Website Fingerprinting in the TOR Network

Documentation and instructions on collecting data, data sets, extracting features, and performing classification

Norbert Landa
norbert.landa@rwth-aachen.de

Martin Henze
martin.henze@rwth-aachen.de

Sebastian Agethen
sebastian.agethen@rwth-aachen.de

Contents

1 Preparations

1.1 Used Software

A number of programs are required to collect data:

Program	Description
Ubuntu	Used Version: 12.04 LTS 32 bit
LUbuntu	Used Version: 12.10 32 bit (weak machines)
TorBrowserBundle	Used Version: 0.2.3.25-15
Chickenfoot	Addon for Firefox(Modified to work with current Firefox)
Scriptish	Addon for Firefox
vino	VNC can be useful in case errors occur when collecting data
tcpflow	We use a modified version of tcpflow (see below)
Matlab	Outlier detection is done in Matlab (see below)
libsvm	Support Vector Machines, used in Matlab
tcpdump, libpcap	Tools for capturing

1.2 Setting up a machine for traffic recording

Install Ubuntu 12.04 LTS 32 bit or Lubuntu 12.10. Use the same username on all machines of your testbed, that makes things easier.
Copy the folders `bin`, `crawling`, `fetches_raw` and `fetches_compiled` to your home directory.
Then run:

```
sudo apt-get install libpcap-dev tcpdump python-numpy python-torctl
 apparmor-utils make screen openssh-server -y
```

Configure your preferences for the remote connection with:

```
vino-preferences
```

Now tcpdump has to be configured. Type:

```
sudo visudo
```

and append:

```
yourusername ALL=NOPASSWD:/usr/sbin/tcpdump
```

After saving, execute

```
sudo aa-complain /usr/sbin/tcpdump
```

Then compile tcpflow:

```
cd bin/tcpflow
./configure
make
```

To optionally enable the automatic login the corresponding option can be enabled in the `System Settings`-Menu.
Depending on the version of the files you currently use, you may have to change the username in some files to fit your username [1]. The files are:

- bin/tor-kill-streams.py
- bin/torstramstatus.py
- bin/tor-browser_en_US/Data/profile/compatibility.ini
- bin/tor-browser_en_US/Data/profile/search.json
- bin/tor-browser_en_US/Data/profile/prefs.js

[1] The current username is landa

- bin/tor-browser_en_US/Data/profile/extensions.ini
- crawling/fetch-and-calculate.sh
- crawling/run-client-torbrowser-newtor.sh
- crawling/copytoserver.sh (also change the IP and path in the SCP command to the values of your data storage location)

1.3 Setting up your own Tor Browser Bundle

First of all check out the Tor Browser Bundle source from the TorProject git. Then get rid of the unwanted modifications is the source code. Currently there is a patch that prevents taking screenshots (in the folder `torbrowser/src/current-patches/firefox/`. Now you can compile the browser bundle.
When finished, copy the `tor-browser_en_US` folder to `/home/user/bin` and start the bundle with the `start-tor-browser.sh` script.
Install the modified Chickenfoot plug-in from the provided file. Restart Firefox and go to `about:config`. Set `dom.max_chrome_script_run_time` to `-1`.
Install the Scriptish plug-in. Depending of your Firefox version you have to take an old version from the Scriptish repository as current versions of Sciptish do not work with old versions of Firefox.
Using Firefox8, we encountered a problem with javascript alert events, unload events, etc., which are blocking the continuation of the script and thereby distorting the results. To solve the problem we use Scriptish to modify the behavior of Firefox.
Add the following scripts to Scriptish.
The first script is called `blockunloadevents.user.js` and designed to block messages that appear when navigating away from a website:

```
// ==UserScript==
// @name           BlockUnloadEvents
// @namespace      http://theaceoffire.8k.com/STOPTHEMADDNESS
// @include        *
// ==/UserScript==
(function() {
      unsafeWindow.onbeforeunload = null;
      unsafeWindow.onunload = null;
})();
```

For the second script, called `alert-killer-test@erikvoldcom.user.js`, we require the Scriptish addon:

```
// ==UserScript==
// @id              alert-killer-test@erikvold.com
// @name            Overwrite Alert
// @description     Overwrites alert()
// @include         *
// @author          Erik Vold
// @run-at          document-start
// ==/UserScript==

unsafeWindow.alert = function(){};
unsafeWindow.confirm = function(){};
unsafeWindow.prompt = function(){};
```

Both scripts replace the original functions by empty ones. Take special note of the line `@run-at document-start`, which is supported by Scriptish only. Greasemonkey only executes the script after the document, i.e., index.html has been loaded, thereby failing to prevent such dialogs, that were called earlier.
Finally adjust the `tor-browser_en_US/Data/Tor/torrc` configuration file to match your setup.

