Instructions on replicating our experiments

Benjamin Greschbach — bgre@kth.se
Tobias Pulls — tobias.pulls@kau.se
Laura M. Roberts — laurar@princeton.edu
Philipp Winter — pwinter@cs.princeton.edu
Nick Feamster — feamster@cs.princeton.edu

September 28, 2016

In this document, we provide instructions on how to replicate the results from our research paper "The Effect of DNS on Tor's Anonymity".[1] Each section discusses the replication of a specific experiment, providing both code and data necessary to replicate. Our project page is available online at https://nymity.ch/tor-dns/.

Contents

[1] Available online at https://nymity.ch/tor-dns/tor-dns.pdf.

1 Replicating AS exposure of DNS requests

Here is how you can replicate the results of Subsection IV.A. First, obtain the Python 2 tool ddptr that we have developed for this measurement task:

```
git clone https://github.com/NullHypothesis/ddptr.git
```

Next, you need two files to continue; (*i*) a list of websites such as the Alexa top 1,000 [1], which you should save as `top-1k.csv`; (*ii*) a file `ipasn.dat`, which enables looking up AS numbers for given IP addresses. The library pyasn documents how you can generate this file [2]. After having fetched `top-1k.csv`, `ipasn.dat` and `ddptr.py`, you are ready to run it as follows:

```
sudo ./ddptr.py --fqdn-file top-1k.csv ipasn.dat 2>&1 | tee top-1k-ddptr.log
```

The resulting log files contains plenty of information, but only the lines "Exposure is . . . " are necessary to create Figure 2. You can extract all exposure values from the log files by running:

```
echo exposure > exposures.csv
grep Exposure top-1k-ddptr.log | awk '{print $NF}' >> exposures.csv
```

The file `exposures.csv` then contains all exposure values, one per line, including a header line. It is easy to take the file and plot it with a tool of your choice.

2 Replicating DNS resolver mapping

Here is how you can replicate the results of Subsection IV.B. First, you need to set up an authoritative DNS server for a domain that is under your control. We set up a bind9 instance that was responsible for the zone *.tor.nymity.ch. We configured bind9 to return the same IP address for any request for *.tor.nymity.ch. Next, you need the Python 2 exitmap scanner [4]:

```
git clone https://github.com/NullHypothesis/exitmap.git
```

Exitmap is a module-based scanning framework. We developed a new module for exitmap that is part of the following git repository. You can clone it, and then copy the module to exitmap's modules directory as follows:

```
git clone https://github.com/NullHypothesis/tor-dns-tools.git
cp tor-dns-tools/exit-resolvers/dnsenum.py /path/to/exitmap/src/modules/
```

You are now ready to start the experiment. On the machine running bind9, you need to run tcpdump, to capture all DNS requests going to bind9:

```
tcpdump -n "udp and port 53" -w dns-requests.pcap
```

Bind9 is now ready to serve DNS requests and tcpdump is capturing them to a file. You can now run exitmap to generate the data:

```
exitmap --build-delay 0.5 dnsenum
```

There are various flags for exitmap that can improve the reliability of your experiment. You might be particularly interested in `--first-hop`.

We set up a cronjob to run exitmap automatically, every six hours. First, edit your crontab:

```
crontab -u "$USER" -e
```

Then, add the following line to the crontab file:

```
0 */6 * * * /path/to/exitmap -f /path/to/exitmaprc dnsenum
```

The file `exitmaprc` had the following content. The key `first_hop` is not necessary, but it makes scanning more reliable. We set it to the fingerprint of a Tor relay under our control.

```
[Defaults]
first_hop = 9B94CD0B7B8057EAF21BA7F023B7A1C8CA9CE645
build_delay = 0.5
analysis_dir = /path/to/analysis_directory/
tor_dir = /path/to/tor_directory/
```

Having set up cron, exitmap will now run periodically, and the data you need will be captured in the pcap file that is created on the machine running bind9. Once you are ready to analyze the data, the `README` file of our GitHub repository[2] discusses how you can turn the pcap file into visualizations:

3 Replicating number of DNS requests on exit relays

Because of the ethical issues of recording data on exit relays, we developed a patch for tshark that allows us to reduce the granularity of timestamps. That way, we are able to record the number of DNS requests in a given time period without payload or further information.

The following listing contains that patch. It only modifies two lines of tshark's code. Save it to the file `tshark-binned-timestamps.patch`, and then obtain the source code of wireshark in version 1.12.1, and place it in the same directory as the patch file.

```
diff -Naur wireshark-1.12.1.orig/epan/frame_data.c wireshark-1.12.1/epan/frame_data.c
--- wireshark-1.12.1.orig/epan/frame_data.c     2014-09-16 18:09:03.000000000 +0200
+++ wireshark-1.12.1/epan/frame_data.c  2016-07-21 19:31:56.392000000 +0200
@@ -310,8 +310,8 @@
   fdata->flags.has_phdr_comment = (phdr->opt_comment != NULL);
   fdata->flags.has_user_comment = 0;
   fdata->color_filter = NULL;
-  fdata->abs_ts.secs = phdr->ts.secs;
-  fdata->abs_ts.nsecs = phdr->ts.nsecs;
+  fdata->abs_ts.secs = phdr->ts.secs - (phdr->ts.secs % 3600);
+  fdata->abs_ts.nsecs = 0;
   fdata->shift_offset.secs = 0;
   fdata->shift_offset.nsecs = 0;
   fdata->frame_ref_num = 0;
```

