BADPASS: Bots taking ADvantage of Proxy AS a Service

Elisa Chiapponi[⋆], Marc Dacier[†], Olivier Thonnard[‡], Mohamed Fangar[‡], and Vincent Rigal[‡]

[⋆] EURECOM, Biot, France
elisa.chiapponi@eurecom.fr
[†] RC3, CEMSE - KAUST, Thuwal, Kingdom of Saudi Arabia
marc.dacier@kaust.edu.sa
[‡] Amadeus IT Group, Sophia Antipolis, France
{olivier.thonnard,mohamed.fangar,vincent.rigal}@amadeus.com

Abstract. Web scraping bots are now using so-called Residential IP Proxy (RESIP) services to defeat state-of-the-art commercial bot countermeasures. RESIP providers promise their customers to give them access to tens of millions of residential IP addresses, which belong to legitimate users. They dramatically complicate the task of the existing anti-bot solutions and give the upper hand to the malicious actors. New specific detection methods are needed to identify and stop scrapers from taking advantage of these parties. This work, thanks to a 4 months-long experiment, validates the feasibility, soundness, and practicality of a detection method based on network measurements. This technique enables contacted servers to identify whether an incoming request comes directly from a client device or if it has been proxied through another device.

Keywords: Web Scraping, Residential IP Proxy, RESIP, Round Trip Time measurement, TLS, Security, Bots

1 Introduction

Nowadays, websites in different domains, such as e-commerce, ticketing, and social media, are engaged in a persistent fight against subtle but damaging actors: scraping bots. They produce a significant amount of traffic towards these websites producing large financial losses, as explained in recent works [23,16].

Lately, scrapers behind these bots have started to take advantage of IP addresses from Residential IP Proxy (RESIP) providers, as explained in [18]. RESIP providers announce to have access to tens of millions of devices belonging to real users. These device IPs[1] can be used as exit points of requests.

This situation is problematic for scraped websites because these IPs addresses belong to legitimate users. Online companies are reluctant to block traffic from IP addresses that could come from potential customers. As a result, more and more scrapers can perpetuate their activities without being stopped.

[1] In the rest of the paper, we will use IP and IP address interchangeably.

To limit the actions of scrapers taking advantage of RESIP services, it is important to find new techniques to distinguish whether or not a connection is proxied through a device or comes directly from it. Very recently, some attempts were made to leverage the behavioral differences between the two types of connections by using machine learning [37]. Differently from it, we propose a RESIP detection method based on the evaluation of the Round Trip Time (RTT) difference of the TLS and TCP packets exchanges.

Our intuition is based on a difference at the transport layer among direct and proxied connections. In the second case, two distinct TCP sessions are built (scraper-RESIP and RESIP-target server). In a direct connection, only one TCP session is created (scraper-server). On the other hand, only one TLS session is put in place in both scenarios. In proxied connections, TLS packets are just forwarded by the RESIP and TLS is end-to-end between scraper and server, as in direct connections.

This difference is reflected on the RTT at TCP and TLS layers. The RTT gives us information on the distance[2] between the sender and the receiver. In case of a proxied connection, the sender of the TCP packets differs from the sender of the TLS ones, creating discrepancies between the two RTT values. In a direct connection, the two values are similar. We exploit this aspect to design our detection method.

Our idea, simple and straightforward in theory, could not work in practice if the typical variations of the TCP RTT were of the same order of magnitude of the differences between the TLS RTT and the TCP RTT. This is why we have run a long and thorough experiment to assess the feasibility of the approach. In this work, we show the successful results we have obtained.

The contributions of this paper are twofold:

- We propose a novel server-side proxy detection technique based on the evaluation of the RTT difference between TCP and TLS RTT values.
- We provide experimental results demonstrating the feasibility, reliability, and practicality of this solution, based on the results of a 4 months-long measurement campaign.

Our paper is structured as follows. Section 2 illustrates the state of the art regarding scraping bots and RESIP services as well as RTT based proxy detection methods. Section 3 describes the experimental infrastructure, the detection method and the details of the campaign. Section 4 provides obtained results while Section 5 discusses the stability and practicality of the method. Section 6 concludes the paper and offers thoughts for future work.

[2] To be exact, the RTT is a measure of time, from which we can infer an approximation of the distance [26,20,32].

2 State of the art

2.1 Scraping bots exploiting RESIP services

Existing anti-bot solutions leverage several different techniques to detect bots [13,14]. In the past years, RESIP companies have emerged and scrapers have started using them for their activities. In [27], Li et al. show that more than half of the bot IP addresses they collected with their honeypots belong to residential networks. As explained in [16], current detection techniques struggle in blocking bots using such IPs.

In 2020, Mi et al. proposed the first comprehensive study of RESIP services [28]. They created an infiltration framework to study 5 RESIP providers and their IPs. Our infrastructure is similar to theirs, but it is used differently: we take network measurements of the connections and we use them to validate a new detection method.

The dataset provided by Mi et al. has been used for subsequent studies [21,17], focused on geo-localization and reputation of IP addresses. Yang et al [39], recently investigated Chinese RESIP services and their IP addresses. Chiapponi et al [15] performed a mathematical analysis of IP addresses hypothesized to belong to RESIP providers, examining the repetitions of IP addresses. Study and detection of software used in devices to enroll them as RESIP GATEWAYs have been investigated in recent publications [29,34].

Very recently, a blog post from the anti-bot company DataDome [37], proposed a new ML based approach to identify RESIP connections. They claim that RESIP IPs exhibit a different behavior compared to residential IPs used only by humans, even when the IPs are shared by the two categories. Thanks to an infrastructure similar to ours, they collect RESIP IPs. Every time they detect one of these IPs sending them requests, the ML model checks if the request exhibits RESIP behavior. This approach, differently from ours based on network measurements, is intriguing and might prove to be effective. However, their blog post only offers a high-level overview of the behavioral features used. Furthermore accuracy, false positive and false negative values are not provided.

