Timing Analysis Attack Simulation in Tor Networks

Davide Berardi, Matteo Martelli

July 13, 2015

Abstract

In this document we talk about blablabla

1 Introduction

Protecting data privacy on the Web is a very hot topic nowadays. Users of the Web may want to surf without the risk that their personal informations can be read by other users. One of the most largely-used architecture for this purpose is the *Onion Routing* and its protocol implementation: *Tor*[5]. In fact, the latter is modeled with several techniques with the aim to provide communication security and data privacy to its network users. Anyway there have been recent papers pointing out the Tor's vulnerabilities. As the Tor community itself stages, "Tor does not provide protection against end-to-end timing attacks"[2], thus the chance for an attacker to eavesdrop a Tor communication traffic and discover the users involved in it by a timing analysis is a well known vulnerability of Tor.

In this kind of analysis, in order to identify the source of a communication, the attacker should be able to trace the outgoing traffic and the incoming traffic from both the entering and exiting node of the communication path. Clearly a timing analysis is feasible only under a certain amount of conditions that are often hard to satisfy. As instance, discovering that a generic user U is connecting to a server S over a Tor communication may require tracing the traffic of many users in the network, as the attacker cannot know which users may be interested in connecting with S. Also, there may be the need of tracing more than just the interested server because the attacker can find a better time relation between the user U and another server S' than between U and the interested server S, thus the attacker could exclude U to be a possible connection source for S. In the section 2 we will better describe how Tor works and how the end-to-end timing attack could be performed.

In order to test the feasibility and the parameters involved in a time analysis attack over the Tor network, we set up a simulation scenario in which a series of simulation runs have been performed and some interesting empirical results have been taken out and analyzed. At the end we will point out how the Tor time analysis vulnerability can be critical and we will introduce some proposals to enhance Tor with the view of preventing this kind of attacks.

2 Tor

Tor is an implementation of the onion routing architecture model. The onion routing consists in a technique that provides anonymous connections over a computer network[7].

Tor is born with the aim to allow people to improve their privacy and security on the Internet. Its architecture is based on the Onion Routing model and it's widely used by many user over the world. Users may be interested on using Tor for different purposes such as avoiding website tracking, communicating securely over Internet messaging services, or just web surfing with the access on the services blocked by their local Internet providers.

The idea behind Tor, and Onion Routing as well, is to protect people against a common form of Internet surveillance known as "traffic analysis". Traffic analysis can reveal information about the network traffic such as the source, destination, size, timing and more of the analyzed traffic packets. This can be possible even if the packets are cyphered because the traffic analysis focuses on the header part of the packets that are in plain text.

Thus, simply listing between the sender and recipient on the network, a traffic analysis can be performed. Moreover, spying on multiple parts of the Internet and using some statistical techniques, some attackers can track the communications patterns of many different organizations and individuals.

In the next paragraphs we will discuss more in details about Tor communications and the flaws of the model.

2.1 Tor Internals

The figure 1 shows how a message is cyphered before the communication begins. The communication source, before sending the message, choses a communication path of nodes which the keys are known to the sender. Then the source node is able to create a stack of encryption starting with the key of the last relay node and then continuing backwards with the keys of the other relay nodes in the chain. In this way every node in the communication path can decrypt the package and read the next hop address. After that the final node receives the message, he can send the response back to the originator of the data stream. In this phase the response message is encrypted sequentially by each node in the chain. With this method each relay node can gain access to the previous and the next node addresses only. Anyway the last node of the Onion Routing path, called exit node, send the message to the end point as plain text.

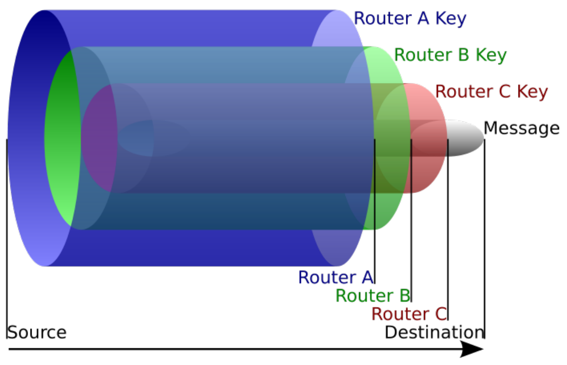


Figure 1: Message encryption layers.

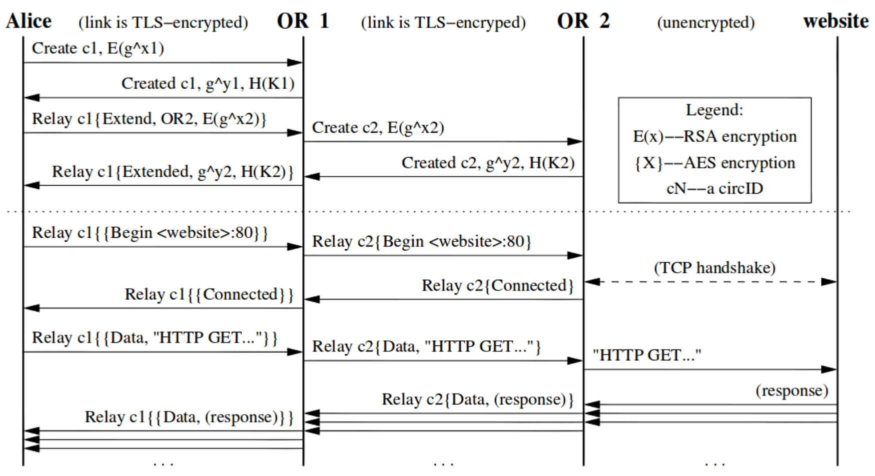


Figure 2: A two relay Onion Router communication.[5]

2.2 Tor attacks

Recent studies showed the existence of different kinds of possible attacks on the Tor network. Apart from Side channel attacks [1] we can classify these attacks in two major families: probabilistic attacks and path selection forcing attacks. The first ones are based on the data analysis that leak from the network. In these cases an attacker that sniff some specific data packets can perform some statistical analysis and may be aware of users identity informations. On the other hand, the path selection forcing attacks are based on some techniques able to route the network traffic through a sequence of nodes handled by the attacker. In our study case we focus on one of the probabilistic attacks: Timing Analysis Attack.

2.2.1 Timing Attack

As Tor development team states Tor is vulnerable to the timing attack analysis:

> *"Tor does not provide protection against end-to-end timing attacks[...]"*

More specifically, an attacker placing between the communication source and the Tor entry node and also between the Tor exit node and the communication destination, can observe the connection time of both end-points and point out their relation. The attacker can, for example, inject some malware code in the communication source machine or in some other entities, i.e. a router, on the network segment before the entry node. Similarly, a way to introduce some eavesdropping point into the communication destination could be found by the attacker.

[1]For example the well known *tor browser attack* .

We will focus on how much this kind of attack can be effective on a real network, considering the amount of the resources available to the attacker.

3 Simulation

In order to analyze the effectiveness of a time analysis attack in a realistic scenario, we decide to consider a simulation environment. We wanted to model the scenario in a way that most of the characteristics of the Tor network architecture would have been considered.