Finally, you can apply the patch as follows.

```
patch -p0 < tshark-binned-timestamps.patch
```

After compiling tshark (which is part of wireshark), you can now run our modified version as follows. The command does not resolve IP addresses; only captures UDP traffic on port 53; only prints timestamps, and no payload; and only prints DNS requests, and no responses. The output of the command is coarse timestamps, one per line, of when tshark captured a DNS request.

```
sudo ./tshark -n -f 'udp port 53' -T fields -e frame.time_epoch -Y 'dns.qry.type == 1 and
    dns.flags.response == 0'
```

4 Replicating DNS requests stats

This is how you replicate the DNS-related statistics from Section V.B. First download and extract the Alexa top one million websites dataset with extracted DNS requests:

```
wget https://dart.cse.kau.se/defector/alexa1mx5-extracted.tar.gz
tar -zxf alexa1mx5-extracted.tar.gz
```

Get the Alexa file we used when gathering the data [1] and the CloudFlare IPv4 addresses[3] at the time of gathering:

[2] Available online at https://github.com/NullHypothesis/tor-dns-tools/tree/master/exit-resolvers.

[3] Available online at https://www.cloudflare.com/ips-v4.

```
wget https://dart.cse.kau.se/defector/top-1m.csv
wget https://dart.cse.kau.se/defector/ips-v4
```

Get and install our dnsstats tool using Go[4]:

```
go get github.com/pylls/defector/cmd/dnsstats
```

Finally, run dnsstats on the downloaded data:

```
dnsstats alexa1mx5-extracted/
```

This produces a large number of statistics, including those used in Section V.B.

5 Replicating DNS to website classification

This is how you replicate the DNS to website classification from Section V.B. First download and extract the Alexa top one million websites dataset with extracted DNS requests:

```
wget https://dart.cse.kau.se/defector/alexa1mx5-extracted.tar.gz
tar -zxf alexa1mx5-extracted.tar.gz
```

Get and install our dns2site tool using Go:

```
go get github.com/pylls/defector/cmd/dns2site
```

For the closed world metrics, run:

```
dns2site -sites 1000000 -instances 5 -open 0
```

For the open world metrics, run:

```
dns2site -sites 500000 -instances 5
```

Without the `-open` parameter, the dns2site tool determines a reasonable open world size based on our conservative power-law distribution, which should be around 433,000.

6 Replicating DefecTor attacks

This is how you replicate the DefecTor results figures from Section VI. First download and extract the traffic traces from our 1,000x100+100,000 WF dataset:

```
wget https://dart.cse.kau.se/defector/alexa1kx100+100k-feat.tar.gz
tar -zxf alexa1kx100+100k-feat.tar.gz
```

Get and install our defector tool using Go:

```
go get github.com/pylls/defector/cmd/defector
```

To generate data for Figure 8, run:

```
defector -sites 1000 -instances 100 -open 100000 -pmin 0 -pmax 100 -pstep 5 -alexa 10000
```

The result is written to stdout and three files: one CSV file for precision, one CSV file for recall, and a log of all output. All filenames capture relevant parameters and are only created upon experiment completion.

To generate data for Figure 9, run as a shellscript:

```
for i in 1 10 100 1000 10000 100000 1000000 10000000 100000000
do
  defector -sites 1000 -instances 100 -open 100000 -pmin 100 -alexa $i
done
```

Manual assembly of the CSV files is needed. For Figures 10a, 10b, 10c, and 10d, run as a shellscript:

[4]Available online at https://golang.org/.

```
# rounds
for i in 0 300 600 900 1200 1500 1800 2100 2400 2700 3000
do
  defector -sites 1000 -instances 100 -open 100000 -pmin 33 -pmax 33 -alexa 10000 -r $i
done
# window size
for i in 90 180 360 540 720 900 1080 1260 1440 1620 1800
do
  for j in 10000 100000
  do
  defector -sites 1000 -instances 100 -open 100000 -pmin 33 -pmax 33 -window $i -alexa $j
  done
done
# Tor network scale
for i in 1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0 10.0
do
  for j in 10000 100000
  do
  defector -sites 1000 -instances 100 -open 100000 -pmin 33 -pmax 33 -scaletor $i -alexa $j
  done
done
# different distributions
for i in conpl realpl conuni realuni
do
  for j in 0 10 100 1000 10000 100000 1000000 10000000 100000000
  do
    defector -sites 1000 -instances 100 -open 100000 -pmin 33 -pmax 33 -simdist $i -alexa $j
  done
done
```

Manual assembly of the CSV files is needed. Generating the figures takes *days* on modern hardware: we spent well over a CPU core-year in total on the figures.

