An Untold Story of Middleboxes in Cellular Networks

Zhaoguang Wang[1], Zhiyun Qian[1], Qiang Xu[1], Z. Morley Mao[1], Ming Zhang[2]
[1]University of Michigan [2]Microsoft Research
[1]{zgw, zhiyunq, qiangxu, zmao}@umich.edu [2]mzh@microsoft.com

ABSTRACT

The use of cellular data networks is increasingly popular as network coverage becomes more ubiquitous and many diverse user-contributed mobile applications become available. The growing cellular traffic demand means that cellular network carriers are facing greater challenges to provide users with good network performance and energy efficiency, while protecting networks from potential attacks. To better utilize their limited network resources while securing the network and protecting client devices the carriers have already deployed various network policies that influence traffic behavior. Today, these policies are mostly opaque, though they directly impact application designs and may even introduce network vulnerabilities.

We present *NetPiculet*, the first tool that unveils carriers' NAT and firewall policies by conducting intelligent measurement. By running *NetPiculet* on the major U.S. cellular providers as well as deploying it as a smartphone application in the wild covering more than 100 cellular ISPs, we identified the key NAT and firewall policies which have direct implications on performance, energy, and security. For example, NAT boxes and firewalls set timeouts for idle TCP connections, which sometimes cause significant energy waste on mobile devices. Although most carriers today deploy sophisticated firewalls, they are still vulnerable to various attacks such as battery draining and denial of service. These findings can inform developers in optimizing the interaction between mobile applications and cellular networks and also guide carriers in improving their network configurations.

Categories and Subject Descriptors

C.2.3 [**Computer-Communication Networks**]: Network Operations

General Terms

Measurement, performance, security

Keywords

NAT, firewall, middlebox, TCP performance, cellular data network

1. INTRODUCTION

Data cellular networks have undergone tremendous growth in recent years due to the increasing popularity of mobile devices such as smartphones, tablets, and eBook readers. Their ever-growing coverage and capacity have enabled a wave of flashy mobile applications ranging from gaming and video to social networking. In contrast to their Internet counterparts, cellular networks bear more constraints due to the scarcity of physical resources in network infrastructure and mobile devices as well as the complexity in managing these resources. Cellular carriers usually deploy various types of middleboxes to make efficient use of these precious resources and protect them from potential attacks. For instance, many carriers use NAT (Network Address Translation) to provide data service to millions of users over a limited public IP address space. They also deploy firewalls to isolate mobile users from rampant malicious activities (*e.g.,* worms and DoS attacks) on the Internet.

Today, cellular network middleboxes and mobile applications are independently managed by two groups of entities: cellular operators (*e.g.,* AT&T, T-Mobile) and application developers. The latter group is often unaware of the middlebox policies enforced by operators while the former has limited knowledge about the application behavior and requirements. Such knowledge mismatch could potentially impair application performance, aggravate energy consumption, or even introduce security vulnerabilities. One illustrative example is that a carrier sets an aggressive timeout value to quickly recycle the resources held by inactive TCP connections in the firewall, unexpectedly causing frequent disruptions to long-lived and occasionally idle connections maintained by applications such as push-based email and instant messaging.

Prior work has studied middleboxes on the Internet by characterizing NAT properties such as mapping type and filtering rule [22, 15, 20, 18, 21], and proposed various schemes to perform NAT traversal. It is unclear whether the NAT box behavior remains the same or the NAT traversal schemes are still applicable in cellular networks. Other work quantified the end-to-end performance degradation (*e.g.,* lower throughput or larger transaction delay) induced by middleboxes [13]. In addition to performance, energy consumption is another critical perspective in the cellular network context. However, no previous studies have investigated how middleboxes affect energy consumption on mobile devices. Several research efforts revealed that cellular infrastructure and mobile handsets are vulnerable to various types of DoS attacks [35, 37, 26]. However, they did not investigate the feasibility of launching these attacks when the targets are behind middleboxes.

In this study, we design and implement *NetPiculet*, a measurement tool for accurately and efficiently identifying middlebox policies in cellular networks. We focus on firewalls and NAT boxes, which are widely deployed by many cellular carriers. Although a carrier may employ many policies, we focus on those that directly impact mobile users and their applications in terms of important properties including connectivity, performance, energy consumption, and security. Informed by our findings, we propose new techniques and modifications to applications to better cope with existing policies. We also offer concrete suggestions on policy changes for carriers to improve the experience and protection of their mobile users as well as robustness of their network infrastructure in response to attacks.

We released *NetPiculet* on Android Market in January 2011 and attracted 393 unique mobile users within merely two weeks. Leveraging the data from these users, we report our findings from 107 cellular carriers around the world. In particular, we studied the policies of two large nation-wide U.S. carriers in more depth and corroborated our findings carefully with controlled experiments. Due to security and privacy concerns, we anonymize their names and label them as Carrier A and Carrier B. We summarize our key findings as follows:

- In some cellular networks, a single mobile device can encounter more than one type of NAT, likely due to load balancing. We also discovered some NAT mappings increment external port number with time which was not documented in any prior NAT study. Accordingly, we develop new NAT traversal techniques to handle both cases.
- Four cellular networks are found to allow IP spoofing, which provides attack opportunities by punching holes on NATs and firewalls "on behalf of" a victim from inside the networks, and thus directly exposing the victim to further attacks from the Internet.
- Eleven carriers are found to impose a quite aggressive timeout value of less than 10 minutes for idle TCP connections, potentially frequently disrupting long-lived connections maintained by applications such as push-based email. The resulting extra radio activities on a mobile device could use more than 10% of battery per day compared to those under a more conservative timeout value (*e.g.,* 30 minutes).
- One of the largest U.S. carriers is found to configure firewalls to buffer out-of-order TCP packets for a long time, likely for the purpose of deep packet inspection. This unexpectedly interferes with TCP Fast Retransmit and Forward RTO-Recovery, severely degrading TCP performance triggered merely by a single packet loss.
- At least one firewall of a major cellular ISP liberally accepts TCP packets within a very large window of sequence numbers, greatly facilitating the traditional blind data injection attacks, endangering connections that transfer relatively large amount of data (*e.g.,* streaming applications).
- Some cellular network firewalls do not immediately remove the TCP connection state after a connection is closed, allowing attackers to extend his attack on a victim even after the victim has closed the connection to a malicious server. This also dramatically lengthens the NAT traversal time to a few minutes, given that the same TCP five tuple cannot be reused quickly.

2. OVERVIEW

Today's cellular data networks are susceptible to DoS attacks [35, 37, 26] and face the problem of IPv4 address depletion, which prompts carriers to deploy NATs and firewalls at network boundaries to protect cellular infrastructure and mobile users from unsolicited traffic from the Internet and to effectively share public IP

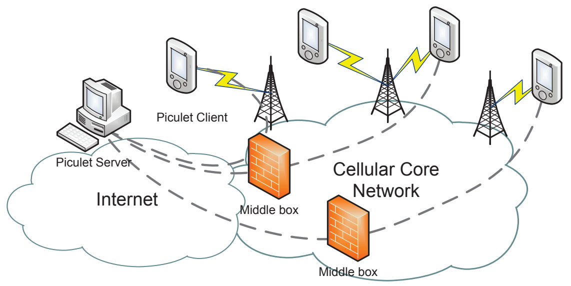


Figure 1: Physical view of the NetPiculet system

Count by # of	Technology		Continent						IP address		
	UMTS	EVDO	EU	AS	NA	SA	AU	AF	Public	Private	Both[1]
Carriers	97	10	46	26	20	11	2	2	25	72	10
Users	246	148	113	35	231	11	2	2	73	316	5[2]

[1] Some carriers assign both public and private IP addresses
[2] A single user is observed to have public IP or private IP at different times

Table 2: Properties of the studied carriers

addresses. However, they also directly impact end-to-end performance, connectivity, energy consumption, and security. Popular mobile P2P applications, *e.g.,* Fring [5] and Tango [6] for video chatting, use NAT traversal to establish direct connections between peers behind NAT. Correctly determining the NAT mapping type and filtering policy is crucial for the success of NAT traversal. Firewalls attempt to block unwanted traffic by detecting anomalous traffic patterns. Such behavior, however, may tamper with TCP's control and feedback mechanism, resulting in longer transaction delay and higher energy consumption. Furthermore, firewalls in cellular networks must accommodate a diverse and large set of mobile devices and applications. Such flexibility may come at the expense of security, leaving opportunities for mindful attackers.

Our goal is to develop a tool that can help application designers gain insight into NAT and firewall policies in cellular networks and make targeted improvements to applications. It can also help carriers fine-tune their middlebox policies to attain better user experience and security. A few technical challenges remain. First, the middlebox policies in a cellular network can be quite complex and are usually proprietary. We need to design a suite of end-to-end probes to accurately infer the policies and quantify their impact. Second, the middleboxes usually block unsolicited traffic from the Internet, making purely server-based probing infeasible. We need to carefully coordinate probing between a mobile client and an Internet server to infer these policies, while ensuring the policies inferred are not due to middleboxes on the Internet. Third, cellular carriers are diverse, often spanning a large geographic area and offering multiple types of subscription plan. We need to make our tool both efficient and user-friendly to attract different types of users from many carriers and locations.