The **shadow simulator**[8] seemed to be a good candidate for our purpose. In the next paragraphs we will introduce shadow and we will illustrate our work based on this simulator.

3.1 Shadow

Most of the simulators used for networks experimentation do not provide the use of real external applications, which their behaviour are often simulated as well. This approach can be reasonable if the analysis is focused on the network layers below the application layer.

In our case we are interested in studying the characteristic of the Tor network which is based on the network application layer. Thus the Tor application is one of the core part of the Tor architecture as it implements the onion routing protocol that let the Tor nodes communicate each others.

It is evident that the Tor application has a significant role in the Tor architecture, thus we chose the shadow simulator as it permits the use of the real Tor applications in a simulated environment. Essentially it allows the execution of a set of local applications that communicates in a simulated computer network. Moreover we could avoid the oversimplification of the system that could have been occurred with a custom implementation.

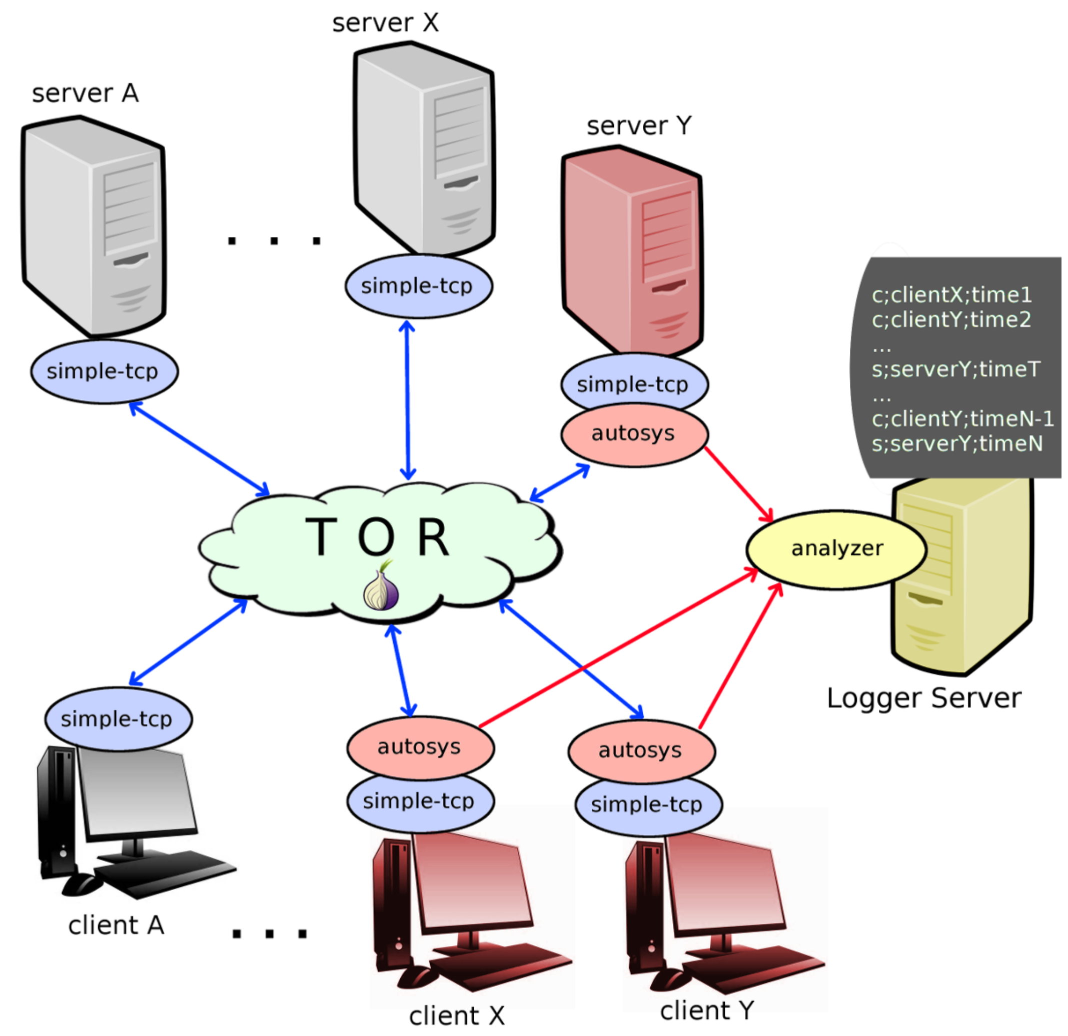
Applications are executed inside the shadow simulator through dynamic libraries called plug-ins. These plug-ins are interfaces used by shadow to trace a selective set of system functions and re-route them to the simulator instead of letting them proceed directly to the kernel. In this way, the simulator is transparent to the application that may function as like it was running in a standard UNIX environment.

The plug-in that allows the execution of the Tor applications is scallion. The latter provides also some useful tools for the virtual Tor network topology generation.

The virtual network is modeled with a XML configuration file, called blueprint, used by shadow to understand the network structure and some network properties such as link latency, jitter and packet loss rate. The blueprint also tells Shadow what software each virtual node should run at its creation. This is specified with a plug-ins list for each virtual node entry of the XML file.

We implemented three shadow plug-ins and their relative applications able to work alongside scallion in order to experiment a time analysis attack scenario. Our plug-ins and the attack scenario will be illustrated in the next paragraphs.

3.2 Simulation Scenario



```
c;clientX;time1
c;clientY;time2
...
s;serverY;timeT
...
c;clientY;timeN-1
s;serverY;timeN
```

Figure 3: Time analysis attack simulation scenario.