7 Replicating the DNS Alexa top 1,000,000 dataset

This is how you replicate the steps we used to gather the Alexa top 1,000,000 dataset used in Section V.B. First, get and install three tools using Go:

```
go get github.com/pylls/defector/cmd/{server,tbdnsw}
go get github.com/pylls/defector/cmd/extractdns
```

We use a worker-server model, where a server instructs workers to browse to a site and return the observed DNS requests. Download an Alexa file with top sites [1] and run the server:

```
server -f data -s 5 -t 30 top-1m.csv
```

The server will instruct workers to collect in total five samples of the sites in `top-1m.csv`, using 30 seconds per site visit, and store the results in the data folder. By default, the server listens on port 55555 on all interfaces.

Download a fresh copy of Tor Browser from https://torproject.org and extract it. Open `Browser/TorBrowser/Data/Browser/profile.default/preferences/` and put the following *at the bottom* of `extension-overrides.js`:

```
user_pref("app.update.enabled", false);
user_pref("extensions.torlauncher.prompt_at_startup", false);
user_pref("extensions.torlauncher.start_tor", false);
user_pref("datareporting.healthreport.nextDataSubmissionTime", "1559373924100");
user_pref("datareporting.policy.firstRunTime", "1559287524100");
user_pref("extensions.torbutton.lastUpdateCheck", "1559287542.7");
```

```
user_pref("extensions.torbutton.show_slider_notification", false);
user_pref("extensions.torbutton.updateNeeded", false);
user_pref("extensions.torbutton.versioncheck_url", "");
user_pref("extensions.torbutton.versioncheck_enabled", false);
user_pref("network.proxy.proxy_over_tls", false);
user_pref("network.proxy.socks", "");
user_pref("network.proxy.socks_port", 0);
user_pref("network.proxy.socks_remote_dns", false);
```

Download the latest release of dumb-init from https://github.com/Yelp/dumb-init/releases. We need a minimal init system to clean up the many processes we will be creating in Docker. Copy the following into a new file named `Dockerfile`[5]:

```
FROM debian:jessie
MAINTAINER Tobias Pulls <tobias.pulls@kau.se>

RUN apt-get update && apt-get install -y \
        xvfb \
        libpcap-dev \
        libasound2 \
        libdbus-glib-1-2 \
        libgtk2.0-0 \
        libxrender1 \
        libxt6 \
        xz-utils \
  xauth \
  psmisc \
        --no-install-recommends

COPY dumb-init*_amd64.deb /
RUN dpkg -i dumb-init*.deb
RUN rm dumb-init*.deb && apt-get clean && rm -rf /var/lib/apt/lists/* /tmp/* /var/tmp/*

ENV HOME /home/user
ENV LANG C.UTF-8

# create user (start-tor-browser.sh prevents us from running as root)
RUN useradd --create-home --home-dir $HOME user

COPY tbdnsw $HOME/
COPY tor-browser_en-US $HOME/tor-browser_en-US

RUN chown -R user:user $HOME \
        && chmod +x $HOME/tbdnsw \
  && setcap 'CAP_NET_RAW+eip CAP_NET_ADMIN+eip' $HOME/tbdnsw

WORKDIR $HOME
USER user
ENTRYPOINT ["dumb-init", "--"]
```

Build the docker container and start a worker:

```
docker build -t pulls/worker  .
docker run --privileged -d pulls/worker ./tbdnsw <IP:port>
```

The worker will repeatedly try to connect to the server at `<IP:port>`. We successfully ran 120 workers each on several 1U blades with an Intel(R) Xeon(R) CPU E5-2650@ 2.00GHz and 62GiB RAM. The server was hosted on a laptop with SSD storage.

Finally, to extract the DNS data from the resulting pcaps use the extractdns tool:

[5] Based on https://github.com/jfrazelle/dockerfiles/tree/master/tor-browser.

```
extractdns -o results/ data/
```

Where data is the folder the server stored the data in and results is the folder to store the extracted data in.

8 Replicating the WF dataset

To replicate our WF dataset used in Section VI, get and install the following four tools using Go:

```
go get github.com/pylls/defector/cmd/{server,tbw}
go get github.com/pylls/defector/cmd/{torlogext,fext}
```

Download an Alexa file with top sites [1] and run:

```
server -f data -s 5 -t 60 -o .torlog top-1m.csv
```

The server will instruct workers to collect in total five samples of the sites in `top-1m.csv`, using 60 seconds per site visit, and store the results in the data folder with the suffix `.torlog`. By default, the server listens on port 55555 on all interfaces.