1.4 Setting up the closed world evaluation

First of all setup the machine for traffic recording. Then run:

```
sudo apt-get install libtool gcc g++
```

Install MATLAB and download libSVM or use the provided version of libSVM. `libsvm`[2] is a library for Support Vector Machines. We will be using it in connection with Matlab.
After downloading, compile the files with

```
make
```

Then, open Matlab and navigate to the matlab subfolder of your libsvm folder. Finally, execute make again. If you get an error, setting absolute paths in make.m helped. If you use the provided version of libSVM you have to adjust the paths.

2 Overview of available scripts

2.1 Chickenfoot

`crawling/scripts/create-chickenfoot-file-modified.py`

This Python script is used to create a Chickenfoot script which contains all the parameters (path, run identifier, number of runs, timeout, hostname) and a list of the URLs that should be visited. The created script is launched on Firefox start-up.

`crawling/scripts/worker.js`

This is the file that basically performs all the work. It is invoked by the Chickenfoot script which holds all the parameters (see above). It iterates *number of runs*[3] times over the URL set. A fetch is considered successful, if:

1. All TOR streams have been closed ten minutes after the initial request for the webpage has been performed
2. Firefox states that the page has been loaded completely
3. Neither soft timeout (see below) nor hard timeout (5 $\times$ soft timeout) have been reached

For each fetch of a webpage the URL and the start time and end time are stored in a file. If the fetch fails, the endtime is replaced by an error code:

-1	Timeout (hard or soft)
-2	Empty page has been loaded (URL is `about:blank`)
-3	Firefox states that page has not loaded completely
-4	JavaScript Exception was thrown
-5	Waited too long for TOR streams to finish

After all requested fetches have been performed, the completion of the crawling process is signaled by writing "1" to the lock file.

`crawling/scripts/util.js`

Contains the following utility functions: `trim()`

`crawling/scripts/close-windows.js`

This file contains a function that periodically calls it selfs and closes all open Firefox windows except the main window. This avoids problems with alert windows, pop-ups, and certificate warnings. Initially, this function is called by `worker.js` (see above).

[2] `http://www.csie.ntu.edu.tw/~cjlin/libsvm/`
[3] This parameter was used in previous work. You should never use another value than 1 to ensure that a new circuit is built after the whole list is downloaded once

crawling/scripts/progress-listener.js

This function hooks into the progress change function which is also used to update Firefox's loading progress bar. It is added as a so-called progress listener by `worker.js` (see above). On every progress change (i.e. one object of the page has been loaded successfully) the soft timeout is resetted.

crawling/scripts/flash-bridge.js (unused)

This implements a work-around to read the progress state of an embeded flash object. The function has to be called for each flash object which is embedded in the page. It expects the DOM index of the queried flash object as parameter. **Note:** This is currently not used as we disabled the Flash plugin (see above).

2.2 Automated Collection (`crawling/fetch-and-calculate.sh`)

The whole process of starting up and tearing down all required tools and programs is performed by `fetch-and-calculate`
It is configured both by parameters (which are specific for each run) and static variables (that only need to be adjusted once for each run environment). The expected parameters are:

1	runidentifer	An identifier for the collection run
2	runs	Number of requests for each page[4]
3	timeout	Timeout in seconds
4	urlfile	Path to the file with the URLs (one per line)

Additionally, the following static variables can / have to be modified in the script `run-client-torbrowser-newtor.sh` file:

PYTHONPATH	Path to additional Python libraries (used to store TorCtl)
TORBIN	Path to TOR binary
TORCONFIG	Path to TOR config file
FIREFOXBIN	Path to Firefox binary
TCPDUMPBIN	Path to tcpdump binary
TORCONTROLBIN	Path to TOR control information collecting script
TORSTATUSBIN	Path to TOR stream status collecting script
BASEPATH	Base path of crawling environment (directory where `run-client.sh` is in)
ETHDEVICE	Identifier for the ethernet device we want to capture on

At startup, the script first kills all required programs as there (accidently) might be left overs from previous runs. It initializes the status files and saves the IPv4 address of the machine it is running on to a file. The order of the list of URLs is randomized. This reduces the chances of the same webpage being crawled more than once within a short time frame. As typically multiple machines are crawling the webpages at the same time, we such ensure that each machine has its own random order.
After the Chickenfoot file for this run has been created (see above), the necessary programs can be started (running in background mode). This includes TOR, `tcpdump`, and Firefox as well as the helper scripts `tor-control.py` and `tor-streamstatus.py` (see below).
The script then busy-waits until the Chickenfoot scripts signals (using a status file) that the crawling has been completed successfully. Finally, a termination signal is sent to the started programs, which are running in background mode.
After all webpages have been downloaded the script calls multiple other scripts (`raw-to-tcp.py`, `parse-tcp.py`, `raw-to-tls.py`, `parse-tls.py`) in order process the traffic dump into more lightweigt files only containing timestamps, packet sizes and directions. These lightweight files are then copied to `/home/user/fetches_compiled`. All files (including the huge raw dump file) are copied to `/home/user/fetches_raw` and compressed.

2.2.1 bin/tor-control.py

This script runs in parallel to the measurements. It connects to TOR's control port and polls the list of connected TOR nodes. For each previously unseen TOR node, it outputs the IPv4 address. Later on, this allows us to filter out non-TOR traffic that might have occured.

2.2.2 `bin/tor-streamstatus.py`

This script also runs in parallel to the performed measurements. It connects to TOR's control port and polls the list of open streams. The total number of open TOR streams is periodically written to a file. This information is used by the Chickenfoot script to distinguish whether a page has been loaded successfully or not.

2.2.3 `bin/tor-kill-streams.py`

This script also runs in parallel to the performed measurements. It terminates all open streams after Chickfoot signals that the page has finished downloading.

2.3 Various Tools

2.3.1 check-urls-exist.py

Checks if for each URL in an URL file (one URL per line) a corresponding output file exists.[5]

2.3.2 tor-entry-nodes.py

This script can be used to generate a comma seperated list of TOR entry nodes which run on a certain node. This can be used if you want to capture on one specific port only (which we are currently NOT doing).[6]

2.3.3 1vsall/1vsall.sh

Performs a 1-vs.-all prediction for all classes, i.e., each class is compared to one class (+999, change this if you have more classes) which contains all the other classes.[7]

2.3.4 1vsall/results.py

Prints for each class the results (classid, true positive, false negative, false positive, true negative) of the 1-vs.-all run.[8]

3 Collecting Data

This section describes the process of collecting the data to be evaluated in Section 4.

3.1 Performing a Collection Run

Change the directory to `crawling`. Run `clear-all.sh` to remove leftovers from previous runs. Start the collections run with

```
sudo ./fetch-and-calculate.sh IDENTIFIER 1 TIMEOUT URLLIST
```

After the machine has finished, you can copy the files to you server with

```
sudo copytoserver.sh
```

3.2 Output of a Collection Run

The various output of a collection run is written to different files and directories, which are described in the following.

crawling/dumps/RUNIDENTIFIER-HOSTNAME.raw

This file holds a raw capture of the networking traffic in `libpcap` format. It can be displayed, e.g., using Wireshark.

[5] This is an old file that I have not used. NL
[6] This is an old file that I have not used. NL
[7] This is an old file that I have not used. NL
[8] This is an old file that I have not used. NL

crawling/ips/RUNIDENTIFIER-HOSTNAME.ownips

This file contains the IPv4 address of the network interface which was used to fetch the data.

crawling/ips/RUNIDENTIFIER-HOSTNAME.torips

This file contains the IPv4 addresses of all TOR nodes that had an open connection with the TOR client during our measurements.

crawling/timestamps/RUNIDENTIFIER-HOSTNAME.log

This file contains information on each fetch of a webpage. Each line represents one fetch, that is the URL, start time, and end time (or error code, see above).

crawling/log/

This folder contains the log files of TOR and Firefox. Might be useful for debugging.

crawling/tmp/

This folder contains temporary data which is useless once the collection run has finished.