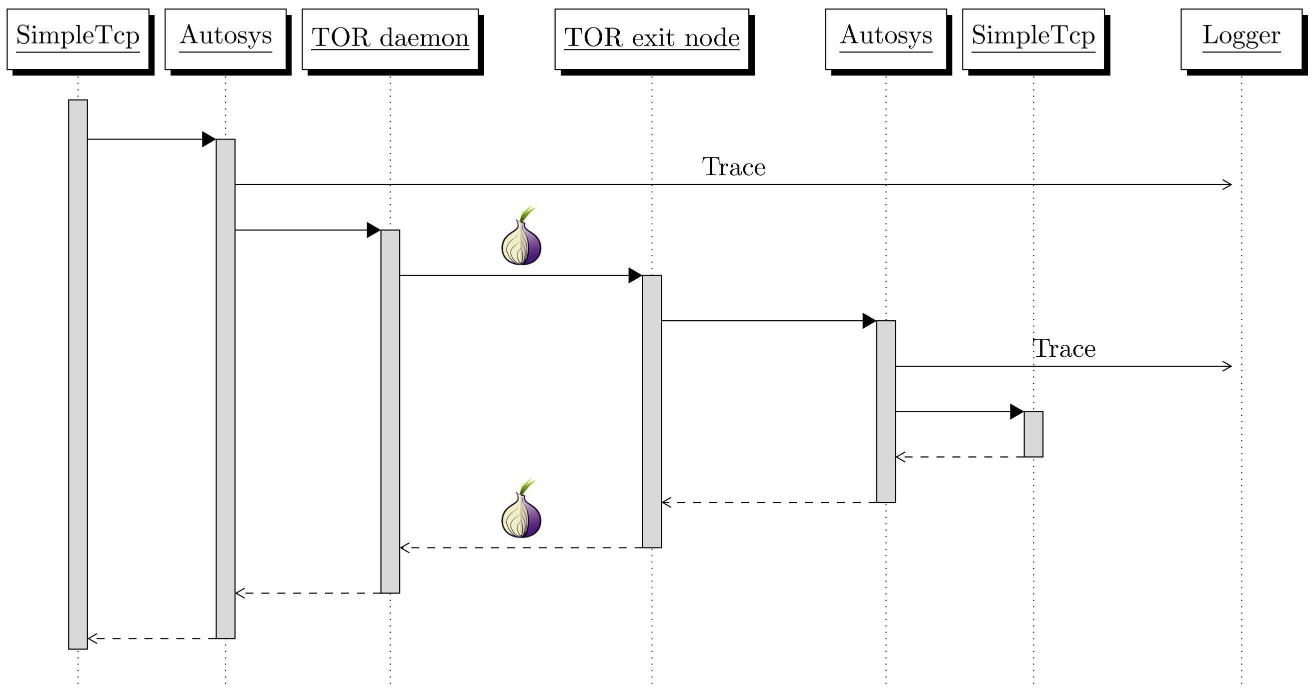


Figure 4: Summarization of the behaviour of the plug-ins system (only one connection of the simpleTcp plugin)

The figure 3 shows an outline of the simulation scenario. More specifically the considered identities of the scenario are:

- **Servers**, that represent usual application servers. They listen on some TCP port waiting for connection requests.
- **Clients**, that represent usual application clients. They try to initialize TCP connections with some servers through the destination port the target servers are listening to.
- **Tor nodes**, that form Tor communication paths. They compose the Tor network.
- **Logger server**, that is a simple server whose read and store incoming UDP status messages.

Furthermore, clients and servers communicate each others at the application layer through an ad-hoc application we built, **simple-tcp**. As the application name itself suggests, simple-tcp permits clients to establish simple TCP connections with servers. Also, simple-tcp is used by clients to send their own IP address to the servers in the TCP messages payload. In this way servers can log some information about the incoming connections: specifically they store source IP address, destination IP address and time of day related to them. This feature is not intended to be used by an attacker but we will see in the paragraph 3.5 how it is useful in the analysis process. Anyway, from the attacker point of view, simple-tcp tries to behave like an usual web-application, simulating a sequence of TCP connections interchange between clients and servers.

In our scenario all the TCP connections pass through the Tor nodes as we are not interested in the traffic outside the Tor network. This is legitimated considering our purpose of testing the feasibility of a time analysis attack in Tor and considering that the internet traffic coming from outside Tor could be filtered by the attacker's tracer software[2], avoiding in this way unwanted data interference.

As a simplification of the implementation we decided that during a simulation a client is able to request several TCP connections to a single server only. An extension of the application that allows a client to connects to more servers may be implemented in future. Anyway this limitation is for now negligible if we consider that a client usually requests a large amount of HTTP fragments and TCP connections to

[2]The Tor exit nodes list is public[3].

the same domain in a certain temporal window. Hopefully this assumption should suggest similar results considering client applications with multi-server requests but this is indeed a critical point to be tested in future works.

Autosys is a tracer application installed on both clients and servers which are interesting for an attacker. In short, when autosys discover an outgoing connection from the hosting client, it sends to a logger server the client IP address and the current time of day. Doing the same when it discovers incoming connections in the server side, autosys may let the attacker relate outgoing clients connections to incoming server connections. We will see in the next paragraph how this application can be used to simulate a time analysis attack.

3.3 Autosys plug-in

To analyze the tor network traffic we need, at least, some information about the incoming and the outgoing connections.
To achieve this result we have more than a solution: in the first one we can place in the last network element that is under the control of an **autonomous system**[3] a connection listener which traces a whole subset of network nodes; otherwise we can place the listener in a, malware-like, invisible proxy on the target machine with the aim to trace every outgoing connection. In both solutions we need to place some correspondent exit tracers on the target end points. This proxies can be implemented as a simple raw-socket based sniffer. Unfortunately Shadow doesn't support raw-socket emulation[4], so we implemented the plug-in as a simple event based TCP proxy. In our attack scenario an instance of the autosys plug-in is placed between the client application and the client Tor daemon, also a second instance is placed between the server Tor daemon and the server application.

Algorithm 1 Autosys main loop

```
 1: function AUTOSYS($sa$, $sb$, $analyzer$)
 2:     while $True$ do
 3:         if Some data can be read from $sa$ then
 4:             $indata \leftarrow$ readall($sa$)
 5:             writeall($sb$, $indata$)
 6:             if $indata$ was a connection initiator then
 7:                 send(< $type(sa)$;$hostname(sa)$;$gettod()$ >, analyzer)
 8:             end if
 9:         end if
10:         if Some data can be read from $sb$ then
11:             $indata \leftarrow$ readall($sb$)
12:             writeall($sa$, $indata$)
13:             if $indata$ was a connection initiator then
14:                 send(< $type(sa)$;$hostname(sa)$;$gettod()$ >, analyzer)
15:             end if
16:         end if
17:     end while
18: end function
```

As we can see in the algorithm 1, where *sa* and *sb* are the two end sockets and *analyzer* is the socket descriptor of the **analysis server** (the logger), when some connection initiator packets flow trough the proxy the current machine time of day is sent to the **analysis server** [5].

[3] That the reason why the plug-in is called "Autosys".
[4] If you launch a simulation with a raw socket based program the simulator inform you about the impossibility of the task.
[5] The simulator have a single clock, so we don't have any problem with the clock synchronization.

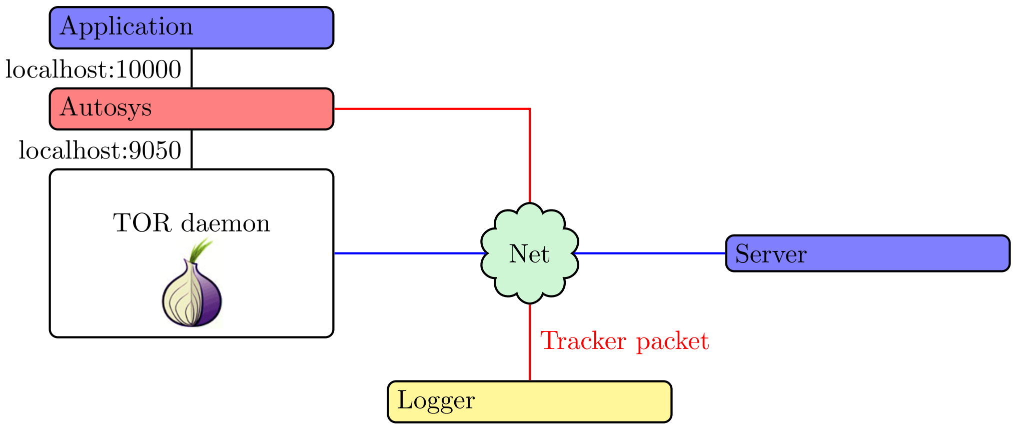


Figure 5: Autosys proxy structure.

