First 5G deployment of Distributed Artificial Intelligence

Orestis Kanaris
Delft University of Technology
Delft, Netherlands
O.Kanaris@student.tudelft.nl

Johan Pouwelse (MSc Supervisor)
Delft University of Technology
Delft, Netherlands
J.A.Pouwelse@tudelft.nl

***Abstract*—**

***Index Terms*—NAT, CGNAT, 5G, Distributed Machine Learning, Mobile Machine Learning**

I. Introduction

II. Problem Description

A. Background

In recent years, the proliferation of mobile devices has reached unprecedented levels, with smartphones becoming an integral part of everyday life. These devices have increasingly powerful hardware, making them suitable candidates for running complex machine-learning models [1], [2]. Machine learning on mobile devices holds excellent potential for many applications, from personalized recommendations to democratizing big tech. One can imagine a world where every smartphone (or personal computer) holder holds their own portion of "Google's" database (and computation), having all smartphones intercommunication and share information to complete a search result, leading to a democratized distributed peer-to-peer search engine, cleansed from the big tech influence and hidden agendas [3].

However, deploying machine learning models on mobile devices presents numerous challenges, including limited computational resources, memory constraints, and the need for efficient communication between devices. The main struggle this paper focuses on is connectivity between devices since the communication in the context of this research will be handled by the IPv8[1]. IPv8 is a networking layer which offers identities and communication with some robustness and provides hooks for higher layers.

Personal devices, specifically smartphones, communicate through home Wi-Fi and mobile networks like 4/5G. Using these networks, the devices usually end up behind a home NAT or a Carrier-Grade NAT (CGNAT). The existence of these NATs makes it harder for the devices to communicate with each other since they lock their discoverability by hiding the devices behind the NAT's private network, forcing the "NATed" device to initiate the connection first. This is not a particularly impossible problem if one of the two peers has a static IP address and is discoverable. It is particularly bad when both peers are behind NATs (even worse when it is the same NAT, a problem common with CGNATs [4]), then both need to initiate the connection first, but none of them is "visible" to the other.

The STUN protocol (RFC3489 [5]) outlines four types of NATs: Full-cone NAT, Restricted-cone NAT, Port-restricted cone NAT, and Symmetric NAT. These categories are further classified in RFC4787 [4] as "easy" NATs, which employ Endpoint-Independent Mapping (EIM), and "hard" NATs, which utilize Endpoint-Dependent Mapping (EDM). EIM ensures consistency in the external address and port pair if the request originates from the same internal port.

As per V. Paulsamy et al. [6], the specifications for these NAT types are as follows:

- **Full-cone NAT:** This EIM NAT maps all requests from the same internal IP:Port pair to a corresponding public IP:Port pair. Moreover, any internet host can communicate with a LAN host by directing packets to the mapped public IP address and port.
- **Restricted-cone NAT:** Similar to Full-cone NAT, this EIM NAT maps an internal IP:Port pair to an external IP:Port pair. However, communication from an internet host to a machine behind the NAT is only allowed if initiated by that machine.
- **Port-restricted cone NAT:** Also an EIM NAT, similar to Restricted-cone NAT but with additional restrictions on port numbers.
- **Symmetric NAT:** This EDM NAT maps requests from the same internal IP:Port pair to a specific public IP:Port pair. However, it considers the packet's destination as well. Consequently, requests from the same internal pair but to different external hosts result in different mappings.

Symmetric NAT is the most "problematic" in the sense that it is the hardest one to establish a connection with if both peers are behind a NAT. Symmetric NATs behave very similar to a hard firewall; that is, they only allow incoming packets from a specific IP:Port pair only if an outgoing packet went to that destination first. The reason that one might use a symmetric NAT is when the administrator does not want to consume a single IP address per user since they theoretically allow up to 65535 simultaneous users. Symmetric NATs also give the fallacy of security, as in being behind a firewall since they

Identify applicable funding agency here. If none, delete this.

