5.1.1, were generated by the evaluation program *preload-eval.py*, that took collected domains as input and compared them with the generated `ext-tranco-top-10000` preload lists.

The time to generate preload lists with the original and the new preload tools were as follows:

- With the original preload tools, collecting domains and generating preload lists took around nine hours.
- Crawling 2000 sites and collecting up to 7000 links in total took around six hours. Domain collection for the collected links took around eleven hours. Thus, in total time to collect domains and generate lists took around 17 hours.

5.2 Evaluation

The generated preload lists were evaluated by comparing their results from the evaluation program *preload-eval.py*.

5.2.1 Size of Preload List

The filtered preload lists showed an average decrease of around 7% domains. This indicated that around 7% of the domains in the preload lists were considered useless. After filtering, when comparing the preload lists that were generated by collecting all and five links respectively for each site, the preload lists that were generated by collecting five links for each site had around 0,6% more domains. This showed

that collecting all links for each site (up to 7000 links in total) resulted in a lot of duplicate domains. Therefore, there is no benefit to collect more than five links for each site.

5.2.2 Time to Create Lists

Every preload list that were generated by using crawling took the same amount of time to generate, since the link limit were 7000 links for all collection runs. When comparing with the original preload lists, the preload lists that used crawling took around eight hours more to generate. A big part of the increased time was the crawling and link collection which took around six hours alone.

5.2.3 Number of Hits

Since all lookup lists vary a lot in size, there was no point to compare the number of hits. Instead, the hit ratio of each result was compared. Filtering a preload list did not have an impact on the hit ratio, e.g, the largest impact on the number of hits was a decrease by around 0,006% hits. The reason behind this was that (nearly) all domains that got hits in the pre filtered preload lists were not removed during filtering.

Comparing the original preload lists with the preload lists that used crawling showed that the hit-ratio increased by around 1%, except for when collecting all links which had the same hit ratio. This was an interesting result, since the hit ratio was expected to decrease since domains in the preload lists were removed. What this result implied, was that the (nearly) all the domains that were removed during filtering were useless domains, since the number of misses did on average increase by 0,04%.

5.2.4 Number of Useless Domains

By comparing all the filtered preload lists, the number of useless domains were on average reduced by around 57%. When looking further into the useless domains, the majority of filtered useless domains were `*.safeframe.googlesyndication.com` domains. There were a great amount of `*.safeframe.googlesyndication.com` domains in the pre filtered preload lists, which all of them were unique and therefore useless.

The preload list generated by collecting all links for each site had the highest amount of useless domains, while the preload list generated by collecting five links for each site had the lowest amount. A hypothesis would be that the most popular sites on the Internet contain the most amount of advertising domains and by collecting a lot of domains from these sites, the preload lists would contain a lot of useless domains. However, this hypothesis would need to be verified.

The preload lists in section 4.6.3 were not evaluated for this metric, since they were generated from lists with missing rows. Therefore, no evaluation on useless domains could be done that would be certain.

5.2.5 Best Preload List

From the results and metrics gathered, the most promising preload list generated would be the one that used crawling and collected five links for each site. This preload list had the least amount of useless domains (around 8% less than the original preload list), while having a high hit ratio (around 1% higher than the original preload list). The amount of misses for this preload list are relatively low (17% higher than the original preload list), considering that the lookup list was around 31% larger than the lookup list for the original preload list. Compared with the original preload list, collecting five links for each site gave an increase of around 7% domains.

By following the patterns shown in the results, it may be possible that by collecting even fewer links for each site and visiting more sites, greater preload lists could be generated.

5.2.6 Crawling Tools

While the tools for crawling showed great results, the collected domains for the preload lists in section 4.6.3 had some missing rows. The issue was solved and used in Section 4.6.4. However, as earlier mentioned, one site had an extra row of domains which had to be removed. The cause of this issue could perhaps be that something went wrong with parsing that specific sites HTML document, since it only was that site that had an extra row. Nevertheless, this is a issue that should be looked into.

Also, the cloudscraper that was used to bypass Cloudflare protection, did not always succeed. Sometimes the error ”cloudscraper.exceptions.CloudflareChallengeError: Detected a Cloudflare version 2 Captcha challenge, this feature is not available in the opensource (free) version.” appeared, indicating that a paid version of cloudscraper also exist. However, this error did not always appear for the same site all the time, since when rerunning the program, the bypass would sometimes work.