2.2 Proxy detection based on RTT

Our detection method relies on the comparison of the TCP and TLS RTT, measured between packets that are exchanged within a TLS session. To the best of our knowledge, we are the first ones to have implemented this technique and to have conducted a thorough measurement campaign to assess its feasibility.

Using some form of RTT measurement for proxy detection has been proposed before, though. For instance, Hoogstraaten [22] suggests that comparing the RTT of the application layer and the transport layer could potentially indicate the presence of a proxy. He advises to calculate the application RTT by retrieving consecutive elements from the server (e.g. HTML page and associate image) using HTTP. As far as we know, this technique was not implemented and this approach is different from the one we propose. Indeed, this method requires changes in the application code and only works if some specific assumptions hold (no caching in the proxy, no parallel requests to retrieve both objects, etc.). Our technique, on the other hand, does not require any modification of the original server code because it leverages the exchanges that normally take place in a TLS connection.

In another blog post [24], the author suggests a proxy detection based only on the measurement of the TLS RTT (ignoring the TCP RTT) which could, possibly, work thanks to an implementation issue in chromium-based browsers. His technique takes advantage of JavaScript code running at the client-side. The code queries 5 times both 127.0.0.1 and 0.0.0.0 with HTTPS. In the case of direct connections, the RTT of the two connections are comparable. When the connection is proxied, there should be a relevant difference between the two measurements. This technique is different from ours for various reasons. Our detection method does not require any code running at the client-side, is independent from the client application or operating system, and is solely based on measurements obtained at the server-side. No additional URL needs to be queried and the comparison of RTT values is performed between the TCP and TLS RTTs of a single connection.

Other works leverage similar approaches. In [38], a RTT at the application layer is calculated by fetching an HTTP object that cannot be cached. A patent [35] performs the detection with the comparison of the RTTs obtained when fetching a cached and a non cached object. Our approach, in contrast to the described ones, works completely at the server-side, does not require fetching any object and uses the more stable TCP and TLS RTTs as opposed to the ones at the application layer.

In summary, previous works have considered using some form of RTT measurement to detect proxied connections. However, they all require either modification of the server code and/or some JavaScript to be executed on the client-side. Our proposal, in contrast, is exclusively based on passive measurement made at the server-side, which does not need to be modified in any way. Furthermore, contrary to our work, none of the previous works compares the TCP RTT to the TLS one.

3 Setup and methodology

3.1 Infrastructure

The goal of our experiments is to validate a detection technique for scrapers using RESIP services. To achieve this result, we have created an infrastructure that reproduces the real-world conditions scrapers experience when using such services. RESIP infrastructures are merely instruments in the hands of scrapers. Scrapers cannot access their internal parts and change their functioning, they can just rent the provided service using a well-defined API. Similarly, RESIP providers take advantage of real people devices that they cannot access directly. Thus, they cannot alter their hardware and they can only use application-level features. These constraints create a fixed environment in which scrapers have to

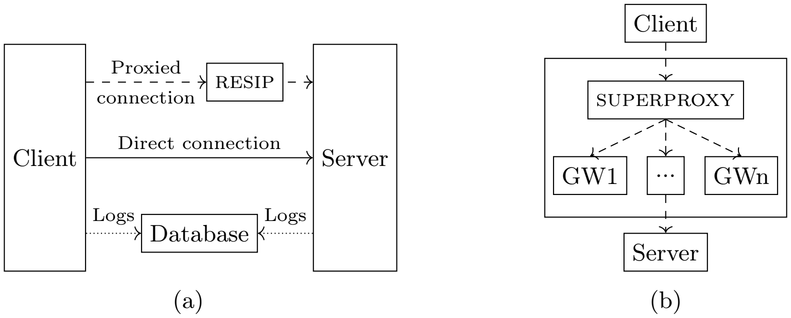


Fig. 1: Infrastructure schema (a) and internal structure of RESIP (b)

send their requests. Our infrastructure reproduces these conditions and enables us to perform network measurements that are representative of those observed in real-world scraping traffic carried out through RESIP proxies.

As shown in Fig. 1a, our infrastructure includes: a client sending requests to a target server, a target server, a RESIP provider, and a database.

The client, which represents the scraper, sends requests either directly to the server (Direct connection) or through a RESIP provider (Proxied connection) of which we purchased the services. On the other side of the connection, the server analyses each received query at the application and network layers. The client and the server locally produce logs and send them to a database where they are aggregated and processed.

22 machines and 4 RESIP providers constitute the core of the infrastructure. Section 3.2 discusses the examined RESIP services. Configuration, location, and roles of our clients and servers are explained in Section 3.3. Section 3.4 and Section 3.5 outline the performed network measurements and the detection technique. Section 3.6 describes the timeline and data storage of the experiment.

3.2 RESIP providers

Thanks to the information provided by analysts and companies working against scraping bots as well as online blogs devoted to web scraping activities, we have identified 4 RESIP providers widely used by scrapers: Bright Data [3], Oxylabs [6], Proxyrack [8], and Smartproxy [10].

The four services offer different packages and options. We subscribed to each of them to have 40GB (Bright Data) and 50GB (the other providers) of (incoming+outgoing) traffic per month proxied through residential IP addresses.

The details about the internal implementation of RESIP services are not known. From the information available on their websites, the four providers appear to have a similar architecture. As displayed in Fig. 1b, the client sends the HTTP/HTTPS request to the SUPERPROXY, through an HTTP CONNECT. The SUPERPROXY forwards the request to one of the residential GATEWAYS (GW1,..., GWn). It is not known if there are other machines in between these two parties. The chosen GATEWAY sends the request to the server. The server receives a request with the IP address of the GATEWAY as source address.