[1] https://github.com/Tribler/py-ipv8

never expose the user to the whole Internet, only to hosts that the user specifically "opted-in" to communicate with. The reason for this need for security is that the Internet lacks any security model. Anybody can freely send you an unlimited amount of data, spam, and malware [7]. A Symmetric NAT, to the average user, will not be an obstacle to their everyday browsing, but it becomes a big problem with peer-to-peer protocols, i.e. BitTorrent —In their 2008 study on fairness for BitTorrent users, J.J.D. Mol et al. [8] discovered that peers behind firewalls encounter greater challenges in achieving equitable sharing ratios. Consequently, they advocated for either puncturing NAT or employing static IP addresses to enhance network performance.

B. Research problem

The central problem of this thesis revolves around the distribution of Machine Learning on 4/5G Networks. To achieve this, one must connect efficiently to other peers through the cellular network.

Specifically, this project introduces the functionality lacking in IPv8 where they have an overlay network and APIs to connect more or less any peer devices, except when a peer is behind a Symmetric NAT. IPv8, as it stands, cannot add in the network peers behind this kind of NAT [9].

To overcome this limitation, this paper introduces a library to improve the proposal of D. Anderson's Birthday Attack blog post [10]. According to that blog post, if both peers send simultaneously $\approx$ 170000 connection-request packets, they have $\approx$ 99.9% probability of connecting. This is not entirely accurate since it doesn't consider the size of the NAT's HashTable nor the timeout time of the NAT. This paper proposes an improvement using data gathered from each provider's cellular data NAT, which is then analyzed to bias the attack to increase its success rate and avoid sending unnecessary packets that would, in turn, sabotage the attack.

The solution is a standalone open-source Kotlin library introduced in the following sections. It is evaluated both as a standalone library and also as part of IPv8, where the machine learning workload of TensorFlow Lite [11] will be distributed on Android mobile phones using the IPv8's ecosystem.

C. Objectives

The primary objectives of this thesis are as follows:

1) Address the NAT puncturing problem to enable seamless connectivity among devices, even when behind NATs or firewalls, by developing a NAT puncturing library in Kotlin.
2) Evaluate the proposed framework's performance, scalability, and resource utilization through experimental validation and benchmarking on Android devices obtained from the Tribler lab[2].

[2]https://www.tribler.org/about.html

III. Methodology

The methodology outlined in this chapter aims to evaluate the effectiveness of a naive birthday attack in establishing peer-to-peer connections within a NATed network environment. The fundamental premise of this investigation lies in addressing the challenge posed by NAT (Network Address Translation) configurations, where initiating communication between peers behind NATs can be cumbersome due to the necessity for one party to initiate communication first.

This study builds upon the approach proposed by Kanaris et al. [9], which suggests a method for peer-to-peer communication through the randomized exchange of packets until a successful "match" is achieved. However, the sheer volume of potential combinations renders this approach impractical. To address this, we leverage the counter-intuitive probability concept known as the Birthday Paradox, which allows for a reduction in the number of attempted combinations while maintaining a satisfactory success rate. Through experimentation and analysis, we aim to assess the feasibility and efficacy of this approach, particularly in the context of mobile network environments.

A. Evaluating Naive Birthday Attack

Algorithm 1 Naive Birthday Attack

Require: On packet received, send an ACK
Require: On packet received, $ack\_rcvd \leftarrow True$
Require: On packet received, store senders port
  $ack\_rcvd \leftarrow False$
  open UDP socket
  $msgs\_sent \leftarrow 0$
  $UUID \leftarrow generate\_UUID()$
  $packet \leftarrow create\_packet(UUID)$
  while $msgs\_sent < 170000$ and no ack_rcvd **do**
    $send\_packet(packet)$
  end while
  if ack_rcvd **then**
    $maintain\_connection(IP, port\_no)$
  else
    Birthday Attack was unsuccessful
  end if

The rule for communicating in a NATed network is that the person behind the NAT must initiate communication first. The assumption is that the Internet works mainly in a Client-Server fashion where the Server is discoverable (has a Public static IP address). This assumption breaks in the case of peer-to-peer communication between two clients behind a NAT since none are discoverable; thus, no one can initiate the communication.