3.4 Analyzer plug-in

Data sniffed by the **autosys** plug-in need to be stored for later analysis, to do so we need another plug-in. This plug-in is called the "Analyzer plug-in", "Analysis server" or simply "Logger" which will collect all the dead-drops generated by the sniffers. The plug-in is implemented as a simple event driven UDP server, every packet received from the sniffer plug-in is so saved in a single place.

```
host_type;hostname;timestamp
```

Figure 6: Analyzer log Structure

Referring to the figure 6 we have the following information in the log:

- Host Type
  c to signal that the traced packet referred to a client
  s to signal that the traced packet referred to a server
- Hostname
  The name of the tracked machine
- Timestamp
  The time-stamp of the packet (when it flows trough the autosys plug-in).

In conclusion this raw data will be passed to the phase two to be analyzed by the scripts.

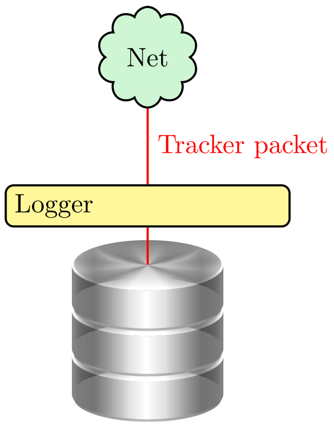


Figure 7: Logger logic structure.

As a subnote this plug-in was implemented as an UDP server for the sake of simplicity and, in opposite to TCP, the low impact on the network.
This can lend us to a little packet-loss but won't impact too much on the time analysis as we will see later on this study.

3.5 Simpletcp plug-in

The last plug-in that we implemented emulate a simple HTTP client/server communication, this plug-in have two operational modes: the client mode and the server mode.

3.5.1 Client mode

This plug-in, in client mode, emulate the communication making some (a customizable amount) connections to the server with a random uniform distribute sleep time in between of every connection (this parameter is configurable too).
The packet transfered by the plug-in simply contains the host-name of the host machine, this is to break the anonymity of the TOR system and analyze the results in later phases.

3.5.2 Server mode

At the opposite end, a server **simpletcp** instance, will:

- Receive the connection packets from the clients.
- Add a time-stamp to the current received packet.
- Save the packet to a common file (in our implementation this data file is called as the host-name of the machine hosting the server). This is used in the analysis of the phase 2, to compute a matching probability between the guessed users and the real users.

3.5.3 SOCKS5 operation

To communicate over TOR we implemented the **simpletcp** plug-in as a **SOCKS5** capable client.
The **SOCKS5** proxies works as described in the RFC1928[9], in a nutshell the **SOCKS5** is a transparent proxy design, it describe an establishment phase in which the proxy is informed about the other host, make the outgoing connection and, when some data is written on the incoming connection (from the client) the proxy will transmit this data trough the outgoing connection.

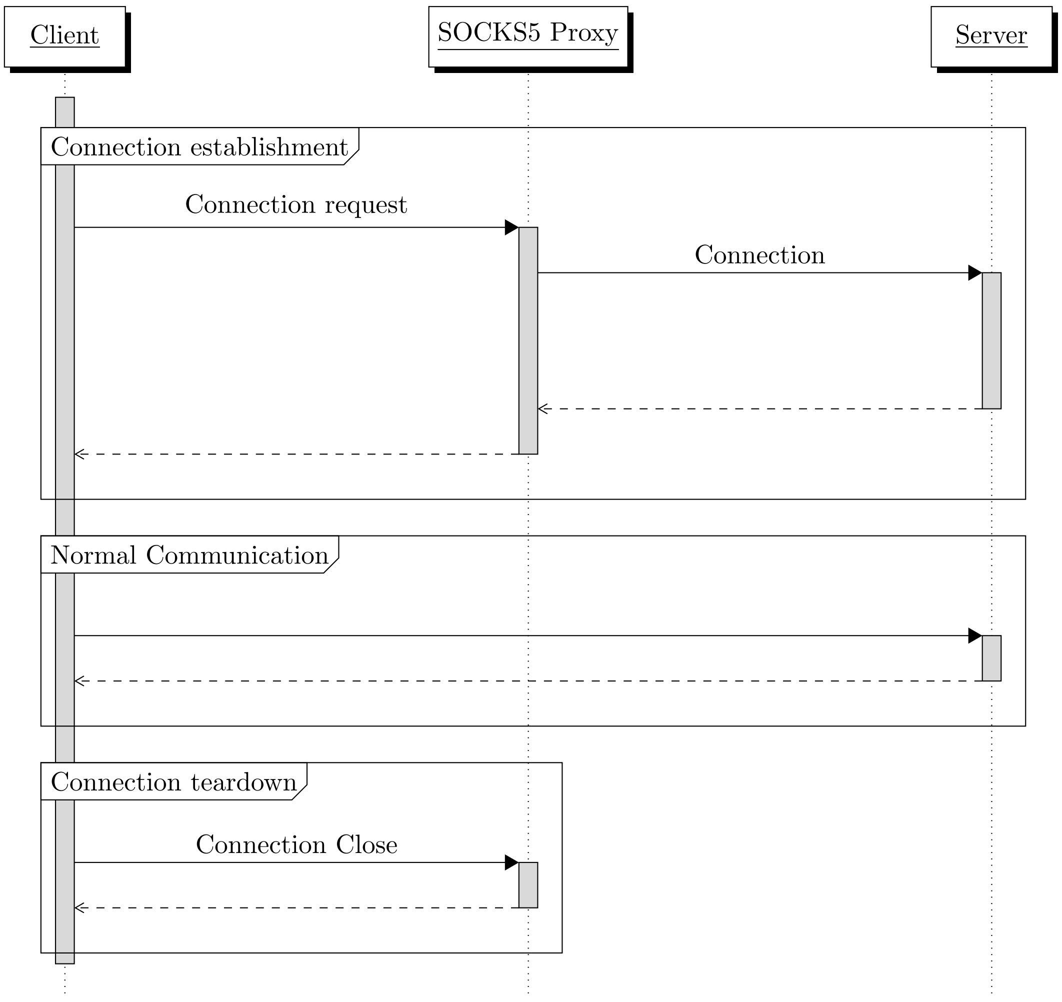


Figure 8: Socks5 Proxy behaviour

This proxy-like approach is required because the TOR interface is implemented as a **SOCKS5** proxy, when some packet flows trough the interface (which is listening by default on **localhost** on port 9050) the TOR daemon will transmit over the TOR network following the anonymity methods described in the previous chapters.

4 Data Analysis

Wrapping around the plug-in model of our system there are some auxiliary software (scripts) which follow and automate all the simulations, from the beginning (building up the simulation scenario) to the post-processing data analysis.

Thus, after a bunch of simulation runs handled as previously explained, we performed a data analysis in order to figure out the effectiveness of a time analysis attack.

In this section we will illustrate the software tools we used for handling the simulation runs and the software tools we used in the analysis phase. Eventually we will present the empirical results obtained.