We have built the *NetPiculet* tool that comprises of client software running on mobile devices inside cellular networks and a dedicated server on the Internet as shown in Figure 1. The server's upstream provider and border router are verified to not impose any restrictive policies. The client software is an Android application that is publicly available on the Android Market. In order to save both user time and device battery, except for the TCP connection timeout measurement running as a background service, the other tests are parallelized and finish within 10 seconds. By informing users about their network polices, *e.g.,* the feasibility of running P2P applications in their networks, the client software is able to attract users around the world. Both the server and client are implemented in C and Java with approximately 3000 LoC in total.

Category	Policy	Main findings and implications
NAT	NAT Mapping type (§3.1.1) Endpoint filtering (§3.1.2) TCP state tracking (§3.1.2) Filtering response (§3.1.2) Packet mangling (§3.1.2)	- A new NAT mapping was identified that increases external port number with time (§3.2.1) - One device may experience more than one NAT mapping in the same cellular network (§3.2.1) - New NAT traversal techniques are proposed to handle the challenges imposed by the above two findings (§3.3)
Firewall	IP spoofing (§4.1.1) Stateful firewall (§4.1.2) TCP connection timeout (§4.1.3) Out-of-order packet buffering (§4.1.4)	- 4 large carriers allow IP spoofing, which weakens the network security (§4.2.1) - 11 carriers timeout idle connections aggressively, which wastes device energy (§4.3.1) - One large U.S. carrier buffers out-of-order packets, which negatively affects TCP performance (§4.3.2) - One large U.S. carrier sets a large TCP sequence window, which facilitates TCP RST attack (§4.3.3) - Some carriers do not clear TCP state immediately after connection close, which invites battery draining attack (§4.3.4)

Table 1: *NetPiculet*'s key functionality, findings, and implications.

Table 1 summarizes the policies inferred by *NetPiculet* and where they are described. Because the TCP state diagram inference is time-consuming, we measure the full TCP state diagram only in local experiments and replace it with a simpler TCP connection timeout measurement in the released tool. Our results are based on two weeks of data collected in January 2011 from about 400 users in 107 carriers. Table 2 breaks down the carriers and users by technology, continent, and IP type. Note that for certain type of tests we may observe smaller number of data points as some tests such as IP spoofing require root privilege which is not available on every phone. Also, NAT tests only consider the carriers that assign private IP addresses. Moreover, we conducted extensive local experiments to validate the findings and quantify their impact for two major U.S. carriers: Carrier A and Carrier B.

Although the policies in Table 1 are associated with either NAT or firewall, we emphasize that this is simply a classification and may have little to do with the actual implementation. A carrier usually has a variety of options in implementing these policies. For instance, a single network device could have both the NAT and firewall functionalities. Our end-to-end probes treat NAT and firewall as black boxes, thus only inferring the *existence* of the policies but not their *inner workings*.

3. NAT POLICIES

In this section, we study the feasibility of NAT traversal in cellular networks. We focus on TCP NAT traversal because it is much more challenging than UDP NAT traversal. Our findings, however, can be easily applied to UDP NAT traversal as well. We first provide some background knowledge on several key NAT properties related to NAT traversal and describe the methodology for measuring these properties. We then present the results from *NetPiculet* clients executed by 316 mobile users in 72 carriers with NAT boxes deployed. We identify a new NAT mapping that increments external port number with time. We also find that a single mobile device may encounter more than one NAT mapping. Since techniques for traversing NAT with such previously unknown behavior were not studied, we design and implement a light-weight scheme to traverse the new NAT mapping with high success rate.

3.1 Methodology

At the high level, existing TCP NAT traversal approaches (*e.g.,* STUNT#1 [22], STUNT#2 [18], NATBlaster [15], and P2PNAT [20]) follow a similar idea. Two clients behind NAT first learn each other's external IP address through an out-of-band channel, *e.g.,* a third party server. Then both clients initiate a TCP connection to each other by sending a SYN packet, which creates a mapping on their own NATs. The destination port, which the SYN packet was sent to, is determined from *port prediction* based on the NAT mapping. The two connections are finally reconciled into one if the traversal succeeds. The key difference among various approaches is the distinct sequence of packets exchanged between two clients during the reconciliation process. Besides the NAT mapping, these approaches also heavily depend on several other NAT properties, which will be described below.

3.1.1 Identifying NAT Mapping

A NAT box maps a TCP connection to an external endpoint (IP address and port) based on the TCP five tuple. Based on previous work [21], the NAT box can map connections from *the same local endpoint* in the following ways:

- *Independent:* external endpoint remains the same for all connections.
- *Address and Port*$_\delta$*:* external endpoint changes when destination endpoint changes.
- *Connection*$_\delta$*:* external endpoint changes for each new connection.

The subscript δ indicates how the external port number changes relative to its previous value. It can be either a fixed value n (usually $n = 1$) or a random value R.

It is important for both clients to know each other's mapping type so that they can predict the external endpoint a new connection will use based on that of a previous connection. This allows each client to send a SYN packet to the appropriate external endpoint of the other client and create necessary connection mapping state on its own NAT.

To discover the mapping type of a NAT box, we follow the approach in previous work [21] to create 12 back-to-back connections to 4 destinations (3 connections to each destination) while using the same local port. Different from prior approaches, we add another 12 connections to the same destination but with different local ports. As explained in §3.2.1, this helps us discover a new time-dependent NAT mapping type.

3.1.2 Identifying Other NAT Properties

Endpoint filtering: A NAT box forwards an incoming packet to a destination based on its existing connection mapping state. A packet will be filtered if its destination address and port are not found in the maintained mapping state. Even if they do exist on the NAT, the packet can still be filtered based on its source address and/or port. This is referred to as the endpoint filtering policy.

To measure endpoint filtering policy, a *NetPiculet* client behind NAT first establishes a TCP connection to the *NetPiculet* server and thus creates the corresponding mapping state on the NAT. Outside the NAT, the *NetPiculet* server varies the source IP address and port and sends back SYN packets to the external endpoint of the connection established earlier. The client checks if it can receive the SYN packets from the server to determine the endpoint filtering policy.

TCP state tracking: A NAT may also track TCP state and filter certain sequence of TCP packets that are considered invalid. We test the following two types of packet sequence that are critical for existing TCP NAT traversal schemes (*e.g.,* STUNT#2, NATBlaster, and P2PNAT):

NAT Mapping	# carriers
Independent	30
Address and Port$_1$	15
Connection$_R$	19
Connection$_T$	5
Address and Port$_T$ & Connection$_T$	3
Total	72

Table 3: NAT mapping results for 72 cellular networks that deploy NAT boxes.

- *SYN-out SYN-in* tests if a NAT allows an incoming SYN packet after an outgoing SYN.
- *SYN-out SYN-ACK-out* tests if a NAT allows a client to send out a SYN-ACK packet after sending a SYN packet.

Filtering response: Recall that at the beginning of NAT traversal, each client sends out a SYN packet to create a connection mapping on its own NAT. However, when the SYN packet reaches the other NAT, it could be dropped or trigger a TCP RST or ICMP packet. The returning RST or ICMP packet may cause the newly-created mapping to be removed from the NAT and thus disrupt the NAT traversal process.

To test the filtering response, the *NetPiculet* server sends SYN packets to some random ports on the external IP address of a pre-established TCP connection. If no response packet is received, it means the NAT box drops the packet.

Packet mangling: A NAT may mangle TCP packets by modifying sequence number, which is vital to STUN#1 and NATBlaster. To test if a NAT box implements this, the *NetPiculet* client sends several SYN packets to the *NetPiculet* server with a predefined sequence number. The server can check if the sequence numbers in the received packets match the predefined value.

3.2 NAT Characteristics

3.2.1 NAT Mapping Results

Table 3 shows the NAT mapping results of 72 cellular carriers deploying NAT. A majority of them exhibit either *Independent* or *Address and Port*$_1$ mappings, which are quite easy to traverse. Surprisingly, 19 of them (26.4%) fall into *Connection*$_R$, which cannot be handled by most existing NAT traversal schemes. This percentage number is significantly higher than the 0.5% number for home NAT boxes [21]. NATBlaster [15] proposes to use the birthday paradox to deal with *Connection*$_R$. However, NATBlaster requires root access to the mobile device, which is usually not easily obtained on mobile devices. The results suggest that NAT traversal is much more challenging in cellular networks.