Download a fresh copy of Tor Browser from https://torproject.org and extract it. Open `Browser/TorBrowser/Data/Browser/profile.default/preferences/` and put the following *at the bottom* of `extension-overrides.js`:

```
user_pref("app.update.enabled", false);
user_pref("extensions.torlauncher.prompt_at_startup", false);
user_pref("datareporting.healthreport.nextDataSubmissionTime", "1759373924100");
user_pref("datareporting.policy.firstRunTime", "1759287524100");
user_pref("extensions.torbutton.lastUpdateCheck", "1759287542.7");
user_pref("extensions.torbutton.show_slider_notification", false);
user_pref("extensions.torbutton.updateNeeded", false);
user_pref("extensions.torbutton.versioncheck_url", "");
user_pref("extensions.torbutton.versioncheck_enabled", false);
```

Open `Browser/TorBrowser/Data/Tor/torrc` and add:

```
LogTimeGranularity 1
UseEntryGuards 0
```

Next, we need to build a custom tor binary for Tor Browser that logs all incoming and outgoing cells using Tor's logging framework. First, get the tor source code:

```
git clone https://git.torproject.org/tor.git
```

Follow the instructions in the `INSTALL` file. Once you can build tor, apply the below patch to `src/or/relay.c` and run make:

```
diff -Naur tor/src/or/relay.c relay.c
--- tor/src/or/relay.c  2016-08-02 18:04:05.796809070 +0200
+++ relay.c     2016-08-02 18:03:50.036572253 +0200
@@ -585,6 +585,10 @@
     return -1;
   }

+  log_notice(LD_GENERAL, "OUTGOING CIRC %u STREAM %d COMMAND %s(%d) length %zu",
+             circ->n_circ_id, stream_id, relay_command_to_string(relay_command),
+             relay_command, payload_len);
+
   memset(&rh, 0, sizeof(rh));
   rh.command = relay_command;
   rh.stream_id = stream_id;
@@ -1453,6 +1457,9 @@
         ;
     }
```

```
   }
+  log_notice(LD_GENERAL, "INCOMING CIRC %u STREAM %d COMMAND %s(%d) length %d",
+             circ->n_circ_id, rh.stream_id,
+             relay_command_to_string(rh.command), rh.command, rh.length);

   /* either conn is NULL, in which case we've got a control cell, or else
    * conn points to the recognized stream. */
```

Copy `src/or/tor` to `Browser/TorBrowser/tor`.

Download the latest release of dumb-init from https://github.com/Yelp/dumb-init/releases. Copy the following into a new file named `Dockerfile`:

```
FROM debian:jessie
MAINTAINER Tobias Pulls <tobias.pulls@kau.se>

RUN apt-get update && apt-get install -y \
        xvfb \
        libpcap-dev \
        libasound2 \
        libdbus-glib-1-2 \
        libgtk2.0-0 \
        libxrender1 \
        libxt6 \
        xz-utils \
  xauth \
  psmisc \
        --no-install-recommends

COPY dumb-init*_amd64.deb /
RUN dpkg -i dumb-init*.deb
RUN rm dumb-init*.deb && apt-get clean && rm -rf /var/lib/apt/lists/* /tmp/* /var/tmp/*

ENV HOME /home/user
ENV LANG C.UTF-8

# create user (start-tor-browser.sh prevents us from running as root)
RUN useradd --create-home --home-dir $HOME user

COPY tbw $HOME/
COPY tor-browser_en-US $HOME/tor-browser_en-US

RUN chown -R user:user $HOME \
        && chmod +x $HOME/tbw \
  && setcap 'CAP_NET_RAW+eip CAP_NET_ADMIN+eip' $HOME/tbw

WORKDIR $HOME
USER user
ENTRYPOINT ["dumb-init", "--"]
```

Build the docker container and start a worker:

```
docker build -t pulls/worker  .
docker run --privileged -d pulls/worker ./tbw <IP:port>
```

Finally, to extract the data from the resulting torlog-files use the torlogext and fext tools:

```
torlogext -o results/ data/
fext results
```

The torlogext file generates cell-files with the format used by Wang et al. [3], and the fext tool extracts features for Wa-kNN.

9 Replicating Our TorPS Simulations

There are two main components you need in order to recreate the work of Section VII: (*i*) TorPS and (*ii*) traceroutes between client ASes and Tor guards and between Tor exits and destinations. (The scripts we mention below have "usage" comments at the top of their files.)

9.1 TorPS

Grab the code for TorPS from the repository https://github.com/torps/torps. Make sure you run TorPS on a computer that has enough memory. Otherwise, you will encounter an out-of-memory segfault (e.g., 4GB of memory is not enough). In order to run TorPS, go to their github page, and follow their instructions for processing Tor consesuses and descriptors. Here's the command we ran for that step:

```
python pathsim.py process --start_year 2016 --start_month 3 --end_year 2016 --end_month 3
    --in_dir in --out_dir out --initial_descriptor_dir in/server-descriptors-2013-02
```

After that you can run your desired simulations. Here's the command we used:

```
python pathsim.py simulate --nsf_dir out/network-state-2016-03 --num_samples 100000 --trace_file
    here.pickle --user_model typical --format normal --num_guards 1 --guard_expiration 270 tor >
    torps_typical_my_final_model_1_guard_100000_samples.txt
```

In this command, a 100,000-sample simulation is run in which a client has "typical" behavior as given by our modified trace_file (called "`here.pickle`"). The number of guards used is one, and guards expire sometime after 270 days. The file `torps_typical_my_final_model_1_guard_100000_samples.txt` will contain your simulations for the 100,000 samples who visit our chosen websites.