A solution to this is as explained in [9]. Both peers should send packets to random ports until a "match" is achieved. A match is when peer A sends a packet from port X to port Y, and peer B sends a packet from port Y to port X in a timeframe smaller than their NAT's timeout. One can understand that the space for this is 65535^2 in a very tight timeframe, which is

almost impossible to achieve, let alone it will take a lot of time.

This can be improved using the Birthday Paradox [], a counterintuitive probability theory concept. It states that in a group of just 23 people, there's a better than even chance that two people share the same birthday. This might seem surprising, as intuition might lead one to think that with 365 days in a year, it would require many more people to have such a high probability of a shared birthday. The paradox arises because we're not just looking for a specific birthday match but any pair of people with matching birthdays. The probability of any two people not sharing a birthday decreases as more people are added to the group, and the opposite, the probability of at least one pair sharing a birthday increases rapidly.

The birthday paradox can be used to reduce the number of combinations of $sender\_port, receiver\_port$ while maintaining a satisfactory match probability. From the Birthday Paradox calculator [12], one can get a 50% success rate of a match after sending 77162 packets, and for a 99.9% success rate, 243587 packets are needed. Due to the nature of NATs (timeouts and a limited number of mapping maintained), these probabilities are unlikely to occur, but this would be the case even if all combinations are attempted.

Using the numbers above, an Android application [13] was developed to attempt to connect two mobile peers using 4/5G (which is by default using a NAT) using algorithm 1.

The results of the evaluation of 10 runs per carrier are shown in table I The evaluation of the naive birthday attack did not show auspicious results. The first conclusion that can be derived is that whether the attack will lead to a connection is very dependent on the telecommunication carrier pair. As one can see, when Vodaphone was one of the peers, there was always a successful attack. Another fascinating result is that only Vodaphone connected with a carrier of the same type, which was also the trial with the most successful connections.

These aside, even though half of the trials resulted in at least one successful connection, it is not satisfactory since one successful attempt out of ten makes this protocol costly in terms of cellular data used and time inefficient since coordinating two users to start attacking at the same time is already hard and error-prone on its own, doing it multiple times to achieve a single connection will deem the protocol unusable.

B. Determining NAT Timeouts

The NAT timeout is divided into two parts: timeout while waiting for a response and timeout between consecutive sends. I.e., how long will the NAT mapping remain active while waiting for the receiver to respond to an outgoing packet, and how long will the mapping remain active if there are no outgoing packets (all outgoing packets have been responded to)?

Starting with the timeout time of waiting for a response, initially, algorithm 2 establishes a lower and an upper bound on the time that the mapping will remain active while waiting for a response. This is achieved by sending a packet to the server, which incrementally waits more and more time to send a response until the NAT finally drops the response.

When the bounds are established, a binary search (algorithm 3 is performed on those bounds to find the price —to the second— timeout of the NAT.

Algorithm 2 Function to find the timeout in an interval of 20 seconds

```
1: function WAITTIMEOUTTEST
2:     delay ← 0
3:     create UDP Socket
4:     do
5:         delay ← delay + 20
6:         sendUDPPacket(delay)
7:     while timeoutMsgRcvr(delay) is true
8:     waitTimeoutBinaryTest(delay − 20, delay)
9: end function
```

Algorithm 3 Binary search on the timeout interval to get accuracy to the second

```
1: function WAITTIMEOUTBINARYTEST(l, r)
2:     while l ≤ r do
3:         delay ← (l + r)/2
4:         sendUDPPacket(delay)
5:         responseRcvd ← timeoutMsgRcvr(delay)
6:         if responseRcvd then
7:             l ← delay + 1
8:         else
9:             r ← delay − 1
10:        end if
11:    end while
12: return l, r
13: end function
```

The test for how much time the NAT mapping can remain idle without outgoing packets ...

The results of multiple runs of these algorithms on different telecom carriers can be seen in section VI-C.

C. NAT Types

The analysis of the types of NATs that the different Dutch Carriers are using can be found in table II. This analysis was performed using an adapted STUN client, stored on GitHub, with the rest of the analysis code used throughout this section [13].

D. Maximum Transmission Unit

The maximum transmission unit (MTU) denotes the maximum size of a single data unit that can be sent in a network layer transaction. MTU is related to the maximum frame size at the data link layer (such as an Ethernet frame).