Another issue was that there were a lot of sites that could not be crawled. This issue was a bit harder to solve, since there were a couple of errors that appeared during crawling runs. Some of the errors were:

- ”Max retries exceeded with URL”.
- 308 Permanent Redirect.
- 400 Bad Request.
- 403 Forbidden.

By finding solutions to the errors above, more domains can be found for the preload lists.

5.2.7 Filtering Tool

The implemented filtering showed promising results. As earlier mentioned, the filtering tool removed on average 57% of the useless domains and most of them were `*.safeframe.googlesyndication.com` domains. It was possible to filter even more domains, however no domain stood out compared with `*.safeframe.googlesyndication.com`. There were a lot of domains of the same type that were both useless and useful at the same time. An example of this was the earlier mentioned *.fls.doubleclick.net domains. Another example was the `*.fastly-insights.com domain type`, which had a lot of useless domains but the amount of domains with hits were much greater. Since these domains have the same format, good and useless domains cannot be distinguished when collecting domains. In this example, `*.fastly-insights.com` domains had to be stored in the preload lists, even though some of these domains were useless.

Preload list	Preload list size	Lookup list size	Hits	Hit ratio (%)	Miss	Useless entries
Original	43340	394541	387623	98	6918	4859
Filtered Original	40625	394541	387623	98	6918	2144
Crawling (all links)	46721	580511	571396	98	9115	6658
Filtered Crawling (all links)	43297	580511	571364	98	9174	3236
Crawling (Top 10k, 6 links)	47875	614682	607207	99	7475	5990
Filtered Crawling (Top 10k, 6 links)	44105	614682	607176	99	7506	2221
Crawling (Top 10k, 5 links)	46994	572976	564637	99	8339	5711
Filtered Crawling (Top 10k, 5 links)	43733	572976	564602	99	8374	2451
Crawling (5 links)	46529	572976	564674	99	8302	4928
Filtered Crawling (5 links)	43565	572976	564639	99	8337	1965

Table 5.1.1: Metrics from the evaluation program *preload-eval.py*.

Chapter 6

Conclusions and Future Work

6.1 Discussion

The evaluation of the results in chapter 5 showed both positive and negative effects of the implemented tools. The positive effects were that the preload list that used crawling and filtering contained less useless domains and more useful domains when compared with the original preload list.

As for the negative effects, the crawling tools significantly increased the domain collection time. With only 2000 crawled sites, the collection time was nearly doubled. Depending on how many sites were to be crawled, the collection time could increase even more. However, an increased collection time did not necessary have to be all bad, since the same preload list in the preload defense proposed by Dahlberg and Pulls [8] were used for a couple of months. This means that there was no rush to generate new preload lists quickly. Another negative effect was that the results did not give a clear indication to what the most optimal amount of links would lead to finding the most amount of useful domains. A great amount of different collection runs has to be made in order to find the right amount of links.

6.2 Conclusion

The goal of this project was to create better preload lists that the Tor Project can benefit from. Better preload lists meant that we had to filter useless domains and find more useful domains. By analysing collected domains and identifying useless domains, filtering was implemented in the preload list generating tool. The filtered preload lists showed promising results when on average around 57% of the useless domains were filtered. As for finding more useful domains, crawling tools were created to crawl sites and collect links. From the collected links, more useful domains could be find. When combining the filtering and crawling tools, better preload lists were generated with regard to useless and useful domains. This project can therefore be regarded as a success.

6.3 Future Work

This project laid the foundation for filtering preload lists and crawling sites. The current tools could be further developed to generate even better preload lists. Examples of future work could be:

- Operate a modified exit relay in the Tor network to evaluate the performance of the improved preload lists.
- Explore how many links, collected from crawling, would generate the most optimum preload lists. There is a large number of different combinations possible to find the right amount of links.
- Chapter 5 discussed that not all sites could be crawled. The tools created in this project could be further developed to crawl sites that can not currently be crawled.
- During domain collection runs in this project, we visited sites with `HTTP`. As for future work, collection runs could be made with, for example, `HTTPS` or `www`, to see if even more domains are found.
- The filtering tool can also be further developed to remove more useless domains.
- Domains can be collected from more locations to see if more useful domains are found.

Bibliography

[1] AdBlock. *Surf the web without annoying pop ups and ads!* URL: https://getadblock.com/en/ (visited on 12/02/2022).

[2] Amazon Web Services. *Alexa Top Sites*. URL: https://www.alexa.com/ (visited on 01/02/2023).

