Discover, Analyze, and Validate Attacks With Introspective Side Channels

by

Zhiyun Qian

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2012

Doctoral Committee:

Associate Professor Z. Morley Mao, Chair
Professor Atul Prakash
Associate Professor Robert Dick
Assistant Professor J. Alex Halderman
Researcher Yinglian Xie, Microsoft Research Silicon Valley

To my wife, my family, and many other important people in my life.

ACKNOWLEDGEMENTS

There are many people to thank during my 5-year PhD life – one of the toughest experiences I have ever had. I would not have been able to make it without the help of others. First and foremost, I would like to gratefully thank my advisor Professor Zhuoqing Morley Mao who has always been of tremendous help at every step in my PhD study. She always devotes most her time and energy to students. I am also amazed by her ability to have a clear big picture of things in mind without losing track of details. With her guidance, I have transformed from a rookie in research to a researcher that can independently conduct research.

I want to greatly thank my lovely wife for being able to accompany me for almost 2 years during my PhD study, especially when I was still in the early stage of the PhD program and had doubts in myself. Her encouragement and support is much needed.

I am grateful to my dissertation committee, Professor Atul Prakash, Professor Alex Halderman, Professor Robert Dick, and Dr. Yinglian Xie for their valuable feedback and help in refining the thesis.

I appreciate the collaboration opportunity with Dr. Yinglian Xie, Dr. Fang Yu, and Dr. Ming Zhang at Microsoft Research who have shaped my research. I have been fortunate enough to learn from them.

I also want to thank my colleagues and friends for making my life in Ann Arbor much colorful and enjoyable, especially Feng Qian, with whom I had significant interaction both academically and socially during the entire 5-year life. Many thanks to my former group members, Dr. Ying Zhang and Dr. Xu Chen, who had helped me through tough times at

the beginning of my graduate school. In addition, I have enjoyed working together with Qiang Xu, Zhaoguang Wang on several research projects and had fun with Junxian Huang, Yudong Gao, Yunjing Xu, Lujun Fang, Jie Yu, Mark Gordon, Xiaoen Ju, Caoxie Zhang, Xinyu Zhang, Li Qian, Xin Hu, Rui Wang, Yan Huang, Hang Zhao, Zheng Liu, and Ason Chiang (and many others).

Finally, I want to thank my parents for leading me on the right path in my career. Thanks again to my wife who has always been sources of love and support. I am very fortunate to have them in my life. This dissertation is dedicated to them.

TABLE OF CONTENTS

LIST OF FIGURES

Figure

LIST OF TABLES

Table

ABSTRACT

Discover, Analyze, and Validate Attacks With Introspective Side Channels

by

Zhiyun Qian

Chair: Z. Morley Mao

Traditionally, the focus of security property "confidentiality" is on users' data (or application-layer information) such as password and credit card numbers. However, as network systems grow in complexity, more sensitive and internal state information is being maintained both within and external to the system, and therefore also subject to being leaked or inferred. One such example is that more features are being pushed to the middleboxes in the network which causes additional state to be kept. The leakage of such internal state can ultimately cause security breaches at the application layer.

In the thesis, a systematically identification of unintentionally revealed internal network state and its impact are presented. A new class of side channels defined as introspective side channels are summarized that can leak such internal state. Such side channels in disguise only leak seemingly trivial information.

The security analysis of the above problem consists of four steps: 1). Measurement (behavior characterization of a target system). 2). Identification of sensitive network and system state. 3). Identification of relevant introspective side channels. 4). Security analysis by connecting the sensitive network state and the relevant introspective side channels. Through these steps, techniques built on side channels are described which can enable a

wide range of security applications to discover, analyze and validate both new and existing attacks. For instance, a sensitive TCP-related state kept on certain firewall middleboxes is discovered to facilitate TCP injection and hijacking attacks. More surprisingly, even without the middleboxes, similar attacks are still possible due to newly identified introspective side-channels on the hosts.

CHAPTER I

Introduction

1.1 Overview

The tremendous growth in the usage of Internet has introduced many unexpected security challenges not foreseen by the initial Internet designers. The introduction of network-level middleboxes (e.g., NAT and firewalls) is an example of after-thought, leading to unexpected interaction with end-host network stacks which were not designed to work with them. It is particularly hard to ensure security considering how fast the Internet is evolving. Motivated by the fact that the security impact of much of the network evolution is not fully understood, in the research, we have identified the following trends related to networks that have interesting and subtle interactions with security: 1) An increasing number of security measures are implemented at the network-level but they may be ineffective or may introduce side effects. 2) Distinct networks with different underlying layers and asymmetric resources are becoming inter-connected (e.g., cellular and Internet) and their interaction can create security problems. 3) Network speed has increased drastically over the years and will continue in the future, which unfortunately provides attackers with more resources to potentially break previous security assumptions.

Guided by the above trends, we explored several important problem domains including Internet security, cellular network infrastructure security, Spam, and TCP/IP security. We are especially fascinated by the fact that a larger amount of network state is stored at not

only the end-host but also various network elements [93, 97, 111]. *Specifically, we explore the security implications of the internal network state revealed unintentionally through the introspective side channels that enable discovery, analysis, and validation of attacks*. For example, I discovered that firewall middleboxes often keep track of sensitive TCP-related state which can be leaked to attackers and cause TCP injection attack.

1.2 Information Leakage at the Network and System Layer

Information leakage is one of the main security threats that we are facing. On one hand, the ever growing computer networks have resulted in massive amount of information being shared easily around the globe. On the other hand, the computer systems and the networks that connect them are becoming ever so complex that securing the information stored in the networked systems can be extremely challenging.

Traditionally, the focus of confidentiality is on users' data (or application-layer information) such as password and credit card numbers. There are various measures that can be taken to secure the information. One of the basic scheme is to encrypt the sensitive data and decrypt it only when authorized users need them. However, in many cases, encryption and decryption incur significant overhead and may not be feasible in all scenarios. Even if the data can be encrypted, the program that processes the data can be compromised to leak the data out. More interestingly, even when the program is not compromised, data can still be leaked through side channels [101]. For instance, even when a communication channel is properly encrypted, the timing of packets in an interactive session can still leak information on which keystrokes are entered.

However, as the networked systems become more complex, the amount of information kept at the network and system layer also grows. Further, much of them can be sensitive and the leakage of which can result in security breach at the upper layer (*i.e.,* the application-layer). My research interest lies in discovering how such sensitive state can be leaked to or inferred by adversaries. Specifically, we focus on discovering relevant side channels and

analyzing how they can enable the information leakage at the network and system layer.

1.2.1 Sensitive internal network and system state

Throughout the study, we have looked at various internal network and system state that are sensitive. They include but not limited to: TCP sequence number, firewall policies, and configuration parameters.

1.2.2 Introspective Side Channels

Introspective side channels are defined as side channels that can leak information about the basic network or system state. For instance, IPID leaks how many packets are sent by a host or middlebox. Similarly, host packet counters leaks the total number of incoming/outgoing packets of special kinds (e.g., error packets). Although seemingly trivial, such information can be used as building blocks for a wide range of applications including enabling serious attacks on TCP.

1.3 Contributions

To the best of our knowledge, this is the first systematic investigation of this critical class of side channels. We develop techniques using side channels as building blocks to enable applications including analysis of the impact of a new network-level spamming technique, discovery of new TCP hijack attacks, and validation of the feasibility of resource-based attacks on cellular network infrastructures. My research has reached out to many Internet service providers and mobile carriers who confirmed our research findings and provided positive response in investigating and improving their networks. The industry we received feedback from includes carriers like AT&T and Orange, and vendors including Cisco and Checkpoint.

1.4 Thesis Organization

Chapter II surveys related security research. Chapter III covers the discovery of new TCP attacks enabled by a certain type of firewall middlebox. Chapter IV discusses how very similar attacks can be carried out even without such firewall middleboxes. Chapter V presents an effective port blocking policy that is revealed to be ineffective against a new spamming technique. Chapter VI talks about how Radio Network Controllers (RNC) can be fingerprinted via signatures measured through timing channel. Chapter VII discusses the limitation and future work as well as concludes the thesis.

CHAPTER II

Related Work and Background

The research that is related to this work can be divided into two main categories: the high-level theme of information leakage enabled by side channels, and then the various problem domains including 1) security of TCP/IP, 2) security of network middleboxes, and 3) cellular networks security. This section serves to position the research in this thesis and highlight the state of the art in the above security domains (*e.g.,* security of TCP).

2.1 General Theme: Information Leakage Enabled by Side Channels

Side channel leaks are known for decades. There are a wide range of known side channels including CPU usage, power usage, shared memory/files, and even electromagnetic waves, *etc.*. A variety of attacks are possible [108, 115, 101, 60]. For example, researchers have shown that it is possible infer keystrokes through electromagnetic waves [108], shared registers [115], and packet size/timing analysis on encrypted traffic [101, 60].

Side-channel-enabled attacks also have different forms. One common form is what we call simple "operation-to-pattern" matching where different operations trigger different observable outputs. The previous examples on keystroke inference falls under this category where individual keystroke or combinations of them can be considered as "operations" and the corresponding electromagnetic waves, registers, and network packet size, and timing side channels are considered "patterns" that can be passively observed by an attacker to

map back the operations. Such mapping is typically obtained offline through controlled experiments on the target applications.

The second form requires an attacker to be more active. For instance, an attacker needs to actively send a network request to a server in order to use the timing channel to infer the server load or if a user has logged in or not [59]. However, besides being more active (and having more resource requirement), this form is still similar to the first one in that it still requires the attacker to use the observed outputs to map directly back to certain operations.

The last form is more subtle and interesting. It requires an attacker to even more actively participate in the attack by occupying or manipulating certain resources or state so that it will interfere with the normal operations, forcing them to behave differently in a observable fashion. One good example is the recent work on idle port scanning [65] which shows that by exploiting the shared resources such as SYN cache (intentionally occupying it) – a security measure implemented in BSD systems, the kernel will have to use SYN-cookies which can be distinguished from a normal SYN-ACK packet. It allows an attacker to infer if a port is open on a target even without the ability to route traffic to it.

The first two pieces presented in the thesis belong to the last form while the last two pieces fall into the second form. In general, we also focus on how subtle information leakage can happen covering a number of problem domains. For instance, we discover previously unknown side-channels to infer TCP sequence number and launch a number of TCP attacks based on that.

2.2 Problem domain 1: Security of TCP/IP

IP address Spoofing. The Internet architecture has no mechanism to ensure packet-level authenticity. A well-recognized problem is that anyone on the Internet may be able to forge arbitrary IP packet headers. Even though various ad-hoc mechanisms such as ingress filtering [87] are proposed, they are hindered by the fact that not every single network has the incentive to implement it to be globally effective. As a result, so far IP spoofing is still

prevalent. According to a recent study by Beverly [56], more than 31% of the networks still allow arbitrary IP addresses to be spoofed.

Common attack vectors using IP spoofing include denial of service [31, 112], indirection [95, 56], and amplification attacks [86]). It is worth noting that even though IP spoofing is largely perceived as a vulnerability and should be prevented, there are a number of legitimate applications built on top IP spoofing, *e.g.,* reverse traceroute [55]. This makes the situation even worse that it is unlikely IP spoofing will be completely stopped in the near future. In our study, we make use of IP spoofing to devise a number of attacks such as TCP sequence number inference attack described in Chapter III and Chapter IV.

TCP sequence number prediction attack. Twenty years ago, certain OSes select the TCP Initial Sequence Numbers (ISN) based on a global counter which is incremented by a constant amount every second. It allows an attacker who has opened a connection to a server to obtain its current global counter and predict its next ISN with high confidence. With this prediction ability, an attacker can spoof the IP of a trusted client when talking to a target server, and complete the TCP 3-way handshake based on the guess of server's next ISN. The problem is fixed after the randomization of ISN is standardized and adopted. However, in the attacks presented in Chapter III and Chapter IV, we show that even though the ISN is completely randomized, it can be successfully inferred.

Blind TCP reset attack. As described in RFC 5961 [31], the attack is possible because a reset (RST) packet is accepted as long as its sequence number falls within the current TCP receive window. In a long-lived connection (*e.g.,* BGP sessions), an attacker knowing the target four-tuple can simply use brute force all sequence number ranges. Watson [112] has analyzed in detail the number of packets needed under various OS/setup taking into consider the source port can be random. A number of proposals, *e.g.,* requiring the RST sequence number to exactly match the expected sequence number, are discussed in RFC 4953 [29]; however, they are not widely adopted likely due to backward-compatibility issue and the fact that source port randomization can already alleviate the problem. Our attack,

especially the one presented in Chapter IV, can greatly facilitate the attack because it allows the right sequence number that can reset the connection to be inferred under one second under common cases.

TCP sequence number inference attack. The first known sequence number inference attack is described in 1999 [16] where the Linux 2.0.X kernel has a bug that silently drops the third packet in the three-way handshake when the ACK number is too small, and sends a reset when the ACK number is too big. Such behavior allows an attacker to infer the correct ACK number in an ACK packet to complete the TCP connection. However, it is an isolated bug that has been fixed since then. The other relevant attack described in Phrack magazine [5] infers the sequence number by relying on the fact that a packet with in-window sequence number can be silently dropped and a packet with out-of-window sequence number will trigger an outgoing ACK packet. The limitations of this work are that 1) it requires sending two orders of magnitude more packets considering the TCP receive window is usually very small (*e.g.,* 16K); 2) it relies on a very noisy feedback channel (*i.e.,* IPID) on the end-host. It is only targeting at long-lived connections where the host has low traffic rate. According to our empirical results, while theoretically possible, such attack is very hard to carry out and can succeed under rather limited conditions due to a large number of packets required to send as well as the noisy side-channel that is leveraged. Instead, the attacks shown in Chapter III and Chapter IV both have success rates of 65% or higher and take at only a couple of seconds at most to complete.

2.3 Problem domain 2: Middlebox security

. There are many different firewall middleboxes deployed in the network that can interact with protocols at various layers. We discuss a few of them:

1) TCP Firewall middleboxes have been introduced for many years [77, 89]. Although they are supposed to provide security, they often in time introduce new security vulnerabilities as well. Previous work has discovered various vulnerabilities on the firewalls them-

selves that range from not properly checking the sequence number of TCP RST packets resulting in DoS attack on active connections [9], to failure to correctly process specially-crafted packets forcing the middlebox to reload or hang. A more complete summary on firewall vulnerabilities can be found in a study done by Kamara *et al.* [79].

In our study, we found that the more functionalities the firewall implements at the TCP layer, the more information it can potentially leak. Since the design of the firewall middleboxes is well-intended, the problem is wide-spread in all major firewall middlebox vendors.

2) NAT boxes that check the five tuples (protocol, source/destination IPs, source/destination ports). Since NAT is designed to translate private IP addresses to public ones, it naturally looks at the TCP four tuples. Typically, this is not a problem as, however, as shown in Chapter III, if the NAT chooses to check the TCP four tuples embedded in a ICMP packet and allow it to pass only when the four tuples match any existing sessions, then an attacker may be able to send a craft an ICMP packet embedding an arbitrary TCP four-tuple to see if that four-tuple is currently active. While this piece of information may or may not be too sensitive by itself, it is an important piece of information for an attacker to conduct other attacks (*e.g.,* TCP reset attacks).

3) Radio Network Controllers (RNC) that are deployed in cellular networks store state related to the radio resource control. For instance, there are parameters controlling how many seconds the channel should be reserved for a particular device. Previous studies have pointed out that such resource control mechanism and the RNC are subject to denial of service attacks [80, 105, 99]. However, it is unclear how to target a specific region with enough number of reachable devices located in that region. In our study, we realize that since different RNCs may have different parameters configured at different locations, such parameters can be used to fingerprint the RNC (and thus the location). More details on this topic is discussed in §2.4 and Chapter VI.

4) Port-blocking firewall middleboxes that block TCP traffic based on destination port numbers. Port blocking is a well known practice commonly employed at the network. One

of the examples is the port 25 (SMTP) blocking practice implemented by the ISPs in an attempt to block spam traffic originated from their networks [10, 22, 27, 28]. As more ISPs perform outgoing SMTP traffic blocking, fewer bot IP addresses are available to spammers. Interestingly, a relative unknown spamming technique which we call triangular spamming actually may be able to bypass ISPs' port blocking practice through triangular routing. It can still leverage those blocked IP addresses to send spam in a stealthy manner without triggering alarms monitoring outbound SMTP traffic.

To understand how many ISPs are vulnerable to triangular spamming, or equivalently, to infer the port blocking policy (one type of sensitive state) implemented in the networks, we devised novel probing techniques relying on side channels. Related to our study is prior work on firewall policy inference such as FireCracker [98]. However, their model assumes that probing vantage point is inside a network that has deployed a firewall, while our model assumes that the probing vantage point is outside of the network, which incurs additional challenges, *e.g.,* more difficult to obtain response when probing into the network.

2.4 Problem domain 3: Security of Cellular Networks

The wide adoption of cellular networks increasingly draws attention to study their vulnerabilities. Various forms of attacks have been revealed in the past few years. Serror *et al.* [99] found malicious traffic from the Internet can overload the paging channel in CDMA2000 networks. Traynor *et al.* [105] presented two attacks on GPRS/EDGE networks by abusing the radio resource setup and teardown process. Lee *et al.* [80] showed well-timed attack traffic can trigger excessive radio resource control messages, which in turn overburden RNCs in UMTS networks. The success of all these attacks hinges on the ability to identify many target devices in a particular cellular network region, *e.g.,* under a RNC (which is also one of the focuses in our work). We use timing side channels to measure signatures that are associated with different RNCs to distinguish IPs in different locations.

Balakrishnan *et al.* [53] found IP address alone cannot be used to determine the geographic location of mobile phone in AT&T's 3G network. Latency measurement appears to be more informative. However, their observations are drawn from latency samples of only three locations.

There have been extensive studies on geolocating IP addresses on the Internet. Most of them assume a list of landmarks with well-known geographic location, extracted from manually-maintained sources like DNS names [88, 84, 45]. They then use latency constraints to multiple landmarks to geolocate an IP address [72]. Unfortunately, these techniques are ill-suited for geolocation in cellular networks because a phone is often far from its first IP hop (*e.g.,* GGSN) not only in terms of geographic distance but also in terms of network latency. Given this, instead of attempting to infer the exact location of a phone, we solve a slightly different (but easier) problem, *i.e.,* determining if two phones are under the same RNC.

There are also studies for localizing mobile devices such as Escort [63], VOR [85], Virtual Compass [54], PlaceLab [61], and Radar [51]. The key difference is that they all require participation and cooperation from the phone, which is not assumed in the attacks mentioned above.

Prior work leveraged phone numbers to create a hit list for the purpose of launching targeted attacks. Enck *et al.* [64] generated a hit list of phone numbers through a combination of sources such as NPA/NXX database and web scraping. However, its effectiveness is not thoroughly evaluated. Moreover, sending SMS or MMS has a number of limitations compared to sending IP traffic directly to phones.

2.4.1 Cellular Network (UMTS) Background

UMTS architecture. A UMTS network consists of three subsystems: Devices, Radio Access Network and Core Network [78]. They form a hierarchical structure where the lowest layer is the device, followed by radio access network consisting of base-stations and

Radio Network Controllers (RNCs). At the highest level is the Core Network which has Serving GPRS Support Node (SGSN) and Gateway GPRS Support Node (GGSN). The latter connects to the Internet. Our focus is on the radio access network—the DoS target.

Radio Resource Control and State Machine. In UMTS, to efficiently utilize the limited radio resources (*i.e.,* physical communication channels), the radio resource control (RRC) protocol uses a state machine to manage the radio resource for each device. Typically, there are three RRC states—**IDLE**, **CELL_DCH** (or DCH) and **CELL_FACH** (or FACH). Here we present a state machine in Figure 2.1 used in a specific carrier. However, each carrier or vendor may have a slightly different state machine. For instance, some vendors may additionally the state transition from allow **IDLE** to **CELL_FACH** (while the one shown in the Figure does not). It was measured in previous studies through controlled experiments [90]. In the threat model presented in Chapter VI, we assume that the specific state machine model is known to an attacker, but we do not assume that the configuration parameters are known ahead of time.

Different states have different amount of radio resource assigned. **IDLE** has no radio resource allocated. To send or receive any traffic, it has to promote to **DCH** where a dedicated channel is assigned to the device. In **FACH**, devices are assigned a shared channel which only provides limited throughput (less than 20kbps). It is designed for applications requiring a very low data rate. There are several configurable static parameters associated with state transitions. For example, Downlink/Uplink (DL/UL) Radio Link Controller (RLC) buffer size is used to determine when the FACH $\rightarrow$ DCH transition occurs (we refer to it as **queue size threshold** thereafter). If the number of bytes sent or received exceeds the threshold, the state transition will be triggered. Another example is the **inactivity timer** which determines when the DCH $\rightarrow$ FACH transition occurs. They are all configured at the RNC and may differ across RNCs in different locations, making them good candidates to construct the network signatures. Note that different RNC implementations may also employ slightly different state machines due to vendor differences. For instance, there could

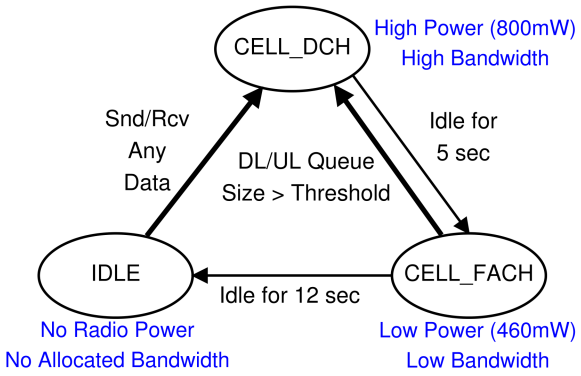


Figure 2.1: State machine for Carrier 1

be a IDLE $\rightarrow$ FACH transition instead of IDLE $\rightarrow$ DCH. In some cases, new states (*e.g.,* CELL_PCH) are introduced to further optimize radio resource. These differences can also be part of the signatures.

It is worth mentioning that in the next generation LTE networks, similar resource control mechanism and control parameters also exist [11].

CHAPTER III

Discovering Firewall-Enabled TCP Sequence Number Inference Attack

3.1 Introduction

In this chapter, we describe how we are able to discover introspective-side-channel-enabled attacks on TCP where we can infer the TCP sequence number of a target connection with the help of sequence-number-checking firewall middleboxes. The introspective side channels being leveraged in the chapter include the packet counter (*e.g.,* error packet counter) and IPID.

TCP was initially designed without many security considerations and has been evolving for years with patches to address various security holes. One of the critical patches is the randomization of TCP initial sequence numbers (ISN) which can guard against off-path spoofing attacks attempting to inject packets with a forged source address (for data injection or reset attacks) [70]. ISN randomization prevents sequence numbers from being predicted, thus arbitrarily injected packets are likely to have invalid sequence numbers which are simply discarded at the receiver.

Firewall vendors soon realized that they can in fact perform sequence number checking at network-based firewalls and actively drop invalid packets even before they can reach end-hosts, a functionality advertised in products from major firewall vendors [62, 77, 89].

This feature is believed to enhance security due to the early discard of injected packets and the resulting reduced wasted network and host resources. Ironically, we discover that the very same feature in fact allows an attacker to determine the valid sequence number by probing and checking which sequence numbers are valid using side-channels as feedback. We name this attack "TCP sequence number inference attack".

Using the sequence number inference as a building block, we design and implement a number of attacks including TCP hijack. In general, all of our attacks require IP spoofing, which is still very common on the Internet according to a recent study [57]. Besides IP spoofing, different attacks may have different requirements. For instance, a long-lived connection inference attack requires only a remote attacker to perform remote scanning and injection of exploits on services that run over unencrypted long-lived connections (*e.g.,* HTTP-based push services [46]). In contrast, TCP hijack requires a piece of unprivileged and lightweight malware residing on the victim.

We implement all except one attacks that we proposed. They are experimented specifically on mobile devices operating under a nation-wide carrier that extensively deploys sequence-number-checking firewall middleboxes. We show that a successful TCP hijacking allows an attacker to take over a connection and inject malicious payload right after the connection is established. For instance, we demonstrated that the attack can return a phishing Facebook login page, as shown in a short YouTube video [47]. We can also inject malicious Javascript to perform actions on behalf of a victim user, *e.g.,* to post tweets or follow other people.

We emphasize that even though our attack is implemented on mobile phones, it is not restricted to mobile devices or mobile networks. The reason for choosing this specific setting is that mobile networks make our experiments easier to carry out, as we have direct access to end devices behind the firewall. Also, the attack model of most TCP hijacking requires the unprivileged malware residing on the victim which fits the smartphone model well in that users often download untrusted third-party apps.

According to our measurement study, such firewalls are deployed in many carriers – at least 31.5% out of 149. This means the sequence number inference attack is widely applicable. It is likely to become more prevalent in the future as such functionality is considered to be advanced and desirable. Moreover, since we exploit the very behavior of sequence number checking — a firewall feature by design, it is unclear how to easily address the problem besides disabling the feature or employing application-layer encryption.

Our study makes the following contributions:

- We discover and report the TCP sequence number inference attack enabled by firewall middleboxes. We also devise techniques leveraging it as a building block to achieve TCP hijacking and a number of other attacks.

- We measure the popularity and characteristics of such middleboxes and found they are widely deployed in major cellular networks throughout the world.

- We survey a broad list of impacted applications ranging from Web-based attacks of directing users to a spoofed login page, application-based attacks of injecting malicious links to Windows Live Messenger chat messages, to attacks against servers in the form of DoS and spamming.

In the rest of the chapter, we describe fundamentals of the TCP sequence number inference attack in §3.2. Next, we discuss the detailed attack requirements and design in §3.3.2, and implementation results in §3.4. In §3.5, we measure how many cellular networks have deployed the sequence-number-checking firewall middleboxes. In §3.6, we describe what applications are impacted. Finally, we discuss what went wrong and conclude in §3.7.

3.2 Fundamentals of TCP the Sequence Number Inference Attack

In this section, we introduce the sequence number inference attack by first describing the behavior of sequence number checking firewalls, then discussing how to use side chan-

nels to infer the sequence number state kept on such firewalls, and finally illustrating the attack by an example.

3.2.1 Sequence-Number-Checking Firewalls

Many stateful firewalls that track TCP state (*e.g.,* SYN-SENT, ESTABLISHED) also track the sequence numbers of the bidirectional traffic. All major vendors including Cisco, Juniper, and Check Point have such products [62, 77, 89]. Typically, once a TCP connection is established, it only allows packets with sequence numbers within a window of the previously seen sequence numbers to go through. As an example illustrated in Figure 3.1(a), when the client and server exchange SYN and SYN-ACK packets, the firewall remembers the current sequence number to be X and Y for client and server respectively. Later packets originated from both sides will have to be in the window of X or Y, otherwise, they will be silently dropped. Such a feature is to prevent arbitrary packets from being injected into the connection. A window is needed because packets may arrive out of order and should still be allowed by the firewall. Note that acknowledgment number is typically not checked by the firewall because packets may or may not even set the ACK flag. In fact, we verified that all major OSes accept incoming data packets that do not have ACK flag set. Based on our observation and experiments with real firewalls (See §3.5), we found that sequence-number-checking firewalls may behave differently in the following ways.

Window size: Ideally, the firewall should acquire accurate state information associated with the end-host and accepts packets if and only if they will be accepted by the end-host. For instance, this requires the firewall to dynamically track the advertised receive window of the end-host, which can be expensive in terms of overhead. In practice, we found that firewalls typically initializes the window size to a fixed value according to the window scaling factor (a TCP option) carried in the SYN and SYN-ACK packet. It is typically calculated as $64K \times 2^N$, where N is the window scaling factor. The maximum possible receive window size is 1G and some firewalls simply use the fixed 1G window directly.

Left-only or right-only window: Some firewalls may only have a left window or right window such as (Y-WIN, Y) or (Y,Y+WIN). As discussed later, we found the nation-wide carrier that we studied indeed has left-only window firewalls because it buffers out-of-order (right-window) packets. Similar behavior was previously reported [111].

Window moving behavior: We found two general cases when the existing window will move: 1) In-order TCP packet arrives. It implies that the window can only move forward. We thus name it *window advancing*. 2) Any packet with an in-window sequence number. For instance, if Z is in (Y-WIN, Y+WIN), it can shift the window to (Z-WIN,Z+WIN). It implies that the window can either move forward or backward. We name this behavior *window shifting*. For the rest of the chapter, we assume the *window advancing* behavior, which is more popular according to our measurement study, unless explicitly stated otherwise.

Such firewall products claim that the sequence-number-checking feature can improve security by defending against connection hijacking [89], which ironically turns out to be the opposite. We demonstrate that as long as the target four-tuple (source/destination IP and port) is known, an attacker can probe using the spoofed target four-tuple to infer the valid sequence number, due to the very behavior that the firewall treats packets with in-window and out-of-window sequence numbers differently. Figure 3.1(b) illustrates such an attack model. The firewall's differentiation behavior, coupled with the ability that an attacker can get feedback regarding which packets are allowed, effectively breaks the non-interference security property [65]. We discuss how to obtain the target four-tuple and feedback below.

3.2.2 Obtaining Four Tuples – Threat Model

We outline three main threat models where the target four-tuple can be known:

(1). On-site TCP injection/hijacking. The unprivileged malware runs on the client with access to network and the list of active connections through standard OS interface (*e.g.,* "netstat" command). It cannot tamper with other applications or OS services. A successful

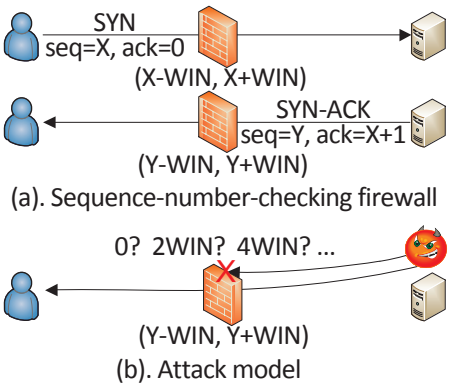


Figure 3.1: Sequence number checking stateful firewall and attack model

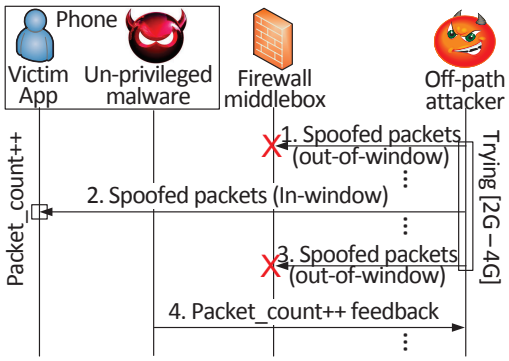


Figure 3.2: An attacker tries to infer sequence number

TCP sequence number inference attack in this case can compromise the security of other applications or even OS services.

Note that the attacker can also carry out other local privilege-escalation attacks under this threat model, but the most known privilege-escalation attacks on Android are still at the application layer without breaking the OS sandbox [67]. In contrast, our attack allows the malware to break the sandbox and compromise the security of other apps. Regardless, our attack provides additional capabilities to the attackers.

(2). Off-site TCP injection. An attacker simply guesses the four tuples. For instance, popular services typically have well-known port numbers and a few load-balancing IP addresses. To attack such services, the attacker only needs to enumerate client IP and port number. This usually works only when the target connection is long-lived, *e.g.,* instant messenger or push notification services.

(3). Establish TCP connection using spoofed IPs. An attacker in this case initiates the connection himself, in which case the four tuples are obviously known. Coupled with IP spoofing, an attacker can use this attack to establish TCP connections with a target server using spoofed IPs (*e.g.,* for spamming or denial-of-service).

3.2.3 Obtaining Feedback – Side Channels

As mentioned, to launch the sequence number inference attack, an attacker needs feedback regarding which packets went through the firewall. We discover two main side-channels that can serve the purpose:

1. OS packet counters: On Linux, the *procfs* [81] exposes aggregated information on the number of incoming/outgoing TCP packets, with or without errors (*e.g.,* wrong checksum). Alternatively, "netstat -s" exposes a similar set of information on all major OSes including Windows, Linux, BSD, and smartphone OSes like Android and iOS. If the packet went through the firewall middlebox, then the incoming packet counter will increment accordingly. Although such counters can be noisy as they are aggregated over the entire system, we show that some of the TCP error counters rarely increment under normal conditions and can be leveraged as a clean side channel.

2. IPIDs from responses of intermediate middleboxes: IPID is a 16-bit field in the IP header. In practice, many OSes, including middlebox OSes, have such monotonically incrementing IPIDs (a known side channel for inferring how many packets a target system has sent [95]). In addition, many networks allow intermediate middleboxes (*e.g.,* routers) to reply with "time-to-live (TTL) expired" ICMP messages (See §3.5.2 for measurement results) to inform the source of a discarded packet due to the TTL field reaching zero. Thus, an attacker can craft packets with TTL values large enough to reach the firewall middlebox, but small enough that they will terminate at an intermediate middlebox instead of the end-host, triggering the TTL-expired messages. By reading the IPID values generated by the intermediate hop before and after sending the spoofed probing packets, an attacker can infer if probing packets went through the firewall.

Both side-channels can serve the same purpose. An attacker can decide which to use depending on their availability and how noisy the side-channels are.

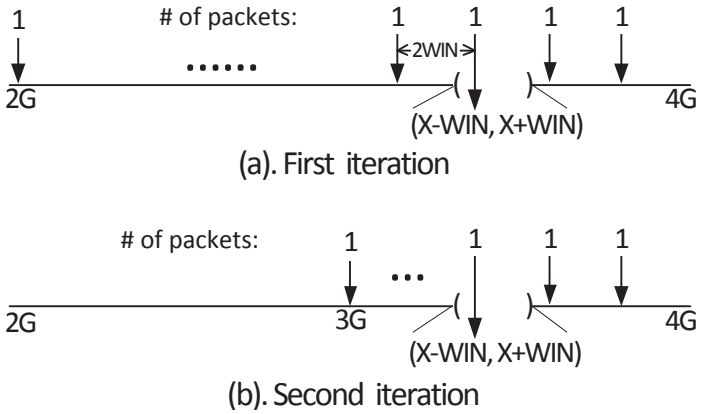


Figure 3.3: Sequence number space search illustration

3.2.4 Sequence Number Inference

Now that we know how to obtain the target four-tuple and feedback regarding which packets are allowed, we need an efficient way to infer the sequence number. A naive approach is to test out each individual window sequentially. In particular, one can check if 0 is in-window, if 2WIN is in-window, *etc.* as shown in Figure 3.1(b). However, that requires $\frac{4G}{2WIN}$ round trips to determine which window the sequence number falls in, which can take too long to finish.

In Figure 3.2, we illustrate a much faster approach – a binary-search-like inference that tries half of the sequence number space at a time (*e.g.,* 0 to 2G) and iteratively narrow down the sequence number. Here we assume the first threat model where the unprivileged malware runs on the client that colludes with an attack server. We also assume that the attacker has prior knowledge of the firewall behavior (*e.g.,* window size), which can be easily obtained offline. Figure 3.2 illustrates the procedure where the attack server first tries the upper half of the sequence number space [2G,4G). As shown in the figure, packets at time 1 and 3 are dropped and only a single in-window packet at time 2 is allowed. Upon receiving the packet, the phone will increase the packet counter. At time 4, after the attack server finishes probing [2G,4G), it can query the malware for the delta of packet counter before and after the probing. Based on the incremented packet counter, the attack server

knows that [2G,4G) is the correct range. Otherwise, it is likely that the other half [0,2G) is the correct one.