A larger MTU is linked with reduced overhead, allowing more data to be transmitted in each packet. Conversely, smaller

	Odido	Lebara	Lyca	Vodaphone
Odido	F,F,F,F,F,F,F,F,F,F			
Lebara	F,F,F,F,F,F,F,F,F,F	F,F,F,F,F,F,F,F,F,F		
Lyca	F,F,F,F,S,F,F,F,F,F	F,F,F,F,F,F,F,F,F,F	F,F,F,F,F,F,F,F,F,F	
Vodaphone	F,F,F,S,F,S,F,F,F,F	F,F,F,F,S,F,F,F,F,F	F,F,F,F,F,S,F,F,F	F,F,F,S,S,F,F,S,F,S

TABLE I: Results of 10 consecutive Birthday Attacks for each pair of Carriers (F= No connection, S = Succesful Connection)

Algorithm 4 Function to find how long a NAT mapping is active while there is no communication going on in a range of 50 seconds

function TIMEOUTBETWEENSENDS
  $delay \leftarrow 0$
  $INC\_SIZE \leftarrow 50$
  create UDP Socket
  $prev\_port \leftarrow null$
  do
    wait($delay \times 1000$)
    sendUDPPacket("$TIMEOUT - TEST$")
    $resp \leftarrow$ timeoutMsgRcvr()
    $port \leftarrow$ extract_port($resp$)
    if $prev\_port = null$ **then**
      $prev\_port \leftarrow port$
    end if
    $delay \leftarrow delay + INC\_SIZE$
  while $prev\_port = port$
  $l \leftarrow delay - (2 \times INC\_SIZE)$
  $r \leftarrow delay - INC\_SIZE$
  timeoutBetweenSendsBinary(l, r)
end function

Algorithm 5 Function to find exactly how long a NAT mapping is active while there is no communication going on

function TIMEOUTBETWEENSENDSBINARY(l, r)
  sendUDPPacket("TIMEOUT-TEST")
  $response \leftarrow$ timeoutMessageReceiver()
  $latestPort \leftarrow$ extract_port($response$)
  while $l \leq r$ **do**
    $midpoint \leftarrow$ floor($(l + r)/2$)
    delay($midpoint * 1000$)
    sendUDPPacket("TIMEOUT-TEST")
    $response \leftarrow$ timeoutMessageReceiver()
    $port \leftarrow$ extract_port($response$)
    if latestPort = port **then**
      $l \leftarrow midpoint + 1$
    else
      $r \leftarrow midpoint - 1$
      latestPort $\leftarrow$ port
    end if
  end while
  return r, l
end function

Algorithm 6 STUN Test, NAT Type Detection, and Getting IP Information

function STUNTEST(sock, host, port, sendData)
  Initialize response data structure
  Convert sendData to hex byte array with headers
  Send byte array to (host, port)
  Receive and decode response packet
  if response matches and transaction ID correct **then**
    Parse attributes like Mapped Address, Source Address, etc.
  end if
  return response
end function
function GETNATTYPE(s, sourceIp, stunHost, stunPort)
  Attempt STUN test with provided or default server
  if initial test fails **then**
    for all server in STUN_SERVERS **do**
      Attempt STUN test with server
    end for
  end if
  Determine NAT type based on test results
  Perform additional tests for refining NAT type
  return NAT type
end function
function GETIPINFO(sourceIp, sourcePort, stunHost, stunPort)
  Create socket with specified source IP and port
  Determine NAT type using GETNATTYPE
  Close socket
  return NAT type, external IP, and external port
end function

MTU values can help decrease network delay by facilitating quicker processing and transmission of smaller packets.

The determination of the appropriate MTU often hinges on the capabilities of the underlying network and may require manual or automatic adjustment to ensure it doesn't exceed these capabilities.

A jumbo frame is an Ethernet frame with a payload greater than the standard maximum transmission unit (MTU) of 1,500 bytes.

Algorithm 7 is used to determine the MTU of each carrier by running this algorithm each time with different sim cards from different providers. It is a binary search which tries to find the precise number of bytes, where one more byte will cause the packet to be split into two. Table III shows the MTU of the different providers tested and whether they support Jumbo frames.