3.3 Processing Collected Data

In order to transfer the collected raw traffic dumps into data which we can use for machine learning, we follow a two stage process. Firstly, we use (modified) third-party tools to parse the raw data into human-readable information. Secondly, we use timing information in order to figure out which information belongs to which fetch of a webpage.

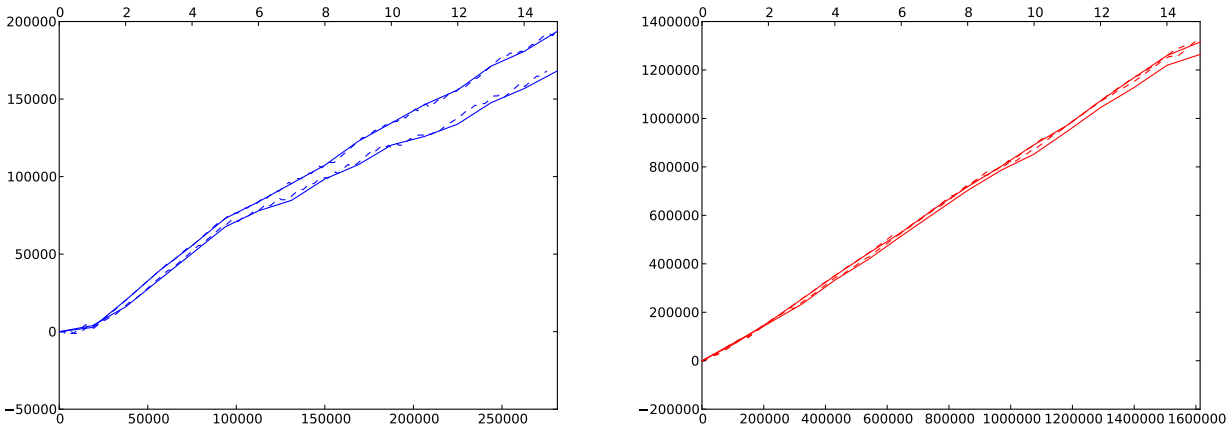
3.3.1 Extracting TCP Packet Sizes

The `crawling/raw-to-tcp.py` script uses `tcpdump` to extract the relevant information (IP addresses, ports, time, payload size) for each TCP packet.
This information is written to a file (`crawling/dumps/RUNIDENTIFIER-HOSTNAME.tcpdump`).
The `crawling/parse-tcp.py` script brings together all the collected information. For each fetch of a webpage, a list of timestamp / packet size pairs is stored. We ignore packets with no payload (e.g., ACK packets). In order to distinguish between incoming and outgoing traffic, we store the sizes of outgoing packets as negative values.
As this scripts are called automatically now, you do not have to call them manually.

3.3.2 Extracting TLS Record Sizes

The `crawling/raw-to-tls.py` script extracts the relevant information (IP addresses, ports, time, record size) for each TLS record. Extracting the sizes of TLS records requires much more effort, as one TCP packet can contain multiple TLS records and one TLS record can spread out over multiple TCP packets. Thus, we have to interpret the data on the basis of TCP flows. However, it might happen (we do not really know why) that the TOR client stops receiving data on a TCP flow and after a while reinitiates this TCP flow with a SYN packet. Commonly used software for parsing TLS records (e.g., Wireshark and tshark) fail at recognizing this accordingly, which leads to a loss of TLS records. We decided to use tcpflow[9] for splitting the trace file into multiple files which each contain the payload of one TCP flow direction. Using the TCP payload it is now possible to parse the whole TLS stream and extract the length information of all application records (`0x17`). If a reinitiation of a TCP flow is detected (i.e. we don't find a TLS header at the expected position in the flow) we mark this and the previous TCP packet as invalid. In order to match positions in the TCP flow (i.e. offset in the file) to TCP packets and thus timestamps, we modified tcpflow as follows. Whenever a TLS record is written to the corresponding file, we write the offset in the file and the timestamp of the TCP packet to the `.time`