In the case of HTTPS connections, RESIP services are supposed to perform SSL tunneling: they should not decrypt and re-encrypt the packets they receive. They act as a circuit GATEWAY by forwarding packets back and forth between 2 distinct TCP sessions while changing IP addresses.

We have experimentally confirmed that this is, indeed, their behavior. To do so, we have established a connection through the 4 RESIP services to one of our servers configured to never send any ACK packet. The three-way handshake between our client and SUPERPROXY always completes successfully, whereas the one between GATEWAY and server does not. This confirms that two distinct TCP connections are created. Then, we have checked if the request sent by the client and the one received by the server have the same encryption and that the Session ID is the same in the "ClientHello" and "ServerHello" messages of the TLS protocol. This check has given us a positive result, confirming that the session is not decrypted and re-encrypted by the proxy.

3.3 Clients and servers

Each of our machines plays both the roles of client and server described in Fig. 1a. In this way, we maximize the number of different client-server paths covered in our experiment. To avoid any possible geographical or vendor bias, our clients and servers are spread all over the world and are rented from two different suppliers: we use 16 machines from Amazon Lightsail and 6 from Azure. We host two machines in each of the following locations: India, Australia, Japan, Germany, Ireland, Canada, USA (Virginia and Oregon), South Africa, United Arab Emirates, and Brazil. The last three locations correspond to the machines acquired from Azure.

We have implemented both client and server algorithms in Python3. The server has been built thanks to the ThreadingHTTPServer and BaseHTTPRequestHandle of the library http.server[4]. We have modified the source code to insert a timeout for connections not completing the TCP handshake. For the client, the library urllib[11] is used both to perform direct and proxied connections.

The client algorithm consists of an infinite loop. According to the speed set in the configuration file, queries are sent to each server in the experiment. Each machine is queried five times, with one direct connection and four proxied ones, one per provider. The query is performed with an HTTPS GET.

To communicate the IP address of the client to the server as well as information on the RESIP provider used, we encode it into the requested URL. We assign a unique numeric code to each machine, from 01 to 22. We also assign a numeric code to each RESIP provider, from 1 to 4. Direct connections have code 0. Every time we need to perform a request, we obtain a random sequence of 5 digits. We XOR this random sequence with the concatenation of client, proxy, and server codes. The final URL is the concatenation of the XORED and the random values.

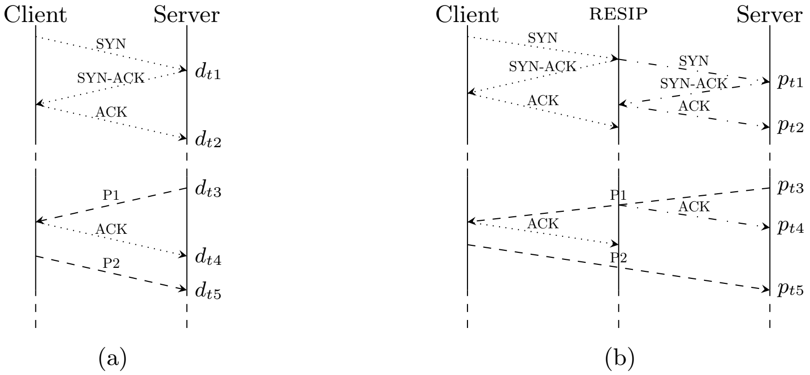


Fig. 2: TCP and TLS packet exchanges used in the detection technique for a) direct and b) proxied connections.

The server algorithm keeps listening for new incoming HTTPS connections on port 443. At launch time, an option can be used to specify if the server uses TLS1.2 or TLS1.3. Every time a new HTTPS GET request arrives, the URL is studied to check if it is part of the experiment (as opposed to, eg. a scanner or a crawler) and to retrieve the client, proxy, and server codes. Requests not passing the check are answered with an error page. Otherwise, a simple page is delivered.

3.4 Network measurements

In each machine, a sniffer is put in place to collect network measurements. The sniffer parses each incoming packet to port 443. It is implemented in Python3, thanks to the library PyShark [25]. It is restarted every hour for the stability of the code and to avoid increasing memory consumption. Information about packets is saved in a structure representing the corresponding stream.

For each incoming stream, we use the RTT to measure how far from each other[3] the parties taking part in the communication are. The explanation of these measurements is illustrated by the schema in Fig. 2. Fig. 2a shows the packets in the TCP (on top) and the TLS (on the bottom) exchanges that we use for detection in case of direct connection. Dotted lines represent TCP packets, dashed lines the TLS ones. Fig. 2b presents the same exchanges in the case of proxied connection. Dotted lines represent TCP packets between the client and RESIP, dash-dotted lines stand for TCP packets between RESIP and server, and dashed lines show the TLS connections. d_{tx} and p_{tx} represent the timestamp measurements of the sending/arrival of a packet at the server for direct (d_{tx}) and proxied (p_{tx}) connections.

[3] For sake of concision, we take the liberty of using the expression "measure of distance" instead of "approximation of the measure of distance" when referring to the RTT in the rest of the paper.

The first measurement we take is the TCP RTT. This value is the RTT between the SYN-ACK packet sent by the server and the corresponding received ACK. In case of direct connection, the TCP connection is created directly between client and server. Thus, the TCP RTT (d_{t2}-d_{t1} in Fig. 2a) is a measure of the distance between these two parties.

By contrast, in a proxied connection, two distinct TCP connections are created (Section 3.2). One connection takes place between client and SUPERPROXY, and one between GATEWAY and server. In this scenario, the TCP RTT (p_{t2}-p_{t1} in Fig. 2b) represents the distance between GATEWAY and server.