Algorithm 7 Function to find the Maximum transmission unit of a carrier

```
function FINDMTU
    icmp ← new Icmp4a()
    left ← 0
    right ← 65507
    while left < right do
        midPoint ← floor((left + right)/2)
        result ← icmp.ping(packetSize = midPoint)
        switch result do
            case Success
                left ← midPoint + 1
            case Failed
                right ← midPoint − 1
    end while
    return right
end function
```

E. Improving the Birthday Attack

Given the cost, the success rate of the Naive Birthday attack, as shown in table I, is not satisfactory. To improve that, one needs to understand the inner workings of each NAT, i.e., how the mapping works, whether there are any patterns, etc.

To answer these questions, an Android mobile client and a kotlin server were developed [14]. The mobile client sends packets containing a UUID[3] to the server from random mobile ports to random server ports. Each UUID, source and destination port are saved in a CSV file. The server which lies behind an unrestricted network does the same; as soon as a packet is received, it stores the UUID inside the packet, the port that the mobile sent it from and the port that the server received it. The two CSVs are then inner-joining on the UUID column, resulting in two crucial columns: the port the mobile believes it sent the packet from and the port the packet came from, i.e. the NAT mapping.

How are the data analyzed, what are the results ? Also advertise that anyone can download the app etc?

IV. System Design

V. Implementation

VI. Evaluation

A. Inner workings of NATs

In this chapter, we delve into the inner workings of Cellular Data NAT across various service providers. Through reverse engineering efforts targeting multiple providers, we unveil the nuanced mechanisms NAT systems employ. Each provider has a different implementation of address mapping strategies, which hinders connectivity on peer-to-peer protocols. By dissecting these NAT architectures, we gain a deeper understanding of their address-mapping strategies, which can be used to one's advantage when they are trying to connect to another peer using cellular data. The analysis is freely available on GitHub [15].

The data gathered using a simple app developed for this research [14] utilises a server with a static open IP address and a phone sending packets to the server containing a UUID. The phone stores the UUID, a timestamp, the port from which it sends the packet, and the port to which it sends it. The server stores the UUID, a timestamp, the phone's port and the port it received on. Then, the two files are joined on the UUID, showing the port the phone opened, and the port the NAT mapped it to. This relationship is then analyzed across runs to understand the address-mapping strategy of each NAT.

1) Lebara Netherlands: The initial observation was that a lot of the sender ports (what the server sees as return addresses) seemed to be following a linear pattern. Initially, some random port was chosen, then the next port's number would be the one of the previous +1, and so on, until a condition was met that would cause it to choose a new random port to start with and then get the consecutive ones and so on.

It was also observed that the initial random ports were often reused across sessions, but those specific numbers were not common across runs. For example, if the first port open was port 12800 and the next x ports, this sequence would be seen multiple times across the run. It is the same with the second, third, etc, random ports and the consecutive ones, but not as frequently.

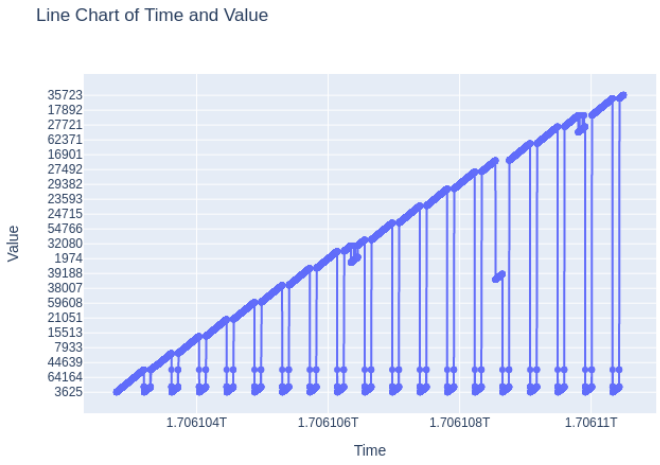

Fig. 1: Port mapping through time on a single Lebara run of $\approx$ 128 minutes

One can see from figure 1 that the port mapping is following a pattern. It starts from the 3625 region —region since it is not the actual starting port—, stays in that region for a bit (incrementing the port numbers by 1), chooses other random regions and then goes back to the 3625 region. One can also notice that the intervals between NAT defaults back to the 3625 region are more or less constant.