[9] `http://afflib.org/software/tcpflow`



(a) `aby.spinchat.de` (b) `bands.rock-im-park.de`

Figure 1: Cumulative curve of packet sizes (dashed) and approximation (solid).

file for this flow. The `crawling/raw-to-tls.py` script is then able to match the offset to the corresponding timestamp and write this information to an output file.
The `crawling/parse-tls.py` script brings together all the collected information. It first has to sort the output file of `crawling/raw-to-tls.py` as we expect a chronological input file. For each fetch of a webpage, a list of timestamp / TLS record size pairs is stored[10]. In order to distinguish between incoming and outgoing traffic, we store the sizes of outgoing packets as negative values.
As this scripts are called automatically now, you do not have to call them manually.

4 Feature Extraction and Outlier Detection

4.1 Feature Extraction

We use the Python scripts `compose-features-tls.py` resp. `compose-features-tcp.py` to extract features from the collected information. Later on, we will choose a subset of these features and pass them to the SVM.

4.1.1 Total Size of Incoming Packets

This feature contains the sum of the size of all incoming packets resp. incoming TLS application records. Values are from $\mathbb{N}_0^+$.

4.1.2 Total Size of Outgoing Packets

This feature contains the sum of the size of all outgoing packets resp. outgoing TLS application records. Values are from $\mathbb{N}_0^+$.

4.1.3 Cumulative Curve of Packet Sizes

This set of features approximates the cumulative curve of the packet resp. TSL application record sizes. We first calculate the cumulative values of packet sizes c_i and the cumulative absolute values of packet sizes t_i as follows:

$$c_i = \begin{cases} s_0, & \text{if } i = 0, \\ c_{i-1} + s_i, & \text{else.} \end{cases} \qquad t_i = \begin{cases} |s_0|, & \text{if } i = 0, \\ t_{i-1} + |s_i|, & \text{else.} \end{cases}$$

The cumulative curve is then described by (t, c) (see dashed lines in Figure 1). We use one-dimensional linear interpolation to extract the desired number of features (currently 15) from this curve (see solid lines in Figure 1).

[10] If a TLS record reaches over more than one TCP packet, we use the timestamp of the TCP packet in which the TLS record starts.

4.2 Outlier Detection

We use Matlab for detecting and removing outlier in the recorded data. Right now we are using the *total size of incoming packets* feature for outlier detection. It is also possible to perform the removal of outliers iteratively for more than one feature.
For each class (i.e., URL), we perform the removal and detection of outlier in three steps:

1. We remove all instances with a total size of incoming packets less than 1172 bytes (which means that not more than two Tor cells have been received).
2. From the remaining set, we calculate the median. All instances that deviate more than 20% from the median are removed.
3. Again, from the remaining set, we calculate the *interquartile range (IQR)*: $IQR = Q_3 - Q_1$. We remove all instances which are below $Q_1 - 1.5 \cdot IQR$ or above $Q_3 + 1.5 \cdot IQR$.

5 Closed world evaluation

5.1 Installation

Run `sudo apt-get install libtool gcc g++` in order to install the required packets. Copy the the folders `evaluation`, `libsvm-3,17` and `mergeCompiledFiles` from the repository to the home folder. Change directory in a terminal to the `libSVM-3.17` folder and run `make`. Copy `svm-predict`, `svm-scale`, and `svm-train` to your `crawling` directory. Add the `libsvm/matlab` folder to path in MATLAB. Run `libsvm-3.17/matlab/make.m` in MATLAB (you may have to repeat this step before evaluation).

5.2 Preparation

Copy the folders from the `fetches_compiled` folder into `mergeCompiledFiles/fetches`. Run `mergeCompiledFile` in order to merge the traffic traces from multiple runs and calculate the features from the merged data. Delete the old data in `crawling/features-tcp` and `crawling/features-tls` and move `mergeCompiledFiles/features-t` and `mergeCompiledFiles/features-tls` into the `crawling` folder.