Time-dependent NAT mapping: There are 8 carriers that were initially classified as *Connection*$_R$ or *Address and Port*$_1$. However, a closer examination revealed that the external ports of the 24 connections created during the test are well correlated with time. To further validate this observation, we conducted controlled experiments in Carrier B's network. We have a mobile client create new connections to the *NetPiculet* server with a random interval between 0 and 60 seconds. Each new connection uses the same destination port but a different local port so that it will be assigned a new external port by NAT. The experiment lasts half an hour. Figure 2 plots the start time and external port number of each connection. It is clear that the external port number increases linearly with time, and it restarts from a small number after reaching the maximum value.

In Table 3, we use *Connection*$_T$ and *Address and Port*$_T$ to denote this new type of mapping. The former means the external port increments with time for each new connection. The latter means

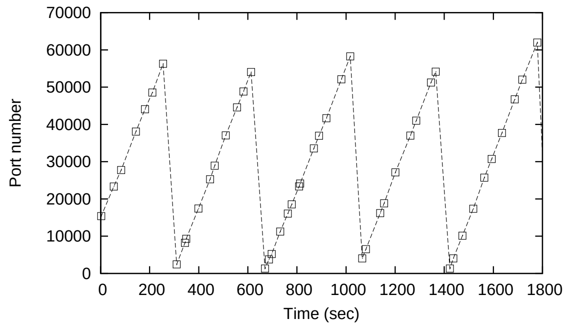


Figure 2: External port number linearly increases over time for new connections in Carrier B's network. (Each square denotes a new connection.)

the external port increments with time if a new connection has a different destination endpoint from the previous connection. Unaware of this new mapping type, existing NAT traversal schemes can easily misclassify it as *Connection*$_R$. To traverse *Connection*$_R$, NATBlaster [15] proposes to use the birthday paradox by initiating 439 connections. Besides the heavy-weight design, it also requires root access to the mobile devices, which normal users do not have. In contrast, as shown in §3.3, by leveraging the time-dependent information we propose a light-weight traversal scheme for time-dependent mapping, which only requires initiating a few connections without any root privilege.

Multiple NAT boxes for a single client: Interestingly, Table 3 shows that the *NetPiculet* clients in 3 carriers encounter two different mapping types. Sometimes, the connections from the same client are even assigned two distinct external IP addresses although the client's private IP address remains the same. To validate this observation, we conducted controlled experiments in Carrier B whose clients experience both *Connection*$_T$ and *Address and Port*$_T$ mappings. By establishing many connections with different five tuples between a mobile client and the *NetPiculet* server, we found that the mapping type encountered by a connection is determined by its source and destination ports. If the two port numbers add up to an even number, the connection experiences *Address and Port*$_T$ mapping. Otherwise, the mapping type is *Connection*$_T$. Similarly, a connection is mapped to one of two external IP addresses depending on the sum of the source and destination port numbers.

NAT boxes are usually deployed at the top of the network infrastructure hierarchy of a cellular network, *e.g.,* near GGSN (Gateway GPRS Support Nodes) [4]. They serve the aggregate traffic from a large number of mobile users. It is likely that a carrier deploys multiple NAT boxes to balance their load based on the hash of the five tuple (source and destination ports in Carrier B). This again may interfere with NAT traversal and will be further discussed in §3.3.

3.2.2 Other NAT Results

Because the *NetPiculet* client requires raw socket to measure the remaining four NAT properties, we are only able to report the results from users who run *NetPiculet* client with root privilege in 20 cellular networks shown in Table 4.

Endpoint filtering: All the 20 carriers (100%) employ *Address and Port* filtering, which requires the source IP address and port of an incoming packet to exactly match the destination IP address and port of an existing connection in NAT. This is the most restrictive filtering policy. Given that only 82% of home NATs use *Address and Port* filtering [21], NAT traversal in cellular networks appears to be more challenging.

TCP state tracking: 18 carriers allow an incoming SYN packet

NAT property	# carriers
Endpoint filtering	20 use Address and Port
SYN-out SYN-in	18 allow, 2 disallow
SYN-out SYN-ACK-out	18 allow, 2 disallow
Filtering response	19 silently drop, 1 sends RST
Packet mangling	0 modifies sequence number

Table 4: NAT properties for 20 cellular networks.

after an outgoing SYN packet. 18 carriers allow an outgoing SYN-ACK packet following an outgoing SYN packet. One carrier in Hungary does not allow either sequence of packets. This suggests that the two underlying mechanisms (TCP simultaneous open and packet forging) of the existing NAT traversal schemes [21] are still viable in most cellular networks that we studied.

Filtering response: 19 carriers drop unsolicited incoming SYN packets silently while only one carrier in France responds with RST packets. This means the NAT traversal process is unlikely to be disrupted by the removal of an existing connection mapping triggered by a RST packet.

Packet mangling: None of the 20 carriers modifies TCP sequence number. Hence, packet mangling does not appear to be an issue for NAT traversal in cellular networks.

3.3 Implications on NAT Traversal

As we have explained, the connections from a single mobile client may be handled by more than one NAT. Even worse, these NATs may have different mapping types and external IP addresses. Because the existing NAT traversal approaches assume a client is behind only one NAT box [21], they will have trouble in determining the correct NAT mapping type and external IP address. For instance, suppose a client obtains two different external IP addresses (IP_s and IP_c) when connecting to the third-party server S and the other client C respectively. Then C will attempt to establish a connection to IP_s learned from S. This attempt will fail because the NAT we studied applies *Address and Port* filtering and only permits connection to IP_s from S.

Handling NAT load balancing: network load balancing is typically performed at the flow level, *e.g.,* based on the five tuple hash, to avoid packet reordering within a flow [16]. If the five tuple hash is configured statically, we need to discover the hash rule and only establish connections through the same NAT during NAT traversal. For instance, in Carrier B, we can ensure that the connections traverse only one NAT by making the sum of source and destination port numbers an even (or odd) number. We consider it as 2-way balancing solely based on our measurement observation. This problem becomes much more challenging if the hash function is more complicated, *e.g.,* based on the real-time load or n-way balancing where (n>2). We leave it for our future study.

Traversing time-dependent NAT: Since the external port number linearly increments over time on this type of NAT, we can predict the port number according to the port increment rate r and elapsed time t between a new and an old connection. We now describe how to establish a connection between client$_a$ and client$_b$ behind an independent and a time-dependent NATs. Suppose client$_b$ learns the external endpoint (1.2.3.4:5678) of client$_a$ through a third-party server S. Client$_b$ first creates two connections to S and 1.2.3.4:5678 respectively. It also records the time interval t between the two connections and sends the predicted port number increase ($\delta = r \times t$) to S. Upon receiving this information, S relays the external port number of client$_b$'s first connection (*e.g.,* 9000) and δ to client$_a$. Client$_a$ then attempts to connect to client$_b$ from the same external endpoint (1.2.3.4:5678) to multiple destination ports in the range of $[9000+\delta-n, 9000+\delta+n]$ (starting from the middle), until

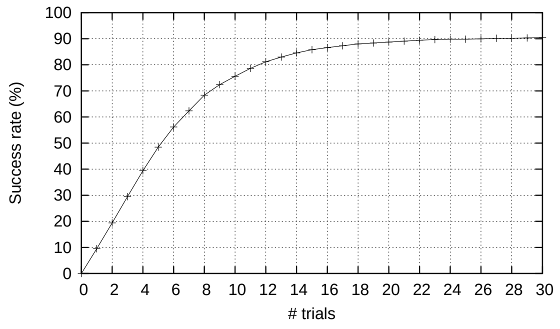


Figure 3: Success rate for traversing independent and time-dependent NATs.

it successfully hits the true external port value of client$_b$'s second connection. Here n is introduced to deal with imprecision in port prediction and is set to 15 by default.

Note that client$_a$ must be behind an independent NAT for the traversal to succeed. Because it is difficult to precisely predict the external port of client$_b$'s second connection, client$_a$ has to try multiple destination ports when connecting to client$_b$. If client$_a$ is behind other types of NAT, *e.g., Address and Port*$_\delta$ or *Connection*$_\delta$, the external port of client$_a$'s trial connections will keep changing. Thus, a trial connection will be rejected by client$_b$'s NAT (because of *Address and Port* filtering) even if it does hit the right destination port.

We implemented this traversal scheme on Android and studied the success rate for establishing a connection between a smartphone in Carrier A and one in Carrier B. We learned from our previous measurements that the two carriers use independent and time-dependent NAT mappings respectively. Figure 3 shows the success rate of NAT traversal through 1,000 runs. The x-axis is the number of trials client$_a$ makes before successfully establishing a connection with client$_b$. It also roughly corresponds to the number of seconds the NAT traversal takes to succeed since we use a one-second timeout between two consecutive trials. The figure shows an 80% success rate within 12 seconds. It is noteworthy that we also tried out two popular P2P video chatting applications on Android, Fring [5] and Tango [6], none of which can successfully establish direct TCP connection in the same experimental environment.

3.4 Summary

To summarize our study on NAT in cellular networks.