In Figure 3.3, we illustrate this example again via the sequence number space view. In the first iteration trying out [2G,4G), a series of packets are sent with sequence numbers on equally spaced interval of 2WIN (with $\frac{2G}{2WIN}$ number of packets sent). Given every 2WIN range is covered by a packet, one and only one packet will be allowed to go through if the current sequence number kept on the firewall indeed falls in [2G,4G). In the second iteration, it continues to try [3G,4G) to further narrow down the sequence number. Even though the number of packets sent at each iteration can be large (especially at the beginning iterations), it is not hard to see that: 1). the search algorithm takes $\log_2 4G = 32$ iterations to complete, which is the same complexity as a standard binary search algorithm; 2). the larger the WIN is, the fewer probing packets are required. We discuss further optimizations to improve the number of iterations and inference time in §3.4.

Note that this example assumes *window-advancing* firewalls. In the case of *window-shifting* firewall, similar procedure still applies yet it only allows an attacker to determine a range of possible sequence numbers instead of narrowing down to the exact value. It is because the first in-window packet already erases the original state of the sequence number by shifting the center of the window away. Nevertheless, it still can allow an attacker to narrow down the sequence number to a much smaller range, which in many cases can be inferred using brute force by the attacker. We omit the details here and focus on window-advancing firewalls given the latter is most commonly observed.

3.2.5 Timing of Inference and Injection — TCP Hijacking

For the TCP sequence number inference and subsequent data injection to be successful, a critical challenge is timing. If a user is in the middle of a session, injected TCP packets may not be "meaningful" at all. Specifically, since the sequence number inference takes time to finish, the server could already send part or all of the response (*e.g.,* HTTP

Table 3.1: Identified TCP sequence number inference attacks and their requirements

Req. ID	Requirement explanation	On-site TCP hijacking					
		Reset-the-server		Preemptive-SYN		Hit-and-run	
		Packet count	IPID	Packet count	IPID	Packet count	IPID
C1	Malware on client with Internet access	X	X	X	X	X	X
C2	Malware can read packet counters	X		X		X	
C3	Malware can read active TCP four tuples	X	X	X	X	X	X
C4	Client has coarsely predictable ISNs	X	X				
N1	A client can spoof another client's IP			X	X		
N2	A shared responsive intermediate hop		X		X		X
N3	Client network has NAT boxes deployed						
N4	Predictable external port if NAT deployed	X	X	X	X	X	X
N5	Additional firewall middlebox deployed					X	X
S1	Legitimate server has stateful firewall	X	X				
S2	Attack server closer to client			X	X		

Req. ID	Requirement explanation	Off-site injection		Spoofed conns
		URL phishing	Conn infer	
C1	Malware on client with Internet access			
C2	Malware can read packet counters			
C3	Malware can read active TCP four tuples			
C4	Client has coarsely predictable ISNs			
N1	A client can spoof another client's IP	X		X
N2	A shared responsive intermediate hop	X	X	X
N3	Client network has NAT boxes deployed		X	
N4	Predictable external port if NAT deployed			X
N5	Additional firewall middlebox deployed			
S1	Legitimate server has stateful firewall			
S2	Attack server closer to client			

response). The injected packets then will likely just corrupt the original response, which may or may not achieve the attacker's goal.

To address the challenge, we design and implement a number of *TCP hijacking* attacks (described in §3.3.2) where injection can happen at deterministic timing, *e.g.,* right after the TCP three-way handshake. This can, for instance, allow an attacker to inject a complete HTTP response without any interference from the original response. In contrast, *TCP Injection* is a general term that does not assume any specific timing of the injection.

3.3 TCP Attack Analysis and Design

Applying the basic TCP sequence number inference as a building block, we detail the design of a number of TCP attacks, each associated with a list of corresponding requirements. We show that they are widely applicable and feasible under many client/server/network combinations.

3.3.1 Attack Requirements

We first introduce two base requirements for all attacks: 1) the ability to spoof legitimate server's IP on the Internet, and 2) a sequence-number-checking firewall deployed in the client's network or anywhere in the network observing traffic flows in both directions. The former is a known problem and still widely prevalent on today's Internet [57], and the latter is required for the sequence number inference.

Besides the base requirements, we provide a complete list of requirements in Table 3.1, only a subset of which are required for any specific attack. We use "C", "N", and "S" to represent client-side, network, and server-side requirements.

Client-side requirements mainly have to do with malware's capability. For instance, C1 specifies that the malware needs Internet access. C2 requires access to the first side-channel (*i.e.,* packet counter) to obtain feedback. C3 specifies that the malware can run in the background and continuously monitor the creation of any new TCP connection. C1–C3 are common capabilities that an unprivileged program has in modern OSes. To be more stealthy, the malware could hide its monitoring activity until the target app (*e.g.,* browser app) is launched. C4 is a byproduct of the design decision made in many UNIX-like OSes (*e.g.,* Linux 3.0.1 and earlier) where the ISN for different connections are not completely independent. Instead, the high 8 bits for all ISNs is a global number that increments slowly (every five minutes) and only the low 24 bits are produced as random numbers. The design is to balance across security, reliability, and performance, and it is long perceived as a good optimization (more details discussed in [17, 6]). The result of this design is that the ISN of

two back-to-back connections will be at most $2^{24} = 16,777,216$ apart.

Network requirements relate to policies in the network. For instance, N1 specifies that client-side IP spoofing is allowed. As discussed, this is fairly common on the Internet [57] and also observable in cellular networks according to a recent study [111]. N2 corresponds to the second side channel to obtain feedback as described in §3.2.3. The requirement further states that such an intermediate hop must be on the path for both the attacker connection and the victim connection for the feedback to be useful (§3.5.2 shows more than half of the networks that have sequence-number-checking firewalls satisfy this requirement). N3 simply describes that a standard NAT is deployed in the client's network. N4 says that the NAT-mapped external port has to be predictable which is a typical requirement for P2P applications [30]. The requirement is necessary for on-site attacks that need externally-mapped four tuples (as described in the next section). A recent measurement study on NAT mapping type in cellular networks [111] shows that the majority of the networks satisfy the requirement. N5 states that there is an additional sequence-number-checking firewall deployed in the network, which is actually what we observe in the nation-wide cellular network. Except for N1, other requirements are mostly network design decisions and cannot be classified as "vulnerabilities."

Server-side requirement S1 states that the legitimate server has to deploy host-based stateful firewall that drops out-of-state TCP packets. Many websites such as Facebook and Twitter deploy such firewalls to reduce malicious traffic. For instance, iptables can be easily configured to achieve this [37]. Note that interestingly this security feature on the server turns out to help enable one of the TCP hijacking attack. S2 requires the attack server's network latency to the victim needs to be smaller compared to the legitimate server.

3.3.2 Attack Design

In this section, we describe in detail each attack and the corresponding requirements. Specifically, we design three classes of attacks for each threat model as described earlier in

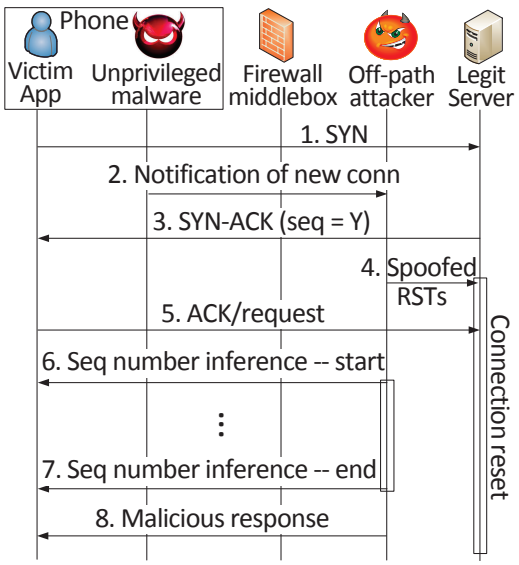


Figure 3.4: Reset-the-server hijacking

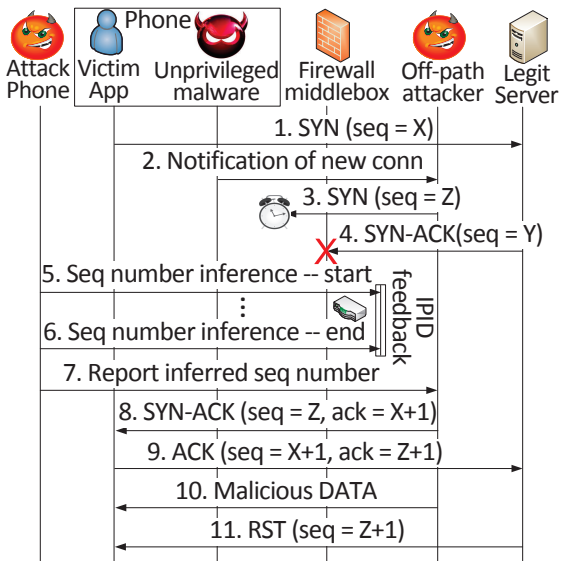


Figure 3.5: Preemptive-SYN hijacking

§3.2.2: 1) On-site TCP hijacking/injection. 2) Off-site TCP hijacking/injection. 3) Spoofed connection establishment. Each class has several attacks with the same goal but different requirements.

3.3.2.1 On-site TCP hijacking

As noted, TCP hijacking allows packets injected right after the connection is established. It is more powerful than the general case of injection but with more requirements. Thus, we focus on the hijacking attack design which also covers the general case of injection. In total, we devise three TCP hijacking attacks and all of which are implemented and tested against the nation-wide cellular network, since all requirements are satisfied in the network (As shown in §3.4).

The first TCP hijacking is **Reset-the-server**. The high-level idea is to reset the connection on the legitimate server as soon as possible to allow the attacker to claim to be the legitimate server talking to the victim. The key is that such reset packets have to be triggered right after the legitimate server sends SYN-ACK. To achieve this, we leverage requirement C4 which allows an attacker to predict the rough range of victim's ISN and send reset packets with sequence numbers in that range. This is helpful because then the

attacker can send much fewer spoofed RST packets (thus with lower bandwidth requirement) compared to enumerating the entire 4G space. Further, after the legitimate server is reset, requirement S1 is necessary as it helps prevent the legitimate server from generating RST upon receiving out-of-state data or ACK packets from the victim. Here we focus on the design of the attack. Implementation and feasibility analysis are covered in §3.4.

Figure 3.4 illustrates the attack sequence. Here the attacker is off-path and not man-in-the-middle. It is positioned between the victim and legitimate server for ease of illustration only. Starting at time 1, the victim app first initiates a TCP SYN. At time 2, the malware discovers the new connection attempt by continuously monitoring the output of "netstat", and it immediately notifies the attack server about the new connection including the four tuples. The malware also starts a new connection to the attack server so that the server knows the current ISN. At time 3, the legitimate server receives the SYN, and replies with a SYN-ACK. At time 4, the attack server floods the legitimate server with a number of spoofed RST packets based on the previously gathered ISN. As discussed earlier in §3.2.5, the RST packets have to arrive before the ACK/request packets at time 5; otherwise, the legitimate server will respond before the attacker can send any malicious content.

From there on, the legitimate server's connection is reset. All future packets from the victim are considered out-of-state and silently dropped due to requirement S1. For instance, the ACK packet received at time 5 is silently discarded. From time 6 to 7, we omit the sequence number inference procedure described earlier in §3.2.4. At time 8, the attack server can inject data using the inferred sequence number.

Table 3.1 summarizes the requirements for the attack. Depending on the side-channel used for feedback, the set of requirements for this attack methodology is (C1,C2,C3,C4,N4,S1) using the packet count feedback, and (C1,C3,C4,N2,N4,S1) using the intermediate hop IPID feedback. Note that N4 is needed because all RST packets need to have the correct external source port number.

The second TCP hijacking is **Preemptive-SYN**. The high-level idea is similar to Reset-

the-server in that it also tries to prevent the legitimate server's packets from reaching the client. The difference is that it does so by turning the firewall middlebox's sequence number checking feature against the legitimate server. Remember that the middlebox initializes the current sequence number from SYN and SYN-ACK packet, if an attacker can preemptively send spoofed SYN packets before the legitimate SYN-ACK packet (*e.g.,* when requirement S2 is satisfied), the firewall will initialize the sequence number according to the spoofed SYN instead of the legitimate SYN-ACK. Spoofed SYN packet is allowed due to TCP simultaneous open [32]. The attacker cannot directly spoof a SYN-ACK packet without the knowledge of a valid acknowledge number. Another difference is that such an attack needs requirement N1 to allow the sequence number inference from the client's network. Specifically, a separate attack phone inside the network is required to spoof the victim's IP and infer the sequence number of the victim's SYN. As described later in §3.4, the firewall is deployed at the *Gateway GPRS Supporting Node* (GGSN) level [114] such that a single attack phone can spoof hundreds of thousands of IPs of other devices. As a result, the attack phone and the victim phone can be in different cities or states as long as they go through the same GGSN. The details are described below.

As shown in Figure 3.5, initially the victim app sends a TCP SYN packet (with sequence number X) at time 1, followed by the malware reporting the new connection. Due to requirement S2, at time 3, the attack server receives the notification and immediately sends a preemptive SYN (with sequence number Z) which reaches the firewall middlebox before the legitimate server's SYN-ACK. Also, note that the preemptive SYN packet does not actually reach the phone (easily achieved with small TTLs set deliberately by the attacker), necessary to prevent the phone from replying with SYN-ACK which triggers connection reset from the legitimate server and prevents the connection from being established. At time 4, the legitimate server's SYN-ACK packet is dropped at the firewall because its sequence number Y is now considered out-of-window of (Z-WIN,Z+WIN), assuming that Y and Z are unlikely close together. During time 5 and 6, the attack phone tries to infer the sequence

number of the victim's original SYN with the intermediate hop feedback. At time 7, after finishing inferring the sequence number, the attack phone reports it to the attack server which then sends a spoofed SYN-ACK with the correct acknowledgment number. Since the victim never actually sees any response after it sends SYN, thinking the delay is likely due to resource issues, it happily accepts the SYN-ACK and replies with ACK to complete the connection.

There is however still one remaining challenge — the ACK packet at time 9 will trigger a reset once it arrives at the legitimate server, which will terminate the connection immediately. To get around this problem, the attack server has to inject data packets immediately following the spoofed SYN-ACK at time 10 so that it arrives before the RST packet at time 11. As long as the data packet is accepted before the connection is RST, the damage is already done. For instance, we verified that in a HTTP session, a small data packet containing an iframe pointing to a malicious URL still makes the browser follow the URL and load the content even through the connection is reset immediately after.

The requirements are (C1,C2,C3,N1,N4,S2) using packet count feedback, or (C1,C3,N1,N2,N4,S2) using intermediate hop IPID feedback. Here N4 is required because the preemptive SYN packet needs to have the correct external source port number as the destination port.

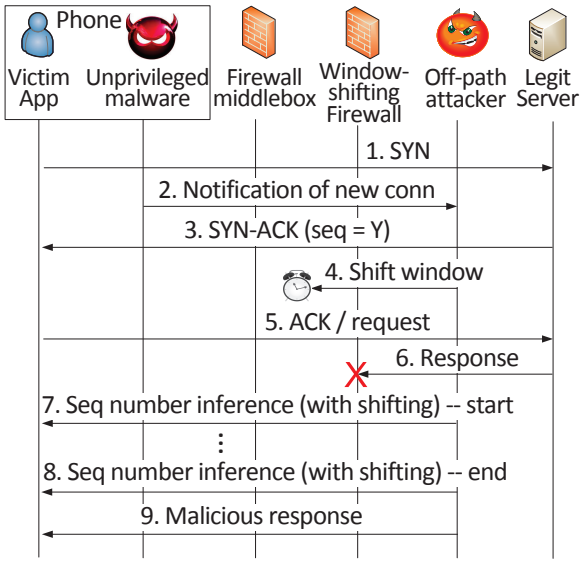


Figure 3.6: Hit-and-run TCP hijacking

The last TCP hijacking is **Hit-and-run**. This attack is possible only when the network deploys two different firewall middleboxes, which is what we observed in the nation-wide carrier elaborated in §3.4. In general, assuming that the sequence number inference is carried out in network external to the mobile device, the two different firewalls have to satisfy the following: a window-shifting firewall is deployed external to a window-advancing firewall. The network may intentionally set up the external firewall for general packet-filtering (which is simpler and potentially cheaper) and the internal one is for more advanced intrusion detection (which requires packet reassembly and incurs more overhead). The problem with this setup is that the window-shifting firewall allows an attacker to intentionally shift the window away from its original position which effectively disallows packets sent from the legitimate server. At the same time, the attacker still can shift the window back when it is necessary to traverse the internal window-advancing firewall to conduct the sequence number inference. This particular two-firewall setup effectively eliminates the requirement C4 and S1 in the Reset-the-server attack. We emphasize that the combined effect of the two firewalls is still a window-advancing firewall and previous two TCP hijacking attacks still work.

Figure 3.6 illustrates the attack process in detail. In this example, we use the setup of the nation-wide network where the internal window-advancing firewall has a left-only window of 1G. However, in the general case, the attack is possible as long as it is a window-advancing firewall. Time 1–3 match that in the Reset-the-server attack. At time 4, however, instead of resetting the connection on the server, the attacker tries to intentionally shift the window away from its original position. Specifically, regardless of the original window's position, an attacker can send an array of spoofed packets with sequence number 4G, 4G-(WIN-1), 4G-2(WIN-1), ..., all the way to 0. It is not hard to see that the center of the window will be deterministically shifted to 0 (we show the feasibility in §3.4). This way, at time 6, the legitimate server's response is highly likely to be dropped by the window-shifting firewall (assuming its sequence number has a low probability of being close to

0). Note that packets sent at time 4 do not need to go further beyond the window-shifting firewall, as easily achieved using a small TTL. These TTL-expired ICMP packets are sent to the legitimate server, which may unintentionally terminate the connection on the server side in extremely unlucky situations. Specifically, the ICMP packet embeds the original TCP header which includes the sequence number. The connection will be terminated only if the sequence number happen to exactly match the one used in the SYN-ACK packet. If that happens, then all client's packets in the future will trigger the legitimate server to respond with RST packets and stop the attack. However, having an exact match of the server's SYN-ACK sequence number is highly unlikely.

At time 7, the sequence number inference is started. However, since the window was shifted to 0 in the sequence number space. Now it is necessary to shift it again in order to allow the attacker's sequence number inference packets to pass through the window-shifting firewall. To do so, we can piggyback the sequence number inference packets along with the packets for shifting the window. For instance, an attacker can infer if the sequence number is in [0,2G) by trying 0, WIN-1, 2(WIN-1), ... up to 2G, which not only can shift the window from 0 to 2G, but also tested the [0,2G) range. Since the internal firewall has a 1G window, only 0 and 1G needs to be sent with a large TTL to go through it. All other packets can have a small TTL so that they only pass through the external firewalle. If either 0 or 1G passes through the internal firewall, then the sequence number falls in [0,2G). Otherwise, it falls in [2G,4G). One additional challenge is that the legitimate server may retransmit its "lost" response packet during the inference. As a result, the attacker has to shift the window back to a "safe" spot to prevent the retransmitted packets from passing through. For instance, one simple way is to shift the window to 0 every time after an iteration (which is what we did in our implementation). Such "position-reset" happens so fast that it is very unlikely the retransmitted packets can catch the "shifting" window.

The requirements are (C1,C2,C3,N4,N5) using packet count feedback, or (C1,C3,N2,N4,N5) using intermediate hop IPID feedback.

3.3.2.2 Off-site TCP injection/hijacking

Off-site attacks do not require the unprivileged malware but they are generally harder to carry out given the challenge to obtain target four-tuple.

However, **URL phishing** is a special case where an attacker can also acquire target four tuples by luring a user to visit a malicious webpage that subsequently redirects the user to a legitimate target website. A successful attack can replace the content of the target website, or if the user is previously logged in, the attacker can inject malicious Javascript to steal authentication cookies or perform actions on behalf of the user.

Here is how it works: assuming the user visited the malicious webpage, the attacker can obtain the client IP. It is also easy to obtain the legitimate website's IP given the common use of only a few load-balancing IPs. The remaining missing information is the source port number used in the next connection to the legitimate website. If the attacker can predict that, he can hijack the connection using the preemptive-SYN technique introduced earlier, *i.e.,*, start sending preemptive SYN packet right after the client is about to be redirected to the legitimate website (*i.e.,* make a connection to the legitimate server). However, many browsers seem to always assign a random local port number for different web pages which makes the port prediction very difficult. To overcome the challenge, we design a simple strategy to intentionally occupy as many local ports as possible so that the next port used is selected from a much smaller pool.

Specifically, the malicious website can instruct the client to open many connections to the malicious site (or any other server) to consume a large number of local ports. In addition, the occupied port numbers tend to be contiguous according to our experiment likely due to the origination from the same Javascript. One challenge is that the OS may limit the total number of ports that an application can occupy, thus preventing the attacker from opening too many concurrent connections. Nevertheless, we found such limit can be bypassed if the established connections are immediately closed (which no longer counts towards the limit). The local port numbers are still not released since the closed connections

enter the TCP TIME_WAIT state for a duration of 1–2 minutes. If an attacker can manage to open enough connections, he can easily use brute force the remaining ports by sending many preemptive SYN packets simultaneously. The rest of the attack works exactly the same as in the preemptive-SYN hijacking. Here the on-device malware is not required since the attacker already knows the target four-tuple.

Long-lived connection inference. Besides URL phishing, another type of off-site injection is to target long-lived connections. Instead of guessing the target four tuples, we discover that it is possible to "query" a network and check if a particular four-tuple is active through a single ICMP packet. If the attack targets at popular services, the server IP and port are typically known, thus the search space is reduced to only different client IP/port combinations. Since many popular services using unencrypted long-lived HTTP connections to implement PUSH services [46], the attack would basically allow remote scanning and injection of HTTP-based exploits.

This attack is possible because NAT boxes maintain state about active or in-session TCP connections, identified by four tuples. Out-of-session packets are denied access. Such behavior can leak information about existing/active sessions (similar to the reason why sequence number can be leaked). For instance, one approach is to use the intermediate hop IPID side-channel again to infer if packets with spoofed target four-tuple can go through. Note that such spoofed packets should not reach far enough to the firewall middlebox, so it does not matter what sequence number the spoofed TCP packets have. In total, the attacker has to send at least three packets (two to get the IPID before and after the spoofed probing and one is the probing packet) to query a single four-tuple, and the results may not be always reliable due to possible IPID noise.

A more efficient and reliable approach we discover is through sending a single ICMP error message (*e.g.,* network or port unreachable) to query a four-tuple. Specifically, since many NAT boxes check the embedded TCP four tuples inside ICMP packets and allow them through only when the four tuples match existing sessions, an attacker can easily craft

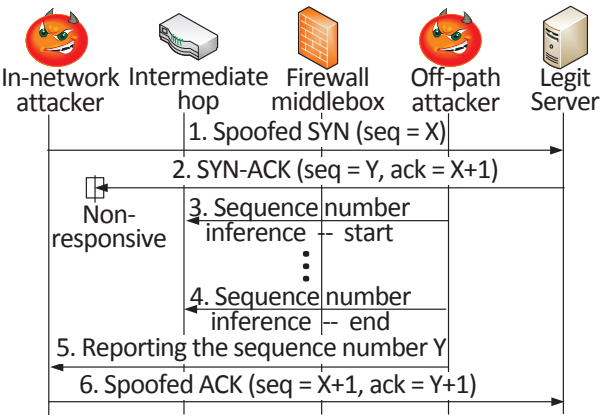


Figure 3.7: Establish TCP connection using spoofed IPs

ICMP packets embedding target four tuples and check if they can go through. More importantly, the source IP address of the ICMP packets themselves do not have to be spoofed. This is because ICMP packets are often sent by IPs other than the two communicating parties such as a gateway. This allows the attacker to receive direct response in the form of a TTL-expired message from the intermediate hop.

3.3.2.3 Establish spoofed connections

The goal of this attack is to establish TCP connections to a legitimate server from an attacker using spoofed IPs. It closely resembles the traditional TCP sequence number prediction attack where an attacker can guess the sequence number of the legitimate server's SYN-ACK and establish connections using spoofed IPs. We are essentially launching the same attack, but here we are "inferring" instead of randomly "guessing" the sequence number. As elaborated in §3.6.3, this attack can be a useful building block of DDoS attack or spamming where each connection has a distinct source IP, thus overcoming IP-based blocking.

The attack sequence diagram is fairly simple as shown in Figure 3.7. At time 1, an in-network attacker sends a spoofed SYN with an unresponsive source IP (more discussion below). At time 2, the server replies with a SYN-ACK back to the spoofed IP. However, as the spoofed IP is unresponsive, the packet does not trigger any response packet. The attack

server then performs the sequence number inference during time 3 and 4. Upon completion, it reports the inferred sequence number to the in-network attacker at time 5, which in turn sends the spoofed ACK packet using the inferred sequence number to complete the TCP handshake with the victim server at time 6.

Here unresponsive IPs are either IPs that may not be currently used by any device, or they drop out-of-state TCP packets on their own (*e.g.,* by host-based firewalls). We found that there are many such unresponsive IPs in the nation-wide cellular network that we tested. The requirements of this attack are (N1,N2,N4).

3.4 Attack Implementation and Experimental Results

We have implemented the complete end-to-end attacks for all three threat models. Below is our experiment setup.

Client platform. We use Android smartphones because it fits the first threat model well and it can easily connect to the nation-wide cellular network with sequence-number-checking firewalls. Other smartphones such as iPhone could also be used since it also satisfies all the client-side requirements. We implement the malware that spawns a service to run in the background and monitors new target connections to attack. To prevent from being scanning the active target connections too aggressively, the malware stops running whenever the screen is off. In fact, it can start the scanning activity only when detecting the target app is at the foreground. We tested the attacks ourselves on three different controlled Android phones (no other people is attacked by the malware) with OS versions of Android 2.2 and 2.3.4 and from three vendors (HTC, Samsung, and Motorola). The default window scaling option is 2 and 4 for Android 2.2 is 2.3.4 respectively.

Network. The experiments are conducted on an anonymized nation-wide carrier that widely deploys firewall middleboxes at the GGSN-level. The carrier satisfied all the network-side requirements (N1 to N4), which allows us to realistically test all attacks except for URL phishing. However, different GGSNs [114] may have slightly different network policy. For

instance, some GGSNs prevent internal hops from replying with TTL-expired messages, thus violating requirement N2. IP spoofing is however allowed in every GGSN which enables an attacker to spoof a large range of IPs (many /16), potentially affecting many users.

Firewall. We found firewalls are deployed in all of the carrier's GGSNs. There are two main types: the first has a fixed window size (*i.e.,* $WIN = 1G$) with window-advancing behavior, the second computes the window size based on the window scaling factor (as mentioned in §3.2.1) with window-shifting behavior. The first one also has a left-only window since it buffers out-of-order packets. In certain GGSNs, only the first firewall is deployed. In others, both are deployed with the second one external to the first one (which enables the hit-and-run hijacking).

Proxy setting. We found that if the proxy is enabled through the *Access Point Name* (APN) configuration [48], then the firewall middlebox is no longer visible, which we suspect is due to the specific network topology setup and is a special case. In general, a proxy is similar to NAT that essentially rewrites the external IP and port. Only the browsers seem to be affected by the proxy setting and thus attacks on mobile apps are not affected. We do not have complete data on how many phones by default have the proxy enabled, but we do know that the Motorola Android phone by default does not use the proxy.

In summary, the diversity of the network and firewall setup implies that carriers may not be fully aware of the potential impact of various network configurations on security.

3.4.1 Side-channel

So far we have introduced the two side-channels — packet counter and intermediate hop IPID, now we discuss them in more details. For the packet counter, we found that Android has all the standard and advanced Linux packet counters accessible through publicly-readable procfs. The following is a list of relevant counters identified.

/proc/net/snmp: InSegs. This is a basic counter that simply records the number of incoming TCP packets received by the OS, regardless if the packet contains error (*e.g.,*

wrong checksum). It is the most straightforward counter but may be noisy as there can be background traffic received by the client during the sequence number inference.

It turns out that it is possible to find other much less noisy counters. The idea is to leverage the mismatch in the logic of identifying error packets between the firewall middlebox and the client. For instance, we can craft packets that look erroneous to the client but perfectly legitimate to the firewall. The result is that the firewall still checks the sequence number, but when the packet reaches the client, it will be dropped and the corresponding error packet counter will be incremented. Note that these error packet counters are much less noisy because they are rarely incremented caused by naturally occurring packet corruption. Specifically, we found the following promising counters on Android.

/proc/net/netstat: InErrs. This counter should be incremented every time when, among other things, a packet with wrong checksum is received. If the firewall lets packets with wrong checksum through, then an attacker can craft such packets and make use of this counter. However, we verified that the firewall in the nation-wide cellular network already drops packets with incorrect checksum.

/proc/net/netstat: PAWSEstab. This counter is incremented when a TCP packet with an old timestamp is received. PAWS, or Protect Against Wrapped Sequences, is a mechanism that relies on timestamp to prevent old packets with wrapped-around sequence numbers from being mistakenly received, a TCP extension standardized in RFC 1323 [75]. All Android phones that we tested have this counter enabled and the firewall does not check the timestamp at all (likely due to overhead concerns). As a result, our implementation uses this counter for all on-site attacks.

For the intermediate hop IPID side-channel, we found that the noise level is quite tolerable. Specifically, the IPID of the intermediate hop increments only when the hop (*e.g.,* router) itself is originating packets (*e.g.,* TTL-expired messages or packets generated for routing protocols). In contrast, packets passing through the hop do not affect its IPID. That means that the IPID should not increment very often. Moreover, since the probing packets

are back-to-back, the window for observing such noise is very small. In practice, we found that sending 1–4 packets per window range is usually enough to overcome the IPID noise.

3.4.2 Sequence Number Inference

Theoretically, the time to complete a binary-search-like probing is $32 \times RTT$. Assuming a cellular RTT of 200ms, the total time should be about 6.4 seconds. However, as observed in our experiments, it also takes time to send a large number of packets to cover the large sequence number space. In addition, we also add padding time during the probing to prevent packets arrive out-of-order. In practice, the binary-search-like probing can take up to 10 seconds to complete with an RTT of 200ms, which can be too long since a user may be able to notice the delay. To speed up the probing, we implement a number of optimizations.

The first optimization is that instead of inferring the exact sequence number, we can stop the inference once we know the sequence number is within a range (*e.g.,* of 256 possible numbers). Later, it will not be difficult to simply brute force all 256 sequence numbers simultaneously. In a binary search, this can reduce $\log_2 256 = 8$ RTTs, which is significant.

The second optimization is based on the observation that the sequence number inference is heavily round-trip-bound instead of bandwidth-bound. As a result, we devise an algorithm that reduces the number of network round trips significantly. The idea is that instead of eliminating half of the sequence number space each iteration, we can eliminate $\frac{N-1}{N}$ of the search space by simultaneously probing N-1 of N equally-partitioned bins. We could send different number of packets in different bins. As an example where $N = 4$, we could send 1 packet each window in the first bin, 2 packets each window in the second bin, and 4 packets each window in the third bin. This way, an attacker could tell which bin the sequence number falls in by looking at the increment of the packet counter. We name the probing technique "N-way search". It is not hard to see the resulting number of iterations can be computed as $\log_N 4G$. For instance, if $N = 4$, $\log_N 4G = 16$, which is only half of

that the original binary search needs.

At a glance, it seems that the bigger N, the better. However, we also note that by increasing N, the total number of probing packets also increases (since it requires more packets for each bin) and so is the inference time. In practice, we use a small N=2 (*i.e.,* binary search) at the beginning few iterations, and use larger N (*e.g.,* $N = 4$) towards the end, which turns out to work very well. When using the packet counter feedback, we found that it takes only about 4–5 seconds to complete the inference when RTT is at around 200ms.

3.4.3 On-site TCP hijacking

We next describe more details on the most critical part of each hijacking attack. We also analyze the bandwidth requirement when necessary (*e.g.,* to reset the server) and present the experimental results in Table 3.2 measured using the Android 2.3.4 OS where we hijack `m.facebook.com` with a Planetlab server acting as the attack server.

3.4.3.1 Reset-the-server

In this attack, the most critical part is to successfully reset the server. As described before, we leverage requirement C4 which tells the attacker that the victim connection's ISN is at most 16,777,216 away (either smaller or larger) from the ISN of the attacker-initiated connection. Since RST packets with any sequence number that falls in the receive window can terminate the connection [112], the max number of required RST can be calculated as $\frac{16777216 \times 2}{server\_rwnd}$ where $server\_rwnd$ represents the server's TCP receive window size. Further, given that the RST happens right after the server sending out SYN-ACK, $server\_rwnd$ is in fact the initial TCP receive window size denoted as $server\_init\_rwnd$. Typically, $server\_init\_rwnd$ is about three to four full TCP packets long as per TCP slow start. For instance, `m.facebook.com` uses 4380, `twitter.com` uses 5840, and the corresponding number of required RST packets is 7661 and 5746 respectively. However, different

websites can have very different values. We found `chase.com` uses 32805 which is almost a magnitude larger. In general, the larger the $server\_init\_rwnd$, the fewer packets required.

Moreover, to successfully reset the server in time, all RST packets have to be delivered between time 3 and 5 as shown in Figure 3.4. If they arrive after time 5, the server may already respond to the client's request. Thus, the valid time window for reset is basically a round trip time between the client and the server. The bandwidth requirement is then computed as $\frac{16777216 \times 2}{\frac{server\_init\_rwnd}{RTT}} \times 40bytes \times 8bits$. In our experiment in cellular networks where $RTT = 200ms$, it will be $327Kbps - 12Mbps$ (as shown in Table 3.2), depending on the $server\_init\_rwnd$ values mentioned above. When RTT is smaller (as on the Internet), the bandwidth requirement will increase proportionally. This is another reason why cellular devices are particularly vulnerable and easy to attack. Although the bandwidth requirement may seem high, it is important to note that bandwidth resource is becoming more abundant and cheaper. For instance, the uplink bandwidth of a standard home Comcast network can be up to 4.2Mbps (tested in our home). The bandwidth requirement can even be distributed across a number of bots. Moreover, the bandwidth requirement is not a hard requirement and the attack can be attempted multiple times. For instance, it will be good enough to use TCP hijacking to steal a user's password just once. In our experiment, we use a Planetlab server acting as the attack server to reset `m.facebook.com`. We are not certain about the exact bandwidth, but the reset success rate is quite good according to our experiment.

As shown in Table 3.2, the success rate of reset-the-server hijacking is 65% after 20 experiments with 7 failures in total. 5 of them are caused by the RST race condition failure. Other 2 are due to sequence number inference failures (*e.g.,* packet loss). As we can see, the success rate is high enough to cause real damage. It takes 4 to 5 seconds to complete the inference when measured with packet count feedback. It takes only 2 seconds using intermediate hop feedback as the probing does not go through the cellular link. The downside is that the latter may not always be available. Nevertheless, since we observe that it takes

Table 3.2: TCP hijacking bandwidth requirements and results

	Reset-the-server	Preemptive-SYN	Hit-and-run
BW required	0.3 – 12Mbps	None	6.6 – 26Mbps
BW factor	server_init_rwnd, RTT	None	WIN, RTT
Success rate	65%	65%	85%
Inference time	4–5s	6–7s	8–9s

more than 10 seconds to tear down a connection after several rounds of retransmission, the inference time is definitely short enough.