To make our modified trace file, we unpickled their included trace file (used `unpickled.py` on `in/users2-processed.traces.pickle`) and modified it so that it would visit our desired websites.

Because we were interested in only DNS resolution, we used the trace file as an indication of when a client had visited a website and with what Tor circuit (or "path" as TorPS calls it), but we were not worried about any further website communications. See `final_model.py` to see how we have the client visit Facebook, Instagram, Google, Google Mail, Google Docs, Google Calendar, Startpage, Twitter, and Ixquick for our new trace file for the TorPS "Typical" model to use.

9.2 Traceroutes

We use the RIPE Atlas platform to perform our traceroutes and use RIPE Atlas's Cousteau Python wrapper.[6]

9.2.1 Traceroutes from client ASes to Tor Guards

Source of traceroutes: One RIPE Atlas probes in each of our five, chosen Tor client ASes.

Destination of traceroutes: All Tor guards listed in the consensus files for March 2016.

The `getGuardsAndExitsFromConsensusMonthDir.py` script parses consensuses and produces two text files: one that contains all of the guard IP addresses used in a particular month and one that contains all of the exit IP addresses used in that month.

The `do_udp_traceroutes.py` file in the directory `clients_to_all_guards_forward` will perform the traceroutes, and it takes as stdin the IP addresses you want to traceroute to. The script has our five client ASes hard-coded as the sources for our traceroutes. For each of these five ASes, the script will schedule traceroutes to all of the guard IPs found by parsing the consensuses for March 2016 above. The script sleeps for 15 minutes when we approach RIPE Atlas's rate limits. Here's how to use it:

```
python do_udp_traceroutes.py < guardIPSet_Mar_2016.txt > mIDs.txt
```

[6] Available online at https://github.com/RIPE-NCC/ripe-atlas-cousteau.

9.2.2 Traceroutes from exit ASes to Destinations

Source of all the following traceroutes: One RIPE Atlas probe in each of the ASes of the exit relays in the consensus files for March 2016. (Please note that RIPE Atlas does not have probes in *all* of the exit relay ASes.)

As mentioned earlier, use `getGuardsAndExitsFromConsensusMonthDir.py` to get a list of all of the exit relays' IP addresses for March 2016. Then use `getASSetFromIPList.py` in order to convert those IP addresses into a pickle set of their corresponding ASes. Here's an example:

```
python getASSetFromIPList.py exit_AS_set_Mar_2016 ipasn_20160729.dat < exitIPSet_Mar_2016.txt
```

`ipasn_20160729.dat` is an IP/ASN lookup database that you can build by following the instructions at https://pypi.python.org/pypi/pyasn. `exit_AS_set_Mar_2016` will contain your output set.

RIPE Atlas allows you to schedule traceroutes to one destination IP address like 8.8.8.8 from multiple different source ASes (e.g., 200 source ASes), and these traceroutes will all fall under one RIPE Atlas measurement ID (vs. 200 IDs, which you could have if you scheduled all of the traceroutes separately), which is very convenient. However, when you go to schedule such a measurement, if just one of your source ASes does not have a RIPE Atlas probe in it (because as we mentioned above, RIPE Atlas does not have probes in all of the exit relays' autonomous systems), the entire measurement will not be scheduled, and you can't tell which AS was the culprit. Thus, we made a script called `getProbeASSetFromRIPE.py` that outputs a pickle set with the ASes that we believe have active probes based on the output from RIPE Atlas's official command-line tool called Magellan.[7] Here's how we used that tool:

```
~/.local/bin/ripe-atlas probe-search --all > allprobes.txt
```

Then we supply `allprobes.txt` to `getProbeASSetFromRIPE.py` which parses it. We then use `get_covered_ripe_ASes_set.py` to find the intersection between RIPE Atlas's active AS set and our desired exit relay AS set, and we use that intersection for our traceroutes.

Recall that to measure the paths from exit relays to DNS resolvers, we emulated a number of different exit relay DNS configurations.

- *ISP DNS:* No additional traceroutes were necessary for this experiment.
- *Google DNS:*

  Destination of traceroutes: 8.8.8.8 (Google's public DNS resolver)

  Run the following command:

  ```
  python do_udp_traceroutes_from_exit_ases.py exit_AS_set_pickle_file 8.8.8.8 > 8888_mIDs.txt
  ```

  `exit_AS_set_pickle_file` is a pickle file with the set of all exit relays ASes. `8888_mIDs.txt` is an output text file that will contain IDs that RIPE Atlas assigns to your measurements, and it'll contain "True" if a measurement was scheduled or "False" if it was not. Make sure you check it at the end to make sure that the measurements were carried out successfully. If they were not, then you'll have to redo them. Measurements can be unsuccessful if you picked an AS that doesn't contain a RIPE Atlas probe or if you've gone over your measurement limits. See the RIPE Atlas website for the rate limits (https://atlas.ripe.net/docs/udm/#rate-limits).