- We discovered a previously-unknown NAT mapping that linearly increments port numbers with time. We designed and implemented a corresponding traversal scheme, which can succeed with high probability.
- A single mobile client may encounter more than one NAT mapping in cellular networks, likely due to load balancing, which requires extra care to ensure the same mapping during the traversal process.
- 19 out of the 72 carriers we studied assign random ports for connections on their NATs, which is the worst scenario for NAT traversal. To better support P2P applications, we suggest operators and vendors implement or configure consistent mapping on their NATs.

4. FIREWALL POLICIES

Firewalls are essential for carriers to protect their networks. As mentioned earlier, without such protection, attackers can more easily launch various attacks targeting either the network infrastructure [27] or mobile devices [33]. Even though some carriers deploy NATs, which naturally protect internal networks by dropping unsolicited incoming packets that do not have existing NAT mappings, they often still configure firewalls to further enhance network security (*e.g.,* by performing deep packet inspection and tracking TCP connection state). In this section, we study the key properties of firewalls in cellular networks from the perspective of their effectiveness to guard against potentially malicious traffic as well as policies that can negatively impact performance and energy. Indeed, we uncovered policies and configuration settings that unexpectedly impact both performance and energy.

4.1 Methodology

NetPiculet consists of several firewall tests targeting different aspects of the firewall. We describe the motivation and methodology of those tests in this section. We study the properties of firewall policies in two aspects: IP spoofing and stateful firewall. IP spoofing policy is one that can be potentially overlooked but in fact can undermine the purpose of NAT and firewall to protect mobile phones from probing and scanning. Stateful firewall can impact reachability, affecting both security and performance.

4.1.1 Testing IP Spoofing

Previous study on Internet IP spoofing [14] has shown that IP spoofing is still widely allowed. However, it unclear how prevalent it is in cellular networks. The unique implication of IP spoofing can be summarized in the following story [31]: There has been a well-known attack on jailbroken iPhones' sshd program where a default password is used. A worm scanned the public IP ranges and propagated via this weakness and eventually infected more than 21,000 phones.

Obviously, networks with NATs or firewalls that block unsolicited incoming traffic are not vulnerable to this worm, as external hosts cannot scan phones inside the cellular network. The ability to perform IP spoofing completely changes this by allowing a single phone to punch many holes on the NATs and firewalls, thus allowing external servers to scan the phones in the internal networks. More specifically, as long as the attacker controls a phone in the network, she can send spoofed SYN packets to an attacker-controlled server which can then send SYN back to the spoofed IP (victim). Besides scanning, attackers can also launch battery draining attacks [33] by continuously sending packets to the target phones.

To determine whether a given phone's network allows IP spoofing, *NetPiculet* client modifies the source IP field in the IP header for outgoing TCP SYN packets. We randomly choose 10 source IP addresses in its own /24 and /16 IP prefixes and send a SYN packet using the chosen IP. The SYN packet is to mimic a legitimate TCP connection establishment request initiated by a device inside the cellular network which will be allowed by the firewall and NAT. We also studied UDP packets. Note that other packets are not of interest as they can be easily blocked by NAT. If the *NetPiculet* server sees the corresponding public IP address translated by NAT, it records the spoofed private IP address, obtained from the payload, and the mapped public IP address pair and consider the carrier as permitting IP spoofing.

4.1.2 Testing Stateful Firewall

Stateful firewalls can determine legitimate packets under different connection states (*e.g.,* states associated with a TCP connection). In each state, only packets matching certain criteria are allowed to traverse through the firewall (for either incoming or outgoing traffic). For mobile carriers deploying NAT boxes (Carrier A and Carrier B), external traffic cannot reach devices directly without devices initiating an outbound connection first. Hence, the *NetPiculet* client and server have to coordinate to reveal the firewall policies for those carriers. We first perform this test in a local environment setting to help us understand properties of interest, *i.e.*, those impact performance and security. We subsequently deployed only tests associated with such properties on *NetPiculet* client to ensure they finish quickly on end-users' phones.

Note that we verified using local experiments, the packet blocking behavior imposed by the firewall is applicable to both incoming and outgoing traffic. However, the net effect of which incoming and outgoing packets are allowed may not be symmetric because of the presence of NAT. Also, the buffering behavior described later in §4.1.2 applies to both directions as well.

Due to the popularity of TCP, we conduct controlled experiments between *NetPiculet* client and server to infer the firewall policies in various TCP states such as SYN-SENT, SYN-REVD, ESTABLISHED *etc.* Specifically, in each state, we perform the following tests:

1. Single packet filter test: Probing using SYN, SYN-ACK, ACK, DATA, FIN, RST packets from the *NetPiculet* server to check if they are allowed.
2. Timeout test for each state: If any such probing packet is allowed, we infer the timeout value in each state by incrementally increasing the interval between consecutive probes.
3. Sequence (SEQ) window test for state transition: for packets that trigger state transitions, we probe with a range of SEQ number and ACK number to verify if they can still enable the correct transitions.
4. SEQ window test for ESTABLISHED state: we vary the SEQ number of DATA packets via binary search to infer the SEQ window size adopted by the firewall.

Given that some of these tests are quite time-consuming, we selected a subset, expected to significantly impact application security and performance, as shown in §4.1.3 and §4.1.4, to be integrated into the *NetPiculet* client.

4.1.3 Testing TCP Connection Timeout

Through local experiments, we found that firewalls deployed by Carrier B have a very short timeout for idle TCP connections. Thus TCP connections of chat applications are frequently timed out. Such applications require long-lived connections, *e.g.,* MSN Talk on Android. To understand the timer values of firewalls, we integrate the timeout inference test into *NetPiculet*.

To infer the timer value on the firewall that times out idle TCP connections, *NetPiculet* client creates multiple connections in parallel to *NetPiculet* server without enabling keep-alive option. Each connection sends a message to the server after a specific amount of idle time. The server responds to the client upon receiving the message. If the connection is still alive when the message is sent, the client should receive the response message from the server. Otherwise, the connection is timed out by the firewall. *NetPiculet* client tries four idle time intervals: 5, 10, 20, and 30 minutes, to bound the inferred timeout timer value within these ranges.

4.1.4 Testing Out-of-Order Packet Buffering

During our controlled experiments to test the SEQ window size, we accidentally discovered that some firewalls buffer out-of-order TCP packets. We inject packets with large sequence numbers to check if the firewall has a SEQ window such that packets with sequence number out of the window would be dropped. Initially we thought packets were dropped whenever the other end did not receive them, after accounting for network loss. We then discovered that after those missing packets arrive, the injected packets with large sequence numbers are released, indicating the buffering behavior. This ensures packets are delivered in order, most likely motivated by the need to perform deep packet inspection (DPI) to detect security attacks, a feature we found to be available in some commercial routers [10]. As we discuss later in §4.3.2, this buffering behavior has significant impact on TCP performance in the presence of packet loss. We implement this test in *NetPiculet*.

To realize this test for detecting buffering in both directions, we intentionally drop one packet from the many packets on the sender side, and observe whether there will be any ACK packet received for beyond the lost packet. If there is no response after certain time window (*e.g.,* 5 seconds), we conclude the firewall does buffer out-of-order packets. 5 seconds is chosen as it is sufficiently large to account for some of the performance impact on TCP, as described in §4.3.2.

4.2 Firewall Characteristics

We next describe the firewall testing results, including results from both local controlled experiments and tests implemented in the *NetPiculet* client deployed in the wild.

4.2.1 IP Spoofing

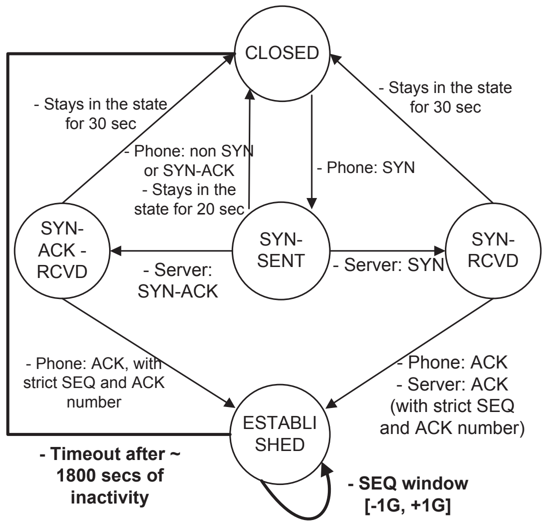
For local experiments, we studied one of the carriers found to allow IP spoofing to quantify how fast one can scan inside a cellular network. We scanned a randomly chosen /24 IP range as fast as possible by sending spoofed SYN packets back to back. We repeat the experiment for 10 times, and the result shows that it takes on average 2 – 3 seconds for all IP spoofing packets to be received by the external server, and then 3 – 4 seconds are spent for the servers to scan and wait for the response. Note that to receive packets, mobile devices have to obtain radio resource first, which normally takes 1 – 2 seconds [32]. The probing packets will likely experience such delay before reaching the scanned devices. In practice, if the attacker is continuously scanning a large range of IPs, the throughput of spoofing is usually the bottleneck, which we found to be around 90 packets/second. Compared to data packets, we do not find any signs of rate limiting on the spoofed SYN packets, as their throughput numbers are very close. In our scanning of the /24 prefix, we found 54 hosts on average responding to the probing.