[3] Banday, M. Tariq. "Recent Developments in the Domain Name System". In: *International Journal of Computer Applications* 31 (Oct. 2011). DOI: 10.5120/3796-5227.

[4] Cloud Infrastructure Services. *What is DNS Hierarchy Architecture with Examples (Explained)*. URL: https://cloudinfrastructureservices.co.uk/what-is-dns-hierarchy/ (visited on 09/28/2022).

[5] Cloudfare. *DNS server types*. URL: https://www.cloudflare.com/learning/dns/dns-server-types/ (visited on 09/28/2022).

[6] ClouDNS. *DNS cache explained*. URL: https://www.cloudns.net/blog/dns-cache-explained/ (visited on 01/19/2023).

[7] Crummy. *Beautiful Soup Documentation*. URL: https://www.crummy.com/software/BeautifulSoup/bs4/doc/ (visited on 12/20/2022).

[8] Dahlberg, Rasmus and Pulls, Tobias. "Timeless Timing Attacks and Preload Defenses in Tor's DNS Cache". In: *USENIX Security*. to appear. 2023.

[9] Dingledine, Roger, Mathewson, Nick, and Syverson, Paul F. "Tor: The Second-Generation Onion Router". In: *USENIX Security*. 2004.

[10] *DoD standard Internet Protocol*. RFC 760. Jan. 1980. DOI: 10.17487/RFC0760. URL: https://www.rfc-editor.org/info/rfc760.

[11] *Domain names - concepts and facilities*. RFC 1034. Nov. 1987. DOI: 10.17487/RFC1034. URL: https://www.rfc-editor.org/info/rfc1034.

[12] EasyList. *EasyList - Overview*. URL: https://easylist.to/ (visited on 12/02/2022).

[13] GeeksforGeeks. *Components of a URL*. URL: https://www.geeksforgeeks.org/components-of-a-url/ (visited on 01/02/2023).

[14] Greschbach, Benjamin, Pulls, Tobias, Roberts, Laura M., Winter, Philipp, and Feamster, Nick. "The Effect of DNS on Tor's Anonymity". In: *NDSS*. 2017.

[15] Haraty, Ramzi A. and Zantout, Bassam. “The TOR Data Communication System”. In: *JOURNAL OF COMMUNICATIONS AND NETWORKS* (2014).

[16] Lowe, Peter. *Blocking with ad server and tracking server hostnames*. URL: https://pgl.yoyo.org/adservers/ (visited on 12/16/2022).

[17] Mullvad. *Mullvad VPN - Privacy is a universal right*. URL: https://mullvad.net/en/ (visited on 12/01/2022).

[18] Mullvad. *Split tunneling with Linux (advanced) - Guider | Mullvad VPN*. URL: https://mullvad.net/sv/help/split-tunneling-with-linux-advanced/ (visited on 12/01/2022).

[19] OpenVPN. *Business VPN | Next-Gen VPN | OpenVPN*. URL: https://openvpn.net/ (visited on 12/01/2022).

[20] Origin, uBlock. *uBlock Origin - Free, open-source ad content blocker*. URL: https://ublockorigin.com/ (visited on 12/02/2022).

[21] Pochat, Victor Le, Goethem, Tom van, Tajalizadehkhoob, Samaneh, Korczynski, Maciej, and Joosen, Wouter. “Tranco: A Research-Oriented Top Sites Ranking Hardened Against Manipulation”. In: *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, 2019. URL: https://www.ndss-symposium.org/ndss-paper/tranco-a-research-oriented-top-sites-ranking-hardened-against-manipulation/.

[22] Pulls, Tobias and Dahlberg, Rasmus. “Website Fingerprinting with Website Oracles”. In: *PETS* (2020).

[23] Pypi. *Cloudscraper*. URL: https://pypi.org/project/cloudscraper/ (visited on 12/23/2022).

[24] Scrapy. *Scrapy | A Fast and Powerful Scraping and Web Crawling Framework*. URL: https://scrapy.org/ (visited on 12/20/2022).

[25] Tor Project. *Research Safety Board*. URL: https://research.torproject.org/safetyboard/ (visited on 01/20/2023).

[26] Tor Project. *Tor Project | Anonymity Online*. URL: https://torproject.org (visited on 10/21/2022).

[27] Tranco. *A research-oriented top sites ranking hardened against manipulation - Tranco*. URL: https://tranco-list.eu/ (visited on 12/04/2022).