- *Local DNS:*

  Destination of traceroutes: The IP addresses of the name servers on the delegation paths for resolving www.facebook.com, www.instagram.com, www.google.com, mail.google.com, docs.google.com, calendar.google.com, www.startpage.com, www.twitter.com, and www.ixquick.com.

  We modified `ddptr.py` to create `dd.py` and get the name servers. With these IP addresses in hand, you can use `do_udp_traceroutes_from_exit_ases_ddptr.py` in order to schedule your traceroutes. Here's an example of a command we ran:

  ```
  python do_udp_traceroutes_from_exit_ases_ddptr.py < facebook > facebook_ddptr_mIDs.txt
  ```

[7] Available online at https://github.com/RIPE-NCC/ripe-atlas-tools.

where "facebook" is a text file that contains

```
192.5.5.241
192.33.14.30
69.171.239.12
```

This script doesn't contain any rate limiting because you won't reach the limits.

- *Status quo:*

  Destination of traceroutes: The current resolvers of the exit relays based on the exitmap tool.

  The mapping tool was used to provide a mapping from exit relays to the resolvers they used during March 2016. Some exit relays had several resolver IP addresses associated with them, so we randomly assigned one IP address to those relays and worked with that. In the mapping file, those relays that had their own IP address listed as the resolver are those that performed their own DNS resolution. Use `map_exit_ip_to_one_resolver_randomly.py` to output a pickle dictionary where the key is the exit relay's IP address as a string and the value is its randomly chosen resolver's IP address as a string. Use `group_resolvers_by_dest_n_convert_exit_ips_to_ASes.py` in order to make efficient use of RIPE Atlas. This script will produce a pickle dictionary where the key is the resolver IP address and the value is the set of exit relay ASes that need to perform traceroutes to that resolver IP address. This script ignores exit relays that do their own resolution and those that use 8.8.8.8 as their resolver because these traceroutes are covered by our other traceroutes for the different configurations above. Finally, use `do_udp_traceroutes.py` in the `forward_exit_ases_to_3rd_party_resolvers` directory to perform the RIPE Atlas traceroutes according to the contents of the pickle dictionary file that you just made. Here's how to use it:

```
python do_udp_traceroutes.py resolver_ip_to_exit_AS_set_dict_pickle_fname
    ripe_as_set_pickle_path > third_party_mIDs.txt
```

  This script also takes in a pickle set (`ripe_as_set_pickle_path`) that contains the ASes that currently have active RIPE Atlas probes in them so that your measurements won't fail due to attempting to schedule a traceroute from an exit AS that doesn't contain a RIPE Atlas probe. This script sleeps for 20 minutes when we start reaching RIPE Atlas's rate limits. The script outputs a text file that provides you with RIPE Atlas's measurement IDs for your traceroutes and whether they were successfully scheduled or not. Here's an example of an output file:

```
True
{u'measurements': [4485078]}
True
{u'measurements': [4485079]}
False
{u'error': {u'status': 400, u'code': 104, u'detail': u'Executing this measurement request
    would violate your maximum daily spending limit of {max} credits.  Please stop some of
    your currently running measurements and try again.', u'title': u'Bad Request'}}
```

Looking back on this traceroute process, the first thing to do would really be to run TorPS and *then* do your traceroutes based on the guard IPs and exit IPs that your simulations actually use! That way, you don't end up performing unnecessary traceroutes. (Note: We only consider the forward direction for all of our traceroutes for now.)

9.2.3 Processing the traceroutes:

We use RIPE Atlas's Sagan library[8] to parse the traceroute results. Use the `processAtlasTraceroutes.py` script under the `traceroutes` directory to parse all of your traceroutes. The purpose of this script is to produce Python dictionaries where the key is the source AS of the traceroute as an integer and the value is

[8] Available online at https://github.com/RIPE-NCC/ripe.atlas.sagan.

another dictionary where the key is the traceroute's destination IP address as a string and the value is the set of ASNs as ints that are along the path from that source AS to the destination IP address.

This script takes in as input the name you'd like to give your output dictionary pickle file, your IP/ASN lookup database, and the RIPE Atlas measurement IDs associated with your traceroutes. It'll output the pickle dictionary and a text version of that dictionary.

Here's an example of a command you might run:

```
python processAtlasTraceroutes.py exit_ases_to_8888 ipasn_20160729.dat < 8888_mIDs_only.txt >
    dict_text.txt
```

In order to get a list of isolated, new-line separated measurement IDs from your result output files from running your traceroutes above, use `mIDs_to_text_file.py` to get a text file that you can input to `processAtlasTraceroutes.py` above.

9.3 How we translate IP addresses to ASes

We use the `pyasn` library to translate IP addresses to ASes. The instructions on their website are straightforward.

9.4 Finding Intersections

After you've collected all of your traceroutes, the next step is to find ASes that appear on both the ingress and egress Tor AS paths. We treat the ingress AS path as a set of ASes and the egress AS path as a set of ASes. The ASes that appear in the intersection of these sets are the ASes that can potentially compromise the anonymity of Tor users. Note: If there are missing sets, we assume that there was no compromise for that particular circuit.