NetPiculet was able to perform the IP spoofing test in 60 cellular networks, among which 4 allow IP spoofing. Despite the low absolute number, the percentage is still significant: 6.7%. Further, all four carriers are large cellular ISPs, covering U.S. and Europe respectively. Thus they can be profitable targets for attackers. Given that specifications and commercial products are already available [11, 9] that can prevent IP spoofing at finest granularity (using subscriber information IMSI/MSISDN available in each packet header), it is surprising to see that these four large carriers allow IP spoofing.

We further investigate the range of spoofable IP addresses (/24 and /16). Surprisingly, each of the four carriers allows /24 and /16 to be spoofed. This is likely due to the fact the NAT box is shared at the very high level of the network infrastructure inside the cellular network, usually co-located with GGSN [4]. We expect IP spoofing prevention to be enforced at GGSNs, as confirmed by commercial products available [9] that advertise such features.

4.2.2 Stateful Firewall

We observe that the stateful firewall policies usually do not strictly follow the TCP specifications (*e.g.,* packets with an invalid ACK number can still traverse the firewall). There are two key reasons for this: (1) It is expensive to check all the low-level details to the degree that a host networking stack does. (2) There are many TCP variants; thus, firewall policies should be flexible enough to accommodate most if not all of them. The latter is an inherent problem of network-based firewalls and intrusion detection systems [36]. In our study, we found significant variations among the stateful firewall policies adopted by various carriers. Next we explain in detail the key properties of the inferred TCP state machine on the firewalls, which serves as the necessary prelude for discussing their implication.



CLOSED
- Stays in the state for 30 sec
- Stays in the state for 30 sec
- Phone: non SYN or SYN-ACK
- Stays in the state for 20 sec
- Phone: SYN
SYN-ACK - RCVD
SYN-SENT
SYN-RCVD
- Server: SYN-ACK
- Server: SYN
- Phone: ACK, with strict SEQ and ACK number
- Phone: ACK
- Server: ACK (with strict SEQ and ACK number)
ESTABLISHED
- Timeout after ~ 1800 secs of inactivity
- SEQ window [-1G, +1G]

Figure 4: TCP State diagram of Carrier A's firewall.

Figures 4 shows the TCP state diagram we inferred for the firewalls in Carrier A. Carrier B has a similar diagram except for some minor differences, which can still have significant implications for applications as explained below.

Carrier A's firewall: In Figure 4, we infer that there are 5 states, slightly different from the TCP states defined on end-host. The figure is fairly self-explanatory. We only highlight the key interesting state transitions identified, as indicated with bold lines.

There are several observations associated with the ESTABLISHED state. We found that even after a legitimate close sequence (FIN, FIN-ACK, ACK), one can still send packets into the cellular networks from external networks, indicating that the firewall did not go back to the CLOSED state. We discuss implications of this unexpected behavior in §4.3.4.

The ESTABLISHED state has an inactivity timer associated with each TCP connection. If the timer expires, the connection state is removed from the firewall and no additional packets are allowed until a new connection is established (starting over from TCP's three-way handshake). This timer value is 1800 seconds which we compare with another carrier and discuss its impact in §4.3.1.

In the ESTABLISHED state, the firewall checks the TCP SEQ of all packets from either direction to ensure that they fall into a window of SEQ from previously seen packets. The problem is that this window size δ seems to be statically configured to liberally accommodate packet reordering. As an example, if the server's first packet has SEQ of n, the second packet must have a SEQ from $[n-\delta, n+\delta]$. The range $[n-\delta, n)$ is to allow retransmission of lost packets. The range $(n, n+\delta]$ is to allow out-of-order packets. Dropping them would cause unnecessary retransmission from the sender. Surprisingly, we found that δ can be as large as 1G (and sometimes 128K), depending on the types of firewall encountered. We also found that different IP ranges go through different firewalls likely due to different network paths. 128K is normally good enough in today's 3G networks as the $delay \times bandwidth$ is typically capped at 0.2s $\times$ 4Mbps = 100KB, which is smaller than 128KB. However, for future networks such as 4G with higher net-

Timeout (min)	(0,5]	(5, 10]	(10, 20]	(20, 30]	(30, ∞)	Total
# carriers	4	7	6	8	48	73

Table 5: Measured TCP timeout timers in cellular network firewalls.

work capacity, a loss of one packet may cause its following packets overflow the 128KB window and be unnecessarily dropped.

Also, another important finding is that for packets that fall within $(n, n+\delta]$, they are not delivered immediately. Instead, they are buffered until in-order packets arrive. This behavior actually leads to security issues as well as performance impact due to unexpected interaction with TCP stack as discussed later in §4.3.3 and §4.3.2.

Carrier B's firewall: Its state diagram closely resembles that of Carrier A's. One major difference is that its firewall is much less restrictive in determining if a TCP connection is established (from SYN-ACK-RCVD and SYN-RCVD to ESTABLISHED). As long as the SEQ of the ACK packet falls into a (-128K, +128K) window from the previous SEQ, the firewall advances the state to ESTABLISHED. This may allow fake connections to be established on the firewall. Also, unlike Carrier A's firewall, Carrier B's firewall does not buffer out-of-order packets.

In the ESTABLISHED state, the inactivity timeout value is much smaller (255 seconds) compared to that of Carrier A's, more seriously affecting long-lived connections such as push-based services as elaborated in §4.3.1. In addition, the SEQ window of 128K for the ESTABLISHED state is much smaller compared to Carrier A's.

4.3 Implications and Recommendations

Given our previous discussions on several interesting observations from the firewall policy inference results, we elaborate on their impact on application performance and security. To quantify such impact we conduct further controlled experiments. We also describe the important implications and recommendations for the firewall.

4.3.1 Energy Impact of TCP Connection Timeout

Long-lived connections (*e.g.,* push-based services such as email) assume persistent TCP connections. Given the inactivity timer on the firewall, periodic keep-alive messages must be sent to maintain the connection. Otherwise, applications' ability to receive timely notifications can be impaired. The default TCP keep-alive timer of 2 hours [7] is clearly too large to help maintain the connection. In fact, we tested MSN Talk, one of the MSN messenger applications on Android, and discovered that Carrier B's firewall terminates idle connections (using spoofed RST packet) after 255 seconds, forcing the application to re-establish the connection immediately to restore the service. This may appear to be equivalent to sending keep-alive messages but transmitting more data and incurring additional delays.

Table 5 summarizes the measured TCP timeout timers of various carriers tested by *NetPiculet*. Among the 73 carriers measured, 11 have a timer shorter than 10 minutes, four of them with a timer of only at most 5 minutes. Since radio resource is allocated every time a keep-alive packet is sent for an idle connection, over time such packets can consume significant amount of energy for end devices and also incur high signaling overhead for cellular networks. To quantify the energy impact, we assume a long-lived connection which regularly sends keep-alive to reset the firewall timer right before it is about to expire. We use existing cellular interface power models [32] to estimate the ratio of energy spent on keep-alive per day for a common smartphone battery capacity (1350 mhA). While the actual energy consumption may vary across different networks and devices, the overall conclusion that significant energy waste due to small timers shown in Figure 5 is generally applicable. For example, more than 17% of the battery capacity is spent on keep-

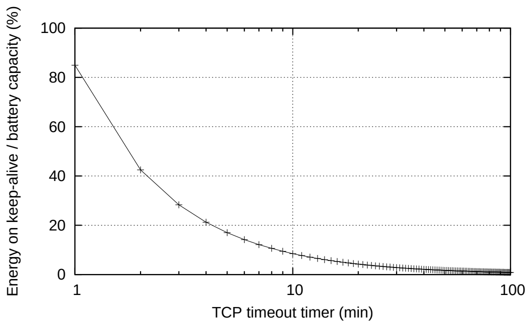


Figure 5: The ratio of energy of sending keep-alive messages per day to the battery capacity (1350 mAh).

alive everyday when the timer is less than 5 minutes, while it drops sharply as the timer becomes longer. The waste is reduced to less than 2% for 30 minute timers.

Network operators must consider an inherent tradeoff when setting the timer for terminating idle TCP connections. On one hand, larger timer values reduce the energy cost and signaling overhead caused by keep-alive packets to maintain the connection. On the other hand, they use up more memory at the firewalls for keeping track of existing connections. Firewalls generally have limited capacity to process concurrent connections; thus, carriers have incentives to set smaller timers to terminate idle connections in order to support more concurrent active users.