3.4.3.2 Preemptive-SYN

During implementation, we found one interesting detail about the intermediate hop feedback where its TTL-expired message can inadvertently terminate the client-side connection. It happens only when a TTL-expired message embedding a TCP header with a sequence number matching the original SYN's sequence number (similar to the hit-and-run hijacking case). Our optimization on the sequence number inference should already alleviate the problem since we stop inference much earlier so that it is unlikely a spoofed packet has the same sequence number as in the original SYN.

Note that there is no bandwidth requirement for this attack as long as requirement S2 is satisfied. Interestingly, according to Table 3.2, the success rate is still measured to be 65% after 20 experiments. However, out of the 7 failed cases, 6 are due to the sequence number inference (likely caused by the noise in IPID side-channel). 1 of them seems to be due to a load balancing change that causes the connection to the attack server to go through a different intermediate hop. However, we observe that this happens very rarely. In terms of the inference time, it takes about 6 to 7 seconds, slightly longer than the Reset-the-server attack, due to the need to send more packets per window to overcome the noise in the IPID side-channel.

3.4.3.3 Hit-and-run

The critical part of this attack is to shift the window in time at the very beginning to prevent legitimate server's packets from going through the firewall. The number of packets required is computed as $\frac{4G}{WIN-1}$ since one packet is sent per $WIN - 1$. Depending on the window scaling factor, WIN is 256K and 1M respectively for the two Android OSes. The bandwidth requirement is basically $\frac{\frac{4G}{WIN-1}}{RTT} \times 40bytes \times 8bits$ or 26Mbps and 6.6Mbps if we plug in the two WIN values (as shown in Table 3.2). One thing to note is that the window scaling factor is incremented every time a new Android version is pushed out, presumably to take advantage of the increasing cellular network bandwidth. This indicates that future attacks will have even lower bandwidth requirement.

As shown in Table 3.2, the success rate is 85% with only 3 failed cases caused by the inference failure. No failure is observed for the initial window shifting likely due to the lower bandwidth requirement with the window scaling factor of 4. Note that we need to shift the window back and forth in each iteration, which means more packets are sent and packet loss is thus more likely. For the same reason, the inference time is a little longer.

3.4.4 Off-site TCP injection

We were not able to implement the **URL phishing** attack on the nation-wide network, which is the only attack we did not implement. The reason is that when NAT is deployed, the attack requires knowing the client's private IP in order to conduct the sequence number inference from the client's network (same as preemptive-SYN). However, without on-site malware, it is difficult to obtain the device IP (*i.e.,* private IP) through mobile browsers. The only way to get device IP seems to be through Java applet which is not supported on mobile browsers. We have confirmed neither Javascript nor Flash can do so. Note that this attack is feasible for cellular carriers using public IP addresses for their mobile devices (there are in fact many such carriers according to a recent study [111]).

We did implement the **long-lived connection inference** using a single ICMP packet and

run a small-scale experiment on the nation-wide carrier to measure the number of cellular IPs actively using Android's push notification service. We pick a particular push server IP 74.125.65.188 and port 5228 (push service port), and choose an entire /16 cellular IPs to probe. For each IP, we enumerate every port within the default local port range for Android: 32768 – 61000. To avoid probing too aggressively, our experiments conservatively rate limit the probing to 6 seconds per IP. Interestingly, using the single-ICMP-packet probing, we found that about 7.8% of the IPs have a connection with the server. That means it is fairly easy to find popular services to attack. Even through the connections are encrypted, it is still possible to carry out connection reset attacks. In fact, this approach is much more efficient than the traditional reset attack where combinations of client port number and sequence number need to be enumerated.

3.4.5 Establish spoofed connections

We implement the attack mostly as described in §3.3.2.3. The only difference is that instead of spoofing a single IP, we spoof as many IPs (for different connections) to a controlled target server as possible. Specifically, we try to spoof all IPs inside a /16 IP range in the nation-wide carrier.

For each IP that we want to spoof, we need to first test if the IP is responsive. To do so, we first send a SYN packet with a spoofed IP from the attack phone inside the cellular network to our attack server which responds with a legitimate SYN-ACK back. If the spoofed IP is responsive, a RST will be generated. Otherwise, we consider the IP to be unresponsive. For any unresponsive IP, we send a second spoofed SYN, this time, destined to the victim server (*i.e.,* a controlled lab server). The rest of the work is to simply conduct the sequence number inference from the attack server using the intermediate hop feedback so that we can spoof a correct ACK packet to complete the connection.

Ideally an attacker can simultaneously spoof many IPs. However, we found that there is only a single shared responsive intermediate hop where all the TTL-expired messages

essentially share a single IPID counter. If we parallelize the process, different experiments probing to the same intermediate hop can interfere with each other. Consequently, we can only pipeline the process as much as possible to make sure there is always one sequence number inference procedure probing to the intermediate hop.

Through our experiments, we found that there are 80% of IPs are unresponsive, which means that there are plenty of IPs an attacker can make use of to establish spoofed connections. We found that we can make about 0.6 successful connection per second on average with more than 90% success rate (the failed cases are mostly due to sequence number inference error).

3.5 Vulnerable Networks

To understand the susceptibility of the existing networks to the described attacks, in this section, we report the measurement results of firewall implementations and availability of responsive intermediate hop, through a deployed mobile application (referred to as *MobileApp*) on the Android market (the malware described earlier was not on the market). The MobileApp measures the network performance and policy and reports the results to users so that they have incentives to run our app. The data are collected between Apr 25th, 2011 and Oct 17th, 2011 over 149 carriers uniquely identified by their Mobile Country Code (MCC) and Mobile Network Code (MNC).

3.5.1 Firewall implementation types

Methodology. We focus on the three firewall implementation properties described in §3.2.1. The three properties are selected based on experiences with the firewalls encountered in real carrier networks as well as a number of trial-and-errors on the earlier deployment of our MobileApp.

To infer the **window size**, we try the following WIN values in order: 2G, 128M, 16M, 1M, 512K, 256K, 64K. Note that testing exhaustively all possible window size values is

too time-consuming as a long timeout (*i.e.,* 4 seconds) is needed for each probing packet to account for long cellular RTTs. Specifically, for each WIN value, our MobileApp server test sequence numbers *X-WIN+2* and *X+WIN-2* to check if they can trigger any response. X is the next expected server-side sequence number. The adjustment by 2 is to accommodate a slightly smaller window implementation from the common values. The reverse ordering by window size is to finish the test more quickly if there is no sequence number checking (*i.e.,* WIN=2G).

To test the **left-only/right-only window** behavior, we always try the left window and then the right one (to be consistent). If the left window probing packet is allowed but not the right one, we conclude it is left-only window. Similarly, we can discover right-only window firewalls. Additionally, we have to eliminate the window-shifting case where the left-window packet can shift the window to the left so that the right window packet may be falsely considered as "out-of-window". Such cases can be detected by the test described next.

To test if the firewall has **window-shifting** behavior, the basic procedure is as follows: once a left-window packet with sequence number *X-WIN+2* is allowed by the firewall, we try to shift it further left by (WIN-1) twice. If both attempts of shifting succeed, we try the sequence number *X-WIN+2* again. If the window is indeed shiftable, its center is already shifted left by 2(WIN-1), making *X-WIN+2* out-of-window (and the packet will be dropped). There are two corner cases that need to be considered to ensure the validity of the results. The first one is that since we do not cover all possible window sizes, the inferred window size WIN may be an under-estimate of the actual window size. We address this explicitly by shifting the window far enough beyond an over-estimate of the actual window size. The second one has to do with resetting the window position to its original value after left window test is done before the right window test.

Results. Overall, out of the 149 carriers, we found 47 carriers that deploy sequence-number-checking firewalls with at least two completed supporting experiments. 10 other

Table 3.3: Sequence-number-checking firewall types

Window Size	Left/Right	Window Moving	# of Carriers
64K	left-only	window-advancing	6
fixed>128M	left&right	window-advancing	5
window scaling	left&right	window-advancing	7
window scaling	left&right	window-shifting	17
window scaling	left-only	window-advancing	10
-2G	left-only	unknown	2

carriers were found to be suspicious but with only one experiment, thus are excluded due to possible errors caused by packet loss. If we consider only the 47 carriers, 31.5% of the carriers are subject to the sequence number inference attack. The nation-wide network we tested is excluded from this analysis because it is somewhat a special case with two different firewalls deployed. We did not look for a similar two-firewall setup in the measurement and thus cannot conclude the number of other carriers with such two-firewall setup. In essence, our experiments test the combined effects of all sequence-number-checking firewalls.

A detailed breakdown of the measured firewall implementations is shown in Table 3.3.

Window size. We can observe three main window sizes: 1). 64K — some legacy firewalls only support this value (window scaling is not supported), 2). window scaling — where the size is calculated based on the window scaling factor, 3). fixed $>$ 128M — could be 1G as found in the nation-wide cellular network. There is one last window size listed as "-2G" which means that the left window is wide open, but no packets are allowed for the right window.

Left-only or right-only window. Interestingly, we discover that many networks have left-only window firewalls. For the nation-wide carrier, it is because the internal firewall buffers out-of-order packets as discussed before. However, we found this may not be the case for other carriers. Upon a closer inspection, we realize that some firewalls actually have an even smaller-than-64K right window set based on the initial receive window size (sometimes below 8K) carried in the client-side SYN (instead of based on the window-scaling factor). This behavior matches the ideal firewall that dynamically adjust the window

size based on the currently advertised receive window. On the other hand, the left window is still kept to be fixed in case old packets are lost and retransmitted. Since we did not test window sizes smaller than 64K, it is possible that some of the left-only window carriers can in fact be left&right. Regardless, such minor variations do not impact the attack as the window size can be obtained offline.

Window moving criteria. We found 17 carriers to have shiftable windows and all with left&right windows, making it difficult to infer the exact sequence number but still susceptible to attacks. The other majority of 30 carriers, however, allow the exact sequence number to be inferred.

3.5.2 Intermediate hop feedback

Methodology. We devise the following probing technique to infer if any intermediate hop is responsive: from the previous experiments we can gather an in-window and an out-of-window sequence number. We conduct two TCP traceroutes with those two sequence numbers respectively. If there is any hop that responds to the first traceroute (with in-window sequence number) but not to the second one, we flag such hop. Additionally, we send two traceroutes (ICMP error messages) embedding a correct four-tuple and a wrong one (with a modified port number). If any hop responds to the correct one but not the incorrect one, we consider the single ICMP packet probing as possible.

Results. Out of all the 47 carriers that have sequence-number-checking firewalls, 24 carriers have responsive intermediate hops that reply with TTL-expired ICMP packets. 8 carriers have NAT that allow single ICMP packet probing to infer active four tuples.

3.6 Vulnerable Applications

The TCP sequence number inference attack opens up a whole new set of attack venues. It breaks the common assumption that communication is relatively safe on encrypted/protected WiFi or cellular networks that encrypt the wireless traffic. In fact, since our attack does

not rely on sniffing traffic, it works regardless of the access technology as long as no application-layer protection is enabled. In this section, we illustrate the broad impact of the attack by a mere glimpse at a number of impacted applications.

3.6.1 Web-based attack

Facebook/Twitter: We found that the login pages for both desktop and mobile browser are not using SSL. They are subject to phishing attack where the login page can be replaced. Further, when users are logged in, webpages by default are not SSL-enabled (unless turned on in the account settings). It allows Javascript injection which simply sends a HTTP post request to perform actions on behalf of the users such as posting a message or following other users. Both Facebook and Twitter servers have host-based stateful firewalls that satisfy requirement S1, which enables Reset-the-server hijacking. In both cases, gaining access to users' social networking account is a huge privacy breach.

Banking: Similar to a previous study [66], we survey 68 banking websites from a keyword "bank" search from Google, 4 of which are found to have non-SSL login page. There is one other website which uses SSL in most pages but not one specific account query page which also contains a login form. Also, one website has a login helper program download link in HTTP that allows the binary to be replaced. In all cases, successful attacks can cause direct financial loss. We also verified that all bank servers deploy host-based stateful firewalls which satisfy requirement S1.

3.6.2 Application-based attack

Facebook app: The latest version of the Facebook app as of this writing was updated on October 5, 2011. We found that it is impossible to replace the login page as it is part of the built-in UI (*i.e.,* not fetched over the network). However, we do find two sensitive connections not using SSL. Even though we did not test our attacks specifically on them, it is quite obvious that they are subject to our attacks.

- The main page (*e.g.,* news feed) is fetched through HTTP (html/text) which is subject to tampering.
- A critical Javascript is fetched through HTTP. An attacker can inject malicious Javascripts to perform actions on behalf of the user just as the web-based attack.

Windows Live Messenger app: The protocol [20] is in plaintext without encryption in most client implementations, which allows an attacker to inject arbitrary messages while a user is logged in. The protocol does not require any nonce carried in the server's notification of incoming messages. We verified that an attacker can indeed succeed in posting malicious links (*e.g.,* to spread virus or spam).

Stocks app: The number one stocks app on the Android market uses Google finance through HTTP to display stock prices. It allows an attacker to inject misleading prices which can cause potential financial loss. Moreover, we verified that instead of blindly injecting HTTP responses to a request (to guess for a particular stock), an attacker can inject "HTTP 301 – Moved Permanently" message to redirect the request to its own server which can read which stocks the app is requesting and send the corresponding fake prices. Unlike a browser with an address bar, such redirection happens transparently.

Advertisement: We tested that advertisements provided through AdMob are fetched over HTTP. An attacker can thus replace the original advertisement with his own to gain revenue. Note that this attack is not intrusive and can be carried out repeatedly to achieve long-term benefits, as long as the malware is kept on the device.

3.6.3 Server-side attack

The "Establish spoofed connections" attack described in §3.3.2.3 allows an attacker to establish connections with a target server using many spoofed IPs. It can be applied in the following scenarios:

Mail server spamming. Using spoofed IPs generally can increase the probability that a spam email is accepted by the mail server since IP-based spam blacklists are unlikely

to catch all bad IPs at once. Without IP spoofing, an IP repeatedly sending spam is likely blacklisted very quickly. We tested that we can successfully deliver emails by simply sending a spoofed data packet (with SMTP commands) to our departmental mail server and acknowledging server's response (via a number of spoofed ACK packets).

DoS of servers. Web server and other public-facing servers are subject to DoS attacks due to a large number of spoofed connections. Note that it is different from SYN flooding in that the connections are actually established, so SYN-cookie-based defense is not effective. We experimented the attack against our own sshd server running on Ubuntu 11.04 (server kernel build) and found that the 0.6 conn/s rate is in fact enough to cause new legitimate ssh connections rejected sporadically when the number of active connections reach a certain limit. We suspect it is due to a security kernel counter-measure triggered to block new connections, which also causes the collateral damage.

3.7 Discussion and Summary

In this chapter, we have seen a diverse set of attacks enabled by the introspective side channels – error packet counters and IPID. However, it would be even more important to understand what actually went wrong and how we can fundamentally correct them. We discuss the following four aspects.

Firewall design. It is interesting and surprising to realize that the more checks the firewall performs, the more information it can leak. For instance, if it checks the four-tuple and allows only packets belonging to an existing session to go through, then an attacker can infer which four tuples are active. If it checks sequence number, then the sequence number inference attack becomes possible. Similarly, if a firewall checks acknowledgment number according to RFC 793 [32] where half of the acknowledge number space is considered valid (as is in the latest Linux TCP stack implementation), then it may allow an attacker to additionally infer the appropriate acknowledgment number, which can help preemptive-SYN attack eliminate the requirement of IP spoofing in the client's network. Our study

suggests that firewall middlebox designs should be carefully evaluated on potential leakage of sensitive network state.

Side-channels. We have summarized two side-channels that serve as feedbacks of the sequence number inference, without which the attack would not be possible. They are intermediate hop IPID and host packet counter. We study whether they are fundamentally difficult to eliminate. For IPID, the answer is negative, as many host OSes such as Linux already use randomized IPIDs. However, for packet counter, it seems that such aggregated information is always available on most OSes and considered harmless. Our study suggests that such information can be abused. One way to mitigate the problem is to add a permission requirement to read such packet counters. However, many users may simply grant the permission. The other important aspect is that the firewall does not check the TCP timestamp option (likely due to overhead concerns) which allows an attacker to leverage the less noisy *PAWSEstab* counter. It suggests a dilemma of the firewall design – it has to tradeoff between performance and the completeness of checks.

Other side-effects. We discover several other notable side-effects of the current host TCP implementation or setup. For instance, the coarse-grained ISN predictability is a byproduct of the Linux TCP implementation. Also, the fact that a server can be kept silent after being reset is caused by the side-effect of the server's host-based firewalls. Interestingly, such implementations and setups are well intended, yet they in fact facilitate the attacks. In the end, we do not think they are the culprit of the problem because even if these two side-effects are eliminated, it prevents only the Reset-the-server hijacking.

HTTPS-only world. In general, SSL should be able to defeat most attacks. Hopefully one of the results of our study is to help push the HTTPS-only world. We do note that even if SSL is employed by the websites, there is a special case where an attack may still succeed. Specifically, when a user types in a URL such as `www.chase.com`, the default browser behavior is to initiate a normal HTTP request first unless the user specifically types in `https://www.chase.com`. It is generally the server that subsequently redirects the

browser to the https site via a "301 – Moved Permanently" HTTP response. Instead of redirecting the browser, an attacker can simply respond directly with a phishing page to the initial HTTP request. In this case, the only difference is that the browser will not show the https icon. However, average users may not notice.

In summary, this chapter has discovered TCP sequence number inference attacks using introspective side channels with the help of firewall middleboxes. We demonstrate that many networks and applications are affected today. We also provide insights on why they occur and how they can be mitigated.

CHAPTER IV

Discovering Generalized TCP Sequence Number Inference Attack

4.1 Introduction

Similar to the previous chapter, this chapter looks at the same attack – TCP sequence number inference attack. The main difference is that we discover attacks that are possible even without the requirement of the sequence-number-checking firewall middlebox. In addition, we summarize a set of new introspective side channels which we name "sequence-number-dependent" packet counters.

To recap, sequence number inference attacks appear only until recently. For instance, in 2007, a study reported in Phrack magazine [5] has claimed that sequence number can be inferred based on how a host treats in-window and out-of-window incoming packets. However, the scope of this attack is rather limited, primarily targeting long-lived connections with rather low success rate (as shown in §4.2.3). In 2012, we have discovered that the sequence number inference attack can be generally applicable, impacting short-lived connections such as HTTP [94]. However, this attack heavily relies on the presence of sequence-number-checking firewall middleboxes deployed in the network. Specifically, the idea is that if a packet has passed the sequence-number-checking firewall, then it implies that the sequence number of the packet is considered within a legitimate window.

This chapter generalizes these attacks by eliminating the strong requirements imposed on them to expose a broader class of attacks. Specifically, we make the following key contributions:

- Building on the threat model presented in the previous chapter, we generalize the sequence number inference attack by demonstrating that it can be reliably carried out without the help of the firewall middleboxes. It provides further evidence that relying on TCP sequence number for security is not an option.
- Distinct from the "error counters" (*e.g.,* packets rejected due to old timestamps) used in the previous chapter, which serves only as an indication of whether a packet is allowed to pass through the sequence-number-checking firewall, in this chapter, we discover a new class of packet counters — "sequence-number-dependent" counters in Linux/Android (1 counter) and BSD/Mac OS (8 counters) — that can directly leak sequence number without requiring the presence of firewall middleboxes, thereby elevating the danger of TCP injection and hijacking attacks.
- We are able to complete the sequence number inference within 4–5 round trips, which is much faster than the one reported in the previous chapter, due to both the property of newly discovered "sequence-number-dependent" counters as well as a more efficient probing scheme. For instance, we show that it takes as little as 50ms to complete, much faster than previously reported results of several seconds and can even eliminate the need of conducting additional TCP hijacking attacks required before, resulting in much higher attack success rate (See §4.4.1).

As a proof-of-concept demonstration, we show our attack allows a piece of unprivileged malware on Android smartphones to hijack a Facebook connection, replacing the login page, or injecting malicious Javascripts to post new status on behalf of the victim user, or performing other actions. All but the TCP hijacking attack works on the latest Linux kernel. TCP hijacking requires kernel version earlier than 3.0.2, which are still the majority case for current Android phones. Besides Android/Linux, we also demonstrated that the attack

is also applicable to the latest BSD/Mac OS. We believe our work presents an important lesson that today's system still exposes too much shared state with poor isolation.

The rest of the chapter is organized as follows: §4.2 explains how to infer TCP sequence number (including both previous study and our discovery). §4.3 covers how we can leverage the sequence number inference as a building block to conduct a number of TCP attacks. §4.4 shows several cases studies demonstrating the impact on specific applications. §4.5 discusses why the problem occurred and concludes.

4.2 TCP Sequence Number Inference Attack

The ultimate goal of the attack is to inject malicious TCP payload into apps running on a victim smartphone or client. It is achieved by a piece of unprivileged on-device malware collaborating with an off-path attacker on the Internet. The main implication of this attack is that websites that do not use HTTPS allow various attacks such as phishing and Javascript injection because the HTTP response can be potentially replaced. Even if HTTPS is used, they are still vulnerable to connection reset attacks as we show the sequence number can be quickly inferred under a second.

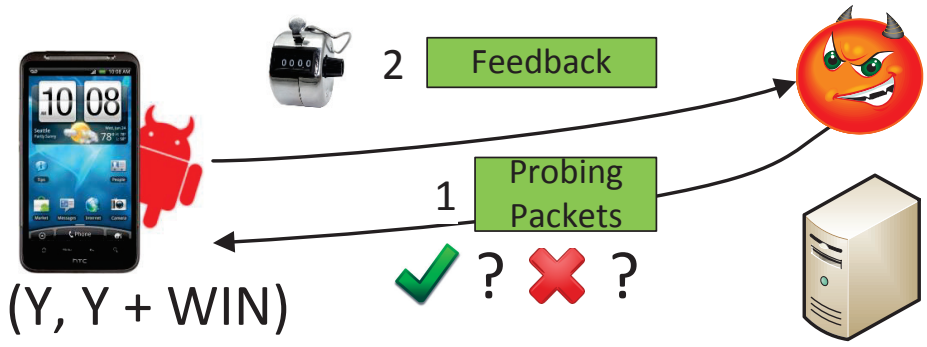


Figure 4.1: Threat model

4.2.1 Threat Model

The threat model is illustrated in Figure 4.1. There are four main entities: 1) The victim smartphone and a target application, constituting the attack target. 2) The legitimate server,

which talks to the victim smartphone using certain unencrypted application-layer protocol (*e.g.,* HTTP). The server can also become the attack target (see §4.4). 3) The on-device malware, which is unprivileged and cannot tamper with other apps directly. 4) The off-path attacker, who is capable of spoofing the IP address of the legitimate server and the victim smartphone. The off-path attacker and the malware collaborate to infer the correct TCP sequence number of the connection established between the target app on the victim smartphone and the legitimate server. Note that different from the threat model described in the previous chapter (§3.2.2), this attack does not require the network firewall middlebox, making our attack model much more general.

At a high level, as shown in Figure 4.1, the off-path attacker needs two pieces of information: 1) the target four tuples and 2) the correct sequence number. The on-device malware can easily tell the current active connections (*e.g.,* through `netstat`), but it does not know the sequence number in use. In this attack model, the off-path attacker can send probe packets using the target four tuples with different guessed sequence numbers. The unprivileged malware then somehow uses certain side-channels to provide feedback on whether the guessed sequence numbers are correct. Guided by the feedback, the off-path attacker can then adjust the sequence numbers to narrow down the correct sequence number.

4.2.2 Packet Counter Side Channels

In this study, we look at a particular type of side channel, packet counters, that can potentially provide indirect feedback on whether a guessed sequence number is correct. In Linux, the *procfs* [81] exposes aggregated statistics on the number of incoming/outgoing TCP packets, with certain properties (*e.g.,* wrong checksums). Alternatively, "`netstat -s`" exposes a similar set of information on all major OSes including Windows, Linux, BSD, Mac OS and smartphone OSes like Android and iOS. Since such counters are aggregated over the entire system, they are generally considered safe and thus accessible to

any user or program without requiring special permissions. IPID side-channel [95] can be considered as a special form of packet counter that records the total number of outgoing packets since it is incremented for every outgoing packet. However, such side-channel is only usable on Windows but is typically very noisy.

Even though it is generally perceived safe, we show that an attacker can correlate the packet counter change with how TCP stack treats a spoofed probing packet with a guessed sequence number. Different from the recent work [94] that uses certain "error counters" as an indication of whether a spoofed packet has passed the sequence-number-checking firewall middlebox, our hypothesis is that the TCP stack may increment certain counters when the guessed sequence number is wrong and remain the same when it is correct, or vice versa. Such counters can directly leak sequence numbers without the help of the firewall middlebox and are thus named "sequence-number-dependent counters" (details in §4.2.4 and §4.2.5). To investigate such possibility, we first need to understand how TCP stack handles an incoming TCP packet and how various counters are incremented during the process.

4.2.3 TCP Incoming Packet Validation

In this section, we provide background on how a standard TCP stack validates an incoming packet that belongs to an established TCP connection. Specifically, we use source code of the latest Linux kernel 3.2.6 as reference to extract the steps taken and checks performed (although the packet validation logic is stable since 2.6.28). More importantly, based on the source code, we discover and summarize "sequence-number-dependent" side-channels on Linux and extend it to BSD/Mac OS.

As we can see in Figure 4.2, we summarize the five main checks performed by Linux TCP stack based on corresponding source code as well as controlled experiment. The check is performed for any incoming TCP packet that is deemed to belong to an established connection (based on source/destination IP and port):

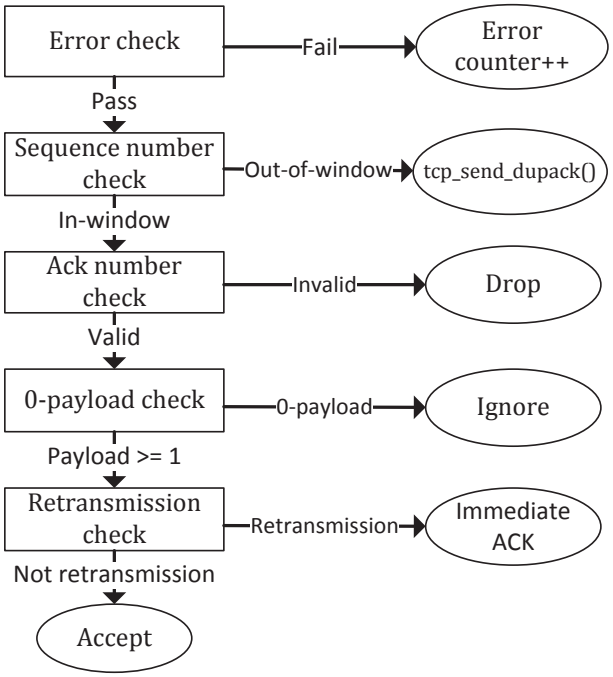


Figure 4.2: Incoming packet validation logic

1). Error check is for the purpose of dropping invalid packets early on. There are a number of specific error checks: 1) MD5 option check, 2) timestamp option check, 3) packet length and checksum check. Each has a corresponding error packet counter. If a specific error is caught, the corresponding host packet counter is incremented and the packet is not inspected further. Otherwise, it goes to the next step.

2). Sequence number check is the most relevant check of interest. It basically checks that if the ending sequence number of the incoming packet is larger than or equal to X, and the starting sequence number is smaller than or equal to X+rcv_win, where X is the next expected sequence number and rcv_win is the current receive window size. If the sequence number is out-of-window, it triggers an immediate duplicate acknowledgment packet to be sent back, indicating the correct sequence number that it is expecting. Otherwise, the next check is conducted.

3). Acknowledge number check is an additional validity check on the packet. A valid ACK number should theoretically be within [Y, Y+$outstanding\_bytes$] to be considered valid. Here Y is the first unacknowledged sequence number and $outstanding\_bytes$ is total

number of outstanding bytes not yet acknowledged. Linux has a relaxed implementation which allows half of the ACK number space to be considered valid (we discuss its impact later). If the ACK number is considered invalid, then it is dropped without further processing. Else, the packet goes through the later non-validity-related checks.

4). At this point the packet has the correct sequence number and ACK number. The stack needs to further check if it has any payload. If it does not have any payload, the packet is most likely silently ignored unless there happens to be pending data that can be piggybacked. In particular, the host cannot send another 0-payload acknowledgment packet for the 0-payload incoming ACK packet which will create endless TCP ACK storm [76].

5). If the packet has non-zero payload, the final check is to detect retransmission by checking if the ending sequence number of the packet is smaller than or equal to the next expected sequence number. If so, it does not further process the packet and immediately sends an ACK packet to inform the other end of the expected sequence number. Since step 2 has already ensured that the ending sequence number cannot be smaller than the next expected sequence number, the only possible ending sequence number that can satisfy the retransmission check is the one equal to the next expected sequence number.

From the above procedure, it is not hard to tell that depending on whether the sequence number is in or out of window, the TCP stack may behave differently, which can be observed by the on-device malware. Specifically, if it is an out-of-window packet with 0-payload, it most likely does not triggers any outgoing packet. However, if it is an in-window packet, it immediately triggers an outgoing duplicate ACK packet. As a result, it is possible to use the counter that records the total number of outgoing packets to tell if a guessed sequence number is in window. Similar observation has been made by the previous study in Phrack magazine [5]. The problem with this approach to infer sequence number is that such general packet counters can be very noisy — there may be background traffic which can increment the system-wide outgoing packet counters. It is especially problematic when the receive window size is small — a large number of packets need to be sent and the probing

is very likely to have limited success. In fact, we have implemented the sequence number inference attack on a smartphone at home connected to the broadband ISP through WiFi with downlink bandwidth of about 10Mbps. Through 20 repeated experiments, we found that the inference always failed because of the noisiness of the counter.

It is also worth noting that the error checks are performed at the very beginning, preceding the sequence number check, which means that the corresponding error counters used by the recent study [94] alone cannot provide any feedback on a guessed TCP sequence number.

4.2.4 Sequence-Number-Dependent Counter in Linux

The reason why the Phrack attack [5] is difficult to carry out is two-fold: 1) the required number of packets is too large, an attacker needs to send at least one packet per receive window in order to figure out the right sequence number range. 2) the counter that records the total number of outgoing packets is too noisy. Interestingly, we show that both problems can be addressed by using a newly discovered set of *sequence-number-dependent packet counters* that increment when the sequence number of an incoming packet match certain conditions.

```
if (TCP_SKB_CB(skb)->end_seq != TCP_SKB_CB(skb)->seq
   && before(TCP_SKB_CB(skb)->seq, tp->rcv_nxt))  {
      NET_INC_STATS_BH(sock_net(sk),
            LINUX_MIB_DELAYEDACKLOST);
      ...
}
```

Figure 4.3: tcp_send_dupack() source code snippet in Linux

Server-side sequence number inference. Specifically, we closely studied the function *tcp_send_dupack()* that is called after the sequence number check (depicted in Figure 4.2). Within the function, we discovered an interesting piece of code shown in Figure 4.3. The "if" condition says if the packet's starting sequence number is not equal to its ending sequence number (*i.e.,* the packet has nonzero payload), and its starting sequence number is "before" the expected sequence number, then a packet counter named *DelayedACKLost* is

incremented (which is publicly accessible from /proc/net/netstat). This particular logic is to detect lost delayed ACK packets sent previously and switch from delayed ACK mode into quick ACK mode [39]. The presence of an old/retransmitted TCP packet is an indication that delayed ACKs were lost.

The question is how "before()" is implemented. In Linux (turns out the same for Mac OS), it basically subtracts an unsigned 32-bit integer from another unsigned 32-bit integer and convert the result into a signed 32-bit integer. This means that half of the sequence number space (*i.e.,* 2G) is considered before the expected sequence number. For instance, two unsigned integer 1G minus 2G would lead to an unsigned integer 3G, when converting to an signed value, it is -1G.

The net effect of the *tcp_send_dupack()* is that it allows an attacker to easily determine if a guessed sequence number is before or after the expected sequence number. Since the *DelayedACKLost* counter very rarely increments naturally (See §4.2.7), an attacker can use this counter as a clean and reliable side-channel.

Binary search. Using this special counter, it is straightforward to conduct a binary search on the expected sequence number. Note that the process is significantly different than the one proposed in the earlier work [94] in that the earlier work still requires sending one packet per "window", which results in a total of thousands or tens of thousands of packets. Here, as illustrated in Figure 4.4, the attacker only needs to send one packet each round and only a total of 32 packets, resulting in hardly any bandwidth requirement.

Specifically, as shown in the figure, in the first iteration, the attacker can try the middle of the sequence number space (*i.e.,* 2G). If the expected sequence number falls in the first half (*i.e.,* bin 1), the *DelayedACKLost* counter increments by 1. Otherwise, (*i.e.,* if it falls in bin 2), the counter remains the same. Suppose the attacker finds that the expected sequence number is in the first half after the first iteration, in the second iteration, he can try 1G to further narrow down the sequence number. After $\log_2 4G = 32$ rounds or packets, the exact sequence number can be pinpointed. The total inference time can be roughly calculated as

$32 \times RTT$. In reality, the number of RTTs can be further reduced by stopping the inference at a earlier iteration. For instance, if it is stopped at 31st iterations, the attacker would know that the sequence number is either X or X+1. Similarly, if the number of iterations is 22, the attacker knows that the sequence number is within [X, X+1024). In many cases, this is sufficient because the attacker can still inject a single packet with payload of 1460 bytes and pad the first 1024 bytes with whitespace (which effectively leaves 436 bytes of effective payload). For instance, if the application-layer protocol is HTTP, the whitespace is safely ignored even if they happen to be accepted as part of the HTTP response.

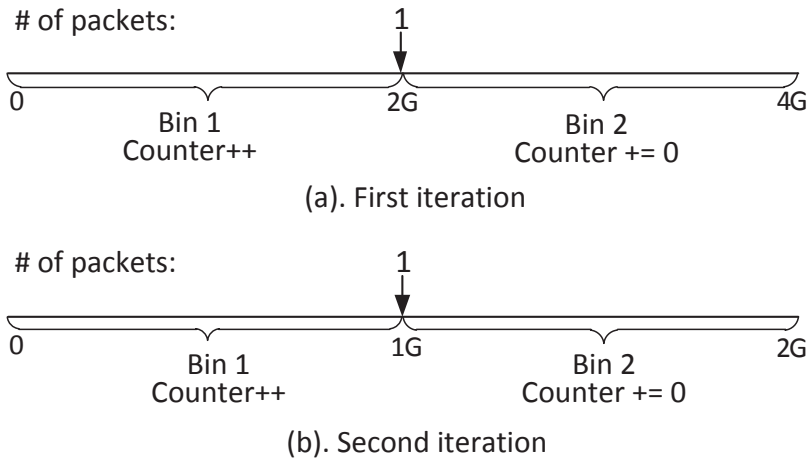


Figure 4.4: Sequence number inference illustration on Linux (binary search)

N-way search. To further improve the inference speed, we devise a variation of the "N-way search" proposed in the recent work [94]. The idea is similar — instead of eliminating half of the sequence number space each iteration, we can eliminate $\frac{N-1}{N}$ of the search space by simultaneously probing N-1 of N equally-partitioned bins. The difference is that inference here requires one or two orders of magnitude fewer packets compared to the previously proposed search.