Network delays can increase the RTT value of the first TCP exchange, both in direct and proxied connections. In this case, the measured TCP RTT is the sum of the real RTT of the exchange plus the delay. To understand how this delay can influence our analysis, we collect the RTT for all packets sent by the server. We calculate statistics of these values that we use in Section 4 to discuss the variability of the TCP RTT within a given connection.

Secondly, we compute the TLS RTT. This value corresponds to the RTT of the TLS layer. The TLS protocol is end-to-end between client and server both in case of direct and proxied connection. Thus, this metric should give us the measure of the distance between client and server in all scenarios.

To obtain the TLS RTT, we consider two packets, P1 and P2. P1 contains a server TLS record after which the server does not send any other TLS records before receiving a specific TLS answer, as per the protocol. P2 is the packet containing the client TLS record that allows the continuation of the protocol. Any couple of TLS packets that satisfy these conditions can be considered.

As explained in Section 3.1, the RESIP architecture can be considered fixed for our scenario. Thus, the only variables that can influence the choice of P1 and P2 are the server and client implementations. Our detection method is server-side and we assume anyone recreating this experiment will have full access and knowledge of the server implementation. In the TLS connection, the server decides which information is needed for the client to complete the connection e.g. accepting the cipher proposed by the client or deciding if client authentication is required. In such conditions, we expect to be able to anticipate all the possible exchanges between client and server to find a couple of packets and cover possible corner cases. For these reasons, having full control over the client in our experiment, contrary to the real-world case, does not constitute a bias.

In our setup, we have a generic HTTP server, which does not require any client authentication and accepts common encryption ciphers. We expect this to be a common scenario for scraped websites that need to be accessed by the largest possible number of clients. In these conditions, we can identify P1 and P2 among the first TLS packets. This is an added value because it enables us to perform detection before any application content is delivered to the client.

Hereafter, we focus on the TLS records we use to perform our measurement. We refer to [31] for an accurate and detailed description of TLS.

Depending on the version of TLS, we use different TLS records to identify P1 and P2. For TLS1.2, the RFC 5246 [19] states that, after sending the "ServerHelloDone" TLS message, the server waits for a client response. We recognize P1 as the packet containing the TLS record encapsulating this message.

After the "ServerHelloDone" message, the client needs to continue the communication. The first TLS message sent by the client must be the "ClientKeyExchange", according to the RFC [19]. We identify P2 as the packet whose TLS record contains this TLS message.

In TLS1.3, if the server agrees on the cipher chosen by the client, it sends the "ServerHello" TLS message. Since the server has already obtained the client-side information for encryption, subsequent data in the message is encrypted. We choose P1 as the packet whose TLS record contains the last encrypted server data sent after the "ServerHello" message.

As explained in Appendix D of RFC 8446 [33], TLS1.3 implementations include a dummy "change_cipher_spec" TLS record to guarantee backward compatibility for middleboxes. This record is sent by the client before its encrypted handshake flight if the client does not offer early data and it does not send a second "ClientHello" message. In our implementation, the server imposes these conditions and thus "change_cipher_spec" TLS record is the first client TLS record sent upon reception of the "ServerHello" TLS message. We identify P2 as the packet containing this record.

We define the TLS RTT as the difference between the sending of P1 and the arrival of P2 at the server. As shown in Fig, 2, this corresponds to the difference d_{t5}-d_{t3} in the case of direct connection and p_{t5}-p_{t3} in the case of proxied one[4].

3.5 The detection method

In this section, we present our approach for detecting connections passing through RESIP services. The method is based on the study of RTT measurements.

As described in Section 3.4, for each incoming connection, the server collects the TCP RTT and the TLS RTT. As previously discussed, these two metrics give us measures of distances. TCP RTT informs about the distance between client and server, for direct connections. In the case of a proxied connection, this value represents the distance between a RESIP GATEWAY and a server. TLS RTT represents the distance between a client and a server in both scenarios.

Our intuition is that the TLS RTT is similar to the TCP RTT in the case of a direct connection. On the contrary, we expect a difference between the two values for a proxied one. When a proxy is used, the TLS RTT represents the sum of the distances between the parties plus the increased distance imposed by the traversal of the RESIP infrastructure, starting from a, possibly, far away gateway.

[4] The reader may wonder why we are not using the arrival of the ACK packet of P1 as the second point of measurement (d_{t4} and d_{t4}, in Fig. 2). In case of proxied connection, there are two distinct TCP connections (client-SUPERPROXY and GATEWAY-server, as explained in Section 3.2). The ACK packet is created by the kernel and it is sent at the arrival of P1 at the GATEWAY, without synchronization with the client-SUPERPROXY TCP connection. Hence, the difference p_{t5}-p_{t4} represents the distance between GATEWAY and server and not the one between client and server.

Indeed, in the RESIP setup, the connection passes through, at least[5], two more points (SUPERPROXY and GATEWAY), before arriving at the destination. In this scenario, the total distance is the sum of the distances client-SUPERPROXY, SUPERPROXY-GATEWAY, and GATEWAY-server. Depending on the geo-localization of the client, SUPERPROXY, GATEWAY, and server, packets could take a much longer route to arrive at the destination, with respect to a direct connection.

We define RTT *difference* as the difference between the TLS RTT and the TCP RTT. If this value is systematically, constantly, and significantly higher for proxied connections than direct ones, it offers to the server a mean to know if an incoming connection comes through such proxy or not, and to act accordingly if deemed appropriate.

3.6 Timeline and data storage

The experiment was started at 15:00 UTC +0 on 12/01/2022. In this work, we will study only the connections performed till 01/05/2022 at 15:00 UTC +0. Thus, the total number of examined days is 110.

Every day at 00:00 UTC +0, each server is restarted and switches from TLS1.2 to TLS1.3 and vice versa. In this way, we obtain the same amount of data for both protocols.