It is of interest to investigate the best way for developers to implement push-based services which are becoming more popular. One approach is to use the push service framework in the SDK. Both Apple and Google provide such framework API, which developers can use to implement push based services [2, 3]. The way it works is that when the third-party application server α has data to push to the mobile device, it informs the push notification server β, which belongs to Apple or Google, β then sends a notification message to the targeted mobile device through a long-lived TCP connection maintained by the framework. The framework provides two benefits. First, since the framework takes care of maintaining the long-lived connection, developers do not need to deal with diverse timer values in different networks. Second, since notifications of different applications share the single long-lived connection, energy cost on the mobile device is lower compared with the case where each application has its own long-lived connection.

It is interesting to note that in Carrier B's network, port 5228 stands out from others with a larger timeout value of 1600 seconds compared to the default of 255 seconds. Port 5228 is actually used by Google's push service framework [3] to send notifications. We suspect that Carrier B intentionally makes this optimization for Google, which reduces the overhead of re-establishing the connection repeatedly. However, verified by our experiments, Carrier B only uses the port number to make the distinction. Therefore, developers can take advantage of this port number to obtain a longer connection timeout value.

4.3.2 Performance and Energy Impact of Buffering

As described before, some firewalls buffer out-of-order packets within a configured range of sequence numbers and deliver them when in-order packets arrive. Two legitimate cases trigger such buffering behavior: 1) packet loss; 2) packets re-ordered along the path. Normally the first scenario happens much more frequently than the second one. It is worth mentioning that Carrier A, a U.S. nation-wide cellular service provider, shows the buffering behavior, and it buffers out-of-order packets for more than an hour.

Disabling "TCP fast retransmit". A major problem with such buffering behavior is that it disables TCP fast retransmission, which

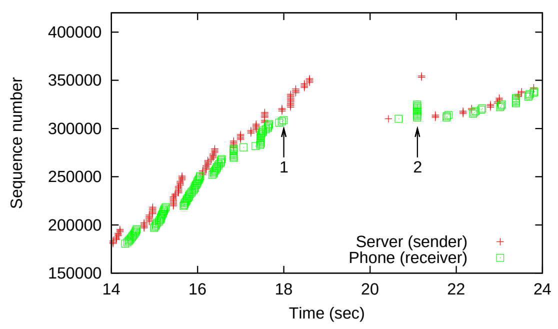


Figure 6: Sender's TCP sequence numbers monitored on server and client in Carrier A's network.

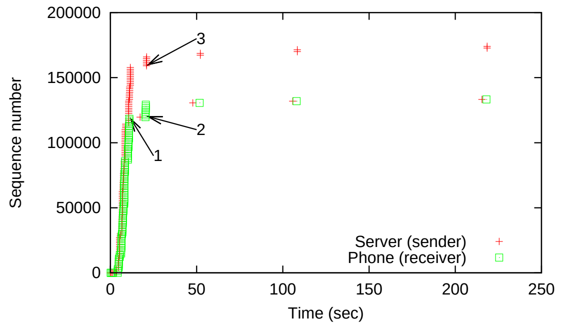


Figure 7: The impact of firewall buffering on TCP flow when FRTO is implemented on the sender side.

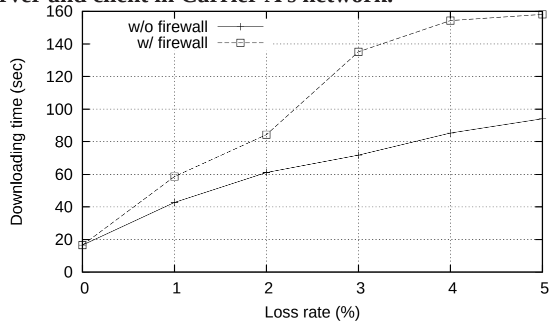


Figure 8: The average downloading time for 1MB file under different loss rates.

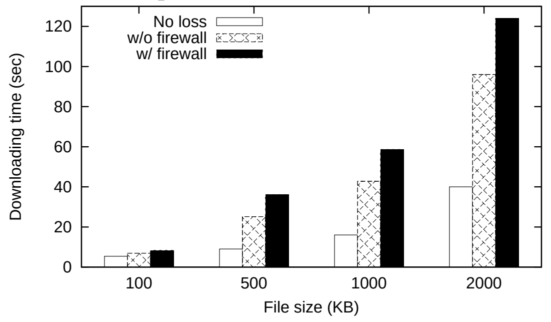


Figure 9: The firewall impact on downloading time for different file size under 1% loss rate.

is used to reduce the time a sender waits before retransmitting a lost segment and without which a sender has to use a timer to detect any lost segment. It works as follows: if a TCP sender receives three duplicate acknowledgements with the same acknowledge (ACK) number, *i.e.*, a total of four ACKs with the same ACK number, the sender can be reasonably confident that the segment with the next higher sequence number was dropped. The sender then retransmits the packet before waiting for its timeout.

Buffering disables fast retransmission because the sender is unable to observe any duplicate ACKs. For instance, if packet 1 is dropped and packets 2, 3, 4 and 5 arrive. The firewall buffers packets 2 – 5 since it never sees 1. This way, the receiver does not see any packets and cannot send any response. As a result, the sender has to resort to timeout-based retransmission.

Figure 6 illustrates the problem using a packet trace from our local experiment in Carrier A's network. A packet loss occurs at arrow 1, and the sender keeps sending more packets to fill the congestion window. Since the firewall does not see the lost packet, it buffers all the later packets, which are considered out-of-order. The sender thus is unable to get any response from the receiver and has to fall back to time-out based retransmission noted as the gap in the figure.

Bad interaction with TCP "Protect Against Wrapped Sequence number (PAWS)". At arrow 2 in Figure 6, the firewall released all the buffered packets upon seeing the retransmitted lost packet. However, those packets were retransmitted later by the sender through slow start, even though they have successfully reached the receiver. We investigated this behavior and found the explanation in RFC [24] where PAWS is defined. PAWS uses TCP timestamp to reject potentially old packets that might corrupt an ongoing TCP connection. With TCP timestamp option enabled, the timestamp on the buffered packets was strictly smaller than the timestamp on the retransmitted packet. Consequently the receiver rejected the buffered packets by sending back duplicate ACKs which then triggered the following retransmission. The pile of packets at arrow 2 also indicates that the firewall has a limited buffer size. We measured the buffer size to be 10 packets per TCP flow and confirmed the setup with the documentation of commercial routers [10].

To quantify the impact of disabling fast retransmit on application performance and energy, we perform controlled experiments to measure the time and energy consumption for downloading files of different size from our server. We use WiFi to emulate the delay of 3G network (400ms RTT derived from a recent study [23]) to have full control over the loss rate and firewall configuration. On the hosting server, a small Perl script is used to control the loss rate by randomly dropping packets with certain probabilities. To emulate the firewall buffering behavior for the same TCP flow, the script buffers the first 10 packets (as we observed in Carrier A's network) with sequence number larger than the packet previously dropped intentionally and drops the remaining ones. Upon receiving a retransmitted packet with the same sequence number as the intentionally dropped packet, it releases all the buffered packets. We vary the loss rate from 0 to 5% and download each file 10 times under each setting.

Figure 8 shows that even with a loss rate of 1%, the downloading time for a 1MB file is increased by 50% due the firewall buffering behavior. And the degradation is worse with higher loss rate. Under the same loss rate, the buffering has less impact on smaller files. As shown in Figure 9 compared with normal loss without firewall, the firewall incurs 21% extra delay for a 100KB file while 44% increased delay for 500KB files for a loss rate of 1%. However, a recent study [1] points out that TCP-based streaming applications that typically send large amount of data contribute to majority of smartphone traffic. So the overall network performance could be significantly affected by firewall with such buffering behavior. Note that the cellular radio interface always stays in high power state during the entire download process. Even when there is no traffic during RTO, the interface still remains in high power state due to the tail time [32]. The energy consumption for downloading the same amount of data increases almost linearly with the downloading time.

Bad interaction with "TCP Forward RTO-Recovery (F-RTO)" Since PAWS makes the buffered packets useless to the receiver, we were wondering whether letting the receiver accept the buffered packet would reduce the negative impact of buffering on TCP performance. Therefore, we disabled the TCP timestamp option on the sender side and repeated the experiments to measure the downloading time. Surprisingly, we found the TCP connection almost hung up after a single packet loss. As we figured out later, it is due to the bad interaction between buffering and F-RTO, a TCP extension defined in RFC [34].

F-RTO is designed to detect spurious TCP retransmission timeouts which cause unnecessary retransmissions when no segments are lost. After a spurious retransmission timeout, the late acknowledgments of the original segments arrive at the sender, usually triggering unnecessary retransmissions of the entire window of segments. This can happen when, for instance, some mobile networking technologies introduce sudden delay spikes on transmission due to actions taken during a hand-off.