Figure 4.5 illustrates the process of a 4-way search. In the first iteration, the search space is equally partitioned into 4 bins. The attacker sends one packet with sequence number 1G, three packets with sequence number 2G, and two packets with sequence number 3G. It is not hard to tell that if the expected sequence number falls in the first bin, the *DelayedACKLost* counter increments by 2 as the two packets sent with sequence number 3G

are considered before the expected sequence number. Similarly, the counter increments by a different amount for different bins. In general, as long as the number of packets sent for each bin follow the distance between two consecutive marks on a circular/modular Golomb ruler [12], the *DelayedACKLost* counter increment will be unique when the expected sequence number falls in different bins.

In the later iterations, however, a much simpler strategy can be used. In Figure 4.5(b), an attacker can just send one packet per bin instead of following the circular Golomb ruler. The reason is that now that the search space is reduced to smaller than 2G, it is no longer circular (unlike the first iteration where the counter increment in the first bin can be impacted by the fourth bin). Now, if the sequence number falls in the first bin, then the counter remains the same; if it falls in the second bin, the counter will increment 1; and so on. We discuss the realistic settings and performance of different "N" in §4.2.6.

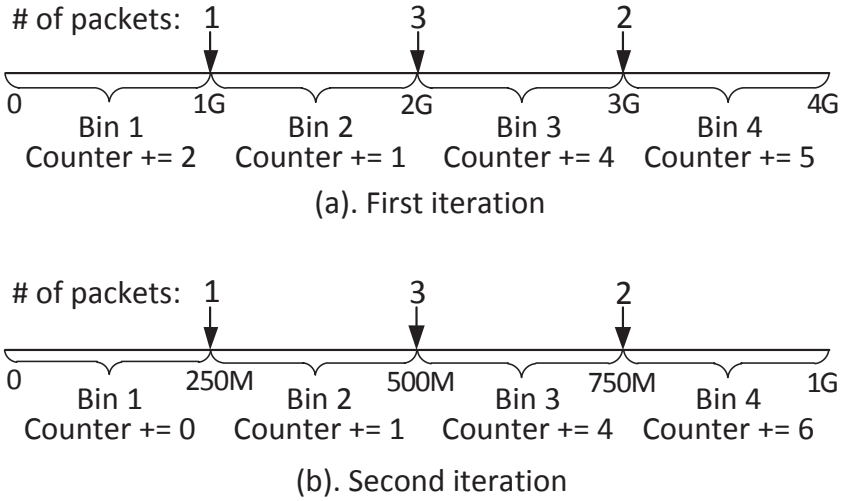


Figure 4.5: Sequence number inference illustration on Linux (four-way search)

Client-side sequence number inference. Sometimes, it is necessary to infer the client-side sequence number, for the purpose of either injecting data to the victim server, or injecting data to the victim client with an appropriate ACK number. The latter is currently unnecessary as Linux/Android and BSD/Mac OS allows half of the ACK number space to be valid [94]. For the former, it turns out that we can still use the same *DelayedACKLost* counter to infer the ACK number.

Specifically, as discussed in §4.2.3, the only ending sequence number that can satisfy the retransmission check is the one equal to the next expected sequence number. When that happens, the TCP stack increments the *DelayedACKLost* packet counter again. The source code of the retransmission check is shown in Figure 4.6.

Since the retransmission check is after the ACK number check, it allows an attacker to send a non-zero payload packet that has the ending sequence number equal to the next expected sequence number with a guessed ACK number. If it does not pass the ACK number check, the packet is dropped and *DelayedACKLost* counter does not increment. Otherwise, the packet is considered a retransmission packet and triggers the counter to increment. Based on such behavior, it is not hard to come up with a binary search or N-way search on the ACK number similar to the sequence number search. In fact, the procedure is mostly identical.

```
if (!after(TCP_SKB_CB(skb)->end_seq, tp->rcv_nxt)) {
        NET_INC_STATS_BH(sock_net(sk),
                LINUX_MIB_DELAYEDACKLOST);
        ...
}
```

Figure 4.6: Retransmission check source code snippet from tcp_data_queue() in Linux

4.2.5 Sequence-Number-Dependent Counters in BSD/Mac OS

Inspired by the newly discovered counter in Linux, we further conducted a survey on the latest FreeBSD source code (version 10), and surprisingly found that at least four pairs of packet counters can leak TCP sequence number. The counters are confirmed to exist in Mac OS as well. This shows that the sequence-number-dependent counters are widely available and apparently considered safe to include in the OS. They are: 1) *rcvduppack* and *rcvdupbyte*; 2) *rcvpackafterwin* and *rcvbyteafterwin*; 3) *rcvoopack* and *rcvoobyte*; 4) *rcvdupack* and *rcvacktoomuch*. They can be either accessed through standard "netstat -s" interface or sysctl API [38].

The first three pairs can be used to infer server-side sequence number. Specifically,

based on the source code, the semantic of *rcvduppack* is identical to that of *DelayedACK-Lost*. *rcvdupbyte*, however, additionally provides information on the number of bytes (payload) carried in the incoming packets that are considered duplicate (with an old sequence number). This counter greatly benefits the sequence number inference. Specifically, following the same "N-way" procedure, the first iteration can be improved by changing the "k packets sent per bin" to "a single packet with k bytes payload". This improvement substantially reduces the number of packets/bytes sent in each iteration, especially when "N" is large (shown in §4.2.6).

The semantic of *rcvpackafterwin* and *rcvbyteafterwin* is similar to *rcvduppack* and *rcvdupbyte*, except that the former will increment only when the sequence number is bigger than (instead of smaller than) certain sequence number X. In this case, X is the expected sequence number plus the receive window size. *rcvbyteafterwin* can be used similarly as *rcvdupbyte* to conduct the sequence number inference.

rcvoopack and *rcvoobyte* differ from the previous two pairs. They increment only when packets arrive out-of-order, or more precisely, when the sequence number is bigger than the expected sequence number yet smaller than the expected sequence number plus the receive window size. Even though an attacker needs to send a lot more packets to infer the TCP sequence number using this counter pair, at least they can be used to replace the original noisy side-channel in the Phrack attack [5] to improve success rate.

rcvdupack and *rcvacktoomuch* are used to determine the client-side sequence number. Specifically, the former increments when the ACK number of an incoming packet is smaller than or equal to the unacknowledged number (SND.UNA). The latter increments when the ACK number is greater than the sequence number of the next original transmit (SND.MAX). The comparison again follows the "unsigned integer to signed integer conversion" such that half of the ACK number space is considered to match the condition.

We currently did not combine the counters together to improve the inference speed. However, we do realize there are potential ways to speed things up. For instance, the

rcvdupbyte and *rcvdupack* allows the client-side sequence number inference to be piggybacked with the server-side sequence number inference.

4.2.6 Inference Performance and Overhead

We have implemented the sequence number inference on both Android (which incorporates the Linux kernel) and Mac OS. We are interested in knowing the tradeoffs between different strategies in picking "N" in the "N-way search".

Generally, as "N" goes up, the number of bytes sent in total should also increase. Since the first iteration in the "N-way" search requires sending more bytes, we pick a smaller "N" for the first iteration and a bigger "N" in the later iterations to ensure that the number of bytes sent in each round is similar. In the Linux implementation, we pick the following pairs of N, (2/2, 4/6, 8/30, 12/84); For Mac OS, we pick (2/2, 4/6, 34/50, 82/228). Here 4/6 indicates that we pick N=4 for the first iteration and N=6 for the later iterations.

As shown in Figure 4.7, we can see that the general tradeoff is that the fewer iterations an attacker wants, the more bytes he needs to send in total. For instance, when the number of iterations is 4, Linux attack requires sending 13.7KB. Due to the advantage of the *rcvdupbyte* counter in Mac OS, it requires to send only 8.4KB. This is a rather low network resource requirement because sending 8.4KB would take only less than 70ms with even just 1Mbps bandwidth. Going further down to 3 iterations requires sending 27.75KB for Mac OS, depending on the available bandwidth and the RTT, the saving of one round trip may be overcome by the increase in the number of bytes.

Next, we pick N=34/50 (4 round trips) for the Mac OS attack, and N=8/30 (5 round trips) for Linux attack (with roughly the same resource requirement), and plot the inference time measured under various conditions. Specifically, we control the RTT between the attacker and the victim in three different settings: 1) victim is in an office environment (enterprise-like) connected to the network using WiFi and the attacker is in the same building (the RTT is around 5-10ms). 2) victim is in a home environment and the attacker is

50ms away. 3) victim in a home environment and the attacker is 100ms away. In Figure 4.8, we can see that the inference time for Android and Mac OS are as low as 80ms and 50ms in the first setting, which are low enough to directly launch injection attacks on HTTP connections with the guarantee that the inference finishes before the first legitimate response packet comes back (also discussed later in §4.3.2). In fact, inference time between 350ms and 700ms is also short enough in certain scenarios (see §4.4.1).

4.2.7 Noisiness of Sequence-Number-Dependent Counters

So far, we have claimed that these sequence-number-dependent counters are "clean" side-channels that rarely increment naturally due to background traffic. To quantitatively support that, we conduct a worse-case-scenario experiment as follows: we open a YouTube video at the background and browse web pages at the same time to see how often the counters increment. Since it is easier to do the multi-tasking on Mac OS, we chose it over Android. The Android counters should increment even less frequently since it is rare that smartphones are used for video streaming and web browsing simultaneously.

Specifically, we pick the *rcvdupbyte* counter (which is equivalent to *DelayedACKLost* on Linux) and run the experiments for about 8.5 minutes. We made sure that the video is long enough that it was not fully buffered before the end of the experiment. To quantify the counter noisiness, we break down the time into 30ms intervals to mimic the window of exposure during one round of probing, and then count how many intervals in which we observe any counter increment. As expected, there are only 10 intervals out of 16896 that have the increment. This indicates that the probability that the counter may increment due to noise and interfere with one round of probing is roughly 0.059%. Even if there are 22 rounds (worse case), the probability that the entire probing will be impacted is only 1.2%.

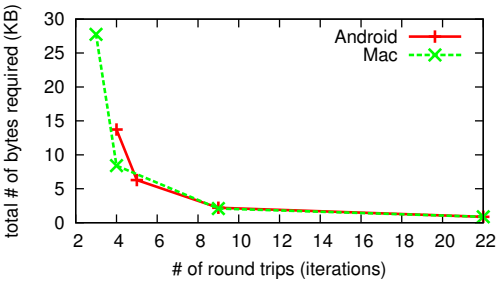


Figure 4.7: Tradeoff between inference speed and overhead

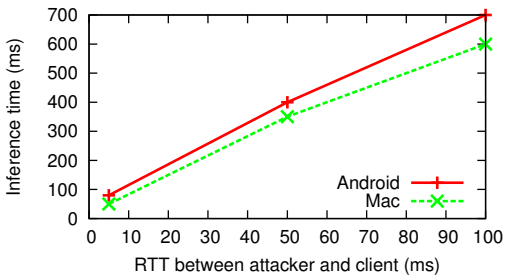


Figure 4.8: Relationship between RTT and inference time

4.3 Design and Implementation of TCP Attacks

In the previous section, we described how to efficiently and reliably infer TCP sequence number using the newly discovered set of sequence-number-dependent packet counters. Since the sequence number inference only takes under a second, it can be fast enough to launch many application-layer attacks. In this section, we discuss four possible TCP attacks that can be launched against a variety of applications. All of the attacks leverage the TCP sequence number inference as the essential building block, but the main difference is in the timing and reliability with slightly different requirements. Even though we have implemented the attack on both Android and Mac OS. We use the Android as the running example.

Injection vs. Hijacking. Using the same terminology as a recent work [94], we define TCP hijacking to be the more powerful attack than TCP injection. Specifically, TCP hijacking allows an attacker to inject packets right after the TCP 3-way handshake. For instance, it enables an attacker to inject a complete HTTP response without any interference. In contrast, *TCP Injection* is more general and does not require this capability.

The four attacks are named as: 1). Client-side TCP Injection. 2). Passive TCP hijacking. 3). Active TCP hijacking. 4). Server-side TCP injection.

4.3.1 Attack Requirements

There are a number of base requirements that need to be satisfied for all of the TCP attacks considered. Note that our attacks have much fewer requirements than the one proposed in the recent study [94]. Specifically, we do not require a firewall middlebox in the network to help the attack, which makes our attack applicable in a much more general environment.

1). Malware on client with Internet access. 2). Malware can run at the background and read packet counters. 3). Malware can read active TCP four tuples. 4). Predictable external port number if NAT deployed. The first three requirements are straightforward. All of the Android applications can easily request Internet access, read packet counters (*i.e.,*/proc/net/netstat and /proc/net/snmp, or "netstat -s"), and read active TCP connections' four tuples (*e.g.,* through /proc/net/tcp and /proc/net/tcp6, or "netstat"). The requirements can be easily satisfied on most modern OSes as well. The fourth requirement is required as the off-path attacker needs to know the client's external port mapping to choose the correct four tuples when sending probing packets. This requirement is also commonly satisfied. Many NAT mapping types allow external port to be predictable to facilitate NAT traversal. For instance, our home router directly maps the internal port to the external port. According to recent measurement studies on the NAT mapping types [73, 111], the majority of the NATs studied do allow have predictable external port. Further, even if the prediction is not 100% accurate, the attack can at least succeed probabilistically.

Additional requirements for passive TCP hijacking are C1 and S1:

C1). Client-side ISN has only the lower 24-bit randomized. This requirement is necessary so that the malware can roughly predict the range of the ISN of a newly created TCP connection. In Linux kernel earlier than 3.0.2, the ISN generation algorithm is designed such that ISNs for different connections are not completely independent. Instead, the high 8 bits for all ISNs is a global number that increments slowly (every five minutes). The design is to balance among security, reliability, and performance. It is long perceived as

a good optimization, with the historical details and explanations found in this article [17]. The result of this design is that the ISN of two back-to-back connections will be at most $2^{24} = 16,777,216$ apart. Even though it is a design decision and not considered a "vulnerability", since Linux 3.0.2, the kernel has changed the ISN generation algorithm such that two consecutive connections will have independent ISNs. The majority of Android systems that are on the market, however, are still on Linux 2.6.XX, which means that they are all vulnerable to the passive TCP hijacking attack.

S1). The legitimate server has host-based stateful TCP firewall. Such a firewall is capable of dropping out-of-state TCP packets. Many websites such as Facebook and Twitter deploy such host firewalls to reduce malicious traffic. For instance, iptables can be easily configured to achieve this [23]. Note that interestingly this security feature on the server turns out to help enable the TCP hijacking attack.

For the active TCP hijacking, the additional requirements besides the base requirements are S1 and C2:

C2). Client-side ISN monotonically incrementing for the same four tuples. This client-side requirement is in fact explicitly defined in RFC 793 [32] which is to prevent packets of old connections, with in-range sequence numbers, from being mistakenly accepted by the current connection. Even though the latest Linux kernel has eliminated the requirement C1, C2 is still preserved.

4.3.2 Client-Side TCP Injection

In this attack, an attacker attempts to inject malicious data into a connection established by other apps on the phone. The essential part of the attack is the TCP sequence number inference which has already been described in detail. Here we describe at a high level how an attacker can inject meaningful data that may potentially experience race condition with the data sent from the legitimate server. For instance, considering the connection under attack is an HTTP session where a valid HTTP response typically follows immediately

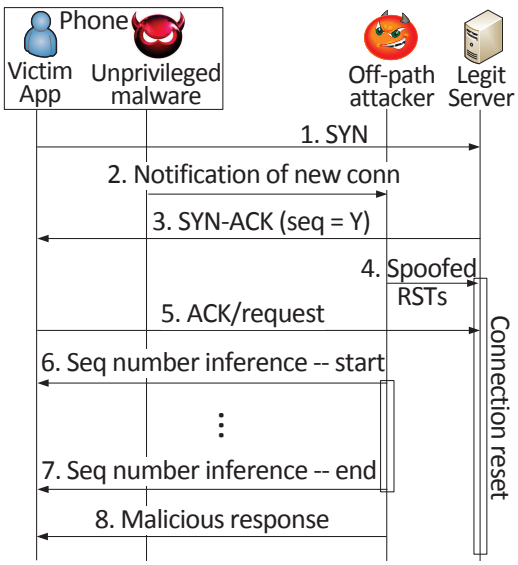


Figure 4.9: Passive TCP hijacking sequence

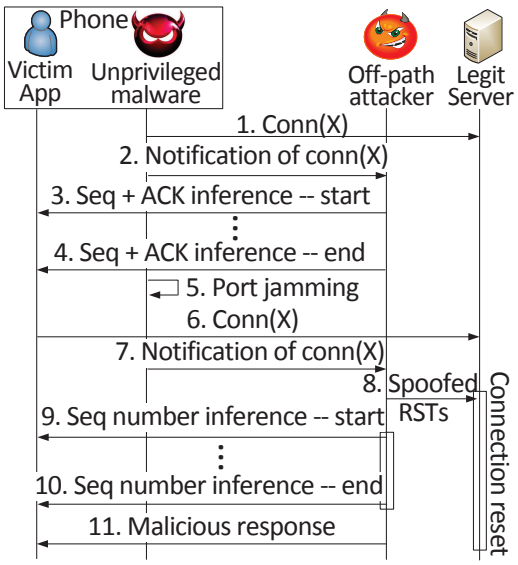


Figure 4.10: Active TCP hijacking sequence

after the request is sent, by the time the sequence number inference is done, at least part of the HTTP response is already sent by the server. The injected HTTP packets likely can only corrupt the response and cause denial of service instead of serious damage.

Even though the timing requirement sounds difficult to satisfy, we did implement this attack against websites like Facebook where we are able to inject malicious Javascript to post new status on behalf of a victim user. The detail is described in §4.4.1.

The idea is to leverage two common scenarios: 1) The server may take a long time to process the request and assemble the response. This is especially common as many services (websites) take longer than 100ms or more to process a request. The fact that the sequence number inference time in certain scenarios (when RTT from the server to the client is small) can be made below 100ms makes the injection attack as powerful as hijacking. 2) A single TCP connection is reused for more than one pair of HTTP request and response. The idea is to use the inferred sequence number for injecting malicious data not on the first HTTP request but the later ones. Both cases allow an attacker enough time to conduct sequence number inference.

4.3.3 Passive TCP Hijacking

Passive TCP hijacking allows an attacker to hijack TCP connections that are passively detected. This means that the attacker can hijack TCP connections issued by the browser or any other app, regardless of how and when they are made. It is the most powerful TCP attack in this study. As demonstrated in §4.4, with this attack, it is possible to replace the Facebook login page with a phishing one.

The high-level idea is the same as proposed in the recent work [94], which is to reset the connection on the legitimate server as soon as possible to allow the attacker to claim to be the legitimate server talking to the victim. The key is that such reset has to be triggered right after the legitimate server sends SYN-ACK. Requirement C1 allows the malware and the attacker to predict the rough range of victim's ISN and send reset packets with sequence numbers in that range. This is helpful because then the attacker is required to send much fewer spoofed RST packets (thus with lower bandwidth requirement) compared to enumerating the entire 4G space. Further, after the legitimate server is reset, requirement S1 is necessary since it helps prevent the legitimate server from generating RST upon receiving out-of-state data or ACK packets from the victim.

The attack sequence diagram is shown in Figure 4.9. Time steps 1 to 3 are the same as the previous attack where the unprivileged malware detects and reports the newly established TCP connection. In addition, the malware also needs to establish a connection to the off-path attacker to report the current ISN value (high 8 bits). With this information, at time 4, the off-path attacker can flood the legitimate server with a number of spoofed RST enumerating the lower 24 bits. Note that the RST packets have to arrive before the ACK/request packets at time 5; otherwise, the server may send back the response packets before the attacker. Of course, the server may need some time to process the request as well, which can vary from case to case, allowing the attacker additional time for completing the reset procedure. After the legitimate server's connection is reset, all future packets from the victim app will be considered out-of-state and silently dropped due to require-

ment S1. For instance, the ACK packet received at time 5 is silently discarded. From time 6 to 7, the attacker conducts the sequence number inference described earlier and injects malicious content afterwards at time 8 with the inferred sequence number.

4.3.4 Server-side TCP Injection

In this attack, an attacker tries to inject malicious payload into a victim connection, destined for the server (as opposed to the client). For instance, as shown in the case study in §4.4, we are able to target at Windows live messenger protocols to inject malicious commands to cause persistent changes to the victim user account, *e.g.,* adding new friends or removing existing friends.

This attack is straightforward by combining the sequence number inference and ACK number inference as described in §4.2. We omit the detailed attack sequence as it does not include other important steps. This attack has no additional requirements besides the base requirements. In general, applications with unencrypted and stateful protocols are good attack targets.

4.3.5 Active TCP Hijacking

In this attack, an attacker also attempts to hijack connections. However, because the latest Linux kernel since 3.0.2 has the entire 32-bit randomized for ISNs of different four tuples, requirement C1 is no longer satisfied. In this case, we show that it is still possible to launch a weaker version of TCP hijacking by "actively" performing some offline analysis as long as requirement C2 is satisfied. As shown in §4.4, we have successfully used the port-jamming-assisted active TCP hijacking to replace a Facebook login page with a phishing one.

Requirement C2 specifies that the ISN for the same four-tuple always increments with time. This implies that as long as an attacker can infer the client's ISN for a particular four-tuple once, he can remember the value for a future connection that reuses the same

four-tuple, and reset the server using the remembered ISN (plus the increment by time) so that the attacker can hijack the connection.

The detailed attack sequence is demonstrated in Figure 4.10, at time 1, the unprivileged malware establishes a connection on its own to a target server of interest (*e.g.,* Facebook server), and notifies the off-path attacker immediately (at time 2) so that it can infer the client ISN of the used four tuples (through time 3 to 4). Now, assuming the attacker knows that a victim app is about to initiate a connection to the same server, an attacker can immediately perform port jamming to exhaust all the local port numbers (at time 5) so that the victim app's connection can only use the local port number that was in the inferred four tuples (we will describe how port jamming can be done later). Now that the victim connection reuses the same four tuples, the malware can immediately notify the off-path attacker (at time 6) which uses the previously inferred client-side ISN to reset the server (at time 7). Subsequently, the attack sequence is identical to the end of passive TCP hijacking.

In the above attack sequence, one critical part is the knowledge of when the victim app initiates the connection to the target website. One simple strategy is to actively trigger the victim app to make the connection through the unprivileged malware. On Android, for instance, any app could directly invoke the browser going to a given URL, before which the attacker can perform the port jamming.

One alternative strategy is to perform offline analysis on as many four tuples as possible so that it can essentially obtain the knowledge of ISN for all possible four tuples going to a particular website (without requiring port jamming). This way, after the offline analysis is performed, the attacker basically can launch passive TCP hijacking on any of the four tuples that have been previously analyzed. Since each client-side ISN inference should take a little over a second, an attacker can infer, for instance, 1000 four tuples in 15 minutes. Even though a connection to Facebook may have 1% probability falling in the range, the user may repeatedly visit the website and the probability that all of the connections failing to match any existing four tuples is likely very low. We have verified that the ISN for the

same four-tuple does increment consistently over time for over an hour. We suspect that the cryptographic key for computing ISN does not change until reboot in Linux 3.0.2 and above.

To jam local ports, the unprivileged malware can simply start a local server, then open many connections to the local server intentionally occupying most of the local port except the ones that are previously seen for inference. One challenge is that the OS may limit the total number of ports that an application can occupy, thus preventing the attacker from opening too many concurrent connections. Interestingly, we found such limit can be bypassed if the established connections are immediately closed (which no longer counts towards the limit). The local port numbers are not immediately released since the closed connections enter the TCP_TIME_WAIT state for a duration of 1 to 2 minutes.

4.4 Attack Impact Analysis from Case Studies

Experiment setup. As discussed earlier, even though our attacks are implemented on both Android and Mac OS, we choose to focus on Android in our implementation and experiments. We use two different phones: Motorola Atrix and Samsung Captivate. We verified that all attacks work on both Android phones, although the experimental results are collected based on Atrix. The WiFi networks include a home network and a university network. The off-path attacker is hosted on one or more Planetlab nodes in California.

We describe four case studies corresponding to the four TCP attacks proposed in the previous section. We also present experimental results such as how likely we can succeed in hijacking the Facebook login page based on repeated experiments.

For all attacks, we implemented the malware as a benign app that has the functionality of downloading wallpapers from the Internet (thus justifying the Internet permission). Since the malware needs to scan netstat (or /proc/net/tcp and /proc/net/tcp6 equivalently) for new connection detection, which can drain the phone's battery very quickly, we make the malware stealthy such that it only scans for new connection when it detects that the

victim app of interest is at the foreground. This can be achieved by querying each app's *IMPORTANCE_FOREGROUND* flag which is typically set by the Android OS whenever it is brought to the foreground. Further, the malware only queries the packet counter when the off-path attacker instructs it to do so. The malware is only used in our controlled experiment environments without affecting real users.

Note that most apps except the browser on the smartphones do not have an indication about whether the connection is using SSL, which means that the users may be completely unaware of the potential security breach for unencrypted connections (*e.g.,* HTTP connections used in the Facebook app).

4.4.1 Facebook Javascript Injection

We build the attack on top of **client-side TCP injection** as described in §4.3.2. Recall that the injection can happen only after the sequence number inference finishes. If the inference cannot be done earlier than the response comes back, the attacker will miss the window of opportunity.

By examining the packet trace generated by visiting the Facebook website where a user is already logged in, we identified two possible ways to launch the Javascript injection attack. The first attack is surprisingly straightforward. Basically, when the user visits `m.facebook.com`, the browser issues an HTTP request which would fetch all recent news. We observe that it consistently takes the server more than 1.5 seconds to process the request before sending back the first response packet. According to our results in §4.2.6, the inference time usually finishes within 0.7s even when RTT=100ms. It allows enough time for an attacker to inject the malicious response in time (or inject a phishing login page as well). As shown in Table 4.1, the success rate is 87.5% based on 40 repeated experiments in our home environment where RTT=100ms. It goes up to 97.5% when the experiment is conducted in the university network where RTT=70ms. The failed cases are mostly due to packet loss.

	RTT_a=70ms[1]	RTT_a=100ms
Succ Rate	97.5% (39/40)	87.5% (35/40)

[1] RTT_a is the RTT between the attacker and the client

Table 4.1: Success rate of Facebook Javascript injection (case study 1)

The second attack is based on the observation that multiple requests are issued over the same TCP connection to Facebook site. Even if the attacker is not able to infer the sequence number in time to inject response for the first request (*e.g.,* Facebook may improve the server processing time in the future), he can still do that for the second request. Specifically, if the user visits the root page on Facebook, the browser on one of the Android phones (Samsung Captivate) will send two HTTP requests: the first request is asking for the recent news; the second request seems to be related to prefetching (*e.g.,* retrieving the friend list information in case a user click on any user for detailed information).

Since there is a delay of about 1s between the end of the first request and the start of the second request. An attacker can monitor if the sequence number remains the same for certain period of time to detect the end of the first response. Furthermore, the second request takes about 100ms to process on the server. A simple strategy that an attacker can employ is to just wait for around 1.1s before injecting the malicious response for the second request. A more sophisticated attacker could also monitor the start of the second request by tracking the current ACK number. Specifically, when the second request is sent, the valid ACK number range moves forward by the number of bytes in the request payload.

In our proof-of-concept implementation, we always inject the Javascript after waiting for a fixed amount of time after the connection is detected, which can already succeed for a few times. However, a more sophisticated attacker can definitely do better.

4.4.2 Phishing Facebook Login Page

We build this attack on top of **passive TCP hijack** which passively monitors if a new connection to Facebook is made. In this case study, we look at how to replace the Facebook login page by resetting the Facebook immediately after it has responded with SYN-ACK.

	One Planetlab node		Two Planetlab nodes	
	RTT_b=70ms[1]	RTT_b=100ms	RTT_b=70ms	RTT_b=100ms
Succ Rate	42.5% (17/40)	47.5% (19/40)	62.5% (25/40)	82.5% (33/40)

[1] RTT_b is the RTT between the attacker and the Facebook server

Table 4.2: Success rate of Facebook login page injection (case study 2)

We assume that the user is not already logged in to Facebook. Otherwise, as described in the previous attack, the server processing delay for the first HTTP request is so long that is is too easy to succeed. When the user is not logged in, the server processing delay will become negligible and the effective time window for reset to succeed is basically a single round trip time. This scenario is also generic enough that can be applied in many other websites such as `twitter.com`.

In Table 4.2, we show how likely the attack can succeed under different conditions. For instance, when there's a single Planetlab node, the success rate is a little below 50%. However, the success rate increases significantly to 62.5% and 82.5% when we use two nodes for latency values of 70 and 100ms respectively, indicating that we have more bandwidth to reset the server. In addition, the result also verified that the larger the RTT, the more likely the attack can succeed.

Note that the 100ms RTT to Facebook may sound very large given the popularity of CDN services. However, the CDNs are mostly for hosting static content such as images and Javascripts, *etc.* For webpages that are highly customized and dynamic (*e.g.,* personalized Facebook news feed), they are very likely to be stored on the main server in a single location (*e.g.,* Facebook main servers are hosted in California). We found that this is a common design for many sites that have significant dynamic content (*e.g.,* twitter).

4.4.3 Command Injection on Windows Live Messenger

Built on the **server-side TCP injection** described in §4.3.4, the case study of command injection attack on Windows Live Messenger is an interesting example of server-side attack carried out on a connection where the user is already logged in. The main connection of Windows Live Messenger runs on port 1863 and uses Microsoft Notification Protocol

(MSNP) which is a fairly complex instant messenger protocol developed by Microsoft [20]. Many Windows Live Messenger clients on the Android as well as on the desktop (including official ones) run the connection in plaintext, thus allowing the attack. Once upon the detection of a vulnerable Windows Live Messenger app running or a connection established to known port numbers and IP addresses that are associated with the app, an attacker can start this attack.

We have verified the commands that are possible to inject into the server include, but not limited to, adding a new friend or removing an existing friend (specified by the account email address), changing the status messages, and sending messages to friends. Given that the messenger client is idle most of the time and the fact that the client-side sequence number inference only takes 2–3 seconds, the attack can be launched fairly easily. The commands can cause serious damage. For instance, the add friend command allows an attacker to add its malicious account as a friend which can subsequently send spam or phishing messages. In addition, after being added as a friend, the attacker can read the friend list (email accounts) of the victim user and either delete them or spam them. Finally, new status posting can be part of the phishing attack against the friends as well.

4.4.4 Restricted Facebook Login Page Hijack

This attack is built on top of **active TCP hijack** as described in §4.3.5, the goal of this attack is still to hijack TCP connections. However, due to a lack of ability to reset the server-side connection in the new version of the Linux kernel, it requires offline analysis on the client-side ISN of the target four tuples.

In our implementation, we made a simple Android test malware that performs the offline analysis right after it is started. The four tuples we target include a pre-selected local port and the Facebook server IP that's resolved for m.facebook.com. After the analysis, which takes a little over a second, it performs port jamming immediately (which takes about 5 seconds). After this, our malware app immediately sends an Intent that asks to

open the m.facebook.com through the browser. An attacker may come up with reasons such as asking a user to use Facebook account to register for the app. When the browser starts the connection to Facebook, the malware works with the off-path attacker to hijack the connection (as described in §4.3.5. We have verified that the Facebook login page can indeed be hijacked following these steps.

The main difficulty in this attack is not about successfully inferring the sequence number. Instead, it requires the user to be convinced that the app indeed has a relationship with the target website (*i.e.,* Facebook) so that they are willing to enter the password into the browser.

4.5 Conclusion

After having demonstrated the attacks, we step back and ask ourselves what have gone wrong. In summary, here are a few lessons that we learned: 1) Seemingly harmless aggregated information in fact does not have enough entropy (*DelayedACKLost*) and can leak critical internal network/system state. The designers of new packet counters in the future should carefully evaluate how such counters may leak critical system state. 2). Our systems today still have too much shared state: the active TCP connection list shared among all apps (through netstat or procfs); the IP address of the malware's connection and other app's; the global packet counters. Future system and network design should carefully evaluate what information the adversaries can obtain through these shared state.

On the defense side, we could consider: 1) removing the unnecessary global state (such as the active TCP connection list and packet counters) or only allow privileged programs to access such state. 2) providing better isolation of the resources such as providing a separate set of packet counters for different apps, or even different source IP addresses for connections in different processes such that the malware will not be able to know the IP of the connection established by another process (which may become feasible when IPv6 is widely deployed). In the extreme case, it will be safest that each app runs in its own virtual

machine.

To conclude, we have demonstrated an important type of TCP sequence number inference attack enabled by host packet counter side-channels under a variety of client OS and network settings. We also offer insights on why they occur and how they can be mitigated.

CHAPTER V

Analyzing Triangular Spamming: a Stealthy Spamming Technique

5.1 Introduction

In this chapter, we analyze the impact of a novel and interesting attack which we call "triangular spamming" using the IPID introspective side channel in the spam domain. The main theme of the chapter is to use side-channel-enabled inference to understand how many ISPs' firewall port 25 blocking policies can be bypassed by triangular spamming.

Despite all the past efforts in spam mitigation, the problem still remains unsolved. There are two categories of spam filtering techniques: content-based and blacklist-based. While content-based filtering is the canonical way, blacklist-based approach (*e.g.,* Spamhaus, Spamcop [36, 35]) is receiving much attention recently because it does not rely on email content and may be more efficient and less susceptible to evasion. While IP-based blacklist is simple and lightweight, compiling and maintaining such a list is challenging due to the changing landscape of compromised hosts: more hosts can become compromised; they could change IP addresses over time; and they may also be patched. As a result, it is not surprising that most IP blacklists provide a very limited coverage of malicious IPs involved in sending spam [100].

Further, as spam detection and prevention techniques evolve, so do spamming tech-

niques. Spammers are increasingly more stealthy by restricting each IP or compromised host to only send very few spam messages to each target in order to stay under the radar [106]. In the meanwhile, ISPs are enforcing the outbound SMTP (port 25) blocking policy for their end-hosts in an effort to reduce spam originated from their networks [27, 28].

In this chapter, we systematically study triangular spamming, a clever spamming technique that has been known for several years, but never systematically studied. Triangular spamming, as its name suggests, involves three main parties, target mail server, original spam sender (or high-bandwidth bot) and relay bot (or low-bandwidth bot). The key idea is that with relay bots' cooperation, the original sender (high-bandwidth bot) can send spam in high throughput while hide its own IP address by spoofing the relay bots' IP addresses. In a recent NANOG survey [22], although people are already aware of such problem, our study shows that most ISPs still do not enforce the correct SMTP blocking policy to prevent triangular spamming.