Initially, only 16 machines from Amazon were part of the experiment. Considering all machines, 10.88 requests/second were sent/received and each RESIP provider was processing 2.18 requests/second. We used these rates to remain below the limits imposed on us by our RESIP subscriptions. On 24/01/2022 at 19:00 UTC +0, we added 6 machines from Azure to our pool. At first, we kept the same rates per client/server. Hence, the rate was 14.96 requests/second and the ratio per RESIP provider was 2.99 requests/second. On 25/01/2022 at 16:00 UTC +0, Bright Data stopped our access to their network and ended our subscription. More details on the motivation for this choice are provided in Appendix A. Since it was not possible to restore this service, on 02/02/2022, we eliminated it from our experiment. We adjusted the rates accordingly and since then, 9.90 requests/second were sent/received for the rest of the experiment. Each RESIP provider processed 2.48 requests/second.

On 07/03/2022 at 00:00 UTC +0, we started collecting more network information to study the variability of our measurements. For each connection, we measure the RTT of all the TCP exchanges.

Occasionally, some machines were restarted by the cloud providers and this resulted in losses of data. We also had some brief synchronization issues, caused by using one port for both TLS1.2 and TLS1.3 server program and switching among them at midnight. Fortunately, this has been an extremely rare event. It has happened, on average, only 1.6 times per machine over the 110 days, with an average loss of only 0.17% of the traffic per machine.

We have created a database with POSTGRESQL [7] to gather the data from the experiment. In the database, we keep a unique record for each connection.

[5] The internal implementations of RESIP services are not known.

This record includes information collected at the client and the server as well as the network measurements.

For each connection, we save the epoch of the client request and the difference between the epoch of the server request and this value, the URL of the connection, the code of the used RESIP, client IP and port, SUPERPROXY IP and port, GATEWAY IP and port, server IP and port. Moreover, we gather the TCP version, TCP RTT, and TLS RTT. We calculate the minimum and maximum RTT of all the TCP packets exchanges and their corresponding positions in the stream.

In total, our clients have generated close to 98M connections but, as explained before, some observed connections were incomplete. Client requests sent when some servers were down only exist in the client-side logs (around 4M). Similarly, some requests are made to our servers from other machines than our clients (eg. from scanners and crawlers) and, for those, we have no matching record in the client logs (around 200K). Moreover, the sniffer program restarts every hour, and incoming connections arriving at the moment of the switch could have incomplete RTT measurements. We only create a record in our database for connections that exist both in the client and server logs and for which we have no missing field. In other words, we ignore connections for which we have no measurement from the sniffer logs. As a result, we use 95% of the total amount of connections started by the clients which sums up to 92,712,461 connections.

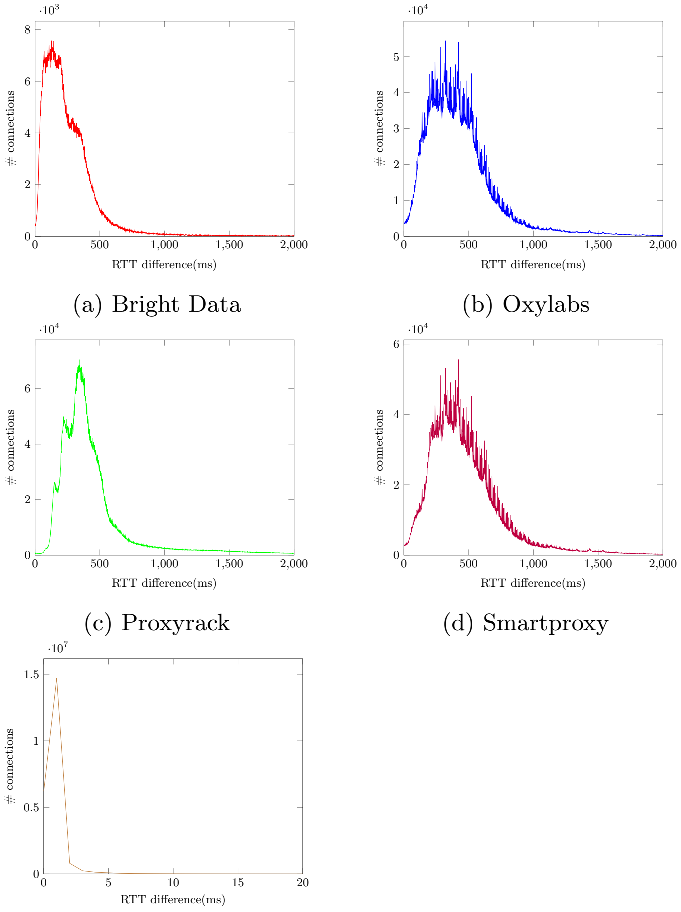
4 Results

In Fig. 3 we show the RTT *differences* for each proxy and for direct connections. To better visualize the RTT *differences* of the majority of the connections, we consider different ranges. 97% of direct connections have an RTT *difference* value lower or equal than 20ms. We use the range [0,20]ms for the x-axis. For proxied connections, we consider instead the RTT differences in the range [0,2000] ms, which amounts to the same percentage of connections.

We can see how for direct connections (Fig. 3e), the difference is always close to zero. In the RESIP plots (Figs. 3a–3d), instead, we can see how the difference varies for proxied connections. It is very important to note that the maximum value of the RTT *difference* (x-axis) in Fig. 3e is 100 times smaller than the ones in the other graphs. The maximum values on the y-axis is at least 3 orders of magnitude larger for direct connections than for the proxied ones. Yet, the total amount of connections has similar values for each proxied and the direct scenario[6]. These results clearly show that direct and proxied connections have dramatically different distributions of RTT *differences*.

Our approach determines if a connection passes through a RESIP provider from the measurement of the RTT *difference*. This measurement is conducted on packets sent and received on the Internet. Thus, network delays could, possibly, negatively impact our approach.

[6] Except for Bright Data for which we have less traffic due to the early end of the service, as explained in Appendix A.