When a retransmission timeout (RTO) occurs, the F-RTO sender retransmits the first unacknowledged segment as usual. Deviating from the normal operation after a timeout, it then tries to transmit new, previously unsent data, for the first ACK that arrives after the timeout given that the ACK advances the window. If the second ACK that arrives after the timeout also advances the window, *i.e.*, acknowledges data that was not retransmitted, the F-RTO sender declares the RTO as spurious and exits the RTO recovery. However, if either of the next two ACKs is a duplicate ACK, no sufficient evidence is found for a spurious RTO. The F-RTO sender retransmits the unacknowledged segments in slow start similar to the traditional algorithm.

This behavior is illustrated in Figure 7. A packet loss occurred at arrow 1, marked on the figure. The sender keeps sending about 30 more packets to fill the congestion window, 10 of which are buffered at the firewall, the rest dropped due to buffer space constraints. At arrow 2, the sender retransmits the lost packet after retransmission timeout which triggers the buffered packets to be released (since it is now in-order). The receiver then receives 11 packets including the retransmitted one. At this point, the sender behaves according to F-RTO. At arrow 3, instead of retransmitting unacknowledged packets in the congestion window, it now tries to transmit new, previously unsent data. If no duplicate ACKs are received, the sender can save the unnecessary effort of retransmitting the unacknowledged packets. However, this optimization fails to consider the case where the newly transmitted packets are buffered at the firewall causing no ACKs to be received. Later, the sender has to repeat the entire process of F-RTO again and further transmit new, previously unsent data. As we can see in the figure, in each round, the retransmission timeout is doubled with only one packet successfully delivered, effectively rendering the connection useless. And this is triggered by only a single packet loss shown by arrow 1.

The fundamental problem here is that this F-RTO algorithm does not expect packets to be buffered by the firewall for so long. Intuitively, one way to address this issue is to limit the buffering time to a small value (*e.g.,* $< 1s$). However, we found that firewalls with Intrusion Prevention Systems (IPS) capability must examine or perform deep packet inspection (DPI) on in-order packets before they can be delivered to provide strong security guarantees. Indeed, many such IPS boxes are sold today by major vendors such as Cisco [10]. At the moment, we did not observe other carriers implementing similar buffering policy yet, but we did observe consistent buffering behavior from 14 users in 8 U.S. states. Further, this behavior can become more prevalent in the future due to increasing security concerns for mobile cellular networks. Since F-RTO was implemented in Linux kernel 2.6.22 in 2007, the majority of Linux servers are likely impacted.

In summary, regardless of the TCP timestamp option, the buffering behavior always disables the TCP fast retransmission which can degrade TCP performance significantly. If the TCP timestamp option is enabled, it can have bad interaction with PAWS. Without the TCP timestamp option, it can have adverse interaction with F-RTO, seriously rendering the TCP connection completely useless.

4.3.3 Exploiting Large Sequence Number Window

We have discussed the property of a SEQ window $[-\delta, +\delta]$ on the firewall to allow legitimate TCP packets through while blocking potentially malicious packets. In general, this window size should not be much bigger than the receiver window size, which is mostly 64K or 128K in most smartphone operating systems today (with the TCP window scaling option). For Carrier A, the window of [-1G, +1G] is too large. This allows arbitrary data packets to be injected assuming the knowledge of the 5 tuple, since the data packets are being buffered instead of being dropped immediately.

As the firewall can buffer 10 packets at most, an attacker can equally spread 40 data packets across the 4G SEQ space where at most 10 of them are buffered at (0, +1G]. Some of the buffer space can be occupied by legitimate packets that arrive out of order. To simplify the discussion, we assume for now that attackers can occupy all the 10 buffer slots. In this case, a TCP connection will always be injected with a packet with any payload within at most 100MB of data transfer. On average, about 50MB of data transfer is needed. The likelihood of success is much higher if this attack is attempted concurrently on many users and at different times of the day. Given that it takes only 40 packets for one trial, it is fairly cheap to blindly launch such attacks. For example, every 500 trials, an attacker can with high probability succeed at least one connection within 500KB of data transfer. Therefore, the large sequence number window property of such firewalls effectively amplifies the severity of blind data injection attacks [12] against TCP. TCP connections which transfer more data are especially vulnerable. The effect of this attack varies depending on different types of applications. For file downloading, it can corrupt the file content and render it unusable or even dangerous (imagine an executable file with altered instructions). For video streaming, however, a single corrupted packet can be tolerated by the video codec without much problem.

4.3.4 Flaws with Closing TCP Connections

For Carrier A, when the state transitions to the CLOSED state, the server can still send SYN packets continuously to drain the phones' battery even though the phone has invoked socket `close()` on its side. This is because some state is not properly de-allocated by the firewall and NAT, which operate independently. More specifically, the NAT still keeps an entry for the previous connection which gets cleaned up only after 20 seconds of timeout if it sees FIN packets. It is easy to keep resetting the timer by sending a packet every 20 seconds. The firewall always allows an incoming SYN packet as it treats it as a new connection. By maintaining the state at the NAT and the firewall, a malicious server can keep sending packets to the phone that it previously communicated with to drain its battery, even though the phone has already closed the connection. Such battery-draining attack can cause up to 22.3 times energy consumption and deplete the battery within several hours [33].

Due to the same problem, the back-to-back connections using the same five tuple could be delayed up to 25 seconds. As a consequence, the whole process of NAT mapping discovery (described in §3.1.1) could take more than three minutes. Comparing the packet traces dumped from both the server and client, we noticed that the SYN packets sent by the client to create new connections are dropped in the network during the delayed time duration.

4.4 Summary

Next we summarize the key findings for the firewall study.

- We discovered that 4 of 60 cellular networks allow IP spoofing, which can make hosts vulnerable to scanning and battery draining attacks even though they are behind the firewall and NAT.
- We discovered 11 of 73 carriers set the TCP inactivity timeout to be less than 10 minutes, which can cost significant amount of energy in maintaining long-lived connections used by push-based services. For carriers, we recommend a 30-minute timeout value. For application developers, we suggest they follow the push service framework in SDK where multiple services share a single persistent connection.
- We discovered the TCP out-of-order buffering behavior in some firewalls causing several unexpected interaction with common TCP behavior defined in the TCP specifications, leading to degraded performance and energy waste. Reducing the buffering time or increasing the buffer size both have negative impact on security. Thus, there is an inherent tradeoff between performance and security.

5. DISCUSSION

As we demonstrate throughout the paper, inappropriate middlebox policies have severe impact on application performance, mobile device energy consumption, and network security. With all the potential impact in mind, carriers should take the responsibility to carefully avoid bad middlebox policies in their networks. Due to some constraints (*e.g.,* cost and security), sometimes they have to configure policies which may not be friendly to applications. However, working towards a neutral network environment, instead of concealing those policies they should inform network users and application developers on the middlebox policies and their associated impact. Despite our focus on cellular network policies, our methodologies for inferring NAT and firewall policies and the identified implications of such policies are also applicable to wired networks. For example, time-dependent NAT mapping may also exist on the Internet, and the new traversal scheme we proposed can be used to traverse such NATs. Firewalls on the Internet may also buffer out-of-order TCP packets, and the resulting bad interaction with TCP extensions would still hold. We leave it as future work to monitor those policies on the wired networks. We notice that some of the bad policies are identified only in a few carriers, but our findings are still valuable in preventing operators from falling into the same pitfalls in the future. The measurement results could also change over time as carriers modify the policies or upgrade their network devices. In this case, our tool can still monitor such policy changes in the long run.

6. RELATED WORK

We heavily draw on prior work on NAT characterization and traversal. MacDonald *et al.* [28] defined in an IETF draft a few key attributes of NAT box, *e.g.,* mapping, filtering, and binding time, as well as their corresponding tests. NUTSS [22] demonstrated the feasibility of TCP NAT traversal through a combination of tricks including port prediction, IP address spoofing, and TCP packet forgery. Ford *et al.* [20] identified a few important properties relevant to NAT traversal and measured them in the wild. In parallel, Guha and Francis [21] tested nearly 100 commercial NAT boxes and compared the effectiveness of several NAT traversal solutions. Majority of the NAT boxes they studied exhibit predictability in port allocation. Makinen and Nurminen [29] measured the NAT properties from one vantage point in each of the six cellular networks. They suggested that the existing NAT traversal techniques would work in cellular networks without modifications. In contrast, we discovered some previously unknown NAT properties not handled by existing techniques and proposed new solutions accordingly. Our focus is on cellular ISPs, and by releasing *NetPiculet* to mobile users, we covered significantly more carriers and users.

While providing security and flexibility in IP address allocation, middleboxes have been shown to have undesirable performance impact on Internet users and content providers. They can inflate transaction time, disrupt long-lived connections, and reduce TCP throughput [13]. A follow-up study revealed that middleboxes may also interfere with ECN negotiation, MTU discovery, and IP and TCP options [30]. More recently, Casado and Freedman [17] investigated web clients behind middleboxes by injecting active content. They find most NATs cover a small number of hosts and their IP addresses are stable over at least several days. They also devised a suite of classifiers for detecting hosts behind middleboxes. In contrast, our work focuses on the middlebox behavior in cellular networks and reveals the implications on radio energy consumption, a unique perspective in cellular network context.