Another observation was when observing the length of the linear increases. During low traffic hours, what was observed

[3]https://docs.oracle.com/javase/8/docs/api/java/util/UUID.html

was that the initial "random" port's number was a multiple of 256 (2^8), then the next 255 consecutive ports will be used, and then another random starting port (again multiple of 256) will be used and so on as can be seen in figure 2.

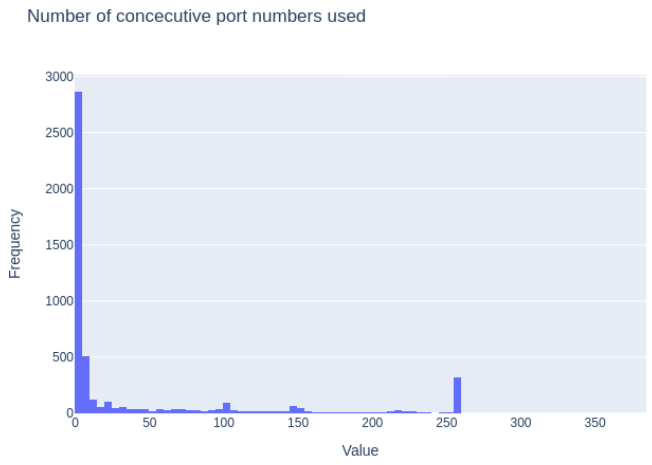

Fig. 2: Frequency of consecutive port numbers used by Lebara

The assumption is that consecutive ports are grouped together in groups of size 256. The exact number of groups cannot be inferred since not all ports were observed, but it seems to span the whole space of 65535 ports. Thus, it is assumed that there are 256 groups of 256 ports. The ports are probably grouped in queues, and then users are assigned to queues. They consume port mappings until the queue runs out of available ports; then, they are assigned to another queue. When the ports are freed or timed out, they return to their queue.

The strategy on who is assigned to which queue cannot yet be inferred, but it is probably either based on the number of consumers in the queue or the queue size. Both of these strategies are reasonable because the same ranges are consumed repeatedly since they timeout and the queue gets full again, and no one is currently using it since it was empty.

Both strategies are also validated throughout the day. On low traffic hours, the test phone was assigned a full queue, which may be either because of the queue size or because no one else was consuming (morning hours in a residential area). Similar to peak traffic hours on the university campus, many times, the phone achieved a significant amount of consecutive ports. This means that there is some strategy on the NAT to give the user as many ports as possible, again, either through the number of consumers on the specific port range or based on the number of available ports on that range.

2) KPN: KPN behaves exactly like Lebara, that is ports are grouped in groups of 256 with consecutive port numbers. The main difference between KPN and Lebara is that KPN has more infrastructure than Lebara —since Lebara is renting infrastructure from KPN— thus, as one can see in figure 3, the test phone managed to consume much more groups of 256 consecutive ports in its entirety than on Lebara. This is likely the case because of the difference in the number of users per infrastructure.