(a) Bright Data (b) Oxylabs

(c) Proxyrack (d) Smartproxy

(e) Direct connection

Fig. 3: RTT *difference* of the connections, divided among RESIP and direct ones

In our experiment, we see connections with negative values for the RTT *difference*. The percentage of the total amount of connections per provider is 2.9%, 0.9%, 1.8%, 0.2%, 1.4%, respectively for direct connections, Bright Data, Oxylabs, Proxyrack and Smartproxy. A negative value of RTT *difference* occurs when the TCP RTT is higher than the TLS RTT. This happens when the SYN-ACK and/or the ACK packets of the TCP connection are delayed but the subsequent packets in the TCP connections are not. Thus, the TLS handshake packets are not delayed.

Similarly, it can also happen that only the TLS packets are delayed while the initial TCP ones are not. In this case, the RTT *difference* increases. This case is visually shown by the long tails of the distributions in Fig. 3. We note that, in the case of proxied connections, the packets participating in the TLS handshake have a longer "journey" than the ones used to compute the TCP RTT. Indeed, the packets travel from the client, through SUPERPROXY and GATEWAY, to the server whereas the latter only travel between GATEWAY and server. Hence, it is more likely to observe *an increase* of the RTT *difference* caused by temporary network congestion rather than *a decrease*. Our experimental results confirm this.

Fortunately, the above-mentioned situations, as reflected by the data of our experiment, are rare. To better understand how the variability of the network could influence our technique, we have studied the variation of the RTT values per connection.

For 56 days, we have collected the RTT of each TCP packet sent by the server and the corresponding ACK. For each connection, we identify the minimum value of RTT and its position within the stream as well as the maximum RTT value and its corresponding position.

For our proxy detection technique to work, the ideal case is to have the minimum RTT in the first exchange and/or to have low variability of its value throughout the connection. Our method could work taking as TCP RTT a RTT value of later exchanges (e.g. the minimum one of an entire connection). However, in this scenario, it would not enable us to detect the RESIP proxy at the very beginning of the connection (to possibly block it).

We have studied the network metrics collected in 45,902,917 connections. The minimum RTT is found in the first exchange in 56% of the connections but, fortunately, the variability is low (relatively to our use case). More than half of the connections (53%) present a difference between the maximum and minimum RTT lower or equal to 50ms.

We have chosen this value to empirically have a threshold above which we categorize the connections as proxied. After experimenting with different values, we found this to be the best threshold. Beyond being higher than more than half of TCP variations, it induces low values for the False Positive (FPR)[7] and False Negative (FNR)[8] rates. Indeed choosing this threshold we obtain a FPR of 0.04% and a FNR of 1.93%. The corresponding accuracy is 99.01%.

In 29% of the connections, the first RTT is the maximum one among all the exchanges. Despite that, the RTT *difference* shows relevant differences in the case of proxied connections. Let us consider the very unlucky case where all first RTTs would have the highest observed value of that connection. If we compute the RTT *difference* with this value and we choose 50ms as the threshold, we obtain a FPR of 0.01%, a FNR of 9.68%, and an accuracy value of 92.78%. Naturally, the percentage of false negatives increases with respect to our previous results (1.93%), but the accuracy remains high, even in this worst-case scenario. These results show that our technique is robust and can confidently detect RESIP connections even in very unlikely worst-case situations.

For each connection, as explained in Section 3.6, we collect the GATEWAY IP address. Studying their distribution per provider with Hilbert Curves[30], we have discovered that they are not uniformly distributed around the world. Naturally, we could fear that this influences the RTT *difference* values calculation.

[7] Direct connections flagged as RESIP over the total amount of direct connections

[8] RESIP connections flagged as direct over the total amount of RESIP connections

More specifically, we could wonder if the connections between clients and servers that are both in a location in which there is a high number of GATEWAYs of a specific provider could result in a small RTT *difference*. In such scenario, indeed, the added distance by the RESIP infrastructure could be small.

Based on our observations, Bright Data has the majority of its GATEWAYs in ARIN [2] and RIPE [9]. We have checked the RTT *differences* for the connections of this provider from our clients in Virginia to our servers in the same location. Oxylabs and Smartproxy GATEWAYs are mainly registered in LACNIC [5]. We have studied the RTT *difference* distributions between our machines in Brazil. For Proxyrack, whose IPs are mostly registered in AFRINIC [1], we have examined the RTT *differences* between clients and servers in South Africa.

The distributions of the above-mentioned combinations are in line with the other distributions of the same providers. Considering our threshold (50ms), the FNR value for these combinations is even lower than the one for all the connections (FNR =0.78%). This data confirms the idea that the detection of RESIP based on the RTT *difference* is a possible and viable method, even when clients, servers, and GATEWAYs are geographically close to each other.

5 Discussion

We propose a new method based on the measurement of the RTT to detect connections coming from RESIP providers. Our results show that this approach is feasible and robust.

The difference between the measurements in case of direct and proxied connections is substantial. 97% of the direct connections have an RTT *difference* lower than 20ms. In the same interval of RTT *difference* ([0,20]ms), Bright Data, Oxylabs, Proxyrack and Smartproxy connections accounts for, respectively, the 0.6%, 0.4%, 0.05% and 0.3% of the total. Moreover, the accuracy of our method remains high even in conditions of network delays.

Our technique can be easily implemented to protect existing servers because it applies to all connections using TLS1.2 or TLS1.3. Nowadays, more than 79% of websites use HTTPS connections and this percentage grows every year [12]. Thus, we can use our method in the majority of the Internet transactions.

Furthermore, our approach does not need any change in the existing software of the server. The measurement can be done outside the server with a sniffer, as we did in our setup.