5.3 Evaluation

The closed world evaluation consists of three steps:

1. Outlier detection and conversion of the data into the format needed by the SVM
2. K-fold cross-validation
3. Calculation of the accuracy

For each step there are several scripts available:

1. :
   - `synchronizedSvmFile.m`: The script prepares the tcp and tls data and uses only traces that are in both data sources. Use this to directly compare TCP with TLS data. Don't use this for final results, as it may filter some valid traces.
   - `preparetcponly.m`: Prepares the TCP data
   - `preparetlsonly.m`: Prepares the TLS data
2. :
   - `svm.m`: K-fold cross-validation of the TCP data
   - `svm_tls.m`: K-fold cross-validation of the TLS data
3. :
   - `evaluation.m`: Calculates the TCP accuracy
   - `evaluation_tls.m`: Calculates the TLS accuracy

6 Data Sets

6.1 Herrmann

We started with a list of 771 URLs, which was used in previous work by Herrmann et al. [**?**]. After excluding some URLs for various reasons (see below), we obtained a list of 728 URLs[11]. Today over 100 URLs cannot be reached.

6.2 Random Datasets

Alexa2k

This data set contains (as of 05/12/2011) 1969 random URLs from Alexa's "Top 1,000,000 Sites" set (retrieved on 24/10/2011). We randomly choose URLs from the Alexa set using `shuf`:

```
shuf -n 1000 alexa-top-1m-2011-10-24.txt > rand1000
```

This measurement was conducted using Firefox 3.6.

Alexa5k

We ran another measurement using the modified version of Firefox 8. A set of 5000 URLs from Alexa's set, as above but excluding those URLs chosen for the Alexa2k set, was determined:

```
#! /bin/bash
#Exclude Alexa2k set:
IFS=$'\n'
for NAME in $(cat remove-these.txt)
do
        sed -ie "\|^$NAME\$|d"  complete-set.txt
done

#Choose 5000 URLs from the modified file
shuf -n 5000 complete-set.txt > alexa5k
```

6.3 Overview of available Datasets

Identifier	Description	# URLs	# Instances
`herrmann`	unmodified Firefox 3.6	728	20
`herrmann-pipelining`	Firefox 3.6 with backported Pipelining patch *This particular Pipelining patch did contain a bug*	191	10
`random`	unmodified Firefox 3.6	1969	10
`herrmann-shuffled`	Firefox 8 with randomized fetch order	728	10
`herrmann-torbrowser`	Torbrowser 2.2.33-3	728	10
`herrmann-firefox8`	Unmodified Firefox 8	728	10
`herrmann-final-shuffled` (Rerun needed, bad settings)	Firefox 8 with randomized fetch order (final patch)	728	20
`alexa5k` (To date incomplete)	Firefox8 (pipelining disabled)	5000	20
`alexa5k-shuffled` (To date incomplete)	Firefox8 with randomized fetch order	5000	20

Randomized fetch order refers to the pipelining patch written by Martin Henze.

[11] `url-files/herrmann`

7 Results

Identifier	# Instances	Features	Av. recognized URLs	Standard deviation
herrmann (Firefox 3.6)	20	TLS: totalIn, totalOut, cumu15	82.074176%	1.051121%
herrmann (Firefox 3.6)	10	TLS: totalIn, totalOut, cumu15	76.126374%	1.100426%
herrmann (Firefox 3.6)	20	TLS: totalIn, totalOUt	37.280220%	1.032405%
herrmann-torbrowser	10	TLS: totalIn, totalOut, cumu15	65.818432%	1.650937%
herrmann-shuffled	10	TLS: totalIn, totalOut, cumu15	55.041322%	1.948169%
random	10	TLS: totalIn, totalOut, cumu15	73.417979%	0.630798%
herrmann-firefox8	20	TLS: totalIn, totalOut, cumu15	77.707756%	1.506870%
herrmann-firefox8	20	TLS: totalIn, totalOut, cumu15, strip32	77.735457%	1.469831%
herrmann-final-shuffled (Firefox 8)	20	TLS: totalIn, totalOut, cumu15	— (69.972414) %	— (1.452918) %
herrmann-final-shuffled (Firefox 8)	20	TLS: totalIn, totalOut, cumu15, strip32	— (69.944828) %	— (1.889043) %
Alexa5k (Firefox 8)	20	TLS: totalIn, totalOut, cumu15	— %	— %
Alexa5k-shuffled (Firefox 8)	20	TLS: totalIn, totalOut, cumu15	— %	— %