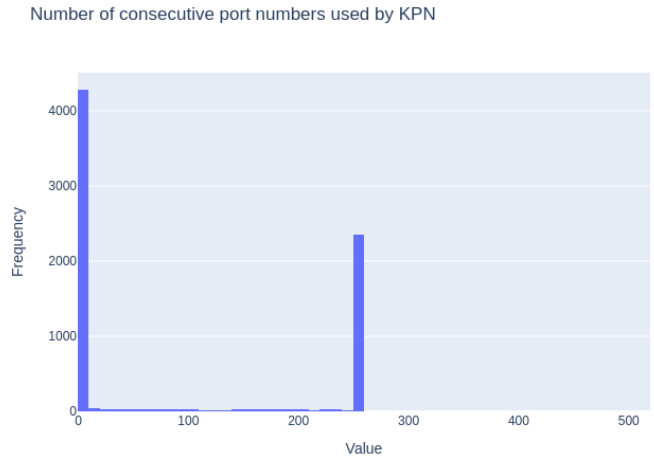

Fig. 3: Frequency of consecutive port numbers used by KPN

The number of subscribers on a single KPN hardware/ IP address makes KPN significantly more predictable than Lebara. During the testing period, the test phone consumed a port group in its entirety 32.3% of the time. On top of that 36.9% of the time, the phone got assigned to a group where the initial port was available (the port number was divisible by 256). This observation gives birth to a strategy of trying port numbers divisible by 256. This may significantly increase the probability of achieving a collision since the phone can perform a request every $\approx 15ms$, meaning that it can try all ports divisible by 256 in under 4 seconds.

3) LycaMobile Netherlands: LycaMobile, although utilizing the network of KPN, employs a different strategy for their address mapping. After analyzing ≈ 288000 mappings, there seems to be complete randomness. No mapping is reused (very few are and in different runs; thus, they are assumed to be a coincidence), and there is no linearity on the mappings, making their address mapping strategy a First Come, First Serve on available ports.

Regarding efficiency, the theory on the inner workings is a FIFO Queue of available ports that all network subscribers subscribe to and "consume" free ports. When a port is freed or time-out, it returns to the Queue. There is no indication of the port numbers being sorted, or eventually sorted, since consecutive ports were consumed so rarely, even on low traffic hours, that it can just be written off as a coincidence.

4) Vodaphone Netherlands:

5) Odido:

B. Nat Types

Knowing the NAT type of the carrier that one is using and also of the peer they want to connect allows one to adapt their connectivity strategy to increase the chance of connecting.

Provider	Type	Area
Lyca NL 4G	Full Cone	Echo Tu delft
Lyca NL 5g	Full Cone	Echo Tu delft
Vodaphone 4G	Restrict NAT	Echo Tu delft
Vodaphone 5G	Restrict NAT	Echo Tu delft
KPN 4G	Symmetric NAT	Echo Tu delft
Lebara 4G	Restrict NAT	Echo Tu delft
Orange Belgium 4G	Symmetric NAT	Spiti tu giorgou Bg
Lyca Mobile Belgium 4G	Restrict NAT	Spiti tu giorgou Bg
MyCall Norway 4G	Full Cone NAT	Oslo Airport
Telia Norway 5G	Restrict NAT	Oslo Airport
Telia Norway 4G	Restrict NAT	Oslo Airport

TABLE II: Nat Types of all the carriers tested and the location of the test

Different strategies should be adopted based on the types, i.e. a Symmetric NAT requires a Birthday Attack, while one can easily connect with a peer behind a Full-Cone NAT using a STUN server. The types of the NATs of various carriers are presented in table II.

C. Timeout of NATs

Understanding a NAT's timeout is crucial for efficient network management and troubleshooting. First, knowing the timeout period for waiting for a response ensures that THE protocol administrators can optimize their network configurations for timely communication. Second, it was useful in understanding patterns of the NAT's behaviour, such as why some port numbers are being reused repeatedly in similar time frames. Thus, aligning the NAT puncturing strategies with the expected timeout duration increases the probability of a puncture. These timeouts are shown in table **??**

Secondly, NATs also have an idleness timeout, i.e. a mapping is deleted if not used. Establishing a connection is costly; thus, maintaining it, even if not needed at some particular instance, is the way to go. To achieve that, one needs to send connection maintenance packets, i.e. empty packets, that will trick the NAT that some communication is still happening. One can use the precise timeout of the NAT to send these packets in such intervals that will not flood the network unnecessarily while also ensuring that the connection stays active. These timeouts are shown in table **??**.

D. Maximum Transmission Unit

Knowing the Maximum Transmission Unit (MTU) of a carrier network offers several advantages. Firstly, it helps optimize network performance by determining the largest packet size that can be transmitted without fragmentation, reducing overhead and latency. Additionally, understanding the MTU enables efficient bandwidth utilisation, as smaller packets may lead to increased overhead and decreased throughput. Knowledge of the MTU facilitates troubleshooting network issues, allowing for more effective diagnosis and resolution.