Our technique does not detect only connections passing through RESIP services. It recognizes all the tunnel techniques that break the TCP connection between client and server. An example of this is SSH forwarding. Both local and remote forwarding do not provide end-to-end TCP connections and are thus detected with our method.

Generally, secure tunneling solutions do not break TCP, but leverage encapsulation. Tunneling providers guaranteeing anonymization typically use Network Address Translation (NAT) and IPSEC. This technique is nowadays really popular and widely used both in private and commercial networks. Neither NAT devices nor IPSEC, however, do break the TCP connection. Hence, requests passing through a VPN or a NAT device are not classified as proxied by our technique. There are methods, however, to detect these tunnels, such as the ones based on the Maximum Transmission Unit (MTU) analysis [36,22]. It is thus not a convenient solution for RESIP providers to switch to it, in order to avoid our detection.

A widely deployed defense for companies is the Web Application Firewall (WAF). WAFS need to break TLS to study the application content and assess if the communication is allowed to continue. Thus, if a client is behind a WAF, both TCP and TLS between client and server are broken and we see comparable TCP RTT and TLS RTT measurements. These connections are not declared as proxied by our method.

The reader could wonder if RESIP providers could break TLS, as WAF do, to avoid detection through the study of the RTT *difference*. This scenario is technically feasible but, in our opinion, unlikely to occur. Indeed, to do this, the proxy has to establish a TLS connection with the client and another one with our server. If the TLS session with the client terminates at the SUPERPROXY, an additional TLS connection has to be created between SUPERPROXY and GATEWAY. This implies that i) the client has to accept a root certificate that enables the SUPERPROXY or the GATEWAY (depending on where the first TLS session ends) to impersonate any server in the world, ii) the GATEWAY devices are now capable of decrypting (and thus monitoring or modifying) the exchanges between the clients and the servers, iii) the GATEWAY must handle two distinct TLS connections and decrypt/re-encrypt in both directions.

It is improbable that RESIP customers would let a third company observe and possibly modify contents of communications that should normally be encrypted between them and the server. Retrieved content could be modified and this could damage their web scraping activity.

RESIP providers leverage home devices, mobile phones, etc. which they do not own and on which they do not have full control. Furthermore, they must consume as few resources as possible to remain invisible to the owner of these devices. Users accept (consciously or not) to run RESIP software on their machines in moments in which they do not need them (e.g. when they are not using them and they are being charged). The additional burden of having to manage 2 distinct TLS sessions plus the decryption/re-encryption in both of them is likely to be a deterrent for devices' owners.

To be able to impersonate any possible end server, the proxy would have to establish certificates on the fly and to push them to the GATEWAY. Moreover, the client would have to accept the root certificate that would make this possible. While technically feasible, this adds a level of complexity and latency that would hurt the RESIP business and would be a major threat to the client once this root certificate is installed.

For all these reasons, we think breaking TLS would be difficult to implement in the RESIP infrastructure. If RESIP providers were to do that, we could still measure the RTT to the end client by serving HTML pages containing objects that the client would need to request to us. By measuring the time between the sending of the page to the client and the arrival of the request we would have a pretty good approximation of the RTT (objects need to be defined in a way that caching by the proxy would not be possible). Injecting objects for the sake of identifying proxies, as proposed in past work [38,24,35,22], could be leveraged for this. Defeating that scenario would require running a browser-like application on the GATEWAY itself which, for the reasons explained above, would increase even more the complexity of RESIP systems.

The reader could now also wonder whether RESIP providers could just produce delays at the TCP level to evade our detection method. This is not a feasible approach. RESIP providers do not own the devices used to proxy out the requests. They simply leverage them at the application level only. They cannot alter the connection settings of the device and/or do kernel level modifications to increase the TCP RTT.

Another point to raise is that an unexpectedly high delay caused by the certificate validation at the client side could increase the number of false positives. This scenario happens in case the server certificate is not signed by a known trusted party, requiring the client to fetch information online to establish a successful certificate chain. In this scenario, since we have full knowledge of the server, we expect to be able to anticipate this and account that all the real clients will have a somehow fixed delay. The chosen threshold can be then modulated to consider this delay. Moreover, we do not expect this situation to happen often. Scraped websites want to be largely accessed and, thus, use certificates that are widely trusted by their clients.

6 Conclusion and future works

In this paper, we provide a new sound method to detect proxied connections through the comparison of the TLS and TCP RTT of a single connection. We show that the method is easy to deploy and stable in case of network delays. We explain how it would be difficult for scrapers behind RESIP to evade it.

The next steps will consist in deploying this detection technique in front of servers suffering from scrapers using RESIP services. This will enable us to assess the real-world effectiveness of the proposed solution.

References

1. AFRINIC, https://afrinic.net/
2. ARIN, https://www.arin.net/
3. Bright Data, https://brightdata.com/
4. http.server, https://github.com/python/cpython/blob/3.10/Lib/http/server.py/
5. LACNIC, https://www.lacnic.net/
6. Oxylabs, https://oxylabs.io/
7. POSTGRESQL, https://www.postgresql.org/
8. Proxyrack, https://www.proxyrack.com/