We focus on three key questions:

1. What are the requirements of triangular spamming, and is today's network vulnerable to such spamming behavior?

2. What are the benefits of triangular spamming, and is it used in the wild today?

3. What are the possible solutions to prevent or mitigate such spamming approach?

As triangular spamming essentially exploits network-level vulnerability, it requires a detailed understanding of network operational practices that are usually overlooked in security research domain. In this chapter, we surveyed the network policy practices in addition to conducting large-scale experiments to verify and explore current network policies of various ISPs. More specifically, we are focusing on the port blocking policy employed by ISPs.

Our study makes the following contributions:

1. We designed an accurate and effective probing technique to discover the networks that tried to block outgoing port 25 traffic but failed to enforce the correct port blocking

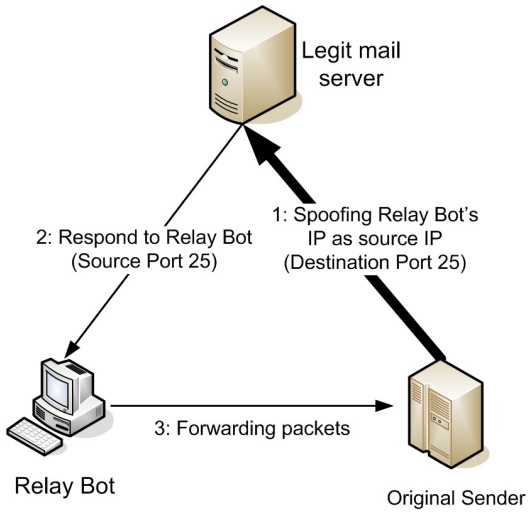


Figure 5.1: Triangular spam delivery example

policy, thus are vulnerable to triangular spamming.

2. We found that that 97% of the blocking networks fall into the above category. Only 44% of such prefixes are listed on Spamhaus PBL [102].

3. We conduct experiments to ascertain the existence of triangular spamming at the mail server side.

4. We systematically evaluated the feasibility and benefits of triangular spamming via experiments of actually deployed setups on Planetlab. Based on the operational experience, we discuss promising prevention and detection approaches to triangular spamming.

The remainder of the chapter is structured as follows: § 5.2 describes the basic requirements and implication of triangular spamming. § 5.3 studies the port blocking policy extensively for thousands of networks. § 5.4 describes our experience and lessons learned from building triangular spamming and deploying it on Planetlab on our own. § 5.5 describes possible detection and prevention techniques and ascertain the existence of triangular spamming. § 5.6 concludes our work.

5.2 Triangular Spamming Mechanism and Implication

As shown in Figure 5.1, triangular spamming exploits IP spoofing to route packets indirectly for the purpose of hiding the identity of actual sending hosts and increasing spam throughput. The spammer picks one or more high-bandwidth bots (or original sender) to send spam directly to target mail server while spoofing the source IP addresses of relay bots. These bots listen for any relevant packets, *e.g.,* those from port 25 from the mail server and forward them back to the original spammer.

5.2.1 Triangular spamming requirement

IP spoofing is allowed at the origin sender network. IP spoofing has long been studied for implications such as DoS attacks [68]. Although the problem has been studied extensively for two decades or so, it is still largely unsolved due to various deployment challenges, *e.g.,* the network policy for enforcing anti-spoofing such as unicast reverse path forwarding (uRPF) [52, 41] is limited by multi-homing, route asymmetry, complexity of managing and updating the filtering rules. Indeed, based on the Spoofer study [56], 31% of the IP addresses studied allow successful spoofing of an arbitrary, routable source address. We perform study from our side to determine the IP spoofing ability as well to try to ascertain the feasibility of triangular spamming.

Traffic from mail server to relay bots are not blocked. As we can observe in Figure 5.1, even though the relay bot does not have to contact the mail server directly, it must receive packets from the mail server in order to relay them back to the original sender. However, if such traffic is blocked, then triangular spamming will fail to operate. In §5.3, by conducting intelligent probing to infer port blocking policy, we show that most ISPs do not block such traffic today. On the other hand, traffic from the relay bot to the original sender can be easily tunneled and encrypted so that it can be hard to detect and filter.

Also, it is generally more difficult for NATed hosts to participate as relay bots given that they will have to be able to receive packets on a specific source port. However, with the

development of NAT traversal techniques such as uPnP [42] (many home routers by default enable this feature), it won't be difficult for compromised hosts to initiate request to add or modify port mappings on their routers. In fact, previous attack has shown that a malicious website can use Flash to control the client's uPnP-enabled router [13]. Also, the port only needs to be larger than 1024 (which will unlikely be in conflict with other applications).

5.2.2 Triangular spamming implications

Port blocking policy bypassing. Many ISPs nowadays are enforcing outbound SMTP traffic (port 25) blocking in an effort to prevent compromised hosts or bots inside their networks from sending spam targeting destinations outside their networks. In the following we denote from a perspective of a given network outgoing packets with destination port 25 as *OUT* traffic and incoming traffic with source port 25 as *IN* packets (which is usually the response packets sent from the mail server) for ease of exposition. The phrase "outbound SMTP traffic blocking" refers to an abstract policy that tries to prevent outbound spam by either blocking IN traffic or OUT traffic or both. The problem is that *only* blocking OUT traffic but not IN traffic (which is the second requirement of triangular spamming) by ISPs is insufficient to fully prevent their internal hosts from participating in spamming activities. Using triangular spamming, those IP address can still be "hijacked" to send spam. Note that for ISPs that do not try to prevent outbound spam, they will not block IN traffic either as it is necessary for outgoing SMTP connection to be established.

Higher spamming throughput compared to sending directly from botnets. Spammers can rent high bandwidth pipes to send spam with higher throughput due to the nature of triangular spamming - most of the traffic is uplink traffic directly flowing from the spammer to the mail server without going through bots (See Figure 5.1). Although the response packet from the mail server has to inevitably traverse bot hosts, it may not be the bottleneck as the spammer can paralellize the connections by leveraging many different bots they may already have access to today.

5.3 ISP Port Blocking Policy Inference and Policy Impact Analysis

How ISPs configure outbound SMTP traffic blocking determines whether triangular spamming can work or not. As we discussed, many ISPs are now enforcing the outbound SMTP traffic blocking policy, but it is unclear what the exact policy is. In this section, we present a systematic empirical analysis on the port blocking policy of various ISPs. More specifically, we intend to study 1) which ISPs currently enforce outbound SMTP traffic blocking, covering as many ISPs as possible, 2) under their current policies, how many are vulnerable to triangular spamming either as sending hosts or as relay hosts as described previously.

5.3.1 Port blocking model

We make several reasonable assumptions about the firewall blocking model in order to design tests to infer firewall policies. First, we assume that ISPs are not blocking port 25 traffic based on packet content (also known as Deep Packet Inspection or DPI) given DPI is more expensive and difficult to operate at line speed. Indeed, direct port 25 blocking is the most commonly enforced policy [27, 28]. We also assume that the blocking is directional and can be configured based on TCP/IP header, *e.g.,* source/destination IP address, source/destination port, protocol types (*e.g.,* TCP or UDP) and TCP flags (*e.g.,* SYN, ACK). This model is commonly employed in most modern firewalls ranging from heavy-weight devices (*e.g.,* Cisco PIX firewall [8]) to host firewall software on PCs (*e.g.,* iptables [23]). For example, a sample firewall rule that blocks outbound SMTP traffic would appear as:

SrcIP	DstIP	SrcPort	DstPort	Protocol	TCP-flags	Action
Any	Any	Any	25	TCP	ALL	Drop

There are two important observations to make here. First, suppose this rule is applied to outgoing traffic, *i.e.,* traffic from ISPs' internal hosts to external networks, it effectively blocks only unidirectional outgoing traffic, implying that packets from an external mail

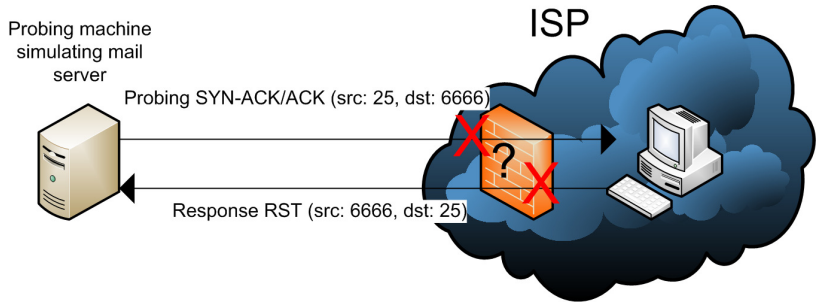


Figure 5.2: Possible outbound SMTP traffic blocking policy

server destined to internal hosts with source port 25 will not be blocked. This motivates our study on inferring current port blocking policies of different ISPs to discover if they are vulnerable to triangular spamming. This problem is illustrated in Figure 5.2 - the ISP can either block OUT traffic, IN traffic, or both. It is known that many ISPs only block OUT traffic due to the simplicity of such policies and the additional complexity of configuring incoming port 25 traffic filtering as mentioned in previous work [56]. For instance, depending on where the firewall is located, there has to be many exceptions in the firewall rules specifically (sometimes separately) for outgoing mail servers and incoming mail servers. As the recent NANOG survey [22] shows, some real-world ISP operators do consider that blocking OUT is simpler and has less impact on the traffic (there is less outgoing traffic than incoming traffic).

Second, we note that a stateful firewall that tracks individual TCP connection states could block IN packets associated with triangular spamming simply because they are "out-of-state". For example, SYN-ACK packets without any prior associated SYN packets will be dropped by such a stateful firewall. However, it is difficult for ISPs to adopt this due to two reasons: 1) it is expensive to keep track of the state associated with a large number of flows, and 2) some out-of-state traffic, *e.g.,* probing, can be legitimate. Note that host firewalls can easily sustain stateful checking, but if the host is already compromised, such firewalls are easily disabled or bypassed. On the other hand, network firewalls are unlikely modified by spammers. Given this key difference, we attempt to distinguish host-based

blocking from ISP-level port blocking as discussed later in §5.3.2.2.

5.3.2 ISPs that block OUT traffic

To study whether most ISPs block OUT traffic instead of IN traffic, we first find a set of candidate ISPs or IP ranges that block outbound SMTP traffic and then use the probing methodology discussed in §5.3.3.1 to distinguish OUT traffic blocking from IN traffic blocking.

5.3.2.1 Experiment design

There are several approaches to discover the outbound SMTP traffic blocking behavior.

Surveying ISPs would be the simplest approach. However, many ISPs treat such information as confidential and only reveal it to new or existing customers. Very few ISPs openly disclose such information (*e.g.,* Sonic.net [28]). ISPs' knowledge of their policies may lack sufficient detail and may also be inaccurate due to misconfigurations. Further, port blocking policy may change over time and may vary depending on location. For example, Comcast was known to enforce such policies [10]. However, our controlled experiment (test using comcast service at home) indicates that Comcast is not blocking outbound SMTP traffic at the moment we conduct the experiment.

The second approach is to obtain control at both end-points by installing a probing program on end-hosts inside various ISPs, which communicate with a server managed by us. For instance, the ICSI Netalyzr [14] requires users to download a Java applet and likewise the Spoofer project [56] requests users to explicitly download a program to run. However, such an approach is more challenging to accomplish wide-scale adoption.

The third approach is to probe with single end control only. We can mimic a mail server sending TCP packets with source port 25 to the other end (on some well-known ports) with SYN-ACK or ACK flags. Depending on the OS and host firewall settings, probed end-hosts may respond with a RST packet (we verified this behavior for Windows XP SP2 and Linux

Ubuntu 9.04). If all live end-hosts respond to the our source port 80 probing but never to our source port 25 probing, it is highly likely that this ISP is blocking outbound SMTP traffic instead of individual hosts doing so. This approach has the benefit of being easily carried out so that we can probe any host or network of interest. However, the choice of which IP address ranges to start with limit its use.

Considering tradeoffs of these various approaches, we adopt a hybrid one combining the second and third approach. In order to obtain control from ISP side, we chose to develop a simple, invisible Flash [1] program that can be easily embedded in Web sites and transparently executed at the client side. Figure 5.3 shows the HTML code to be inserted into Web pages (Note that IP address of the server is used to avoid an additional lookup overhead). We inserted it at various university department and educational Web sites in the U.S. and China to obtain a variety of client IP addresses.

Note that simpler HTML code like can achieve the same goal. However, direct port 25 access in HTML is blocked by browsers like Firefox due to security reasons, *i.e.,* One can craft forged HTML Form Post formatted to send out spam emails. However, Flash is in a completely different domain from the browser and is allowed to initiate outgoing port 25 connection by default. If our Flash program indeed fails to establish the connection, then it is most likely blocked by firewalls at the host or at the network. To distinguish between these two, more data points from that network are needed.

The choice of Flash is supported by the observation that 99% of modern browsers deployed [43] have the Flash plugin installed. Thus almost every client can execute the program which simply tries to initiate an outgoing port 25 connection and terminates immediately upon success. Logging by our server will record this along with the initial download of the Flash script via HTTP. This allows us to distinguish IP addresses that succeeded in the test from those that failed to connect to the port 25 on our server.

```
<object width='0' height='0'>
<param name='movie' value='http://OURSERVER-IP/flash.swf'>
<embed src=``http://OURSERVER-IP/flash.swf' width='0' height='0'>
</embed>
</object>
```

Figure 5.3: HTML code snippet

5.3.2.2 Probing results

As shown in Table 5.1, based on our two months of data collected, we gathered about 21,131 unique IP addresses (excluding 2,749 local IP addresses) in our Web log spanning across 7,016 BGP prefixes. Based on a simple DNS name heuristic, we classify the prefixes into educational institutions and ISPs since our clients are mostly students who likely access through home or school networks. 341 of them are educational institutions, 2987 are ISPs and 3691 are unknown (We randomly probed IP addresses within the prefix with some threshold, if none of them has a DNS name, then it is classified as unknown). Although 3,563 (51%) prefixes contain at least one IP address blocked for outbound SMTP traffic, only 2,600 prefixes (37%) have all IP addresses blocked. Interestingly, there are 622 IP addresses that connected to port 25 without connecting to our web server. We suspect that these are spammers probing for open relays.

For many prefixes, we only have limited samples (IP addresses) and the blocking behavior may not represent the prefix-level policy, *i.e.,* it is possible the host firewall blocks the outbound port 25 traffic which is not easily determined by the Flash script. As a result, we conduct further probing to verify that the ISPs are indeed blocking outbound SMTP traffic in those IP ranges. Extensive probing (piggybacked in our IN/OUT blocking described in Section 5.3.3) for every IP address in the prefix range is carried out to avoid incorrect conclusions caused by outliers, *i.e.,* host firewall rather than ISP firewall blocking traffic. Although we could also develop some randomized probing algorithm, the problem is that we do not know when is sufficient to stop and even if we stop at some threshold number of responses, we may still miss the rest IP addresses with different behaviors.

The results show that about 688 prefixes have at least some /24 sub-ranges blocking

outbound SMTP traffic, assuming the policy is configured at most at the granularity of /24, matching the finest routing granularity on the Internet. Out of these 688 prefixes, 25 are educational institutions, 483 are ISPs and the remaining 180 are unknown.

To illustrate the diversity of our dataset, we look up the country for IP addresses from IP whois database [45]. They span across 127 different countries. Due to a lack of space, only countries with more than 100 IP addresses are shown in Table 5.2. As expected we observe that most IPs are from the U.S. and China matching the locations of the hosting Web sites. At the prefix level, we analyze the percentage of blocking prefixes as verified using probing and show the diverse policy across countries. Since our instrumented Web sites are most likely visited by universities and home users, we expect that many prefixes should perform outbound SMTP traffic filtering at least at some sub-ranges. The results show that the top two countries for enforcing such port blocking policy are Turkey and Canada. Compared to the top two countries, the U.S. has a lower filtering enforcement rate. But it is still better compared to the remaining countries. China and Korea have the worst blocking percentages, implying that ISPs in those countries visible in our data do not pay much attention to spam prevention through network-based filtering. This is consistent with previous findings that China and Korea are two big sources of spam emails.

Table 5.1: Summary of IPs gathered from the Web flash experiment

	# of IP addresses	# of prefixes
# of IP in web log (Baseline):	21131	7016
# of IP in port 25 log:	13576	4280
# of IP in web log but not port 25 log:	7555	3563
# of IP in port 25 log but no web log:	622	397

5.3.3 ISPs blocks OUT but not IN traffic

Based on the previous results, we get an idea about how prevalent the outbound SMTP traffic blocking policy is on today's Internet. We delve deeper in the results to infer whether ISPs that block OUT traffic neglect to block IN traffic, where the latter is a necessary

Table 5.2: Distribution of IPs and prefixes that are blocked based on country

Country	# of IPs	# of prefixes	# of blocking prefixes	% of blocking prefixes
China	6259	1006	4	0.3%
Korea	341	145	2	1.3%
India	1274	547	9	1.6%
Iran	270	89	3	3%
UK	198	120	8	6%
Germany	118	81	6	7%
Australia	638	162	13	8%
US	10499	2714	252	9.3%
Canada	274	151	53	35%
Turkey	150	87	36	41%

requirement for serving as a relay in triangular spamming.

5.3.3.1 Probing design

As shown in Figure 5.2 and discussed previously, it is easy to infer that the ISP is preventing outbound SMTP traffic, but non-trivial to discern at which direction blocking takes place. To recap, we can first send a TCP SYN-ACK probe packet to some hosts in the IP range of interest with source port 80 and destination port 80 (or any other well-known ports). Depending on the OS and whether the port is open, the host may respond with a TCP reset (RST) packet. If we receive the corresponding RST packet, this shows that hosts will respond to probes to unused ports. We then immediately send another TCP probe packet but source port set to 25. If we do not observe any response this time, assuming it is not the host firewall that blocks the traffic, it would be the ISP that blocks either IN traffic (which is our probe traffic) or OUT traffic (which is the RST response sent from the probed host). Note that it is also possible that the ISP spoofed the RST packets uniformly as their policy and in this case, we will conservatively think that the ISP is not blocking port 25 while in reality spoofing RST can be a form of blocking. As a result, we may underestimate the port blocking prefixes. However, since the latest large-scale study [113] did not report the exact same RST behavior (they discover the most popular RST injection

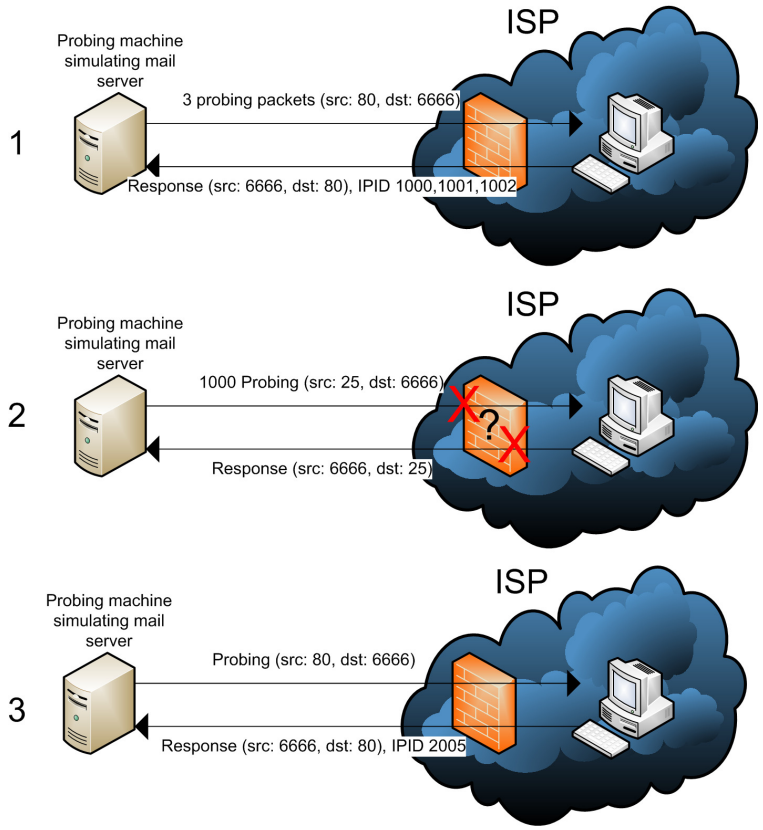


Figure 5.4: Outbound SMTP traffic blocking policy inference

is after SYN and SYN-ACK packet and in the same direction), we believe such behavior (RST after SYN-ACK in the opposite direction) is rare if deployed at all.

Making use of the properties of IPID values (ID field in IP header) generated by the end-host as many previous studies have done, we devise a simple approach to distinguish the IN or OUT traffic blocking. Figure 5.4 shows this probing methodology.

At step 1, suppose we already know that the ISP is blocking outbound SMTP traffic but have no idea whether it is IN or OUT blocking. We send several probing packet (*e.g.,* 5 packets) with source port 80 to some well-known ports. If we receive responses, we record the IPID values of the responses and detect the presence of a monotonically increasing pattern using a simple algorithm similar to nmap [40]. Let X be the last IPID value received. At step 2, we send a burst of packets (*e.g.,* 1000) with source port 25 to the same destination port and expect no response for these packets. At step 3, we send more probe packets again with source port 80 and examine the resulting IPID values in the response packets. If

these values are roughly starting from $X + 1000 + E$ where E is the expected IPID value increase due to other packets sent by the host between steps 2 and 3, then we can infer that the ISP performs OUT traffic blocking instead of IN traffic blocking. This is based on the conjecture that the increase in IPID values indicates that the host did receive our probe packets and responded to them, resulting in increase in IPID values. We did not receive the response packets due to ISP's firewall blocking such OUT packets. On the other hand, if the IPID value has not increased by what is expected, we conclude that the ISP imposes IN traffic block and possibly also combined with OUT traffic filtering. Note that such a conclusion is unlikely to be incorrect due to the previously verified monotonically increasing IPID values.

Note that here we assume that the host probed has system-wide monotonically increasing IPID values which may not hold. For example, the IPID values can be random or always set to 0 for response packets that do not belong to the same TCP flow in recent Linux kernels. However, for Windows XP SP2 and SP3 that we tested (arguably still the most prevalent OS at the time we conduct probing), they all have such system-wide monotonically increasing IPID behavior. In fact, Windows 7 also has such property. Our probing results discussed next also verify this behavior for a large fraction of probed IP addresses. Hosts that do not have this property are not probed further. As long as we have sufficient number of samples from a prefix we can still infer the ISP-level policy.

Our probing test technique is summarized in Algorithm 1.

5.3.3.2 Results

We take candidate prefixes generated from our Web Flash experiment for the probing test algorithm to infer the ISP's policy. Some prefixes can be very large (*e.g.,*/11 or /12), requiring significant time to probe every single IP inside them. Instead, we probed only a subset of IPs in such prefixes due to time constraints. To prevent triggering any firewall alarms, for each prefix, we conservatively spawn only a single-threaded process to perform

Algorithm 1 IN or OUT traffic blocking probing test algorithm

```
Input: Prefix p that has blocking behavior,
repeat {For each IP ip from the prefix p where except ip ended with last octet 1 or 254 or 255}
   response1 = Probe(ip, 80, 80);
   response2 = Probe(ip, 25, 80);
   if(response2 == succ) notBlocking;
   else if(response1 == fail) unknown;
   else if(response1 == succ) {
      blocking;
      IPIDs = probeIPIDs(ip, 80, 80);
      if(increasing(IPIDs) == false) {
         IPIDNotIncreasing;
         continue;
      }
      burstProbe(ip, 25, 80);
      IPID = probeIPIDs(ip, 25, 80);
      if(IPID ≈ IPIDs[last] + E + 300)
      {  OUT-traffic-blocking;  }
      else IN-traffic-blocking;
   }

until [All ip in prefix p has been probed]
```

probing. As a result, on average, it takes 2 - 4 days to probe an /16 prefix. But since we are parallelizing the probing for different prefixes, we can still gather results in reasonable amount of time. We were able to probe most IPs for smaller prefixes, *e.g.,* prefixes smaller than /15. For larger prefixes, we covered about 25% (for some /11 prefixes) to 80% of their IPs.

Table 5.3 is an example of our probing result for prefix 24.247.80.0/20 which belongs to the Charter ISP [7]. We sub-divide the prefix into /24 prefixes based on the assumption that finest policy granularity is at the /24 level. Each row represents the result for a particular /24. Each column shows the number of IP addresses within the /24 for a particular category. We can see that clearly most /24s are entirely OUT-traffic-blocking with only few exceptions. For instance, the fifth row has 7 IP addresses detected as OUT-traffic-blocking but only 1 IP was found to be not blocking outbound SMTP traffic, generating a potentially inconsistent configuration policy within the /24. However, we do know that it

is common that ISPs allow customers to unblock port 25 for "power users" [34]. In this case, we believe that the few unblocked port 25 IPs are such exception cases. An anomaly is shown in the ninth row indicating that 24.247.88.0/24 has no IPs blocked for outgoing port 25. In fact, all 19 IPs are not blocked for either IN or OUT. Upon further investigation, we found that this /24 are static IP addresses (with DNS name such as 24-247-88-64.static.bycy.mi.charter.com) as opposed to dynamic IP addresses (with DNS names such as 24-247-80-0.dhcp.bycy.mi.charter.com). Usually static IPs are business-level customers who pay more for access to open ports [44]. Further, as expected we found that none of the IPs are found to be IN-traffic-blocking.

Interestingly, we only find 22 prefixes out of 688 blocking prefixes (3%) appear to deploy IN-traffic-blocking policy. We identify prefixes as IN-traffic-blocking if at least some /24 IP range have the number of IN blocking IPs is twice (or more) the number of both non-blocking ones and OUT blocking ones. In fact, when the number of IN blocking IPs is twice or more the number of non-blocking ones, it is almost always true that the OUT blocking IPs will be 0. However, it is still possible that some IPs are not blocked due to reasons such as customer's requests of sending outbound SMTP traffic. A quick analysis reveals that these IN blocking prefixes belong to US, Japan and European countries like German, Sweden, UK, Belgium and Italy, mostly concentrate in European countries. There is only one educational instititution - umass.edu. Some ISPs in the US also block IN traffic such as verizon.com. However, there are only 2 out of the 131 verizon prefixes that we probed perform IN blocking.

Note that some ISPs block outbound SMTP traffic (could be IN or OUT blocking) on demand based on the SMTP traffic volume to prevent abuse. This can also reflect in our probing results that sometimes /24 prefixes have more non-blocked IPs than blocked ones. Typically, this is because the majority of the hosts are not sending spam, so outbound SMTP traffic from them is not blocked. Blocked hosts are likely ones detected to be sending spam. In this case, triangular spamming stealthily eliminates outgoing SMTP traffic from

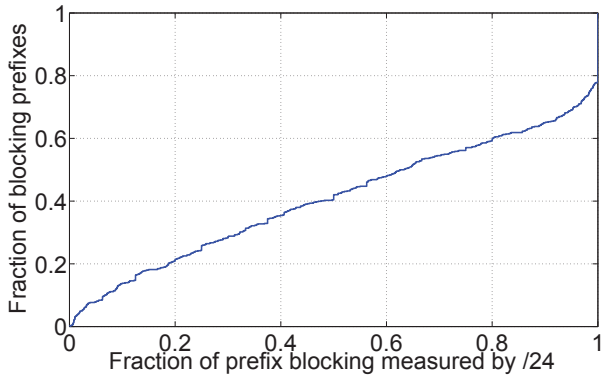


Figure 5.5: Distribution of blocking /24 subnet pctg (OUT SMTP traffic)

compromised bots, increasing the difficulties of detecting outgoing spam for ISPs.

Next, we analyze the results for OUT blocking prefixes. Besides the 22 IN blocking prefixes, the remaining 666 blocking prefixes are OUT blocking. Similarly, we consider a /24 as OUT blocking if the number of blocking IPs within the prefix is at least twice the number of non-blocking ones. Figure 5.5 plots the distribution of the percentage of blocked /24s within a prefix for all prefixes with at least some blocked /24s. We can observe that 20% of the prefixes are blocking all of their /24 sub-ranges and about 40% of the prefixes have about 50% /24 sub-ranges blocking outbound SMTP traffic. This shows the surprisingly non-uniform policy configuration at the ISP level and demonstrates room for improvement, as discussed later in §5.5. From now on, we will assume it is OUT blocking instead of IN blocking whenever we refer to blocking if not specifically mentioned.

Table 5.3: An example of IN/OUT blocking probing results for 24.247.80.0/20

IP prefix	# of OUT blocking	# of non-blocking	# of unknown (*e.g.,* host down)	# of IN blocking
24.247.81.0/24	14	0	238	0
24.247.82.0/24	13	0	239	0
24.247.83.0/24	7	0	235	0
24.247.84.0/24	7	1	244	0
24.247.85.0/24	6	0	246	0
24.247.86.0/24	10	0	242	0
24.247.87.0/24	9	1	242	0
24.247.88.0/24	0	19	233	0
...	...	...	...	...

We analyze the inconsistent policy configuration behavior for prefixes with nonuniform

Table 5.4: IN/OUT traffic blocking probing results of 171.64.0.0/14 (stanford.edu)

IP prefix	# of OUT blocking	# of non-blocking	# of unknown (*e.g.,* host down)	# of IN blocking
...	...	...	...	...
171.66.120.0/24	0	232	20	0
171.66.121.0/24	0	199	53	0
171.66.122.0/24	0	228	24	0
...	...	...	...	...
171.66.128.0/24	4	0	248	0
171.66.129.0/24	0	0	252	0
171.66.130.0/24	5	0	247	0

outbound SMTP traffic filtering setting for its subnets. We found that the following three types are most common ones:

1. Dynamic IP vs. static ranges (or unblocked dynamic ranges), examples shown in Table 5.3.

2. Sub-ranges delegated for other purposes. An example is shown in Table 5.4. Within the Stanford university's /14 prefix, some prefixes ranging from 171.66.120.0/24 to 171.66.127.0/24 have been assigned for other purposes. Many of corresponding reverse DNS names of IPs for such prefixes are changed to names like 'lcgsl.highwire.org' instead of '*.stanford.edu'. These IP blocks appear to be used for the Stanford University Press, which likely requires more admissive SMTP traffic policies.

3. Legitimate mail servers reside in the prefix, sometimes even co-located with a blocking /24 prefix. For example, we found that in a university prefix 128.118.1.0/24 contains several machines allowing outbound SMTP traffic, which are legitimate mail servers according to the MX records. Other than these machines, however, 28 other hosts are blocked for OUT traffic (The rest didn't respond), indicating that the policy made exceptions for these mail server machines.

5.3.3.3 Correlation with Spamhaus PBL

Spamhaus PBL [102] is a popular blacklist widely used for identifying IP or IP ranges that should not deliver unauthenticated SMTP emails. The list includes both dynamic and

static IP ranges and is gathered either from ISPs (ISP operators may volunteer to contribute to the list) or through other analysis. We are curious to know if the blocking prefixes we identified are already on PBL. If so, even if triangular spamming were used, they will still eventually be blocked at the mail servers (since the spoofed IP addresses fall into PBL). Surprisingly, out of 666 prefixes, there are only 296 (44%) of them are listed on PBL. It is possible that ISPs may think that since they already block the user IP ranges for OUT port 25, there is no need to report these IP ranges to PBL. However, with triangular spam, it is not the case. It is still useful to report these IP ranges to PBL which can be considered as another layer of defense.

To understand what kind of prefixes is missed by PBL, we looked at three prefix types as described previously. Out of 23 blocking educational institutions, PBL only listed 1 of them. We imagine the reason is that many universities have departmental mail servers which are difficult to be captured in a large prefix. The only one university prefix that gets on PBL is 130.18.78.0/23 where its DNS names are of the form ts3.dialup.msstate.edu. For 466 blocking ISPs, only 194 of them are listed. For the rest 176 unknowns, 101 are listed. This shows that prefixes without DNS name have the highest listed ratio, indicating that they are more likely to be bad prefixes. For ISP prefixes, we found that PBL is still largely incomplete.

We summarize the findings from our extensive probing based analysis. We found that most ISPs today are not careful in blocking incoming SMTP traffic (with source port 25), despite some effort in reducing spam originated from their networks by blocking outgoing traffic destined to port 25. This opens many prefixes as relay nodes in triangular spamming scheme, resulting in these nodes participating in spamming in a quite stealthy way. Our designed probing methodology enables marking of specific prefixes vulnerable to triangular spamming for subsequent detection purposes.

5.4 Experience and Analysis on Triangular Spamming

The previous section shows that today's network policies allow the possibility of carrying out triangular spamming. To better understand its operational model, as the next step, we build an actual triangular spamming infrastructure deploying at the Planetlab environment to explore various tradeoffs. In particular, we focus on the following questions:

1. Does triangular spamming require significant engineering effort and how easily it can be deployed (maliciously installed) on the relay bot and/or original sender?

2. Does triangular spamming really work in the real world (via Planetlab deployment)?

3. How much bandwidth utilization or throughput benefit can there be by using triangular spamming?

4. What property of the system can be leveraged for detection?

5.4.1 Implementation

Figure 5.1 shows that triangular spamming requires two separate components: one on the original sender and one on the relay bot. We build both components under Linux. Linux is chosen due to ease of development and deployment on Planetlab testbed. The development effort involves about 1700 lines of C code for the original sender and about 700 lines of C code for the relay bot. For the component on the original sender, it can either be deployed on a spammer-owned machine or some bot with good network connectivity. One can imagine that the number of such machines is likely smaller compared to common bots. For example, based on the Torpig study [103], about 22% of the infected hosts are in corporate networks that tend to have better bandwidth support than dial-up and cable networks.

Next we discuss in detail the implementation and design choice for each component.

5.4.1.1 Component on the original sender

To support IP spoofing, one can either modify the mail sending program directly or implement in a transparent fashion independent of the mail software. The latter is the preferred approach adopted by us because one can write mail sending program independently of triangular spamming infrastructure. Our design of the component is shown in Figure 5.6. We intercept outgoing packets destined to port 25 and dynamically rewrite the source IP address to the relay bot's IP. We also modify the source port due to specific constraints of the Planetlab, as it only allows one to intercept incoming packets destined to certain reserved port ranges.

Note that the relayed packet has its source IP address set to the relay bot's IP (instead of the mail server's IP) because ISPs of relay hosts may prohibit IP spoofing. In fact, Planetlab does not allow IP spoofing [26]. As a result, we have to rewrite the source IP address to the mail server IP address upon receiving the packet at the original sender. We know which server IP address to rewrite to because we keep track of the mapping between spoofed source IP addresses and destination mail server IP addresses. Similarly, we rewrite the destination port to the actual port used by the original sender.

Since TCP is used, the sender and the receiver will take care of retransmission for lost packets. The relay bot is therefore stateless.

We use iptables' built-in support to intercept and deliver packets to the user-level program that can modify the packets and re-inject them. Since it involves additional kernel-to-user and user-to-kernel transition for every packet, it is not as efficient compared to a kernel module based approach. We chose user space implementation for simplicity, and the associated overhead is not an inherent limitation.