As for jumbo frames, their presence further enhances network efficiency by supporting larger packet sizes than standard MTU, thereby reducing the overhead of transmitting data. However, it's important to ensure compatibility with all devices and networks involved to leverage the benefits of jumbo frames fully.

The MTU of various carriers and whether their network supports jumbo frames is presented in table III.

E. Improved Birthday Attack Evaluation and Findings

...

1) Lebara:

2) Lyca:

3) Odido:

4) Vodaphone: The evaluation results of 10 runs per carrier using the improved Birthday attack are shown in table **??**. The success rate difference is shown in table **??**

VII. Discussion and Future Work

VIII. Conclusion

Appendix

1) Orange Belgium:

2) LycaMobile Belgium:

3) Orange France:

4) SFR France:

5) Telia Norway:

6) MyCall Norway:

References

[1] M. S. Louis, Z. Azad, L. Delshadtehrani, S. Gupta, P. Warden, V. J. Reddi, and A. Joshi, "Towards deep learning using tensorflow lite on risc-v," in *Third Workshop on Computer Architecture Research with RISC-V (CARRV)*, vol. 1, 2019, p. 6.

[2] J. Dai, "Real-time and accurate object detection on edge device with tensorflow lite," in *Journal of Physics: Conference Series*, vol. 1651, no. 1. IOP Publishing, 2020, p. 012114.

[3] Tribler, "msc placeholder: "swarming llm": decentralised artificial intelligence · issue 7633 · tribler/tribler." [Online]. Available: https://github.com/Tribler/tribler/issues/7633

[4] C. F. Jennings and F. Audet, "Network Address Translation (NAT) Behavioral Requirements for Unicast UDP," RFC 4787, Jan. 2007. [Online]. Available: https://www.rfc-editor.org/info/rfc4787

[5] J. Rosenberg, C. Huitema, R. Mahy, and J. Weinberger, "STUN - Simple Traversal of User Datagram Protocol (UDP) Through Network Address Translators (NATs)," RFC 3489, Mar. 2003. [Online]. Available: https://www.rfc-editor.org/info/rfc3489

[6] V. Paulsamy and S. Chatterjee, "Network convergence and the nat/firewall problems," 2003.

[7] M. Zolotykh, "Comprehensive classification of internet background noise," 2020.

[8] J. Mol, J. Pouwelse, D. Epema, and H. Sips, "Free-riding, fairness, and firewalls in p2p file-sharing," 2008.

[9] O. Kanaris and J. Pouwelse, "Mass adoption of nats: Survey and experiments on carrier-grade nats," 2023.

[10] D. Anderson, "How nat traversal works - nat notes for nerds," Apr 2022. [Online]. Available: https://blog.apnic.net/2022/04/26/how-nat-traversal-works-nat-notes-for-nerds/

[11] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

Provider	MTU (BYTES)	Allows Jumbo Frames?	Area
T-Mobile			
Lebara 4G	65507	Yes	Echo Tu delft
Lyca 4G	1473	No	Echo tu delft
Vodaphone			
KPN 4G	1445	No	Echo Tu delft
Orange Belgium 4G	1472	No	Spiti tou giorgou
Lyca Mobile Belgium 4G	42987	Yes	Spiti tou giorgou Belgium
MyCall Norway 4G	65507	Yes	Oslo Airport
Telia Norway 5G	65507	Yes	Oslo Airport
Telia Norway 4G	65507	Yes	Oslo Airport

TABLE III: The MTU of various carriers and whether they accept Jumbo frames at the location of testing

[12] Fast-Reflexes, "Fast-reflexes/birthdayproblem-python: Implementation of a solver of the generalized birthday problem in python." [Online]. Available: https://github.com/fast-reflexes/BirthdayProblem-Python

[13] O. Kanaris, "NAT measurements gathering with Naive Birthday Attack for connecting smartphones," Dec. 2023.

[14] ——, "NAT Mapping data Gathering and analysing tool," Feb. 2023.

[15] ——, "Cellular Network NAT Reverse Engineering and Exploration," Apr. 2024. [Online]. Available: https://https://github.com/OrestisKan/telecom-analysis