4.1 Netbuilder and Launcher Scripts

As described in 3.1 The Shadow simulator use a XML file to describe the network, we automatize this process with a python script called the net-builder script.
This script will so build a scenario based on some input parameters as:

- The number of TOR relays in the simulation.

- The number of TOR exit nodes in the simulation.
- The number of TOR 4authorities[6] nodes in the simulation.
- The number of clients (simpletcp) of the simulation.
- The number of servers (simpletcp) of the simulation.
- The percentage of clients tracked by an autosys plug-in.
- The percentage of servers tracked by an autosys plug-in.
- The density of the network-requests.

When the others parameters are self-esplicative, the density parameter is somewhat that deserves more that two words on. This parameter specify the time interval between every connection made from the simpletcp plug-in to emulate the behaviour of different type of networks. We defined three behaviours to our network study:

- Slow:
  minimum: a mean of 800 milliseconds of sleep time between every connection,
  maximum: a mean of 2 seconds of sleep time between every connection.
- Average:
  minimum: a mean of 80 milliseconds of sleep time between every connection,
  maximum: a mean of 1 second of sleep time between every connection.
- Fast:
  minimum: a mean of 20 milliseconds of sleep time between every connection
  maximum: a mean of 100 milliseconds of sleep time between every connection.

This normal distributed values are computed for every simpletcp client (which will compute a random uniform distributed value as described in 3.5.1).

The second auxiliary script we used is a bash script that helped us in the automatic run of the simulations. This script will run a bunch of simulation using the algorithm 3.

Algorithm 2 Launcher script

```
for ($simulation\_run \leftarrow 1$; $simulation\_run <= steps$; $simulation\_run++$) do
    for ($sim\_id \leftarrow 1$; $sim\_id <= simulations\_per\_step$; $sim\_id++$) do
        for all $density$ in $(slow, fast, average)$ do
            if The client trace percentage is not fixed then
                $client\_trace\_value \leftarrow sim\_id/simulations\_per\_step$
            end if
            if The server trace percentage is not fixed then
                $server\_trace\_value \leftarrow sim\_id/simulations\_per\_step$
            end if
            if A configuration is present for $< sim\_id, density >$ And the percentages are fixed then
                Use the previous configuration
            else
                Generate a new configuration with net-builder
            end if
            Launch the Shadow Simulator with the appropriate configuration.
        end for
    end for
end for
```

[6]A 4 Authority node is simply the database that keep track of the state of the TOR network and the list of the TOR relays/exit-nodes

From this algorithm we can see that we can use this script to create a set of simulations with some linear modifications to the numbers of the traced clients or the number of the traced servers (or a combination of the two). We can also declare more than one run on this bunch of simulations, this can be useful to compute the mean values from the same configuration file, else we can generate a new configuration file starting from some input parameters.

After the launch we can pass the simulation raw data to the analyzer script.

4.2 Analyzer Script

The analyzer script takes a log file, generated previously by the logger server, as input and tries to ascertain which clients were communicating with which server during the simulation. The log file is a list of entries that are formatted as described in figure 6 and the figure 9 shows an example of it.

```
...
c;client10;1420000000
s;server7;1420008031
c;client6;1420005867
s;server9;1420146660
s;server6;1420205384
s;server8;1420252482
c;client0;1420680882
c;client1;1421017740
s;server7;1421023888
s;server2;1421156205
c;client8;1421160529
s;server3;1421318345
s;server0;1421332488
c;client7;1421487295
c;client4;1421634744
s;server9;1421726485
c;client2;1421827747
...
```

Figure 9: Analyzer log file fragment example.

The script scans the log file and for each line checks whatever the entry is related to a client or a server. If the entry refers to a client connection request it scans in a nestled loop the server connection entries that may be related to that client. The nested loop does not scan the whole file until its end, but it stops after a sufficient amount of read entries. In particular, the amount of entries to be read is measured in time: from a client entry, the scan goes ahead until the difference between the time-stamp of the current scanned entry and the time-stamp of the starting client entry is not greater than a fixed threshold thr_{max}. For our experiment we chose a limit of 6 seconds as it is an enough high latency value for a connection acceptance. Likely, the average latency for a connection response, even passing through the Tor network, is considerably smaller than 6 seconds but anyway we can over-esteem a bit this value without unexpected bad consequences. On the other hand, an under-esteem of this value would threaten the analysis results.

Furthermore, from a client entry the nestled scan starts reading from the entry that is 100 ms far from the client start point. This lower threshold, thr_{min}, is needed in order to filter those server entries that registered a connection time-stamp that results too young for being related to the client start point. This lower limit is more critical: a too low limit would guarantee false positive results and a too high limit would drop the right results. We will see later that after some tests, a value around 100ms seemed to maximize the clients servers relation matching.

The analysis proceeds by recording a set of possible server candidates per each client connection request. Each server candidate s assumes a $pmatch(creq, s)$ value which indicates how much a server candidate can be the real [7] server that accepted the *creq* client request. The *pmatch* value is defined as following:

$$pmatch(creq, s) = 1 - \frac{\Delta_t(creq, s) - thr_{min}}{thr_{max} - thr_{min}} \tag{1}$$

where $\Delta_t(creq, s)$ is the time distance between the entry time-stamp of the server s related to the entry time-stamp of the client connection request *creq*.

As instance let us consider the server candidates related to the first client entry, *client*10, of the example log fragment shown in figure 9: *server*7 is not considered at all in the *pmatch* calculation as Δ_t is lesser than thr_{min}; then the *pmatch* values for the first server candidates related to the client request c;client10;1420000000 are shown in figure 10:

```
candidate        pmatch
server9          0.992
server6          0.982
server8          0.975
server7          0.846
server2          0.823
server3          0.769
server0          0.794
...
```

Figure 10: *pmatch* values for server candidates related to *client*10

Clearly the *pmatch* probability is higher when the server connection is closer to thr_{min} meaning that the likelihood of a server candidate to be the real server for the related client decreases when the server connection acceptance attempted far from 100ms.

Another fact that may be considered in a time-analysis is that if a server accepts a connection request from a client after a certain time Δ_t, that server will likely accept again another connection from the same client after a time that is close to Δ_t if the Tor communication path is the same as before[8]. Thus, as the *pmatch* is defined as the Δ_t normalization, let us define the *gap* average related to a server s that is a candidate for a client c as follows:

$$gap_{AVG}(c, s) = \frac{\sum_{i=0}^{N(c,s)} |pmatch(creq_{i+1}, s) - pmatch(creq_i, s)|}{N(c, s)} \tag{2}$$

where $N(c, s)$ is the number of c connection requests that have been likely accepted from s.

At the end of the scan phase, each client collects a set of server candidates with their relative average *gap* and *score* that is defined as follows:

$$score(c, s) = \frac{\sum_{i=0}^{N(c,s)} pmatch(creq_i, s)}{gap_{AVG}(c, s) + 1} \tag{3}$$

After that every server candidate related to a certain client gained their own score, they are sorted, in a descending order, by the score itself. The first of the list is elected as the best candidate to be the right server that accepted the connection requests from its corresponding client.