5.4.1.2 Component on the relay bot

It is rather simple to implement the component on relay bot given its functionality is to simply relay packets to the original sender by rewriting the destination IP (the source IP

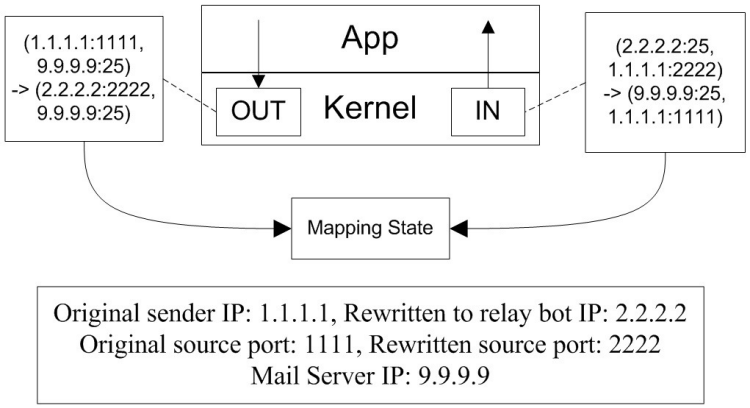


Figure 5.6: Component on the original sender

is also rewritten to avoid unnecessary IP spoofing). However, it is important to understand under which condition the bot relays packets. For instance, it is obviously unnecessary to relay non-SMTP packets by only relaying packets with source port 25. As long as the original packet is not dropped and the user's SMTP traffic is not impacted (assuming the user also uses port 25), it is safe to relay them. However, sometimes, additional care has to be taken. Consider the original packet is not dropped and successfully received by the kernel, the kernel will think of the packet as out-of-state. Depending on the operating system (OS), it may generate a TCP reset packet upon receiving such packets. Reset packets are undesirable because they can terminate the TCP connection at the mail server side even though the original sender does not intend to.

Fortunately, many OSes do not generate such packets. For instance, we have tested and verified that Windows XP SP2 and Windows XP SP3 (arguably still the most prevalent OS in use as of writing) by default do not generate such reset packets for closed ports. We suspect that the "silent drop" behavior is the correct behavior from security and privacy point of view. It is also the case for newer Linux kernels. However, we do discover that Windows Vista SP1 and Windows 7 will generate the reset packets.

In any case, even if the OS generates reset packets, our program can still try to drop such packets or drop the original incoming packet. However, doing so may interfere with the user's SMTP traffic.

5.4.2 Real-world deployment on Planetlab

We have successfully deployed triangular spamming on Planetlab and have experiments running correctly. Using tcpdump trace, we are able to verify that the server believes it is talking to the relay bot (setup on Planetlab node) instead of the original sender.

5.4.2.1 Limitation of Planetlab environment

Unfortunately Planetlab nodes generate reset packet instead of "silently dropping" out-of-state TCP packets likely due to their customized kernel. As a result, we have to modify the server to drop such reset packets to prevent the mail server side connection from being reset. However, such limitations of the Planetlab environment are unlikely present on compromised machines because the attacker would have full control over the machines and can easily alter their behavior.

5.4.2.2 IP spoofing blocking study using Planetlab

Now that we have the basic infrastructure running correctly, we are more interested in knowing whether triangular spamming works in reality, meaning that if we spoof IP addresses of Planetlab nodes across the world, will our spoofed traffic be detected or dropped at certain ISP along the path via uRPF [41]? More specifically, although our university does not block spoofed traffic, intermediate ISPs may still block our traffic. Similarly, one may argue that spammers can always find an ISP that does not filter spoofed traffic, but this is only true for their home ISP, may not necessarily hold for intermediate ISPs. We empirically study from our vantage point to check whether our spoofed traffic will get dropped.

The Spoofer project [56] shows that 80% of filters are implemented a single IP hop from sources and 95% of the blocked packets are observably filtered within the source's Autonomous System. Their result is promising for triangular spamming, *i.e.,* as long as spammers can locate an ISP allowing IP spoofing (which is quite likely given that 33%

of the tested IPs from the Spoofer study can spoof arbitrary IP addresses), it is unlikely that spoofed traffic is blocked in the middle of the path. This is because in practice implementation of such IP-spoof prevention techniques are often limited by multi-homing, route asymmetry and other factors.

In our experiment, we attempt to spoof all available Planetlab nodes' IP addresses and use them to connect to multiple mail servers across various countries. Note that although the Planetlab node will generate a reset packet immediately after it receives the SYN-ACK packet from the mail server, it still relays the SYN-ACK packet back to the original sender so that we know the IP spoofing succeeded. We record each <spoofed IP, destination IP, isSucc> pair in our trace and the results are shown in Table 5.5.

Table 5.5: IP spoofing results - spoofing Planetlab node IPs

Destination	Location	Spoofing-succ count	Failed Count	Failed IP & Location
Local Mail server	US	313	0	N/A
Hotmail	US	313	0	N/A
Yahoo	US	312	1	116.89.165.133 (Korea)
Gmail	US	312	1	116.89.165.133 (Korea)
Yahoo.com.cn	China	6	307	All except some servers in the US
University server	France	313	0	N/A
University server	India	312	1	116.89.165.133 (Korea)
University server	Japan	312	1	116.89.165.133 (Korea)
University server	Brazil	313	0	N/A
University server	Korea	313	0	N/A

The result is mostly consistent with that of Spoofer [56]. Once the homing ISP allows IP spoofing, it is likely to allow arbitrary routable IP address spoofing. For example, we can successfully spoof arbitrary Planetlab node IP addresses (except a node in Korea) to most popular U.S. mail servers except Yahoo. The result is shown in Table 5.5. We have double verified by repeating the experiments several times for the failed IP addresses. Interestingly, not only India and Japan filter the spoofed Korea IP, Yahoo and Gmail in the US also filtered them. This is contradicting with Spoofer's results about arbitrary IP spoofing assertion that may be caused by insufficient diversity of spoofed nodes, although the Korea node seems to be an only exception here. For Yahoo mail server in China, the IP spoofing almost always failed. We verified that the result is consistent for other servers in China too, indicating

that there is some specific filter along the path to China. However, despite these failed IP spoofing cases, the results are still very promising for triangular spamming as the filtering is sporadic and virtually non-existant for US destinations.

5.4.3 Bandwidth utilization analysis

5.4.3.1 Bandwidth utilization shifting

As we can see in Figure 5.1, the bandwidth utilization behavior completely changes from the perspective of relay bots. Without triangular spamming, a bot directly initiates a SMTP connection to the mail server and sends spam messages using its uplink bandwidth. With triangular spamming, all the uplink traffic is shifted to the original sender and such traffic is invisible to the relay bot's network, effectively lowering the bandwidth utilization of bots.

Further, since relay bots are most likely DSL or cable modem hosts, their downlink bandwidth is usually much higher than uplink bandwidth. For instance, certain ADSL connection has a downlink bandwidth of 8 Mbit/s and a uplink bandwidth of 1.0 Mbit/s. Although the relay bot still needs to forward packets to the original sender consuming its uplink bandwidth, the forwarded packets are SMTP response packets and the SMTP-response traffic is much smaller than SMTP-send traffic. Analyzing randomly sampled spam from our local mail server log, we estimate that the size of SMTP-send traffic is about 5 to 10 times of the SMTP-response traffic, depending on what the spam message contains. As a result, we can conclude that triangular spamming allows spammers to associate many more concurrent connections with a relay bot without triggering bandwidth-usage-based anomaly detectors. We will show that in the next section, we can use selective forwarding to even reduce the bandwidth usage on relay bot even further.

For relay bots whose IPs are blocked for outgoing SMTP traffic, there is a clear advantage of using triangular spamming. It is primarily because IP addresses are scarce resources and blacklists nowadays can identify malicious IPs rather quickly so that the IP addresses

may be rendered unusable. As we have shown in previous results, the IP ranges that block outgoing SMTP traffic are not necessarily listed on blacklists. This gives spammers strong incentives to use such IP addresses given they could still successfully deliver spam.

5.4.3.2 Spamming strategy and techniques

In this section, we show that triangular spamming offers an opportunity to improve spamming throughput (i.e. the number of emails sent per second). Concisder the following two spamming strategies:

Strategy 1: All bots directly send spam at their full speed.

Strategy 2: Triangular spamming is used where only high bandwidth bots send spam.

Strategy 1 is the baseline for comparison. This strategy provides good overall throughput since it utilizes the distributed resources of the botnet. However, it has two noticeable disadvantages: first, it will expose the high bandwidth bot IPs; second, even the low bandwidth bots risk of being detected at its hosting ISP if they are sending spam at full speed. On the other hand, bandwidth-usage-based detector may not be able to catch high-bandwidth bot since it is spoofing different IP addresses. Moreover, spammers may also rent their own high-bandwidth machines in spammer-friendly ISPs. For strategy 2, the high bandwidth bot can hide its own IP address while sending at full speed. For low bandwidth bot, given their bandwidth limitation, we think it might be a good idea to conserve their spamming activity. Instead, they should be mostly focusing on relaying server responses back to the sender.

Now, we envision two spamming techniques under Strategy 2 that can help improve throughput for triangular spamming:

Technique 1. Selectively relaying packets at the relay bot - reducing unnecessary network bandwidth usage.

Given that the common case is that senders can successfully deliver emails. It is not really necessary for the sender to receive the response from the mail server. We have

verified using our triangular spamming prototype that the relay bot needs to relay only the TCP SYN-ACK packet to the high bandwidth bot for spamming. This technique can significantly reduce both the uplink bandwidth usage of relay bot and the total bandwidth usage of high bandwidth bot. Depending on the email size, the SMTP-response (incoming traffic) at the high bandwidth bot is around 1/5 - 1/6 of the total traffic when the email body size is around 1700 bytes (this is relatively large spam email size likely in HTML format). It is possible that some messages are larger such as image spam and some spam message is smaller - many spam messages only contain a few words then a link to a scam website or a message contact. For cases where spam messages are smaller, it is a clear benefit in bandwidth usage reduction.

Note that above is somewhat an ideal case, there are some minor issues at the TCP layer that need to be fixed. First, from the mail server's perspective, it may not receive any TCP ACK messages for its response packets since the high-bandwidth bot never gets them in the first place. But in reality, the mail server's initial congestion window is large enough to hold all the outgoing packets without receiving any ACK (although it may cause the mail server to unnecessarily retransmit the response packets). One possibility is to let the relay bot to ACK mail server's response packets directly without burdening the high-bandwidth bot. Similarly, at the sender side, although the initial congestion window at the high-bandwidth bot is also typically large enough to hold all outgoing packets without getting any ACK. It would again cause unnecessary retransmissions that waste bandwidth resources. In order to mitigate this issue, we have two options: 1) let the relay bot relay the ACK packets from the mail server to the high-bandwidth bot or 2) spoof the ACK packets locally at the high-bandwidth bot. The second option has the potential problem of not able to detect packet loss (since we always spoof the ACK without knowing whether it is received by the mail server), although this may rarely happen. The first option will use some bandwidth to relay the ACK packets but the size of ACK packets should be relatively small and it is simpler. We have successfully implemented the first option and verified that the emails can

be successfully received by mail servers.

Technique 2. Aggressive pipelining.

The SMTP protocol is interactive and I/O bound as each SMTP session typically involves many round trips limiting the aggregated throughput. Thus, SMTP has incorporated the pipeline support as introduced in RFC2920 [33] in 2000 to pipeline the commands to reduce the overall session time. In the extreme case, one may send all commands in a single batch to the server. However, as specified in RFC, the EHLO, DATA, VRFY, EXPN, TURN, QUIT, and NOOP commands can only appear as the last command in a group since their success or failure produces a change of state that the SMTP client must accommodate. In order to test the pipelining support in today's mail server, we pick a set of popular mail servers (both open source and commercial) including: sendmail, Java Apache Mail Enterprise Server (JAMES), Gmail, Hotmail, Yahoo mail, and AOL mail. Interestingly, only two mail servers, Gmail and AOL mail, strictly enforce the RFC. All other mail servers allow full pipelining (sending all commands in a single batch). For Gmail and AOL mail, we have to wait after the server processes each 'critical' command such as EHLO before we can issue the next set of commands. Normally we know that the server has finished processing a command by observing its response. However, if RTT is large, spammers will have to wait for very long before they can move on to the next set of commands. But based on our experiments on a variety of mail servers that we tested, the next set of commands will be accepted as long as the server has finished processing the previous 'critical' command. This means that it is possible to aggressively pipeline the commands such that the next set of commands arrive just after the server finishes processing the previous 'critical' command. Typically, the processing time of the 'critical' command should be smaller than the wide area RTT which can be hundreds of milliseconds.

Algorithm 2 and 3 have the pseudo-code that illustrates different pipelining approaches. In Algorithm 3, when $t1 = t2 = 0$, it becomes full pipelining.

Here, since the EHLO command is relatively simple to process, the processing time

Algorithm 2 Normal pipelining

```
send("EHLO [hostname]");
recv_and_process(response);
send("MAIL FROM: <sender@aaa.com>\r\nRCPT TO:
<receiver@bbb.com>\r\nDATA\r\n");
recv_and_process(response);
send("[actual data]\r\nQUIT\r\n");
```

Algorithm 3 Aggressive pipelining

```
send("EHLO [hostname]");
sleep(t1);
send("MAIL FROM: <sender@aaa.com>\r\nRCPT TO:
<receiver@bbb.com>\r\nDATA\r\n");
sleep(t2);
send("[actual data]\r\nQUIT\r\n");
```

is usually very small. However, for the next set of commands (MAIL FROM to DATA), there are three commands combined together, which may take the mail server longer to process. By carefully choosing delay $t1$ and $t2$ in Algorithm 3, one can potentially increase the throughput for every single connection and possibly the overall spamming throughput.

Next, we try to quantitatively study the impact of delay $t1$ and $t2$ on the throughput. We conduct the throughput experiment on Emulab where 25 pc3000 machines with 1Gbps network interface are used. Each machine has a single 3GHz core. We pick one machine as the sender and the rest of the machines as potential receiving mail servers running the open source mail server JAMES. We spawn a large number of threads for concurrent connections for each mail server. Each thread continuously sends emails with a new TCP connection. As shown in Figure 5.7, the throughput increases as the number of mail servers increases, indicating that the initial bottleneck is at the mail server side. In the experiment, we choose the delay $t1$ and $t2$ to be 0ms, 50ms and 100ms respectively and draw the throughput curve accordingly. Figure 5.7 shows that it is difficult to gain higher throughput when the the corresponding RTT is relatively large. Without aggressive pipelining, to achieve high throughput, spammers may need to pick a large number of concurrent mail servers (which could be possible). With aggressive pipelining, one may be able to achieve significant

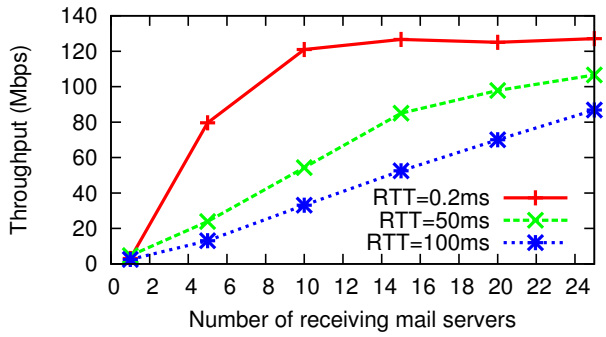


Figure 5.7: Impact of RTT on the spamming throughput

throughput improvement with the same number of mail servers by reducing the RTT. For the case of 100ms and 50ms RTT, the throughput improvement is about 1.5X - 2X with the same number of mail servers.

In reality, on the high-bandwidth bot, two of the following can happen: 1. Network bandwidth is the limiting factor (network bandwidth can be fully utilized). 2. CPU is the limiting factor (too many concurrent connections may cause context switches to occur too frequently such that the bandwidth may not be fully utilized). The throughput saturates or grows slowly as the number of concurrent connections increases.

Technique 1 applies to case 1 given that it can reduce unnecessary messages being received at the high-bandwidth bot. At the high-bandwidth bot, it is likely that the uplink and the downlink are shared (Ethernet rather than ADSL). If the network bandwidth is the bottleneck, using this technique can free up additional bandwidth to deliver spam messages.

Technique 2 applies to case 2 as shortening each individual RTT can help improve the overall throughput. Intuitively, t1 and t2 have to be at least greater than the server processing time for the corresponding commands. To get an idea of this value, we empirically vary t1 and t2 and target at one Gmail server which doesn't allow full pipelining. We perform the measurement on both peak hours (noon) and off-peak hours (mid-night). We found that during peak hours, t1 = 400ms and t2 = 800ms are often large enough to ensure successful email delivery. For off-peak, t1 = 20ms and t2 = 40ms are large enough. The difficult question is what delay value for t1 and t2 to pick in practice. Without triangular spamming,

since each bot can only send one or two spam messages (to avoid being blacklisted), there is no easy way for them to reuse the learned processing time. One possibility is to let bots coordinate the learned processing time. But this can be inefficient. Another possibility, offered by triangular spamming, is to use the measured processing time from one or more previous connections. The reason that it can work under triangular spamming setting is that it is easy to share the measured processing time information across multiple connections (all with different spoofed IPs) given all connections happen on the same physical machine (the high-bandwidth bot). More specifically, when triangular spamming starts, we open multiple connections for each target server. There are some bots that relay packets back earlier than others. We can use the RTT value observed from quick bots as an approximation for the processing time. One potential problem to consider is that we should avoid making too many concurrent connections to the same server because it will likely overload the server and inflate the processing time. So it is a good idea to spread the connections across multiple mail servers. A simple way to do so is to spread the connections across multiple IP addresses/machines exposed by a single mail provider, or sometimes even a single IP address may also correspond to multiple servers internally.

To study the feasibility of the second technique, we again use the Planetlab to measure how diverse RTT values can be, *i.e.,* quick bots vs. slow bots, in a globally distributed environment simulating a botnet. We use a machine in a university to act as the original sender, as university networks are typically well-provisioned. The idea is that if there are indeed many slow bots, we can use the second technique to reduce RTT and increase throughput.

Figures 5.8 through 5.11 show the RTT distribution for different target mail servers. We can see that for Hotmail and Gmail servers, the RTT distribution is quite diverse ranging from 50ms to 300ms. If we assume that we only need a single connection to compute the approximate processing time, it can improve the throughput significantly.

For the local mail server experiment, we simulate the scenario where triangular spam-

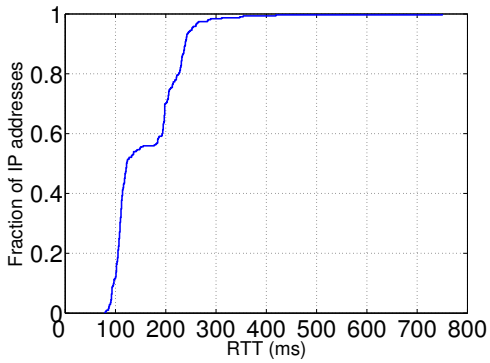


Figure 5.8: Hotmail RTT

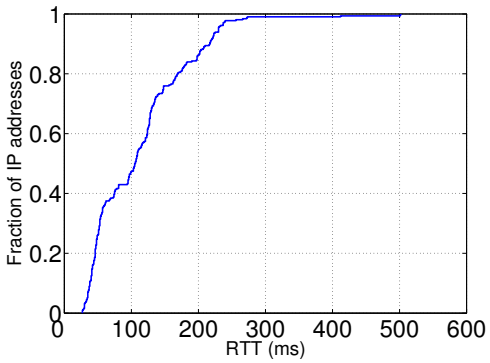


Figure 5.9: Gmail RTT

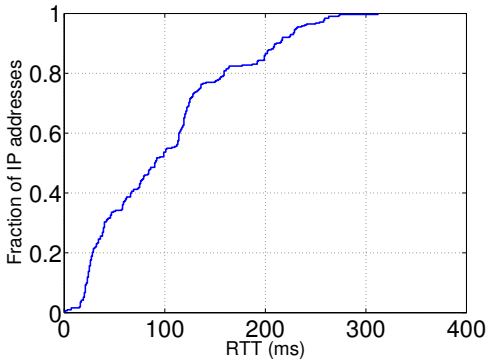


Figure 5.10: Local mail server RTT

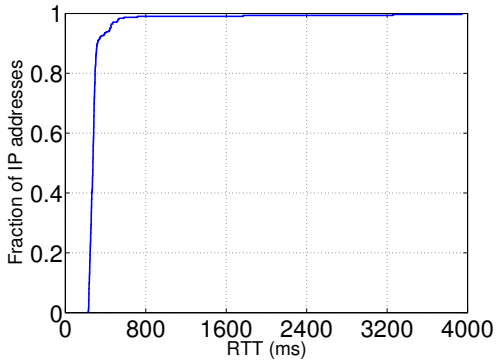


Figure 5.11: Indian server RTT

ming is carried out within the same ISP or organization as the victim mail server. In this case, although the direct RTT between our original sender and the local mail server is only 0.4ms, the RTTs observed are much larger due to triangular routing. However, the increased stealthiness achieved by triangular spamming has the cost of affecting throughput due to large RTTs. Aggressive pipelining could help to improve the throughput of each individual connection significantly.

For the Indian mail server experiment, we simulate the scenario where spammers are targeting a mail server far away from the original sender. We can see that the RTT is clustered at around 200 - 300ms, for 82% of IPs studied, which is mostly bounded by the RTT between the original sender and the target mail server. In fact, the smallest RTT is 227ms, indicating that it could be effective to use aggressive pipelining. But some initial measurement of the processing time has to be done rather than in parallel (where the processing time measured from quick bots can be used for slow bots).

5.4.4 Implication on detection

We observe that although the IP address can be spoofed, some properties exhibited by the original sender may not be easily imitated. For instance, they may run different operating system and resulting in different OS fingerprints. Also, the network delay between the target mail server and the original sender can be different from the delay between the target mail server and the spoofed host. If we can probe the spoofed host in real time to detect deviations in such properties, we may be able to discover triangular spamming. In this section, we briefly discuss several properties promising for detection. Detailed detection results will be shown in §5.5.

5.4.4.1 Round Trip Time difference

As we have shown in Figures 5.8 through 5.10, RTT can differ widely across relay bots. However, from the target mail server's perspective, it does not know the original sender's IP address and can only observe two other RTTs. One is active RTT between itself and the relay bot by direct probing. The second is passive RTT observed locally by observing the delay between sending SYN and receiving SYN-ACK. If no triangular spamming is involved, the two RTT values should be comparable.

However, in the presence of triangular spamming, the passive RTT is calculated as $t_1 + t_2 + t_3$ where t_1 to t_3 correspond to the network delays of the three steps shown in Figure 5.1. The active probed RTT can be calculated as $t_2 + t_2'$ where t_2' is the reverse path network delay of step 2. For simplicity, we assume t_2' to be roughly the same as t_2 (similarly for t_1 and t_3 as well) which allows us to calculate the likelihood of detecting RTT differences. If we compare the passive RTT with the active RTT, the difference is $(t_1 + t_2 + t_3) - 2 \times t_2 = (t_1 + t_3 - t_2)$. Although triangular inequality is shown to be invalid sometimes [109], we show that the chances that $t_1 + t_3 - t_2$ is close to 0 would still be small.

To understand how likely we can observe large values for $t_1 + t_3 - t_2$, we again conduct

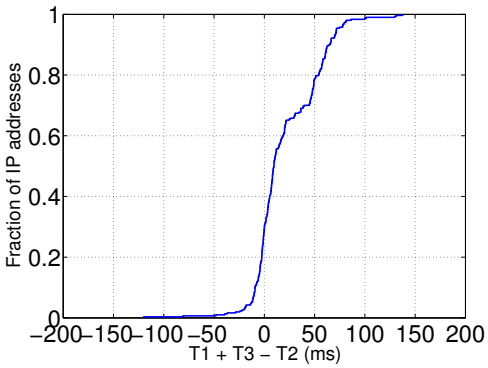


Figure 5.12: Hotmail passive/active RTT difference

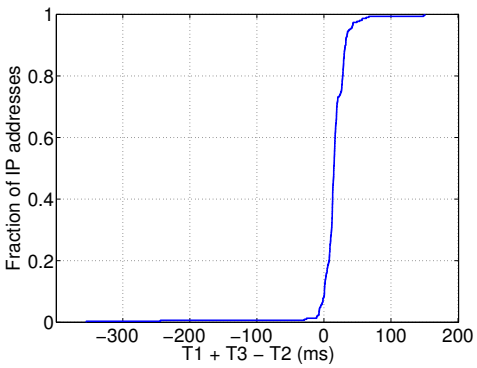


Figure 5.13: Gmail passive/active RTT difference

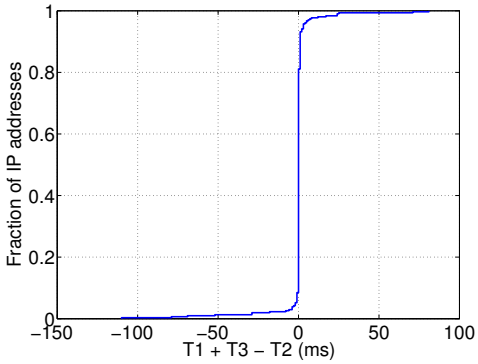


Figure 5.14: Local mail server passive/active RTT difference

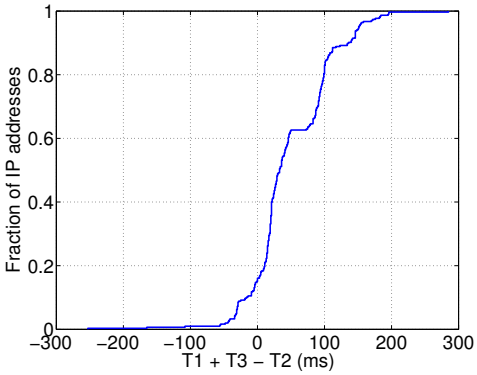


Figure 5.15: Indian server passive/active RTT difference

experiments on Planetlab. First, we measure $t_1 + t_2 + t_3$ as previously described. Second, we measure $2 \times t_1$ by probing from the original sender to the target mail server. Last, we measure $2 \times t_3$ by probing from the original sender to the Planetlab nodes. The distribution of the value $t_1 + t_3 - t_2$ is shown in Figures 5.12 through 5.15.

The results show that for Hotmail server, about 20% of the IP addresses exhibit a difference between passive RTT and active RTT of 50ms or larger. Depending on the relative deviation from the passive RTT, 50ms can be sufficiently large to be considered as an anomaly. For Gmail servers, the difference is much smaller, so is the absolute RTT value. For the local mail server, little difference is found between the passive RTT and the active RTT. This is expected because original sender and the target mail server are very close such that t_1 is close to zero and t_2 and t_3 are approximately the same.

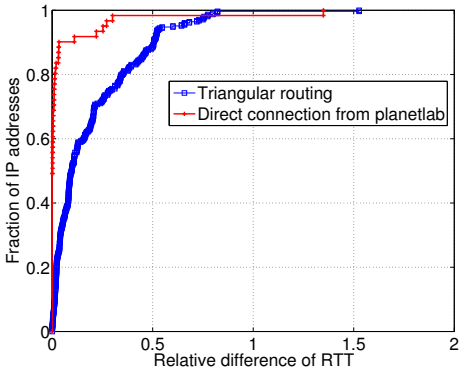


Figure 5.16: Hotmail relative deviation from passive RTT

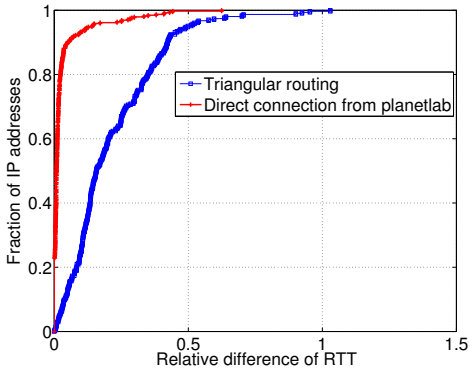


Figure 5.17: Gmail relative deviation from passive RTT

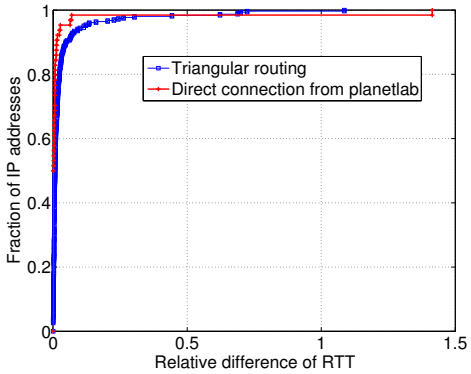


Figure 5.18: Local mail server relative deviation from passive RTT

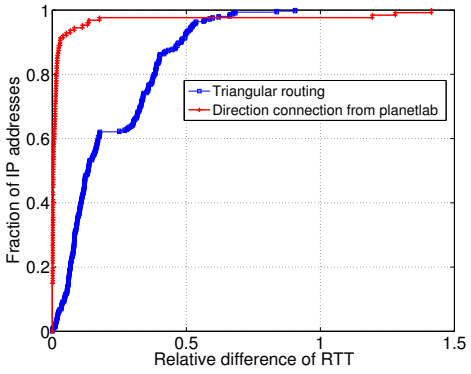


Figure 5.19: Indian server relative deviation from passive RTT

From Figure 5.16 to Figure 5.19, we can see the distribution of relative difference of the active and passive RTT computed by $\frac{t_1+t_3-t_2}{t_1+t_2+t_3}$. However, in order to know if the difference is large enough to be an anomaly, we need to know the baseline of normal RTT variation in a short period of time (within the order of seconds). Although it is known that the RTT variation can be very large over time, in triangular spamming detection, we can set up a real-time active probing infrastructure to probe the active RTT and compare it against the passive RTT. If the real-time RTT variation is small, it is still possible to detect such relatively large and stable RTT difference introduced by triangular routing. We measure the RTT variation by sending 3 consecutive probes with 1 sec interval from the Planetlab nodes. The results clearly show that 95% of the times the relative difference will be smaller than 0.1. This implies that the relative RTT difference can be a useful feature for detecting triangular spamming.

5.4.4.2 TTL value difference

Similarly, TTL values generated from the original sender can differ from the ones generated by the relay bot as observed via active probing. Previous work has studied the effectiveness of using TTL value to detect spoofed DDoS traffic [110]; thus, we do not repeat the study of measuring the difference in TTL values. Here we point out one key difference between DDoS attack and triangular spamming: spoofed DDoS attacks usually have no control over the hosts with spoofed IPs, but in triangular spamming, the relay bots coordinate with the original sender. It is thus harder to detect triangular spamming. More specifically, the original sender can craft a starting TTL such that the receiver cannot tell whether it is generated by a different host other than the relay bot.

However, this coordination has to be done on a per-destination and per-relay-bot basis which is likely high overhead and may severely degrade the spamming throughput. This is not a huge problem in DDoS attack since the attack is typically targeted and well prepared before the actual attack. The difficulty lies in obtaining the correct initial TTL at the real

sender.

In conclusion, we think TTL can be a useful feature for detecting triangular spamming despite the robustness problem outlined above.

5.4.4.3 OS fingerprint

It is also possible to collect lightweight passive OS fingerprints from the first SYN packet using tools such as p0f [25]. Clustering the IP addresses by fingerprints can help detect traffic associated with the same original sender. However, this can be evaded since the original sender can easily modify its kernel to mimic different types of OS fingerprints.

5.4.4.4 Port blocking behavior

If the OUT SMTP traffic is blocked at the ISP, there should not be any SMTP traffic generated from such IP addresses. As a result, if we can identify that an incoming IP is blocked for OUT traffic, then it is highly likely that triangular spamming is used. The limitation is false negatives. This can only detect cases where the relay bots are indeed blocked for OUT traffic which is not a requirement for triangular spamming.

5.5 Detection and Prevention

In this section, we discuss how to apply the previously discussed techniques using data on our departmental mail server to detect any evidence of triangular spamming and its prevalence. We also discuss possible prevention techniques.

5.5.1 Experiment setup

Data source. Monitoring a mail server at our university department mail server for 8 days from 2009.11.9 - 2009.11.16, we see about 360,000 emails, of which 233,746 (87.6%) emails are spam emails (according to SpamAssassin) from 200,347 distinct IP addresses.

Each log entry has four pieces of information: timestamp, sender IP, spam tag, and spam score output by SpamAssassin.

Spam filter - SpamAssassin. Our mail server runs SpamAssassin [2] as the spam filtering system. It employs several detectors including Spamhaus [36] (IP-based blacklist). Every email is labeled based on a computed score combining results from all detectors. If the score exceeds a fixed threshold (5.0 in our case), the corresponding email will be labeled as spam.

Real-time probing experiment. We probe in real-time to gather the TTL and RTT information to help detect triangular spamming. Two sets of probes per destination port per host are conducted, one with source port 25 and the other with source port 80. The destination port is chosen from the most popular ports potentially open on the hosts, *e.g.,* port 25, 80, 22. We limit probing to at most 4 different destination ports to limit the overhead. Probing is stopped if any destination port responds. A lack of response from source-port-25 probes in the presence of responses from other source ports (*e.g.,* port 80) indicates that the IP address is blocked for SMTP traffic.

5.5.2 Detection results

Given that triangular spamming can abuse port-25-blocked IP addresses for sending spam, we are interested in knowing the prevalence of such IP addresses seen by our mail server and the usefulness to correlate this with other features such as TTL discrepancy from real-time probing. For instance, if their port 25 is blocked, does that mean the actively probed TTL value is more likely to be deviate from the passively observed TTLs?

Our results show that on average about *4%* of IP addresses observed by our mail server are blocked for port 25 based on our real-time probing, indicating likely presence of triangular spamming. We then compute various correlations to verify whether such blocked IPs differ from other IPs in any particular properties. Note that if these IP addresses are involved in triangular spamming, they might also relay our probing packets such that we

can still get response. However, the original sender may think our probing packet is invalid and thus do not care to respond.

Spam ratio. The spam ratio of these blocked IP addresses are 99.9%. Compared to the overall 87.6% spam ratio, it is evident that these IP addresses are much more likely to send spam. In fact, we found only 4 blocking IP addresses (out of 7246) sent legitimate emails. Upon a closer look, 1 of them is likely false negative by SpamAssassin based on its DNS name (18.151.195.200.static.copel.net) and it is listed on multiple blacklists. The other three do appear to be legitimate mail servers. However, we found that these three servers have a particular security policy that results in no response to our SYN-ACK packet if the packet has source port 25. They do however respond to SYN packets with source port 25. We plan to incorporate SYN probing in our real-time probing in the future to more reliably determine the port 25 blocking behavior.

TTL or hop count value difference. For these blocked IP addresses, we compare the actively probed TTL value using source port 80 with the passively observed TTL value. Using TTL value directly can result in inaccuracies due to differences in initial TTL value settings. Instead we infer the hop count values using previously established approach [82] to compute the absolute difference in hop count for the two TTL values. Besides triangular spamming, the discrepancy could also stem from firewall or gateway responding to our probing, which is not easy to discern from anomalous triangular routing. Figure 5.20 plots the distribution for hop count difference for blocking and non-blocking IPs. It indicates that blocked IP addresses are more likely to have a larger hop count difference, likely caused by triangular spamming.

RTT difference. Similar to TTL, we plot the relative difference of RTT for blocking and non-blocking IP addresses shown in Figure 5.21. We can see that clearly blocking IPs tend to have a larger relative difference. In fact, more than 50% of the blocking IPs have a relative difference greater than 0.1 which is already larger than the expected difference as shown in our planetlab RTT study.