9. RIPE, `www.ripe.net`
10. Smartproxy, `https://smartproxy.com/`
11. urllib, `https://github.com/python/cpython/tree/3.10/Lib/urllib/`
12. Usage statistics of Default protocol https for websites, `https://w3techs.com/technologies/details/ce-httpsdefault`
13. Azad, B.A., Starov, O., Laperdrix, P., Nikiforakis, N.: Web Runner 2049: Evaluating Third-Party Anti-bot Services. In: Proc. of DIMVA 2020 (2020)
14. Carielli, S., DeMartine, A.: The Forrester New Wave™: Bot Management,Q1 2020. Tech. rep., Forrester (2020)
15. Chiapponi, E., Dacier, M., Catakoglu, O., Thonnard, O., Todisco, O.: Scraping airlines bots: Insights obtained studying honeypot data. Intl. Journal of Cyber Forensics and Advanced Threat Investigations **2**(1), 3–28 (2021)
16. Chiapponi, E., Dacier, M., Thonnard, O., Fangar, M., Mattsson, M., Rigal, V.: An industrial perspective on web scraping characteristics and open issues. In: 2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks - Supplemental Volume (DSN-S). pp. 5–8 (2022)
17. Choi, J., Abuhamad, M., Abusnaina, A., Anwar, A., Alshamrani, S., Park, J., Nyang, D., Mohaisen, D.: Understanding the proxy ecosystem: A comparative analysis of residential and open proxies on the internet. IEEE Access **8**, 111368–111380 (2020)
18. DataDome: Bot IP addresses: 1/3 of bad bots use residential IPs. Here's how to stop them. (2022), `https://datadome.co/bot-management-protection/one-third-bad-bots-using-residential-ip-addresses/`
19. Dierks, T., Rescorla, E.: The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246, RFC Editor (2008), `http://www.rfc-editor.org/rfc/rfc5246.txt`, `http://www.rfc-editor.org/rfc/rfc5246.txt`
20. Gueye, B., Ziviani, A., Crovella, M., Fdida, S.: Constraint-Based Geolocation of Internet Hosts. IEEE/ACM Transactions on Networking **14**(6), 1219–1232 (2006)
21. Hanzawa, A., Kikuchi, H.: Analysis on Malicious Residential Hosts Activities Exploited by Residential IP Proxy Services. In: Information Security Applications. pp. 349–361. Springer International Publishing (2020)
22. Hoogstraaten, H.: Evaluating server-side internet proxy detection methods (Msc Thesis) (2018)
23. Imperva: Bad Bot Report 2021. Tech. rep., Imperva (2021)
24. incolumitas: Is this a valid method to detect Proxies? (2021), `https://incolumitas.com/2021/11/26/is-this-a-valid-method-to-detect-proxies/`
25. KiwiNet: pyshark, `https://github.com/KimiNewt/pyshark`
26. Landa, R., Clegg, R.G., Araujo, J.T., Mykoniati, E., Griffin, D., Rio, M.: Measuring the Relationships between Internet Geography and RTT. In: 2013 22nd International Conference on Computer Communication and Networks (ICCCN). pp. 1–7 (2013)
27. Li, X., Azad, B.A., Rahmati, A., Nikiforakis, N.: Good bot, bad bot: Characterizing automated browsing activity. In: 2021 IEEE Symposium on Security and Privacy (SP). pp. 1589–1605 (2021)
28. Mi, X., Feng, X., Liao, X., Liu, B., Wang, X., Qian, F., Li, Z., Alrwais, S., Sun, L., Liu, Y.: Resident evil: Understanding residential ip proxy as a dark service. In: 2019 IEEE Symposium on Security and Privacy (SP). pp. 1185–1201 (2019)
29. Mi, X., Tang, S., Li, Z., Liao, X., Qian, F., Wang, X.: Your Phone is My Proxy: Detecting and Understanding Mobile Proxy Networks. In: Proc. of NDSS 2021 (2021)

30. Munroe, R.: Map of the Internet (2006), https://xkcd.com/195/
31. Oppliger, R.: SSL and Tls: Theory and Practice, Second Edition. Artech House, Inc., USA, 2nd edn. (2016)
32. Percacci, R., Vespignani, A.: Scale-free behavior of the Internet global performance. The European Physical Journal B - Condensed Matter and Complex Systems **32**(4), 411–414 (2003)
33. Rescorla, E.: The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, RFC Editor (2018)
34. Tosun, A., De Donno, M., Dragoni, N., Fafoutis, X.: RESIP Host Detection: Identification of Malicious Residential IP Proxy Flows. In: 2021 IEEE International Conference on Consumer Electronics (ICCE). pp. 1–6 (2021)
35. Turgeman, A., Lehmann, Y., Azizi, Y., Novick, I.: Detection of proxy server, United States Patent US10069837B2 (2019), https://patents.google.com/patent/US10069837B2
36. ValdikSS: Detecting VPN (and its configuration!) and proxy users on the server side (2015), https://medium.com/@ValdikSS/detecting-vpn-and-its-configuration-and-proxy-users-on-the-server-side-1bcc59742413
37. Vastel, A.: How to Use Machine Learning to Detect Residential Proxies (2022), https://datadome.co/bot-management-protection/how-to-use-machine-learning-to-detect-residential-proxies/#ML-collecting-dataset
38. Webb, A.T., Narasima Reddy, A.L.: Finding proxy users at the service using anomaly detection. In: 2016 IEEE Conference on Communications and Network Security (CNS). pp. 82–90 (2016)
39. Yang, M., Yu, Y., Mi, X., Tang, S., Guo, S. Li, Y., Zheng, X., Duan, H.: An Extensive Study of Residential Proxies in China. In: Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (2022)

A Bright Data end of subscription

In our experiment, we legitimately bought 4 RESIP services to test their infrastructures. Subscriptions were made and paid online. Three out of four providers gave us access to the pool of residential IPs upon payment.

On the other hand, Bright Data did not enable us to use the residential IPs directly after the payment. They asked us to participate in a recorded interview in which we had to explain the motivations of our work and how we wanted to use their infrastructure. We communicated to them that we wanted to test our client and server machines with their infrastructure. The scope was a project we were developing with a third party that did not want to be named at the time. We told them that we would simply perform requests from our client to our server machines, as we did.

However, 13 days after the beginning of the experiment, they paused our subscription telling us that our scenario (targeting our own machines) could “expose their users IPs, which can become a privacy issue”. They told us that we would need to disclose additional information about what we were doing and whom we were working with. Since we did not accept to do so, they completely stopped the subscription and they refunded it.