Let us notice that a candidate score highly depends by the number of connection acceptances estimated for that candidate as by as the *pmatch* sum gained, meaning that more connection acceptances of a server are close to the client requests (plus the thr_{min}) more the score for that server is high. Also, the score is highly attenuated by the gap_{AVG} if the connection latencies often differs between each-others.

The analyzer script also collects information about the real connections logged by the simple-tcp applications. With this information the analyzer can tell us how much the estimated results are close to the real ones. Obviously, in a real time analysis attack the real connections information cannot be

[7] The term "real" should not be misunderstood with non-simulated but should be intended as non-estimated.

[8] If the connections are going to the same server, the Tor daemon could reuse the same old Tor communication path for a maximum than 10 minutes[1].

obtain by the client and server applications themselves but in a simulation environment, we can better figure out how a real time analysis attack may be effective.

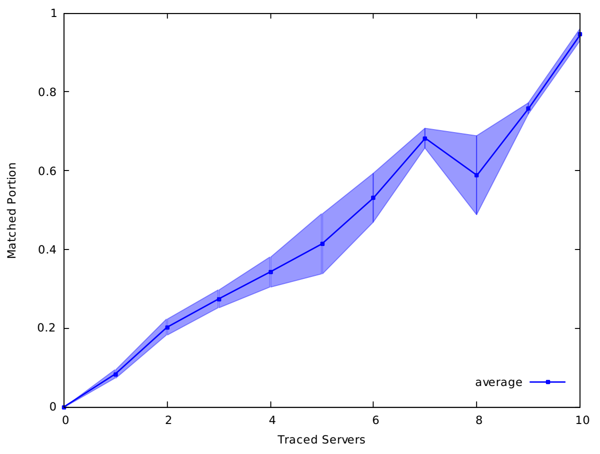
Moreover by a set of simulations that provide the use of the real connections logged data an attacker may be able to understand the matching accuracy of the analysis, how distinguish the matched servers among the others and also evaluate some parameters like thr_{min} and thr_{max} before experimenting the attack in a real scenario. With this in mind, the script calculates the *matched_accuracy*, a parameter that indicates how much the matching of the right server (those that correspond with the real data) is correct in terms of identified connections:

Algorithm 3 Matching accuracy calculation

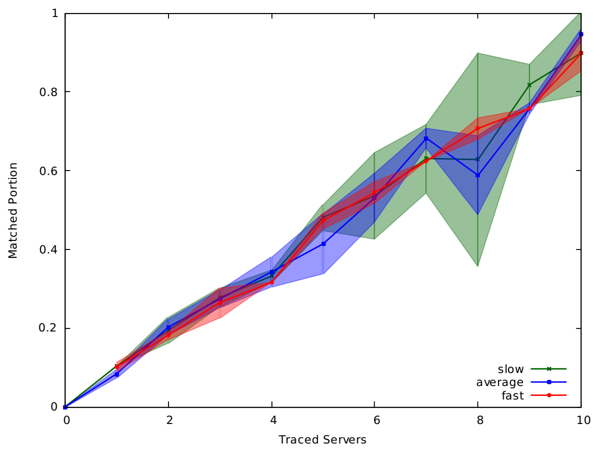
```
$matched \leftarrow 0$
for all $c$ in $traced\_clients$ do
    $accuracy_c \leftarrow null$
    $realsv \leftarrow GetRealServer(c)$                          ▷ The server which $c$ connected to
    $s \leftarrow GetBestServer(c_{SERVER\_CANDIDATES})$
    if $s = realsv$ then
        $N \leftarrow GetConnectionNumber(c, s)$                  ▷ The number of traced connections between $c$ and $s$
        $M \leftarrow GetConnectionNumber(c, realsv)$             ▷ The number of real connections between $c$ and $s$
        $accuracy_c \leftarrow \frac{MIN(M,N)}{MAX(M,N)}$
        break
    end if
    if $accuracy_c$ then
        $matched\_clients \leftarrow matched\_clients + 1$
        $matched\_accuracy \leftarrow matched\_accuracy + accuracy_c$
    end if
end for
$matched\_accuracy \leftarrow matched\_accuracy / matched\_clients$
$matched\_portion \leftarrow matched\_clients / traced\_clients$
```

The $accuracy_c$ contribution is calculated as the distance between the amount of the estimated matched connections and the amount of the real connections. The last but not least parameter is the *matched_portion* that indicates how many traced clients found their right server.

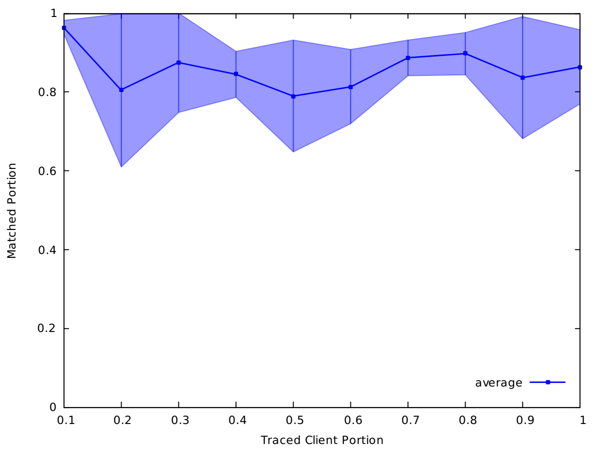
In the next paragraph we will look in details both the *matched_accuracy* and the *matched_portion* results obtained in different situations in order to understand how a time analysis attack can be effective.



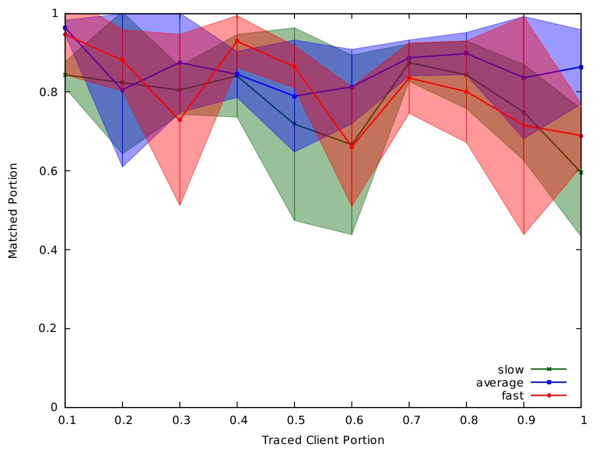
(a) Matched portion related to the traced servers number



(b) Same as frame *a* but with different densities



(c) Matched portion related to the traced clients number



(d) Same as frame *c* but with different densities

Figure 11: Matched portion charts

4.3 Empirical Results

In order to retrieve an interesting results set we decided to conduct 3 different simulation macro-bunches with the following characteristics:

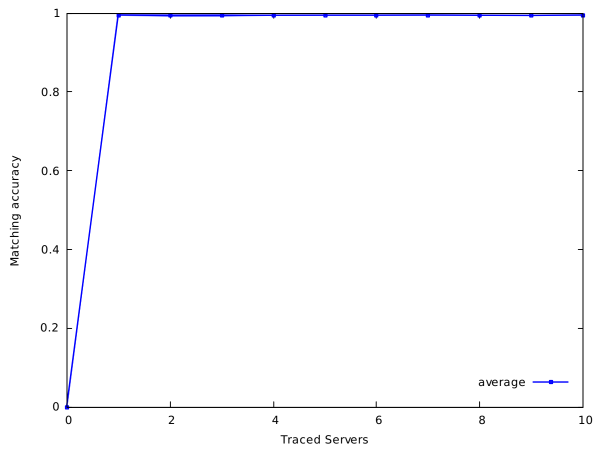
1. The number of traced clients was fixed to almost 100% of the network and the number of traced server was increasing by one to another simulation starting from 0 and ending to the 100% of the network.