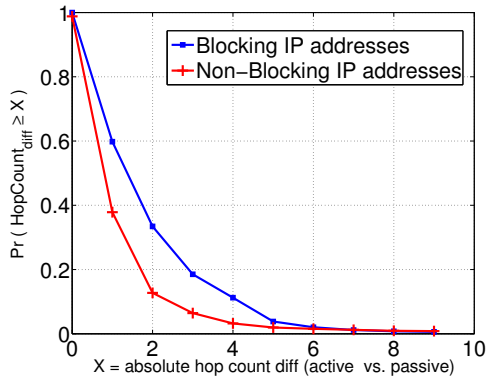


Figure 5.20: Absolute hop count for blocking and non-blocking IPs

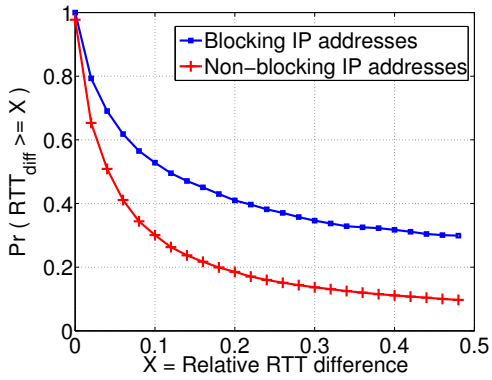


Figure 5.21: Relative RTT difference for blocking and non-blocking IPs

OS fingerprints. We group a set of IP addresses into clusters based on the passively observed OS fingerprints. Here we include hop count in the OS fingerprint (which we call signatures) to consider IPs with same OS fingerprint but with different hop count as different senders. The results show that although there are 52,860 different signatures out of 233,746 IP addresses, the entropy of the signature is only 10.7 bits, indicating that it may not be diverse enough. We also found that the blocking ports are often with different signatures, likely caused by the fact there are multiple original senders involved in the current triangular spamming architecture. In fact, there can be a hierarchical structures where there are a set of high bandwidth bots acting as original senders and each of them peers with multiple relay bots.

Blocking prefix correlation with a popular mail service provider. By correlating with a month-long mail service provider's sampled log trace using the coarse-grained blocking prefix information, we gathered 93,359 IP addresses of which 97% are sending purely spam. While there are certain legitimate mail servers mixed in the blocking prefixes, it is possible to filter them based on their DNS MX records (reverse look up their hostname based on IP, and then look up the MX record for that domain name). Further, 95% of these IP addresses appear only in 5 or fewer days and 54% of them only appear in a single day in the month-long trace, indicating that they are more likely to be spam hosts. It is known

that stable IP addresses are tend to be legitimate while appear-once IP addresses are likely to be spamming IP [106].

From the results above, we can conclude that despite the stealthy behavior of triangular spamming, it also exposes information that can be leveraged for detection. Features proposed above are lightweight enough that they can be easily collected during run-time to help classify spam.

5.5.3 Prevention

Two straightforward ways exist to prevent triangular spamming. The approach of disallowing IP spoofing at every ISP is not feasible given the scale of the Internet and a lack of unified configuration enforcement. The other more realistic way is to have ISPs that block OUT traffic also block IN traffic. However, it does put the management burden on ISPs to correctly configure the firewall rules. Another less obvious prevention method is to deploy stateful firewalls at ISPs to prevent relay bot from relaying out-of-state TCP traffic or just focus on port 25 related traffic to limit imposed overhead. For instance, it is possible to push such functionality into the modem so that it does not have to be deployed at some centralized point to cause performance problems. The question is whether stateful firewall is a desired feature for customers. We think arguably that this can be the right decision and most users will not likely be impacted, just as the case for outbound SMTP traffic blocking.

5.6 Discussion and Summary

To conclude, this chapter has highlighted the practice of triangular spamming, a stealthy and fairly efficient spamming technique that can be relatively easily carried out on today's Internet. Specifically, we analyze its impact by measuring how many networks have the SMTP port blocking policies that can be bypassed by such spamming activities. The measurement is done through probing techniques based on the IPID introspective side channel.

Note that triangular spamming exploits the network level security vulnerabilities (IP spoofing and insufficient firewall port blocking policy). We believe this is just one instance where application protocols are misused due to the underlying network vulnerabilities. Other attacks remain possible.

CHAPTER VI

Validating the Feasibility of Targeted DoS Attacks in Cellular Networks

6.1 Introduction

In this chapter, using timing side channel, we validate the feasibility of targeted DoS attacks in cellular networks to see if there exist distinct signatures observable in different locations. The idea is that different network elements may have different configuration parameters to accommodate the load difference in different areas. It turns out that the timing channel can be leveraged to infer these parameters to validate the assumption that devices in a target location can be identified for the purpose of DoS attack.

Data cellular networks are perennially constrained by limited radio resources due to ever-increasing user demand. To enable efficient resource reuse, there exists built-in support for intelligently allocating radio resources among multiple users and releasing them when they are perceived no longer actively in use. However, the mechanism for implementing the radio resource management primitives requires complex and heavy signaling procedures, rendering cellular networks vulnerable to a variety of low-rate *targeted DoS attacks* [80, 105, 99]. Due to the low bandwidth property, they can be launched from merely one or more external Internet hosts and can deny service to a large number of legitimate users in a particular area. In Figure 6.1, this class of attacks is illustrated at the bottom left

side. Different from another class of attacks such as HLR attack [104] (on the top right of the figure) that requires access to cellular botnets, the low-rate targeted DoS attacks are considered much more realistic given the weaker requirements.

Although targeted DoS attacks are seemingly serious threats, they are still hypothetical. In particular, an important requirement, *i.e.,* locating a sufficient number of mobile devices in the same area (the so-called hit-list) is not fully investigated. Previous work [64] on exploiting SMS functionality to overload cellular voice service proposed using phone numbers under specific area codes to generate the hit-list. However, phone numbers cannot be directly translated into IP addresses needed for launching targeted DoS attacks on cellular data services and have limited location accuracy (detailed in §6.5.3).

To bridge this gap, we develop localization techniques to identify IP addresses under a given area via active probing and fingerprinting. It works by controlling a probe phone under the target area to measure the baseline signature of that location, then searching for other devices with sufficiently similar signature to the baseline. Our observation is that it is possible to actively measure characteristics and configuration parameters associated with network elements that are similar at the same location and distinct across locations. We empirically evaluate this approach on two large UMTS carriers in the U.S. (anonymized as Carrier 1 and Carrier 2 for privacy concerns). We find the approach promising in identifying a set of near-by mobile IPs with high accuracy. This is particularly true for big cities that often have unique configurations or features to satisfy their load requirement. In certain cases, it is even possible to uniquely identify a big city such as NYC with only a single feature. Thus, our work demonstrates that the threat of targeted DoS attacks is real.

Besides exposing the potential abuse of network information to enable this class of targeted attack against cellular networks, our technique can also support legitimate privacy-preserving applications that rely to the knowledge of the number of nearby users to determine whether a user should send his location information while ensuring k-anonymous properties [71]. More generally, our technique opens up a new direction in understanding

how critical infrastructures like cellular networks can leak information about their networks which leads to privacy implications, *e.g.,* in the form of location exposure.

In this work, we make the following contributions:

- We conduct the first large-scale empirical study on the feasibility of targeted DoS attack against commercial cellular data networks (with overview shown in Table 6.1), using data collected through our deployed mobile app on major smartphone platforms (with details of the app covered in §6.3). We show that 80% of the devices keep their device IPs for more than 4 hours, leaving ample time for attack reconnaissance.
- We develop novel techniques to map IP addresses of mobile devices to a geographic area, using a combination of network features including static configuration settings, topological properties, and dynamic features.
- Using the proposed network signatures, we empirically validate the evidence of diverse network signatures across Radio Network Controllers (RNCs). We show that in big cities, the signature is typically unique enough to allow an attacker to locate enough IPs to impose 2.5 to 3.5 times the normal load on the network.

The rest of the chapter is organized as follows. IP-related feasibility analysis is described in §6.3. §6.4 describes the methodology for localization, and §6.5 discusses the evaluation results. Possible defense solutions are described in §6.6. Finally we conclude in §6.7.

6.2 Attack Overview

RNC-level Localization. At a high level, our intuition is that network elements (*e.g.,* RNC) in different locations may be configured differently, therefore exhibit distinct network signature. Since geographically close-by devices are governed by the same RNC, they also share the same signature. We discuss a more complete list of features of our proposed network signature in §6.4.2.1. Since most features identified are associated with the RNC, the localization granularity is naturally at the RNC level. Typically, a big city such

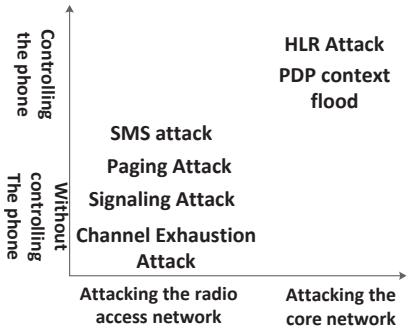


Figure 6.1: Cellular attacks

Category	Measurement target
MobileApp	Cellular IP reachability analysis (§6.3.1)
MobileApp	IP duration analysis (§6.3.2)
Probing	Active vs. Idle IPs (§6.3.3)
MobileApp	IP locality analysis (§6.4.1)
MobileApp	Phone-number-based localization accuracy (§6.5.3)

Table 6.1: Attack feasibility analysis

as NYC has multiple RNCs, while a few small cities may share the same RNC. To clearly illustrate the attack, we outline the steps.

1. The attacker has access to a *probe phone* in the target location. This phone measures the network signature of that location and uses it as the baseline to match against signatures of other probed phones. The probe phone could either be compromised or owned by the attacker. Note that we do not assume access to a cellular botnet. Otherwise, it is trivial to obtain the location information (*e.g.,* GPS) from the phone.

2. The attacker first exclude IPs that are unlikely under the target area through a coarse-grained localization step. For instance, the attacker can exploit the approximate IP locality in cellular networks, *i.e.,* each GGSN responsible for a geographic region assigns distinct IP blocks to phones (See §6.4.1).

3. The attacker probes the remaining IPs to measure their associated signatures (as a fine-grained localization step) with certain time and resource constraint. The signatures are matched against the baseline signature. The ones that are sufficiently similar are considered to be under the same RNC.

4. After identifying enough IP addresses for the target network location, the attacker can launch the targeted DoS attack.

Carrier count	Private IP		Public IP	
	Reachable[1]	Not	Reachable	Not
180	40	63	52	25

[1] In-NAT device-to-device probe allowed

Table 6.2: Reachability for 180 UMTS carriers.

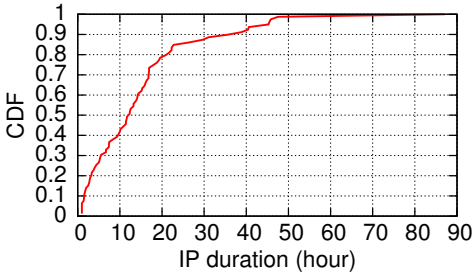


Figure 6.2: CDF of IP duration

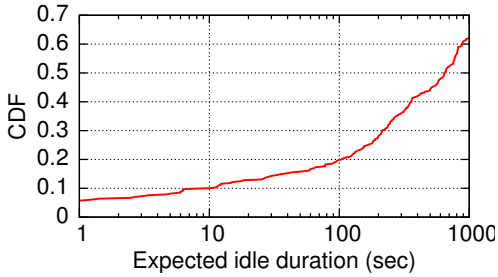


Figure 6.3: CDF of idle duration

6.3 IP-related Feasibility Analysis

In this section, we investigate the feasibility of our localization technique and the targeted DoS attack in general. First (§6.3.1), we study how many carriers allow cellular IPs to be probed directly (a requirement for scanning/fingerprinting and the DoS attack itself). Second (§6.3.2), we quantify how long a device keeps its cellular IP address (to ensure IPs are still valid after fingerprinting). Third (§6.3.3), we measure how many devices are online versus active (to estimate the additional attack load that can be introduced).

MobileApp. To answer the above questions, we collect empirical data using a mobile app we deployed on major platforms (referred to as *MobileApp* thereafter) which measures various aspects of any cellular network including throughput and port blocking behavior. It also reports additional information such as Carrier, GPS, IP address, timestamp, Signal Strength, and 7-digit phone number prefix (with user's consent). As of March, 2011, there are 100,223 users who have voluntarily downloaded and run our app all over the world. There is a small subset consisting of 1,683 users who opted to run the periodic test that report the device's IP address every one hour and allow the signature measurement to be conducted from our server. We focus on the data set collected over a 2-week duration from Nov 7th to 21th, 2010. The complete list of analysis based on the *MobileApp* is summarized

in Table 6.1.

6.3.1 IP Reachability

Some carriers deploy NAT or firewall to ensure traffic originated from the Internet cannot reach the mobile devices directly. They affect signature-based localization as well as the set of existing attacks assuming direct reachability to the phones. Based on the MobileApp data set, we collected reachability information of 180 UMTS carriers in different continents as summarized in Table 6.2. Among them, 77 of them assign public IP addresses to devices, 52 of which are directly reachable from the Internet; 103 carriers assign private IP addresses to devices (with NAT), 40 of which are reachable because they allow either device-to-device probing within the same network or one device to spoof other device's IP address (4 out of the 40 allow IP spoofing). In summary, regardless of public or private IP carriers, the percentage of carriers that allow direct reachability is $\frac{40+52}{180} = 51\%$. This result motivates our localization technique.

In those cases where carriers do not allow phones to be probed in any way, existing attacks discussed in [80, 105, 99] will also be deterred. However, denying reachability is not without any cost. For instance, it may hurt peer to peer applications as well as applications that require running servers on the phones.

6.3.2 IP Duration

It is critical to understand how long a device keeps the same IP address as it takes time to measure the signature and identify the location of an IP. Previous work has shown that when NAT boxes are deployed, the public IPs assigned by NAT can change frequently in cellular networks [53]. For instance, they show that with the same private IP address, the public IP address can change on the order of tens of seconds. However, device IPs (IPs directly assigned to devices) often change much less frequently in contrast to NAT-assigned IPs. Typically such IPs only change when the device reboots or the network interface is

switched (*e.g.,* WiFi to 3G). Carriers do not have incentives to frequently reassign new device IPs as it incurs overhead.

Indeed, from the *MobileApp*'s periodic test data consisting of (deviceID, timestamp, deviceIP) tuples, we quantify the expected IP duration of each device as shown in Figure 6.2 and find the duration of the device IP is fairly long, with more than 80% devices keep their IP addresses longer than 4 hours.

6.3.3 Online Devices vs. Active Devices

We define *online* devices as the ones reachable but may or may not be actively using the network. *Active* devices as the ones reachable and also actively using the network. The ratio of active to online devices determines the upper bound of additional load an attacker can impose on the network.

To quantify the number of online and active devices, we conduct a TCP SYN probing experiment on 12 sampled /24 IP prefixes used in the cellular network operated by Carrier 1 in the U.S. in March, 2011 for two hours at different time-of-day (one hour at noon, one hour after midnight). We probe all IP addresses in the range every 30 seconds. Though the analysis does not cover all the IP ranges, we believe the result is still representative based on randomly sampled IP prefixes.

For validation, we confirm that TCP SYN packets can indeed elicit response from the phone instead of the firewall with spoofed responses. We verify this by launching two different "TCP traceroutes" from the server to the phone – if the probing packet is non-SYN packet, the traceroute will discontinue at the firewall hop (the firewall drops out-of-state TCP packets) while SYN packet can move forward until it reaches the phone and elicit response. To infer if a phone is actively using the network, we exploit the IP ID field in the IP header in the response packets [96]. In practice, many operating systems (such as Windows family) increment a system-wide IP ID counter for every outgoing IP packet. If two consecutive response packets have IP ID values X and $X + 1$, it means the phone is

idle for the 30-second probing interval. Otherwise, it is **active**.

On average, there are 22% of the probed IPs respond (other IPs are likely unassigned at the moment we probe). Out of the responsive/online devices, the active ratio is 20% during day time and 13% at midnight, showing some time-of-day pattern. The ratio indicates that an attacker can introduce at most 5 times the normal load at day time, assuming all IPs can be correctly located. Even if the attacker finds only half of the online phones, he can still impose 3 times the normal load on average, or $\frac{50\%}{20\%} = 2.5$ to $\frac{50\%+20\%}{20\%} = 3.5$, depending on how many identified IPs are overlapping with the ones that are already active. Either way, the attack load is significant enough to cause damage assuming a common over-provisioning factor of 1:2 [69].

From the probing, we can also infer the average idle duration for each IP address. This information helps us understand how likely there will be background traffic interfering with the feature measurement described in §6.4.2.2. Figure 6.3 shows that 85% of the IP addresses have an idle duration of 100 seconds, which are long enough for most measurements.

In summary, we have studied various aspects of real networks and collected evidence demonstrating that the localization scheme and the targeted DoS attack are indeed feasible.

6.4 Localization Methodology

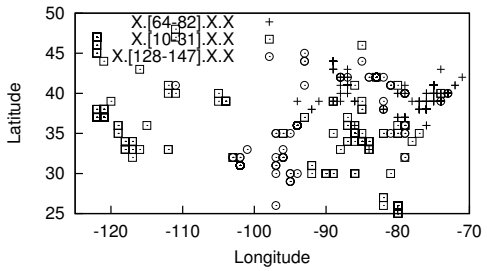


Figure 6.4: IP address locality

Feature	Type	Measurement cost
Min-RTT	Dynamic	Medium
DCH tail time	Static	Medium
FACH DL queue size threshold	Static	Medium
FACH UL queue size threshold	Static	Medium
FACH queue consumption rate	Static	High
Avg promotion delay	Dynamic	Low
RRC State machine	Static	Low
Max number of RLC retransmission	Static	High

Table 6.3: Features of the network signature.

6.4.1 Coarse-grained Localization using IP Locality

IP address is known to have locality on the Internet – close-by IPs in the address space tend to share geographic proximity. However, as pointed out by previous work [114], a large geographic area may share the same set of IP pools managed at a GGSN location. We verify this using 13,784 samples of GPS-IP pairs of Carrier 1 and 17,589 pairs of Carrier 2 from the MobileApp data, during the time period from Sep 2009 to Nov, 2010.

Figure 6.4 shows Carrier 1's private IP address locality. We observe the same private IP range is indeed shared by a large geographic area. For example, X.64.X.X – X.82.X.X are always allocated to devices from the East Coast, while IPs in X.128.X.X – X.147.X.X are mostly assigned to devices on the Central or Northeast. In total, we find 4 such disjoint IP ranges which likely correspond to different GGSNs [114]. Similar results are found for Carrier 2. In summary, device IPs only show coarse-grained geographic locality. Nevertheless, it allows the attacker to exclude a large range of IPs without ever probing them.

For an attacker without prior knowledge about which IP range corresponds to the target area, we devise a simple technique to obtain such knowledge — one can reset the network interface programmatically to continuously obtain new IP addresses. It takes about 30 seconds to get a new IP address every time. In our experiment, this technique allows us to obtain the IP pool of X.128.X.X – X.147.X.X within about an hour.

6.4.2 Fine-grained Localization using Network Signatures

From coarse-grained IP locality, the attacker can obtain a set of candidate IPs only knowing their rough location at the granularity of several states in the U.S. To narrow down to a more fine-grained target location, the next step is probe these IPs directly to obtain their network signatures.

6.4.2.1 Network Signature Definition

Here we define network signature as a set of measurable features that are potentially distinct across different locations. We discuss why the signature exists and how we select them:

- **Incremental changes**: different hardware components are purchased and upgraded at different time to improve capacity of the network or to introduce new technologies [4, 3], thus likely introducing heterogeneity in the RNC hardware. For example, min-RTTs are reduced when HSUPA are deployed on top of HSDPA compared to when HSDPA deployed alone [74]. Also, different RNC products can have slightly different state machine behavior as explained before. As 4G LTE network technology is gradually deployed, we expect even more heterogeneity across network locations.
- **Connectivity between RNC and base-stations**: this may vary across locations. The network delay variation may be due to either RNC processing delay difference or base-station-to-RNC link delay difference (*e.g.,* link congestion) [80, 15], thus affecting the observed state promotion delay.
- **Configurations**: Different RNCs have different static parameter settings to accommodate the load at different locations. For instance, big cities often configure resource-recycling parameters (*e.g.,* inactivity timer) more aggressively to accommodate more users.

The general criteria for selecting features is that any *measurable properties* associated with RNC can be considered. We list the potential set of features in Table 6.3, most of which have been introduced in §2.4.1. Although it may not be complete, it is a good starting point. These features can be classified as static or dynamic. Static features are configured statically on the RNC. They are thus very stable and relatively easy to measure. Dynamic features, on the other hand, are likely to vary over time thus harder to measure. Also, some features are more expensive to characterize. For example, the maximum number of allowed RLC retransmission [83] and the FACH queue consumption rate may take many measurements to correctly identify.

In our current study, we pick the 5 features highlighted in Table 6.3 (and described in §2.4.1) that are relatively easy to measure. The methodology for measuring them is described in the next section. We did not include the RRC state machine feature because we did not find a corresponding ground truth RNC to test our measurement methodology. The local RNC on which we rely for testing did not implement the CELL_PCH state. A dedicated attacker can however spend more effort and take advantage of the remaining features to refine the signature.

6.4.2.2 Measurement Methodology

Our probing methodology assumes control at only one end (*i.e.,* the server side). It is different from a previous study [92] that devised a methodology to measure similar features but with complete control on both ends. In our case, we have to overcome additional challenges such as background traffic on the device which may distort the measured result. Moreover, when state promotion delay is measured from the server, there is an additional variable, *i.e.,* paging delay that needs to be factored out. Finally, without controlling the device, it is hard to control the size of the packet responded by the device (thus hard to measure the UL queue size threshold). For instance, we can use a large TCP SYN packet to probe a device, but the size of the response packet is fixed (40 bytes TCP RST). To control the uplink size, we use UDP probing where the response packet is 24 bytes larger than the original probing packet because the triggered *ICMP port unreachable* packet embed the original UDP packet and wrap it with a new ICMP and IP header. The ICMP response packet is limited to at most 576 bytes as per RFC [58]. We will discuss how to use TCP and UDP probing by adjusting the probing packet size to measure DL and UL queue size threshold respectively.

DCH inactivity timer. As described in §2.4.1, this parameter determines when to release radio resource (after N seconds of inactivity). The procedure to measure inactivity timer is as follows: (1). The attacker first sends a large probe packet (*i.e.,* 1200 bytes). If the

phone is in IDLE or FACH, it will trigger a state promotion to DCH; otherwise, it will reset the inactivity timer [91]. (2). Then the attacker waits for some interval, X seconds before he sends the next large packet. (3). If the next packet incurs an RTT sufficiently large to include the state promotion, this indicates the power state has changed from DCH to FACH after X seconds of inactivity, and therefore the inactivity timer is X seconds. (4). Else, if the next large packet experiences a small RTT, indicating that the power state is still in DCH, it then set $X = X + \Delta$ and go back to step (2). We set the initial X to be 2.5s which is small enough to catch all the known values of inactivity timer. We use a measurement granularity of 0.2s (Δ = 0.2s). It is not useful to increase the granularity further since the variation of normal cellular RTT will prevent us from getting more fine-grained results.

As mentioned, one challenge is background traffic, which may prematurely reset the timer, inflating the measured inactivity timer. The counter strategy is to continue the measurement only when the initial state is in IDLE. The assumption is that if the phone is not using network now, it is likely to continue to stay idle in the near future (demonstrated in Figure 6.3). After all in a real attack, the timer inference can finish much more quickly given the attacker can profile the timer value in the target RNC offline using his attack phone. Specifically, instead of measuring all possible timer values, he only needs to make a binary inference – whether the tested IP has a target timer or not. Nevertheless, the above general methodology assumes no prior knowledge of the timer value; we use it to understand how diverse they can be across locations.

The other source of noise is fast dormancy [107], a mechanism implemented on some devices to release radio resource before the timer expires and save battery life on the phone, causing early transition from DCH to IDLE. However, since the power state transitions directly from DCH to IDLE instead of to FACH, the next observed RTT is much larger and thus distinguishable from the non-fast-dormancy case. More importantly, it is not always implemented or triggered every time. Thus, we can repeat the measurements several times and discard the fast dormancy cases. In practice, we find UDP probing rarely encountered

fast dormancy (compared to TCP probing), likely due to more bytes being exchanged in the UDP case.

FACH $\rightarrow$ DCH DL/UL queue size threshold. As described before, there exists a queue size threshold for DL and UL respectively. Using TCP probing, the size of the TCP RST response is fixed, allowing DL packet size adjusted independently without affecting the UL packet size (ensuring UL threshold is not exceeded first). The following depicts the procedures on the server side to measure DL queue size threshold: (1) send a large packet to make sure the state will be in DCH; (2) wait for $T = Timer$ seconds to let the power state enter FACH; (3) send a packet with size Y; (4) If a state promotion is triggered (identified by a large RTT), then record Y as the queue size threshold and exit; (5) else, try a larger packet size $Y = Y + \xi$ and go back to (3). We set the initial $Y = 20$ and $\xi = 30$ in the MobileApp experiment. In the later large-scale probing experiment, we change $\xi = 10$ bytes in an attempt to measure more fine-grained queue size threshold. Nevertheless, the later experiment confirms that the results with $\xi = 30$ are still valid (without missing any values in between).

Measuring UL queue size threshold is more difficult, because distinguishing between DL threshold and UL threshold being exceeded is hard — in both cases we just observe a large RTT indicating that state promotion is triggered. Fortunately, since we now already know the DL queue size threshold via TCP probing, we can send UDP packets with size smaller than the DL threshold so that they are guaranteed not to exceed the DL queue threshold. If there is still a state promotion triggered, then it has to be the UL packet that exceeds the UL queue size threshold. This is why we have to measure the DL threshold before the UL threshold. There are two constraints in the results: 1). The UL response packet is always 24 bytes larger than the DL probe packet (as described earlier), therefore, we can measure the UL threshold up to 24 bytes larger than the DL queue threshold. 2). The max size of the UL response packet is 576 bytes (as discussed earlier). As a result, the UDP probing can only measure up to 576 bytes for the UL threshold.

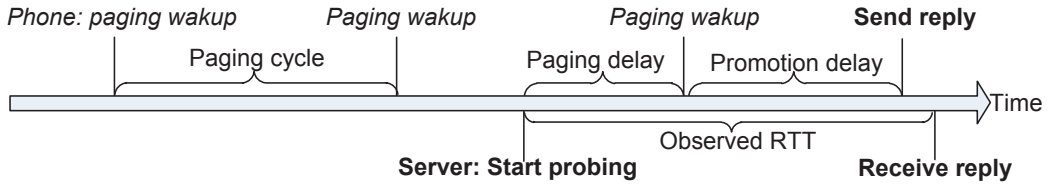
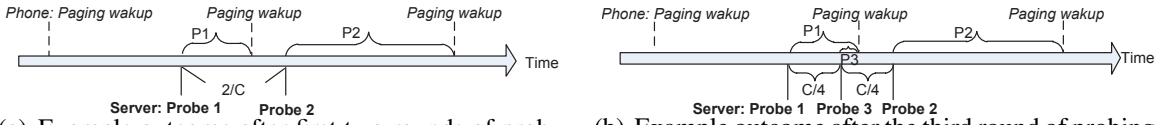


Figure 6.5: An example of paging delay.



(a) Example outcome after first two rounds of probing

(b) Example outcome after the third round of probing

Figure 6.6: Illustration of paging delay elimination to measure promotion delay.

IDLE $\rightarrow$ DCH Promotion delay. Although it is easy to measure promotion delay from the phone, it is much more difficult to do so from the other end. This is because when a downlink packet is to be received in **IDLE** state, it may get delayed significantly due to a process called paging [50].

Figure 6.5 shows an example where the phone is in **IDLE** state, only periodically wakes up to listen for incoming packets. If a packet happens to arrive between two paging occasions, it will be delayed until the next paging occasion. In the worst case, the delay can be as large as a full paging cycle length. The paging cycle length is determined by Core Network and is stable across RNCs [50]. According to the 3GPP specification [49], the protocol should support several values: 0.64s, 1.28s, 2.56s, *etc.* In practice, we observe it to be 2.56s for Carrier 1 and 1.28s for Carrier 2. It is stable and validated through a hardware power-meter: every 2.56s a power spike is seen, and we can observe that the promotion process must wait after the spike.

To overcome the effect due to paging delays, we design a probing technique using binary search that can distinguish the paging delay from the estimated promotion delay. It requires several rounds of probing (the exact number of rounds depends on the accuracy desired). The high-level idea is that we gradually narrow down the exact paging occasion. The more we probe, the closer to the paging occasion the probing will be. We validated the

inferred promotion delays match the duration of power spikes using power-meter. Below is the detailed methodology:

We use the following notations: we denote C as the paging cycle, S_i as the start time of i^{th} probe mod by C ($0 \leq S_i \leq C$), P_i as the paging delay of the i^{th} probing, and T_i to be the RTT of the i^{th} round.

Initially, without any knowledge of the paging occasion, we conduct the initial two probes spaced out with an offset of $\frac{C}{2}$. More specially, $S_2 - S_1 = \frac{C}{2}$. An example outcome is shown in Figure 6.6.(a). We can infer $P_1 < P_2$ based on the fact that $T_1 < T_2$, and paging occasion should be somewhere between $S1$ and $S2$. Despite potential variation in promotion delay (on the order of hundreds of milliseconds), the spacing between P_1 and P_2 of $\frac{C}{2}$=1.28s should be large enough to overcome the noise.

We select the next S_i to bound the paging occasion further. For example, in Figure 6.6.(b), the start time is selected to be $S_3 = \frac{S_1+S_2}{2}$ given $T_1 < T_2$. Note that here the difference between P_3 and P_2 is even bigger than the difference of P_1 and P_2, making each step of the algorithm robust to the noise of promotion delay variation.

The process is repeated until we are able to bound the S_i to be very close to the paging occasion. We pick i to be 8 since it is able to bound the distance to be less than $\frac{2.56s}{2^8} = 10$ms. Even though each individual promotion delay may vary significantly, the average is expected to be stable, thus collecting multiple samples of promotion delay is useful. After inferring the paging occasion, we are able to recover each previous paging delay P_i, and the corresponding promotion delay can then be computed simply as $T_i - P_i$.

Note that it is possible there are background traffic during the 8 rounds of probing since it spans $8 \times 30 = 240$ seconds. However, such background traffic only matters when the measured promotion delay is very small, indicating the device is already in DCH instead of IDLE. Such cases can be easily excluded and the attacker can simply try again in the next round.

Minimum RTT in DCH. Although it has been shown that cellular network RTT is highly

variable, we find the minimum (over 40 measurements) is surprisingly stable within the same area (See Figure 6.9). It is also diverse in different geographic areas. Our local experiment shows that the standard deviation of 100 min-RTTs (each is computed over 40 RTT measurements) measured in 6 hours is only 2.003. In fact, computing min-RTT over 30 measurements is also acceptable as its standard deviation is 2.07. 10 measurements will however increase the standard deviation to 3.46.

Validation. We thoroughly validate the proposed methodology using local experiments over two different UMTS carriers across several types of phones. The results are consistent and match with those collected assuming direct access to the phone. For instance, we verified that the UL/DL queue size threshold measurements are consistent with or without direct access to the phone.

6.5 Evaluation

In this section, we evaluate the effectiveness of the signature-based localization through empirical data collected from the MobileApp as well as a large-scale probing experiment. We focus on three main evaluations.

1. We quantify the distribution and entropy of each feature.

2. We compare the signature-based localization with the state-of-the-art phone-number-based approach.

3. We evaluate the scalability, *i.e.,* time requirement, to locate enough number IP addresses.

Data collection. We collect data from two main sources. One is from the periodic test of the MobileApp, which allows us to obtain ground truth RNC, timestamp, and signature to evaluate two major UMTS carriers in the U.S. covering 20 RNCs in Carrier 1 (shown in Figure 6.9) and 43 RNCs in Carrier 2 from Nov 8th to Nov 19, 2010. In this dataset, we collect all five features except the UL queue size threshold, which can be measured in the actual attack via UDP probing as discussed in §6.4.2.2. The other dataset is from large-

scale probing on Carrier 1 in March 2011, the goal of which is to understand the signature distribution more comprehensively without knowing where each IP is located. The two datasets jointly allow us to evaluate the accuracy of our localization technique.

RNC ground truth. Even though we run our MobileApp on the phone, there is still no direct way to obtain the current RNC the phone is connected to. As an approximation, we obtain "location area code" (LAC) — an identifier that covers an area that is usually comparable to an RNC [19] through standard platform APIs.

RNC/LAC coverage estimation. To understand the challenge to distinguish different RNCs, we first need to estimate the number of RNCs and their coverage. To do so, we leverage the OpenCellID project [24], an open source project aimed to create a complete database of LAC number and Cell IDs worldwide. The data shows 1596 unique LAC numbers for Carrier 1 with about 3/4 of the U.S. covered. A rough estimate based on area uncovered leads to about 2000 LACs in the entire U.S. Since the number of GGSN locations is estimated to be 4 (See §6.4.1), a single GGSN should cover about 500 LACs/RNCs. Typically, a big city such as NYC has multiple RNCs (to accommodate the load), while a few small cities may share the same RNC (verified by LAC and location data from OpenCellID).

Evaluation methodology. As described in §6.2, in the actual attack, an attacker measures the baseline signature from his controlled probe phone and then compare it against signatures of other probed devices in real time. We simulate this process by considering each signature sample in the MobileApp as a baseline (where we know its ground truth RNC) and compare it with other samples for a similarity match. Static features should have exactly the same value to be considered a match. For dynamic features, *e.g.,* average promotion delay and min-RTT, we allow a **signature distance threshold parameter** (**SD-threshold** in short) to tolerate some variation. For example, if we set the min-RTT SD-threshold to 5ms, then all samples within +/- 5ms of the baseline sample are considered to match the baseline. For min-RTT and promotion delay SD-threshold, we select a large range of them

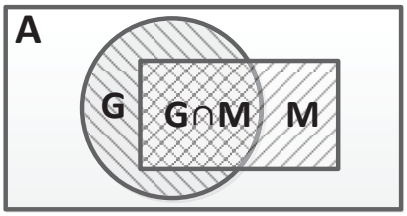


Figure 6.7: Illustration of coverage and accuracy

from (5ms, 50ms) to (20ms, 400ms). We discuss later how set the SD-threshold to balance between the coverage and accuracy metrics.

Coverage and Accuracy metrics. As illustrated in Figure 6.7, the sets A, G, and M represent different signature samples. A represents signature samples for all RNCs. G denotes all the signature samples under the target RNC. M corresponds to the signature samples inferred to be under the same target RNC (using the localization technique). Ideally, localization should output an M that is equivalent to G. However, in reality, M and G may only partially overlap. Based on the notations, we define two metrics to evaluate the performance of the localization: $Coverage = \frac{|G \bigcap M|}{|G|}$ and $Accuracy = \frac{|G \bigcap M|}{|M|}$. Intuitively, coverage quantifies the fraction of phones that can be successfully identified under the target RNC. Higher coverage implies that the attack can impose higher load on the network. Accuracy quantifies the fraction of targeted phones that are indeed under the target RNC. Higher accuracy implies less wasted attack traffic.

6.5.1 Evaluation using the MobileApp Data

In this section, we mainly discuss the findings about the characteristics of the features and city-level properties (*e.g.,* big cities usually have distinct values). We also briefly discuss how we can compute the localization coverage metric using just the MobileApp data.