Besides the characterization work focusing on the usage of middleboxes, several studies investigated various security issues in cellular networks. Compared to their wired counterparts, cellular networks have relatively scarce network resource, making them vulnerable to various DoS attacks. Some attacks can be carried out by exploiting the complex and heavy-weight procedure of establishing a communication channel between a handset and the cellular infrastructure. Serror *et al.* found the paging channel can be easily overloaded in CDMA networks [35], Traynor *et al.* proposed attacks abusing the setup and teardown processes between mobile devices and base stations in GPRS/EDGE networks *et al.* [37], and Lee *et al.* showed well-timed traffic can trigger excessive radio resource control messages [26]. Compared with these studies that identified and exploited resource bottlenecks, our work sheds light on the feasibility of such attacks in real cellular networks that are protected by middleboxes. Moreover, we discovered some unexpected behavior of middleboxes, which can make cellular networks and devices vulnerable.

The popularity of cellular data network has inspired a growing number of measurement studies in this area as well. WindRider [8] proposed to capture network neutrality violations by monitoring application and system information on mobile devices. A previous study [23] leveraging dataset from a deployed mobile application compared the network and application performance of four major U.S. cellular carriers with the inputs from over 30,000 mobile users. Falaki [19] collected traces from 255 users and observed dramatic diversity in their usage behaviors. Our work differs in its emphasis on middlebox characteristics and implications on mobile users.

Finally, besides many previous edge network or end host based measurement systems, our work is heavily inspired by Netalyzr [25], a network measurement and debugging service that provides an extensive measurement of edge-network properties collected from a signed Java applet.

7. CONCLUSION

We have presented *NetPiculet*, the first system that effectively discovers cellular network policies focusing on NAT and firewall policies. Beyond the known properties of such middleboxes, we also focus on unique aspects of cellular networks as they impact mobile devices, mobile application developers, and cellular ISPs from perspectives of energy, performance, and security. Based on the data collected in January 2011, we studied more than 100 cellular networks around the world, illustrating how a diverse set of policies can lead to several important implications for network carriers as well as mobile application developers. Results from *NetPiculet* demonstrate the importance of understanding such policies especially due to potential complex and unexpected interaction among various policy components. In the long run, *NetPiculet* is highly valuable to help make cellular network policies less opaque and expose their impact as networks and applications continue to evolve.

8. REFERENCES

[1] Allot: Video streaming dominated mobile data traffic in first half 2010. `http://www.indiatelecomtracker.com/archives/2669`.

[2] Apple Push Notification Service. `http://en.wikipedia.org/wiki/Apple_Push_Notification_Service`.

[3] Building Push Applications for Android. `http://dl.google.com/googleio/2010/android-push-applications-android.pdf`.

[4] Cisco Mobile Exchange. `http://docstore.mik.ua/univercd/cc/td/doc/product/wireless/moblwrls/cmx/mmg_sg/cmxdesc.htm`.

[5] Fring. `http://www.fring.com`.

[6] Tango. `http://tango.me`.

[7] tcp(7) - Linux man page. `http://linux.die.net/man/7/tcp`.

[8] WindRider: A Mobile Network Neutrality Monitoring System. `http://www.cs.northwestern.edu/~ict992/mobile.htm`.

[9] Cisco GGSN Release. `http://www.cisco.com/en/US/prod/collateral/iosswrel/ps8802/ps6947/ps5413/prod_bulletin0900aecd802e0859.html`, 2005.

[10] TCP Out-of-Order Packet Support for Cisco IOS Firewall and Cisco IOS IPS. `http://www.cisco.com/en/US/docs/ios/12_4t/12_4t11/ht_ooop.html`, 2006.

[11] Interworking between the Public Land Mobile Network (PLMN) supporting packet based services and Packet Data Networks (PDN). 3GPP TS 29.061 V6.15.0, 2008.

[12] Improving TCP's Robustness to Blind In-Window Attacks. `http://tools.ietf.org/html/rfc5961`, 2010.

[13] M. Allman. On the Performance of Middleboxes. In *Proc. ACM SIGCOMM IMC*, 2003.

[14] R. Beverly, A. Berger, Y. Hyun, and k claffy. Understanding the Efficacy of Deployed Internet Source Address Validation Filtering. In *Proc. ACM SIGCOMM IMC*, 2009.

[15] A. Biggadike, D. Ferullo, G. Wilson, and A. Perrig. NATBLASTER: Establishing TCP Connections Between Hosts Behind NATs. In *Proc. of ACM SIGCOMM ASIA Workshop*, 2005.

[16] Z. Cao, Z. Wang, and E. Zegura. Performance of HashingnBased Schemes for Internet Load Balancing. In *INFOCOM*, 2000.

[17] M. Casado and M. J. Freedman. Peering through the shroud: The effect of edge opacity on IP-based client identification. In *Proc. Symposium on Networked Systems Design and Implementation*, 2007.

[18] J. L. Eppinger. TCP Connections for P2P Apps: A Software Approach to Solving the NAT Problem. `http://reports-archive.adm.cs.cmu.edu/anon/isri2005/CMU-ISRI-05-104.pdf`.

[19] H. Falaki, R. Mahajan, S. Kandula, D. Lymberopoulos, R. Govindan, and D. Estrin. Diversity in Smartphone Usage. In *Proc. ACM MOBISYS*, 2010.

[20] B. Ford, P. Srisuresh, and D. Kegel. Peer-to-Peer Communication Across Network Address Translators. In *Proc. of the USENIX Annual Technical Conference*, 2005.

[21] S. Guha and P. Francis. Characterization and Measurement of TCP Traversal through NATs and Firewalls. In *Proc. ACM SIGCOMM IMC*, 2005.

[22] S. Guha, Y. Takeda, and P. Francis. NUTSS: A SIP-based Approach to UDP and TCP Network Connectivity. In *Proc. of SIGCOMM'04 Workshop*, 2004.

[23] J. Huang, Q. Xu, B. Tiwana, Z. M. Mao, M. Zhang, and P. Bahl. Anatomizing Application Performance Differences on Smartphones. In *Proc. ACM MOBISYS*, 2010.

[24] V. Jacobson, R. Braden, and D. Borman. TCP Extensions for High Performance. `http://tools.ietf.org/html/rfc1323`, 1992.

[25] C. Kreibich, B. Nechaev, V. Paxson, and N. Weaver. Netalyzr: Illuminating The Edge Network. In *Proc. ACM SIGCOMM IMC*, 2010.

[26] P. P. Lee, T. Bu, and T. Woo. On the Detection of Signaling DoS Attacks on 3G Wireless Networks. In *Proc. IEEE INFOCOM*, 2007.

[27] P. P. C. Lee, T. Bu, and T. Woo. On the Detection of Signaling DoS Attacks on 3G Wireless Networks. In *Proc. IEEE INFOCOM*, 2007.

[28] D. MacDonald and B. Lowekamp. NAT Behavior Discovery Using STUN. `http://tools.ietf.org/html/draft-ietf-behave-nat-behavior-discovery-08`.

[29] L. Makinen and J. K. Nurminen. Measurements on the Feasibility of TCP NAT Traversal in Cellular Networks. In *Proc. of the 4th EURO-NGI Conference on Next Generation Internet Networks*, 2008.

[30] A. Medina, M. Allman, and S. Floyd. Measuring Interactions Between Transport Protocols and Middleboxes. In *Proc. ACM SIGCOMM IMC*, 2004.

[31] P. Porras, H. Saidi, and V. Yegneswaran. An Analysis of the iKee.B iPhone Botnet. In *Proc. of International ICST Conference on Security and Privacy in Mobile Information and Communication Systems*, 2010.

[32] F. Qian, Z. Wang, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck. Characterizing Radio Resource Allocation for 3G Networks. In *Proc. ACM SIGCOMM IMC*, 2010.

[33] R. Racic, D. Ma, and H. Chen. Exploiting mms vulnerabilities to stealthily exhaust mobile phone's battery. In *Proc. of SecureComm*, 2006.

[34] P. Sarolahti and M. Kojo. Forward RTO-Recovery (F-RTO): An Algorithm for Detecting Spurious Retransmission Timeouts with TCP and the Stream Control Transmission Protocol (SCTP). `http://tools.ietf.org/html/rfc4138`, 2005.

[35] J. Serror. Impact of paging channel overloads or attacks on a cellular network. In *Proceedings of the 5th ACM workshop on Wireless security*, WiSe, 2006.

[36] U. Shankar and V. Paxson. Active Mapping: Resisting NIDS

Evasion Without Altering Traffic. In *Proc. IEEE Symposium on Security and Privacy*, 2003.

[37] P. Traynor, P. McDaniel, and T. La Porta. On Attack Causality in Internet-connected Cellular Networks. In *Proc. of 16th USENIX Security Symposium*, 2007.