2. The number of traced servers was fixed to almost 100% of the network and the number of traced clients was increasing by one to another simulation starting from 0 and ending to the 100% of the network.

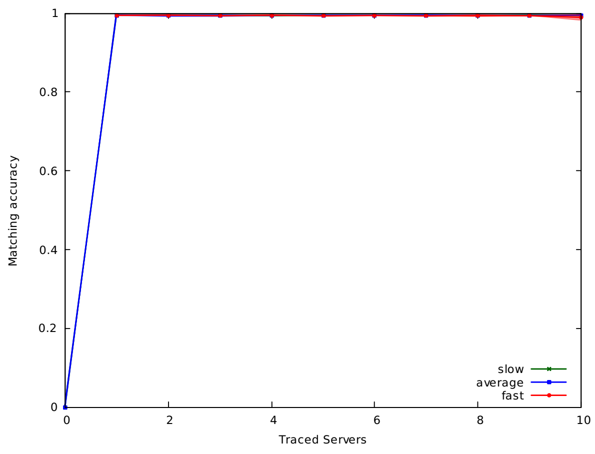
3. Both numbers of traced clients and traced servers (traced network portion) was increasing by one to another simulation starting from 0 and ending to the 100% of the network.

Each macro-bunch was executed 4 times in order to understand the results average and their errors. Also, for each of those simulation macro-bunches, 10 simulation sub-bunches were launched in order to provide a sufficient granularity in the traced entities escalation. Moreover for each sub-bunch 3 simulations were executed with 3 different values of the connections density: **slow**, **average**, **fast**.

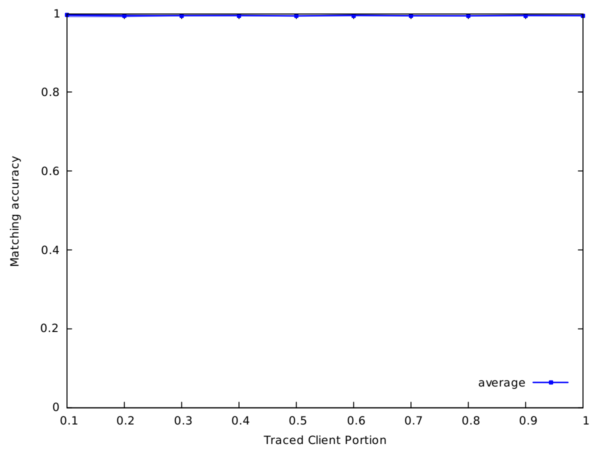
The figure set 11 shows some charts about the matched portion results. More specifically the figure 11a, shows the test with the fixed number of traced clients and the variable number of traced servers. As we can see the matched portion is almost linearly dependent by the number of traced servers. This



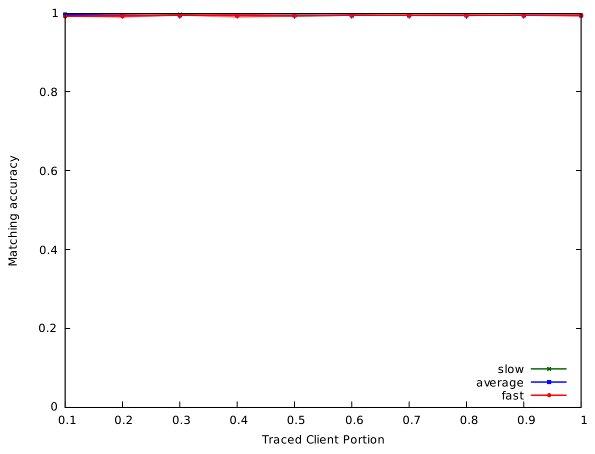
(a) Matched accuracy related to the traced servers number



(b) Same as frame a but with different densities



(c) Matched accuracy related to the traced clients number



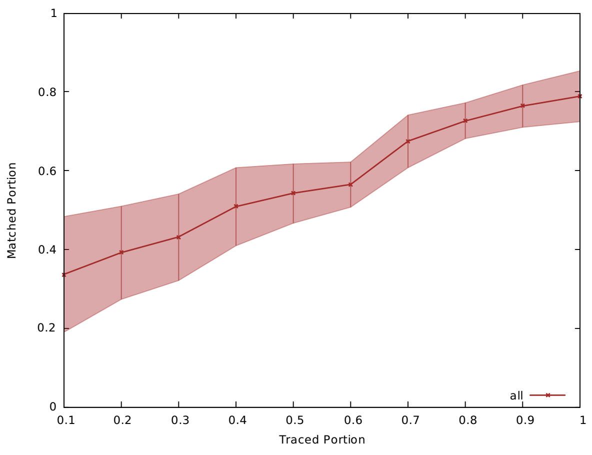
(d) Same as frame c but with different densities

Figure 12: Matched accuracy charts

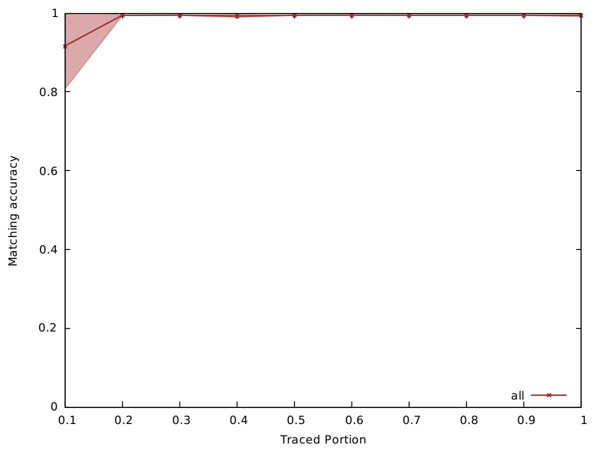
behavior was expected, in fact with a small amount of traced servers most of the clients would not find their right server as its connections are probably not logged. Also, it is interesting that increasing the connections density (11b) the function trend seems to be more precise. This may be given by the fact that the gap_{AVG} tends to be more accurate increasing the number of connections in the same temporal window, thus increasing the score of the right servers. Anyway it is comforting that while increasing the connections density, the best candidate estimation does not seem to be affected by the connections noise coming from the other server candidates. Figures 11c and 11d show the experiments with a fixed number of traced servers at 100% and a variable number of traced clients. We can see that the matched portion tends to be constant around 80% (as it was in figure 11a with a hight portion of traced servers) at the variation of the number of traced clients. This was also expected as the analysis is conducted from the clients side (it searches the right servers of the traced clients) and the matched portion is thus relative to the number of traced clients.