9.4.1 Finding Intersections for ISP DNS

Use `intersect_isps.py` to get the intersections (compromising ASes) for this exit relay configuration. `command_100k.sh` in the `isps` directory contains the commands we used. This script takes as input your pickle dictionary of guard AS sets from your traceroutes, the Tor client AS of interest, the name of your output pickle dictionary, and your IP/ASN lookup database. It also takes in your TorPS simulation text file. It outputs a pickle dictionary with the intersections it found and also a text version of it.

Starting from the beginning of the TorPS simulation text file, the script will pick out the guard IP address and the exit IP address that Tor chose for that particular circuit. It will use the guard IP address to find the ingress AS path set in the input dictionary your provided. It will take the exit IP address, convert it to an AS, use that AS to create a set of one, and look for an intersection between the ingress set and that set of one. It will output your results.

9.4.2 Finding Intersections for Google DNS

Use `intersect_8888.py` to get the intersections for this exit relay configuration. `command_100k.sh` in the `8888` directory contains the commands we used. This script takes the same inputs as the ISP DNS script above plus a pickle dictionary of the egress AS paths from the exit relay ASes to 8.8.8.8. This time the exit IP address from the TorPS simulation will be converted to an AS and the exit AS will be used as a key into that egress dictionary to find the appropriate AS path set for that exit relay's path to 8.8.8.8. The script will output your results.

9.4.3 Finding Intersections for Local DNS

Use `intersect_ddptr.py` to get the intersections for this exit relay configuration. `command_100k.sh` in the `"ddptr"` directory contains the commands we used. This script takes the same inputs as the ISP DNS script above plus nine pickle dictionaries of the egress AS paths from the exit relay ASes to mail.google.com, www.google.com, calendar.google.com, docs.google.com, www.facebook.com, www.instagram.com, www.ixquick.com, www.twitter.com, and www.startpage.com. The exit IP address from the TorPS simulation will be used to

identify which of the nine dictionaries should be used for the egress path, and it will be converted to an AS and used as a key into the correct dictionary to get the AS path. The script will output your results.

9.4.4 Finding Intersections for Status Quo

Use `intersect_whole_enchilada.py` to get the intersections for our appromixation of the "status quo" Tor exit relay configurations. `command_100k.sh` in the directory `status_quo` contains the commands we used:

```
#!/bin/bash

python intersect_whole_enchilada.py
      ../../../traceroutes/clients_to_all_guards_forward/results/clients_to_all_guards_forward_dict.pickle
      ../../../traceroutes/exit_ases_to_third_party_status_quo/results/exit_ases_to_third_party.pickle
      ../../../../fresh/fresh_measurements/forward_exit_ases_to_3rd_party_resolvers/final_exit_ip_to_one_resolver_list_dict.pickle
      ../../../traceroutes/exit_ases_to_8888/results/exit_ases_to_8888.pickle
      ../../../traceroutes/exit_ases_to_calendar_google/ddptr/results/exit_ases_to_calendar_google_ddptr.pickle
      ../../../traceroutes/exit_ases_to_docs_google/ddptr/results/exit_ases_to_docs_google_ddptr.pickle
      ../../../traceroutes/exit_ases_to_facebook/ddptr/results/exit_ases_to_facebook_ddptr.pickle
      ../../../traceroutes/exit_ases_to_google/ddptr/results/exit_ases_to_google_ddptr.pickle
      ../../../traceroutes/exit_ases_to_instagram/ddptr/results/exit_ases_to_instagram_ddptr.pickle
      ../../../traceroutes/exit_ases_to_ixquick/ddptr/results/exit_ases_to_ixquick_ddptr.pickle
      ../../../traceroutes/exit_ases_to_mail_google/ddptr/results/exit_ases_to_mail_google_ddptr.pickle
      ../../../traceroutes/exit_ases_to_startpage/ddptr/results/exit_ases_to_startpage_ddptr.pickle
      ../../../traceroutes/exit_ases_to_twitter/ddptr/results/exit_ases_to_twitter_ddptr.pickle compromises_1guard_100K_7922
      7922 ../../../../fresh/forPyASN/ipasn_20160729.dat < ../../torps_typical_my_final_model_1_guard_100000_samples.txt >
      output_dict_1guard_100K_7922.txt
. . .
```

This script takes the same inputs as the Local DNS script above plus the 8.8.8.8 egress pickle dictionary, the status quo egress pickle dictionary, and the pickle dictionary you produced in the traceroutes phase when you mapped a resolver to an exit relay. The script will use the exit relay IP address to figure out what type of resolution should be used. If, according to the mapping, an exit relay does its own resolution, the script will use the Local DNS dictionary for the egress path. If, according to the mapping, the exit relay uses 8.8.8.8 for resolution, the script will use our 8.8.8.8 dictionary for the egress path. If, according to the mapping, the exit relay uses another resolver, the script will use the status quo egress dictionary. For all of these dictionaries, the exit relay's IP address will be converted to an AS before being used as a key into those dictionaries. The script will output your results.