The charts shown in figure 12 shows the matching accuracy results in the same order of the matched portion experiment shown in figure 11. Clearly the matched accuracy is almost constant around its maximum value, meaning that the clients who correctly found their best candidates matched almost all the connections from those servers. Moreover the matching accuracy does not depend from any of the factors considered: traced clients number, traced servers number and connections density.

The charts in figure 14 show the results of the tests where both traced servers and traced clients were variable (variable traced portion). This represents the most realistic scenario and we can notice that

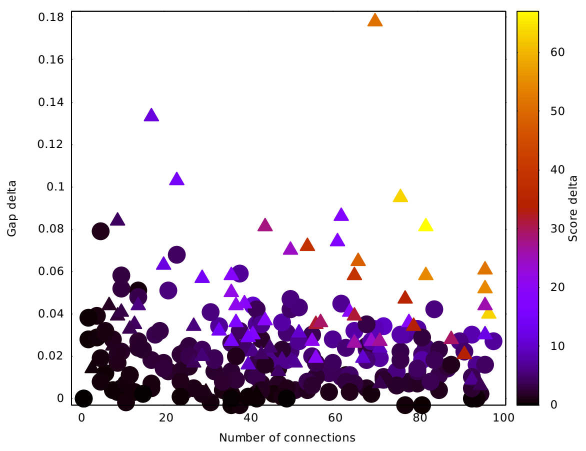


(a) Matched portion related to the traced portion

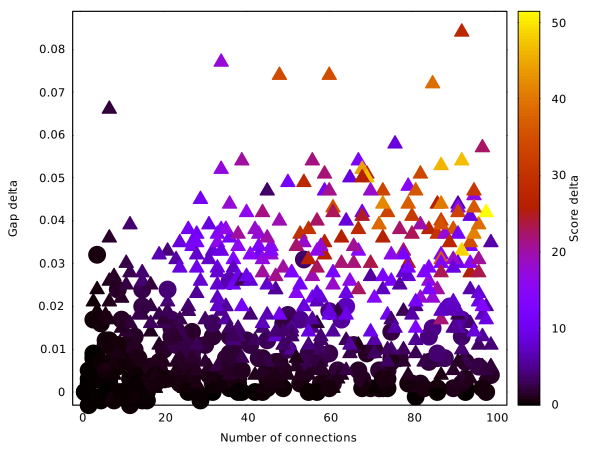


(b) Matched accuracy related to the traced portion

Figure 13: Charts about matched portion and matched accuracy at the variation of both traced clients and traced servers



(a) Trusted matching chart with 40% of traced portion



(b) Trusted matching chart with 90% of traced portion

Figure 14: Charts about trusted matching parameter

the matched portion trend seems to respect the sum of the trend results of the other two experiments (figure 11). We can see that a time-analysis attacker should be interested to trace as much Tor network node as possibile as the matched portion almost linearly scales with the traced portion. As istance if an attacker is able to trace an half of the Tor network nodes he may understand the relations between the clients and servers of around a quarter of the Tor network. Moreover, the matching accuracy still remains almost costant around 1, meaning that of that quarter of the Tor network the attacker may be able to correctly understand the end-points of almost all the connections.

Until now we have not faced yet the problem about how an attacker can distinguish the clients that found a right matching among the others. We based our matching portion evaluation comparing all the results with the real connections logged by the simulations. This is obviously not possibile in a real scenario thus we tried to understand if an attacker may distinguish the right matchings basing on some parameters.

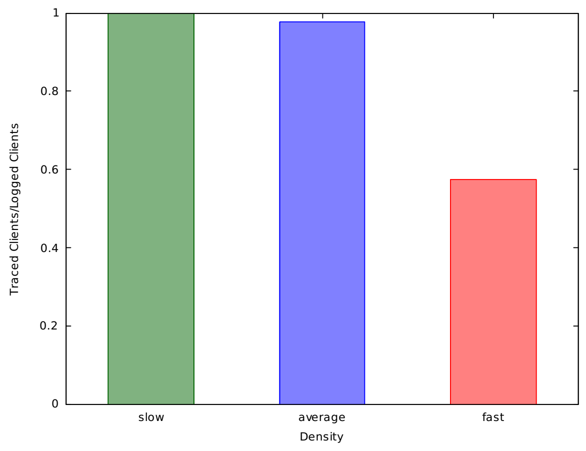


Figure 15: Communication density related to the logging capacity

TODO BLABLABLA SU DENSITY

5 Future Works

As we stated in the previous sections, this study only cover a little part of the big field of the timing-analysis attacks[9] so this work haven't covered some interesting points and study that can refine or give some more efficient countermeasures to the problem.

One of the interesting future works is the test and a benchmark of the so-called *Astoria*[10] Tor client. That client rethink the path-selection algorithm of the standard tor clients to avoid the kind of attacks we described here, that algorithm operates on an upper level that choose the routers minimizing the risk that two nodes falls under the realm of the same autonomous system.

Another future work is the comparison of the impact of the attack described above between the Tor network and the other anonymous network implementations like i2p or freenet.

A really intresting work is the study of the, old but currently really inspirative, Mix Network model[4]: the spiritual predecessor of the onion routing model, this network model is immune to this kind of attack, because the packets will be transmitted and mixed in random ways, this can be a starting point to make an efficient non-low latency onion routing network. Or, hopefully, a powerful mix of the two implementations.

An implementation work related to our model can be the modification of the simpletcp plug-in to make it capable of connecting and download from multiple servers in a single instance, and the changes required to the analysis method/algorithm to trace that kinds of connections.

As a side note, this experiments can be repeated with a bigger tor network, but, as what transpire from the paper*AS-awareness in Tor path selection*[6] this problem is not related to the proportion of the network.

6 Conclusions

References

[1] *Tor Manual.* `https://www.torproject.org/docs/tor-manual-dev.html.en`.

[9]Or, more in general, side channel attacks; There is a specific field in cryptanalysis that study this kind of attacks.

[2] *Tor: Overview.* `https://www.torproject.org/about/overview.html.en`.

[3] *TorStatus - Tor Network Status.* `https://torstatus.blutmagie.de`.

[4] David L Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 24(2):84–90, 1981.

[5] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. Technical report, DTIC Document, 2004.

[6] Matthew Edman and Paul Syverson. As-awareness in tor path selection. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 380–389. ACM, 2009.

[7] David Goldschlag, Michael Reed, and Paul Syverson. Onion routing. *Communications of the ACM*, 42(2):39–41, 1999.

[8] Rob Jansen and Nicholas Hopper. Shadow: Running tor in a box for accurate and efficient experimentation. In *Proceedings of the 19th Symposium on Network and Distributed System Security (NDSS)*. Internet Society, February 2012.

[9] Marcus Leech, Matt Ganis, Y Lee, Ron Kuris, David Koblas, and L Jones. Rfc 1928: Socks protocol version 5. Technical report, RFC, IETF, March, 1996.

[10] Oleksii Starov, Rishab Nithyanand, Adva Zair, Phillipa Gill, and Michael Schapira. Measuring and mitigating as-level adversaries against tor. *arXiv preprint arXiv:1505.05173*, 2015.