9.5 Getting the Results

Use `command.sh` under the directories `numbers/AS_Numbers_March_2016/`X to get your results, where X represents the five client AS numbers. Here's the `command.sh` for client AS 2856:

```
#!/bin/bash

# 8888 ##########
python ../../../../fresh/numbers/number_of_compromised_streams_per_sample.py
     ../../../intersections/AS_Analysis_March_2016/8888/compromises_1guard_100K_2856.pickle >
     8888_comp_streams_2856.txt

python ../../../../fresh/numbers/time_to_first_compromise_per_sample_only_for_comp.py
     ../../../intersections/AS_Analysis_March_2016/8888/compromises_1guard_100K_2856.pickle >
     8888_time_first_comp_2856.txt

python ../../../../fresh/numbers/popular_ASes_or_IXPs.py
     ../../../intersections/AS_Analysis_March_2016/8888/compromises_1guard_100K_2856.pickle >
     8888_culprits_2856.txt


# ddptr ##########
python ../../../../fresh/numbers/number_of_compromised_streams_per_sample.py
     ../../../intersections/AS_Analysis_March_2016/ddptr/compromises_1guard_100K_2856.pickle >
     ddptr_comp_streams_2856.txt
```

```
13
14 python ../../../../fresh/numbers/time_to_first_compromise_per_sample_only_for_comp.py
       ../../../intersections/AS_Analysis_March_2016/ddptr/compromises_1guard_100K_2856.pickle >
       ddptr_time_first_comp_2856.txt
15
16 python ../../../../fresh/numbers/popular_ASes_or_IXPs.py
       ../../../intersections/AS_Analysis_March_2016/ddptr/compromises_1guard_100K_2856.pickle >
       ddptr_culprits_2856.txt
17
18
19 # isps ##########
20 python ../../../../fresh/numbers/number_of_compromised_streams_per_sample.py
       ../../../intersections/AS_Analysis_March_2016/isps/compromises_1guard_100K_2856.pickle >
       isps_comp_streams_2856.txt
21
22 python ../../../../fresh/numbers/time_to_first_compromise_per_sample_only_for_comp.py
       ../../../intersections/AS_Analysis_March_2016/isps/compromises_1guard_100K_2856.pickle >
       isps_time_first_comp_2856.txt
23
24 python ../../../../fresh/numbers/popular_ASes_or_IXPs.py
       ../../../intersections/AS_Analysis_March_2016/isps/compromises_1guard_100K_2856.pickle >
       isps_culprits_2856.txt
25
26
27 # status_quo ##########
28 python ../../../../fresh/numbers/number_of_compromised_streams_per_sample.py
       ../../../intersections/AS_Analysis_March_2016/status_quo/compromises_1guard_100K_2856.pickle
       > status_quo_comp_streams_2856.txt
29
30 python ../../../../fresh/numbers/time_to_first_compromise_per_sample_only_for_comp.py
       ../../../intersections/AS_Analysis_March_2016/status_quo/compromises_1guard_100K_2856.pickle
       > status_quo_time_first_comp_2856.txt
31
32 python ../../../../fresh/numbers/popular_ASes_or_IXPs.py
       ../../../intersections/AS_Analysis_March_2016/status_quo/compromises_1guard_100K_2856.pickle
       > status_quo_culprits_2856.txt
```

The results are the number of compromised streams per sample, the time-to-first-compromise, and a count of the compromising ASes so you can see which ASes were the most popular "culprits."

Doing `wc -l` on your time-to-first-compromise files will reveal the number of samples out of 100,000 that were compromised. For our graphs, we set the time-to-first-compromise number for the samples that were uncompromised to April 1, 2016 as a placeholder as further explained in our paper.

The format for the compromised streams per sample file is: Sample #, # of compromised streams for month of March, # of non-compromised streams, # of streams that I had (at least partial) traceroute info for out of 372 possible streams, 372

The format for the time-to-compromise file is: Sample #, number of seconds till the first compromised stream (Sample #'s not in the file are samples that weren't compromised in March.)

The format for the "culprit" AS file is: AS #: # of times it appeared in compromised streams for March 2016.

References

[1] Amazon Web Services. *Alexa Top Sites*. URL: https://aws.amazon.com/alexa-top-sites/ (cit. on pp. 2, 3, 5, 7).

[2] Hadi Asghari. *pyasn – Python IP address to Autonomous System Number lookup module*. URL: https://github.com/hadiasghari/pyasn (cit. on p. 2).

[3] Tao Wang, Xiang Cai, Rishab Nithyanand, Rob Johnson, and Ian Goldberg. "Effective Attacks and Provable Defenses for Website Fingerprinting". In: *USENIX Security*. USENIX, 2014. URL: https://nymity.ch/tor-dns/pdf/Wang2014a.pdf (cit. on p. 8).

[4] Philipp Winter. *exitmap—A fast and modular scanner for Tor exit relays*. URL: https://github.com/NullHypothesis/exitmap (cit. on p. 2).