Static features. We find that although inactivity timer and queue size threshold do not have many distinct values, they already show some interesting diversity patterns even from the limited samples.

For Carrier 1, we observe two main inactivity timer values: around 3s and 5s. For

the queue size threshold, we also observe two main values: around 480 and 510 bytes. 480 tends to correlate with the 5s inactivity timer, while 510 tends to correlate with the 3s one. Besides these, there is one exceptionally large queue size threshold at around 1000 bytes observed at three places – Bay Area, Dallas, and Chicago Area (*i.e.,* big cities). In Figure 6.8, we can clearly see the distribution of different queue size threshold values. Marker 1, 2, and 3 represent 480, 510, and 1000 bytes respectively. It is interesting that Marker 2 and 3 are always at big cities while Marker 1 is often present in small cities. In the cases where two Markers are closely located (*e.g.,* Marker 1 and 3 in Dallas), we observe that Marker 1 is in fact at a nearby small city.

For Carrier 2, we observe similar patterns. Two main values of inactivity timer: 3s and 8s, and two main values for queue threshold: around 110 and 300 bytes. However, there is also an exception of 1000 bytes which is only observed in New York City (multiple LACs in the city and different users consistently show this value), likely due to its higher load requirement. We also observe interesting patterns around big cities, *e.g.,* Los Angeles is configured with 300 bytes while near-by California cities are configured with 110 bytes.

Despite the limitations of the small number of observed values, we note that the features are promising in distinguish big cities which often have bigger queue size threshold than their near-by cities. Coupled with dynamic features, the signatures will be even more unique.

Dynamic features. Figure 6.9 shows the distribution of Carrier 1's two dynamic features – promotion delay and min-RTT. Each data point is an average over 8 measurements of promotion delay (after subtracting the normal RTTs) and 40 measurements of RTT. We employ 95^{th} percentile filtering to exclude outliers of promotion delay due to measurement noise. Data points with the same shape indicate that they are from the same LAC. We can see from Figure 6.9 that min-RTT is very stable in most cases, even though a user can move around in nearby locations connecting to different base-stations with varying signal strength.

Figure 6.8: Carrier 1's queue threshold

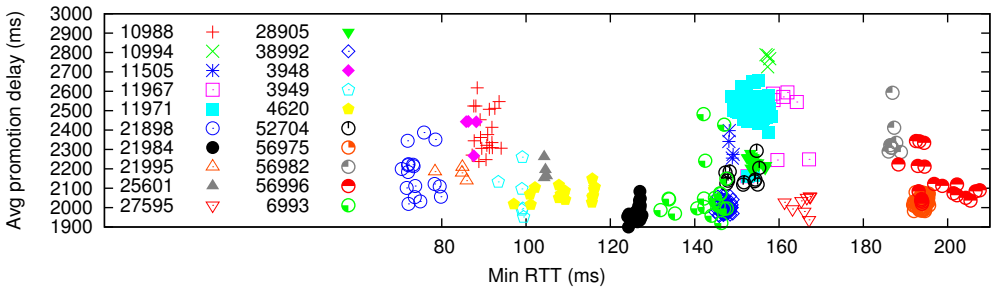


Figure 6.9: Carrier 1's 20 different RNCs' dynamic features

For promotion delay, it is still reasonably stable but less stable than min-RTT. In Figure 6.9, we aggregate enough data points during off-peak hours (3AM - 6AM where promotion delays should be stable), which have relatively stable range of values and do show diversity across different LACs. In the cases where two different RNCs have similar signatures (*e.g.,* at min-RTTs close to 150ms in the figure), we may need to combine static features to distinguish them.

Coverage calculation. Coverage is relatively simple to calculate using only the MobileApp data. As defined in §6.5, $Coverage = \frac{|G \bigcap M|}{|G|}$. G denotes all the signature samples under the target RNC. We use the samples collected in MobileApp data (where we know the ground truth RNCs) to serve as G. $G \bigcap M$ is the signature samples correctly inferred to be under the target RNC. They both can be derived from the MobileApp data.

For accuracy, however, using the MobileApp data alone is not sufficient because it is possible there are other unknown RNCs which have similar signatures, *i.e.,* collisions. This motivates our subsequent measurement on the general distribution of the signatures in an entire GGSN.

Inact. timer	queue threshold DL	queue threshold UL	Prob.	Inact. timer	queue threshold DL	queue threshold UL	Prob.	Inact. timer	queue threshold DL	queue threshold UL	Prob.	Inact. timer	queue threshold DL	queue threshold UL	Prob.
5s	480	484	41.1%	3s	510	>=534	4.3%	3s	480	244	1.7%	3s	1010	244	1.4%
3s	510	244	28.8%	3s	480	>=504	2.7%	3s	<=340	244	1.6%	3s	1010	>=576	1%
5s	480	>=504	6.9%	3s	480	484	2.6%	5s	<=340	244	1.5%	Others			6.4%

Table 6.4: Static feature probability distribution from large-scale probing

6.5.2 Evaluation via Large-Scale Probing

In this section, we focus on two results. First, we characterize the signature distribution of all five features via large-scale probing. Next, based on the distribution, we calculate the localization accuracy for the MobileApp data where we have the ground truth RNC.

Evaluation methodology. In our large-scale probing data, we measure all 5 features but only 4 features are measured in the MobileApp data (UL threshold was not included). When we join two datasets together, we need to make up for the missing feature in the MobileApp data. We choose a very conservative way to do this – we fill in a UL threshold such that the (DL,UL) threshold combination is the least unique possible. This way, the signature is never more unique than it should be, thus the resulting localization accuracy can only be worse.

Signature distribution measurement for Carrier 1. Previous results only provide a limited view on the signature distribution. To get a more comprehensive view, we sample $\frac{1}{10}$ of an entire GGSN IP range in Carrier 1 comprising 20 /16 IP prefixes to measure all five features.

First, we use *Shannon entropy* – a metric measuring the uncertainty associated with a random variable, to measure how "random" the network signature variable is and how many RNCs can be distinguished on average in our context. Entropy is defined as $H(X) = -\sum_x p(x)\log_2 p(x)$, where X is the signature variable and x is a possible value for X. Here we break X into S and D representing static and dynamic features respectively. Thus $H(X) = H(S, D) = H(S) + H(D|S)$, by the law of entropy.

For $H(S) = -\sum_s p(s)\log_2 p(s)$, we need the s and $p(s)$ which are both shown in Table 6.4. There are in total 11 different static feature s with at least 1% probability (ap-

pearance). Plugging in the probability $p(s)$, entropy $H(S)$ is calculated to be 2.3 bits, indicating that we can distinguish on average $2^{2.3} = 5$ different groups of RNCs, using static features alone.

A closer examination of the table shows that, despite the inactivity timer taking on only two distinct values of 3s and 5s, the queue size threshold values are much more diverse than those from the MobileApp dataset (especially when combined with inactivity timer values). The distribution is biased towards the top two combinations where DL threshold values are 510 and 480, which match the values observed in our MobileApp dataset. This means that some RNCs will be harder to distinguish than others. Specifically, big cities with smaller inactivity timer and bigger queue size threshold are typically more distinct, considering that 41.1% of the static features contain 5s inactivity timer and 480 DL threshold (likely in small cities). The results indicate that big cities are more easily identified and targeted.

For $H(D|S)$, it can be computed as $\Sigma p(s) \times H(D|S = s)$ by enumerating different static feature s. To compute $H(D|S = s)$, we slice the two-dimensional dynamic feature space into equal-sized bins using various threshold choice of min-RTT and promotion delay (as described in §6.5). Plugging in the probability associated with each bin, the computed $H(D|S)$ varies from 2.5 to 5 bits. This means the combined entropy $H(S, D)$ is between 4.8 and 7.3 bits, which can distinguish 28 – 158 different groups of RNCs on average. Remember this is without including other potential features in Table 6.3 which can provide additional entropy. Using the calculated entropy number and the estimated RNC number (500 per GGSN), we know on average there are $500/158 = 3.2$ to $500/28 = 17.9$ RNCs left indistinguishable to the attacker. Again, given the distribution is nonuniform, the actual accuracy will depend on which RNC is targeted.

Accuracy calculation using both the MobileApp and probing data. We already discussed how to compute coverage in §6.5.1. Now we discuss how to calculate $Accuracy = \frac{|G \bigcap M|}{|M|}$. First, accuracy calculation requires knowing M — the signature samples inferred to be under the target RNC. We perform the following procedure to output M — for each

LAC	Signature		Location	Phone-number	
	Coverage	Accuracy		Coverage	Accuracy
11956	52%	41%	Queens, NY	N/A[1]	
11974	50%	83%	Newton, NJ	N/A[1]	
26507	50%	3%	Ann Arbor, MI	N/A[1]	
52704	54%	88%	Arlington, TX	15.5%	6%
52818	50%	75%	Arlington, TX		
52903	51%	77%	Fort Worth, TX	0%	0%

[1] No such phone number data from MobileApp

Table 6.5: Coverage and accuracy comparison

RNC's signature sample in MobileApp (termed as **baseline sample**), we go through all the probed signature samples (without knowing their ground truth RNCs), and compare them against the baseline sample. The signatures that match the baseline close enough are then included in M. Now the only unsolved part is $|G \bigcap M|$ — the number of correctly inferred signature samples, which is in fact semantically equivalent to $|G| \times coverage$ — out of all the signature samples that are under the target RNC, the covered ones are effectively the ones correctly inferred. Since we already know $coverage$, all we need is $|G|$ at this point. We can estimate $|G|$ based on the population distribution observed in our MobileApp. Specifically, if we know the fraction of users that are under a particular RNC out of all the users, then we can rewrite $|G|$ into $|N| \times p$ where $|N|$ is total number of signature samples in the GGSN, and p is the fraction of population that are under the target RNC. Therefore, $Accuracy = \frac{|G| \times coverage}{|M|} = \frac{|N| \times p \times coverage}{M}$. Since our dataset is not complete enough to calculate p that is representative, we resort to the uniform population distribution where every p is equal only for the purpose of evaluation.

Coverage and accuracy results. As described in §6.5, the SD-threshold for deciding if two samples are close enough can be tuned according to the desired attack impact and the resource constraint. The more relaxed the SD-threshold, the more samples are considered under the target RNC and thus likely to include more correct samples (and better coverage and higher attack impact). However, this is at the cost of lower accuracy (*i.e.,* other non-related signature samples will be mistakenly inferred to be under the RNC). As an example,

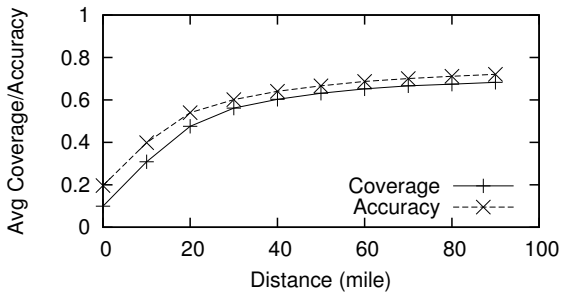


Figure 6.10: Phone-number-based coverage and accuracy in small cities

we tune the SD-threshold to achieve 50% coverage (which is equivalent to 2.5 to 3.5 times the normal load), the corresponding accuracy is shown in Table 6.5 for the 6 RNCs that are under the probed GGSN from the MobileApp data.

Here LAC 52704 and 52818 (Arlington) both belong to the "Dallas-Fort Worth-Arlington" metropolitan area that has a large population. The accuracy is high because their DL queue size threshold of 1010 is very unique. LAC 52903 (Fort Worth) also belongs to the same metropolitan area. However, its DL threshold is only 510 bytes. The reason why its accuracy is high is that its min-RTT is also very unique to distinguish it from most other LACs. LAC 26507 has a very bad accuracy because the promotion delay is unstable and requires a large threshold value to maintain the coverage rate. Further, its static features are also the most commonly observed ones, further decreasing the accuracy. From the attacker's perspective, he can also perform such signature distribution sampling offline regularly to estimate how distinct the target RNC's signature is before the attack. To further improve the accuracy, we discuss an optional step to further filter IPs that are in other RNCs (in §6.6). Finally, the last two columns in Table 6.5 are the results of the phone-number-based approach described in the next section.

6.5.3 Comparison with Phone-Number-Based Localization

In the state-of-the-art phone-number-based localization [64, 105], the first seven digits of the phone number (area code) are used to map out its corresponding physical location

(and carrier name) using the public NPA/NXX database [18]. The fundamental difference between signature- and phone-number-based approach is that phone number is a static property — users carry the same phone number to different places, even when they switch network providers. In contrast, network signature is dynamic and up-to-date, thus robust to user mobility. Based on the data we collected (described later), we find 75.4% of the carrier names returned from NPA/NXX database are incorrect, leading to extra effort for correction, assuming there is a way to do so.

Data & evaluation methodology. From our deployed MobileApp, with user's permission we collected 15,395 entries consisting of first 7 digits of phone number, carrier name, GPS location from users all across the U.S. as ground truth. We use the data to calculate coverage and accuracy and compare them directly with the signature-based approach. Ideally we want to map each GPS location to its RNC and do an RNC-level comparison on the coverage and accuracy. However, the NPA/NXX database returns only city-level location. Thus, we approximate the RNC-level comparison through city-level comparison instead. To begin, we map each GPS location to its corresponding city via a city-to-GPS database, and map the phone number prefix to its city via the NPA/NXX database. We term them *GPS-city* and *queried-city* respectively.

We treat big cities and small cities differently. For each big city big, which is defined to have multiple RNCs (or LACs from OpenCellID), we compute its city-level coverage as $\frac{X}{Y1}$ where X is the number of cases where *GPS-city* $=$ *queried-city* $= big$, and $Y1$ is the number of cases where *GPS-city* $= big$. Similarly, we compute accuracy as $\frac{X}{Y2}$ where X is the same as before, and $Y2$ is the number of cases where *queried-city* $= big$. Note that since a phone number can only map to a city-level location, its RNC-level coverage and accuracy need to be divided further by n — the number of RNCs in that city. The coverage and accuracy are then $\frac{X}{Y1\times n}$ and $\frac{X}{Y2\times n}$. Here we conservatively use $n = 2$ to account for any inaccuracy when mapping from LAC to RNC. In reality, n is likely bigger and makes their results even worse. For small cities that may share a single RNC with other cities, we

need a different method. Specifically, if the *GPS-city* and *queried-city* are within a certain physical distance, we consider them to be under the same RNC.

Coverage and accuracy results. For big cities, the average coverage and accuracy are 18.8% and 18.4% respectively for phone-number-based approach. Both values are much lower than the signature-based results in Table 6.5. In comparison with our signature-based localization, if we set the SD-threshold to allow the same accuracy, the coverage of the two big cities in Table 6.5 can increase to 60% – 80%, which is equivalent to 3 to 4 times the normal load. For small cities, Figure 6.10 illustrates how average coverage and accuracy over all cities change with different distance threshold. At 10 miles, which is our locally observed distance to observe two different RNCs in small-city area, the coverage and accuracy are about 30% and 40% respectively. At 30 miles, they are around 56% and 60%, indicating that the phone-number-based approach can perform better in the small city case as the signature-based approach has less distinct signatures in small cities.

As a more direct comparison, we look at the intersection between the phone-number-based results and signature-based results from MobileApp, arriving at these following cities: Arlington and Fort Worth. As we can see in Table 6.5, the phone-number-based coverage and accuracy are much worse than the the signature-based approach. For Arlington, $coverage = \frac{X}{Y1\times 2} = \frac{11}{46\times 2}$; $accuracy = \frac{X}{Y2\times 2} = \frac{11}{92\times 2}$. For Fort Worth, $coverage = \frac{X}{Y1\times 2} = \frac{0}{30\times 2}$; $accuracy = \frac{X}{Y2\times 2} = \frac{0}{3\times 2}$ ($X, Y1, Y2$ are defined earlier). Surprisingly for city "Fort Worth", both metrics are 0%, showing how ineffective phone-number-based approach can be. Other cities do not have enough phone number data points in MobileApp, thus are marked as N/A.

Other aspects.

Note that NPA/NXX database does not provide mappings from phone numbers to IPs. Therefore, the only way to attack the data channel using phone numbers is through MMS messages, as SMS uses control channel instead. To launch the attack, a large number of MMS messages must be sent regularly.

Category	Signature-based localization/IP traffic	Phone-number-based localization/MMS
Localization overhead	× Scanning large range of IPs	✓ Crawl database, correct carrier name
Localization accuracy	– Better in large cities	– Better in small cities
Technique extensibility	✓ More features to improve accuracy	× Difficult to extend
Stealthiness in actual attack	✓ Hard for users to notice	× Easy for users to notice or disable
Feedback in actual attack	✓ Immediate feedback	× No feedback
Timing control	✓ Easy to control	× Can get queued and delayed
Current defense effectiveness	– Can be hard to traverse NAT	– MMS blocked or rate limited
Attack Detectability	– Probing traffic can be detected	– MMS flooding can be detected

Figure 6.11: Comparison between signature-based and Phone-number-based attacks

Different from sending direct IP traffic, sending MMS has a number of short-comings (shown in Table 6.11). We illustrate a few of them: 1). Repeated MMS messages are easy for users to notice and disable or even report to their carriers as abuse. In contrast, sending direct IP traffic is much less noticeable to users. 2). We find that sending MMS through the email interface (*e.g.,* 123456789@mms.att.net) provides no feedback on the success of message delivery, thus potentially wasting significant effort by attacking phones that received no MMS at all (*e.g.,* no MMS capability). 3). The timing of MMS delivery is very unpredictable, making the attack coordination difficult. In our local experiment, we sent MMS messages repeatedly every 30 seconds over 4 hours at night, which presumably should incur little delay. In contrast, we observe significant delays, causing the phone 22.7% of the time to be idle and making the attack significantly less effective. In a real attack where the MMS gateway/server is more loaded, it is expected to observe even longer delays. In fact, since MMS still uses SMS to push the notification [21], it is likely that the control channel is overloaded first.

6.5.4 Attack Scalability Analysis

An important question is how fast the attacker can scan a sufficient number of IPs. If it takes too long, the collected IP address may change (*e.g.,* users may power off the phone). With a simple analysis, we estimate that it takes 1.2 hours to scan enough IPs with 800kbps of bandwidth. The duration can be significantly shortened using more attack machines and

bandwidth.

Input. The prefix ranges found to be allocated for a target location under a GGSN: X.128.X.X – X.147.X.X.(§6.4.1) are used as input. This corresponds to 1,300,500 IPs.

Min-RTT probing. The attacker can first use min-RTT measurements to filter irrelevant IPs as min-RTT has the smallest probing bandwidth requirement and the highest entropy. Although in total we need 30 – 40 probes (as described in §6.4.2.2) to determine the min-RTT, we need only 1 probe to first check if the IP is assigned or not. Probing one IP with one probe requires bandwidth of $\frac{40byte*8bit}{RTT} = 1kps$ (assuming $RTT = 0.3s$). Probing 800 IPs simultaneously requires $1kbps \times 800 = 800kbps$. Probing each IP once takes about $\frac{1{,}300{,}500}{800} \times 0.3s = 8min$.

Since many IPs are not allocated at run time, they can be dropped easily upon the first probe. Considering the online IP ratio of 22% (§6.3.3), around 286,110 IPs are left. Probing the remaining IPs with additional 29 back-to-back measurements will take $\frac{286{,}110}{800} \times (0.3s \times 29) = 51min$. In total, it takes roughly 59 minutes to scan the entire GGSN IP range.

Static feature probing. After min-RTT probing, on average $\frac{1}{8}$ of IPs (around 35,763) remain since the entropy of min-RTT is 3 bits. The attacker can then measure the static features to further narrow down IPs to probe. Given that the attacker knows the exact inactivity timer and the queue size threshold in the target location from the probe phone in that area, it can directly look for these values during scanning. Depending on the timer and queue size threshold, a complete round to measure both values takes a large packet (1200 bytes) to trigger state promotion to DCH, a small packet (40 bytes) to test the inactivity timer, and a packet with size matching the queue size threshold (suppose it is 500 bytes) to see if it forces the state to go back to DCH. The time to finish such a round would be the $promotion\ delay(2s) + inactivity\ timer(5s) = 7s$. The throughput requirement is then $\frac{(1200+40+500)*8bits}{7} = 1.94kbps$. Scanning 412 IPs simultaneously would require $1.94kbps \times 412 = 800kbps$ bandwidth and $\frac{35{,}763}{412} \times 7s = 10min$ to finish.

Promotion delay probing. After the static feature probing, there are on average $\frac{1}{5}$ IPs

(around 7,152) remaining for entropy of 2.3 bits. Then the promotion delay can be measured. It takes only 10.7bps uplink on the server side, since it only needs to send and receive one 40-byte packet every 30 second. It takes 4min to complete 8 rounds of probing (8 rounds is selected in §6.4.2.2). The attacker can easily target the rest simultaneously using $7,152 \times 10.7bps = 76kbps$ bandwidth. The estimated time to finish probing should be 4min since all of them can be done in parallel.

In total, the time to complete the measuring of all features is $59 + 10 + 4 = 73min$. Considering 80% of IPs keep their IPs for more than 4 hours (Figure 6.2), the attacker has enough time to launch the attack thereafter for at least several hours.

6.6 Discussion and Defense

Impact of device difference on dynamic features. Ideally, end-devices should have little impact on the dynamic features such as RTT and promotion delays. However, we do realize that different phones supporting different technologies may co-exist. For instance, some phones may support only HSDPA while their network supports both HSUPA and HSDPA (*i.e.,* HSPA). This means that these "older" phones may experience slower network performance (*e.g.,* longer RTT and promotion delay) thus exhibiting different network signatures. In practice, however, it is likely that there will always be one or two "dominating" technologies that are most widely used in the end-devices and can be considered as the target for an attacker. For instance, devices are frequently updated given that smartphones are becoming cheaper and quickly out-dated. Regardless, such phone differences do not impact the static features such as queue length threshold and inactivity timer.

Impact of base-station difference on dynamic features. Besides device differences, different base-stations can also impact the measured RTTs or promotion delays, with bounded noise. Indeed, from the MobileApp data, we find that in a short duration (*e.g.,* peak hours), the measured promotion delays do not vary significantly under the same RNC considering the samples are collected across multiple base-stations already. We do find one instance

where the promotion delay can vary more significantly in different base-stations. This means that we either have to relax the SD-threshold to cover enough signature samples, or need some way to account for it. As a preliminary investigation, we find that the difference could be caused by different base-stations under different load, indicated by significantly differing average RTTs (min-RTTs are still the same) which can be used for calibration. We leave for future work to systematically address this.

Optional final step to narrow down RNCs. In cases where the attacker wants to ensure that the inferred IPs indeed belong to the target RNC, an optional step that the attacker can take is to actually try these IPs out. More specifically, the attacker can choose a subset of the total IPs and launch a mini-attack against the RNCs (which should increase the processing delay on the RNC) and observe the changed baseline promotion delay from his probe phone. If the promotion delays of the tested IPs match the perturbed baseline promotion delay, it is likely that they indeed belong to the same RNC. The attacker can repeat this step multiple times to perform more fine-grained filtering.

Next generation LTE networks. As discussed earlier, LTE networks still have the built-in radio resource control mechanism with associated configurable parameters [11]. For instance, they allow various timers such as inactivity timer, discontinuous reception (DRX) cycle timer to be configured [11]. Fundamentally, due to the scarce nature of the radio resource, the corresponding radio resource control protocol has to provide mechanisms for sharing the resource smartly and flexibly among a large number of users. It is conceivable that network signatures exist for LTE networks as well since different network locations may again use a different set of configurations to accommodate the load difference.

Defense. The network signatures can be considered as a type of side channel that leaks information about the network location. A common defense is to eliminate such side channel. For instance, instead of using the fixed configuration parameter for all devices at all times, carriers can assign parameters probabilistically within a range, which makes it significantly harder for an attacker to infer the actual parameter. However, this may reduce the effec-

tiveness of the parameter for accommodating workload and still requires scrutinization. As network defenses, the carrier can disallow direct probing from the Internet or other phones. However, it also has negative impact as discussed. As host-based defenses, one simple solution is to use host firewall that drops unsolicited probing packets to prevent information leakage. This could be an effective solution in the short term. As smartphones evolve, however, they may open more ports to accept incoming connections, making host firewall protection less effective. We leave for future work to systematically study how to defend against such information leakage in critical infrastructures such as cellular networks.

6.7 Summary

This chapter has discussed how cellular devices can be located under specific RNCs that exhibit distinct signatures across different locations. This study helps validate the assumption in previous studies on targeted DoS attacks that assume they can create a hit-list of reachable mobile IP addresses associated with a target location. Our approach uses the timing introspective side channel to measure the parameters and signatures that are associated with RNCs. The technique is applicable for a large set of current cellular networks we studied. This exposes a new attack vector that cannot be easily eliminated. From a thorough empirical analysis, we offer suggestions on how to defend against such reconnaissance effort by attackers.

CHAPTER VII

Conclusion and Future Work

7.1 Contributions

The thesis introduces the notion of introspective side channels that can leak internal network and system state. It makes the following contributions, which help discover, analyze, and validate a number of new and existing attacks:

1. Off-path TCP sequence number inference attack enabled by firewall middleboxes. We discover two new introspective side channel that allows the attack: host packet counters and IPID from intermediate hops. We are the first to report the TCP sequence number inference attack using state kept on middleboxes and attacks built on it. We demonstrate that many networks and applications are affected today. We also provide insights on why they occur and how they can be mitigated.

2. Collaborated off-path TCP sequence number inference attack enabled by new host packet counters. We discover that even without the firewall middleboxes that store the TCP sequence number state, the very similar attack is still possible. The new introspective side channel that we discovered is again host packet counters. This time, however, unlike the previous counters, it is a specific counter that actually can directly leak information about the sequence number. We found that an unprivileged malware on the device can easily get access to such information and make use of the side channels.

3. Investigation of triangular spamming. Our work is the first to highlight the practice

of triangular spamming, a stealthy and fairly efficient spamming technique that can be relatively easily carried out on today's Internet by thoroughly investigating its feasibility in the presence of existing network policies and ease of operational deployment. We bring attention to the community that today's SMTP traffic blocking policies can be bypassed to carry out such spamming activities while protecting the identities of hosts actually sending out spam messages. Through a clever design of a probing technique that relies on known side channels, we are able to identify a large number of networks that only block SMTP traffic in the outgoing direction (*i.e.,* outgoing traffic with destination port 25), which are vulnerable to triangular spamming. The introspective side channel here is the IPID side channel from windows hosts.

4. Targeted DoS attack in cellular network. Motivated by the recent attacks against cellular network infrastructure at aparticular target region (*e.g.,* signaling attack), we focus on a simple but important question of how to create a hit-list of reachable mobile IP addresses associated with the target location to facilitate such targeted DoS attacks. Our approach uses network signatures through active probing and is applicable for a large set of current cellular networks we studied. The introspective side channel that we rely on is the timing channel. The result of this work exposes a new attack vector that cannot be easily eliminated. From a thorough empirical analysis, we offer suggestions on how to defend against such reconnaissance effort by attackers.

Note that these studies are all relying on introspective side channels that can leak information about the network and system state. We believe that these introspective side channels have a larger impact that can affect potentially many applications that are dependent on the lower layer state.

In summary, there are quite a few interesting lessons that we learned: 1) Seemingly harmless aggregated information can leak critical internal network/system state. For instance, besides packet counters, IPID side-channel has been shown to be another dangerous side-channel [5, 94]. 2). It is especially problematic if aggregated statistics do not have

enough entropy on its own (*DelayedACKLost*), which allows an attacker to manipulate it as a reliable information leakage channel. 3). Future system and network design should carefully evaluate what information the adversaries can have access to, especially through seemingly unexpected channels. 4). The design of TCP-level firewalls can be very tricky. The more advanced features there are, the more information that it can potentially leak.

On the defense side, we also summarize a number of ways to defend against similar attacks to the ones presented in the thesis: 1) Eliminate, obfuscate, or provide access control on the side channels. Depending on the specific side channels, it may or may not be possible to eliminate a side channel. For instance, IPID is something that can be eliminated. However, host packet counters are something useful for providing additional functionalities. When the side channel is timing, one could intentionally perturb the timing of events to prevent information from being leaked through timing channel. 2) Eliminate or protect or isolate the sensitive shared state. For instance, one can remove certain unnecessary redundant states (*e.g.,* the TCP sequence number state kept on the firewalls) to minimize the footprint. However, this would have functionality impact (*e.g.,* packet screening functionality based on sequence numbers). 3) Build systems that are resistent to certain state being leaked. For instance, even if TCP sequence numbers can be inferred, SSL can be employed to defend against any attacks on breaking the integrity of the communication.

7.2 Future Work

There are a number of natural future directions that are interesting for further investigation:

- **Future design to prevent side-channel-enabled attacks.** We have discussed on how to defend against the attacks on existing systems. However, it is equally critical to design new systems that are resistant by design to such side-channel-enabled attacks. For instance, one important design principle is to isolate as much state as pos-

sible. One design for the future system is that different source IP addresses are used for connections in different processes. This would disallow malware to know the IP address of a connection established by another process (which may become feasible when IPv6 is widely deployed). In the extreme case, a system can be designed such that each app runs in its own virtual machine environment. A more systemical study is required to understand what is the sweet spot in tradeoffs between the extent of isolation and the functionalities (easiness of sharing between applications).

- **Other information leakage through introspective side channels.** As networked systems become more complex, there are a large number of places where sharing happens. If an attacker can get access to the certain shared information with low profile, he may be able to use that as side channels to infer other information with high profile. It is especially interesting to see how information such as the TCP sequence number state is leaked through seemingly harmless aggregated statistics such as the packet counters. It is interesting to check if there are other such aggregated information that can be leveraged to infer sensitive information.

- **Smartphone security.** As smartphones become more and more popular, privacy and information leakage have also been more concerning. The threat model of the smartphone platforms is that users may download potentially malicious third-party apps that are typically sandboxed. Such malicious app can potentially get access to information such as location and contact list, but it should not be able to tamper with other apps on the phone (*e.g.,* the browser app). Through introspective side channels, the malware can, in some sense, "break" the sandbox and compromise the security of other apps.

- **Cloud security.** Cloud's shared infrastructure presents a similar challenge information leakage. For instance, it is interesting to explore side channels that allow a malicious tenant to infer information about another tenant. In particular, the threat

model that a malicious tenant can utilize resources from not only within the cloud (internal) but also outside of the cloud (external), resembles the threat model of the malware-enabled TCP attacks. In such a threat model, an attacker can not only passively monitor or infer information, but also actively perturb certain state and observe the change.

- **Large-scale infrastructure security.** Today's networks such as cellular networks have significant infrastructure deployed in order to support a large number of users. At the same time, they also keep significant amount of state information that can potentially be exposed to attackers. There are much work can be done to understand how such information can be leaked to attackers and what the security implications are.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Adobe flash player. http://www.adobe.com/products/flashplayer/.

[2] The apache spamassassin project. http://spamassassin.apache.org/.

[3] AT&T: 80 percent of network now upgraded to HSPA+. http://www.engadget.com/2010/11/17/atandt-80-percent-of-network-now-upgraded-to-hspa/.

[4] AT&T: wireless capacity upgrades complete in Hillsborough, Pasco. http://www.bizjournals.com/tampabay/stories/2010/10/11/daily38.html.

[5] Blind TCP/IP hijacking is still alive. http://www.phrack.org/issues.php?issue=64&id=15.

[6] CERT Advisory CA-2001-09 Statistical Weaknesses in TCP/IP Initial Sequence Numbers. http://www.cert.org/advisories/CA-2001-09.html.

[7] Chater cable, internet, telephone. www.charter.net/.

[8] Cisco pix 515e firewall quick start guide, version 6.3. http://www.ciscosystems.org.ro/en/US/docs/security/pix/pix63/quick/guide/63_515qk.html.

[9] Cisco Security Advisory: Cisco Secure PIX Firewall TCP Reset Vulnerability. http://www.cisco.com/en/US/products/products_security_advisory09186a00800b1397.shtml.

[10] Comcast takes hard line against spam. http://news.zdnet.com/2100-3513_22-136518.html.

[11] Evolved universal terrestrial radio access (e-utra); radio resource control (rrc); protocol specification. http://www.3gpp.org/ftp/Specs/html-info/36331.htm.

[12] Golomb ruler. http://en.wikipedia.org/wiki/Golomb_ruler.

[13] Hacking the interwebs. http://www.gnucitizen.org/blog/hacking-the-interwebs/.

[14] ICSI Netalyzr. http://netalyzr.icsi.berkeley.edu/.

[15] iPhone Troubles Might Go Beyond Signaling. http://www.lightreading.com/document.asp?doc_id=190764&piddl_msgid=226225#msg_226225.

[16] Linux Blind TCP Spoofing Vulnerability. http://www.securityfocus.com/bid/580/info.

[17] Linux: TCP Random Initial Sequence Numbers. http://kerneltrap.org/node/4654.

[18] Local Calling Guide. http://www.localcallingguide.com/index.php.

[19] Mobility management. http://en.wikipedia.org/wiki/Mobility_management.

[20] MSN Messenger Protocol. http://www.hypothetic.org/docs/msn/.

[21] Multimedia Messaging Service. http://en.wikipedia.org/wiki/Multimedia_Messaging_Service.

[22] Nanog survey - isp port blocking practice. http://seclists.org/nanog/2009/Oct/727.

[23] netfilter/iptables project homepage. http://www.netfilter.org/.

[24] OpenCellID. http://www.opencellid.org/.

[25] p0f, a Versatile OS Fingerprinting Tool. http://lcamtuf.coredump.cx/p0f.shtml.

[26] Planetlab acceptable use policy (aup). http://www.planet-lab.org/aup.

[27] Port 25 blocking. http://www.postcastserver.com/help/Port_25_Blocking.aspx.

[28] Port 25 (sonic.net). http://sonic.net/support/faq/advanced/port_25.shtml.

[29] RFC 4953 - Defending TCP Against Spoofing Attacks. http://tools.ietf.org/html/rfc4953.

[30] RFC 5382 - NAT Behavioral Requirements for TCP. http://tools.ietf.org/html/rfc5382.

[31] RFC 5961 - Improving TCP's Robustness to Blind In-Window Attacks. http://tools.ietf.org/html/rfc5961.

[32] RFC 793 - TRANSMISSION CONTROL PROTOCOL. http://tools.ietf.org/html/rfc793.

[33] Rfc2920, smtp service extension for command pipelining. http://tools.ietf.org/html/rfc2920.

[34] Sbc email problem. www.sfsu.edu/~helpdesk/sbc/index.htm/.

[35] Spamcop. http://www.spamcop.net/.

[36] Spamhaus. http://www.spamhaus.org/.

[37] Stateful Firewall and Masquerading on Linux. http://www.puschitz.com/FirewallAndRouters.shtml.

[38] sysctl Mac OS X Manual. https://developer.apple.com/library/mac/#documentation/Darwin/Reference/Manpages/man3/sysctl.3.html#//apple_ref/doc/man/3/sysctl.

[39] TCP Delayed Ack in Linux. http://wiki.hsc.com/wiki/Main/InsideLinuxTCPDelayedAck.

[40] Tcp/ip fingerprinting methods supported by nmap. http://nmap.org/book/osdetect-methods.html.

[41] Unicast reverse path forwarding. http://en.wikipedia.org/wiki/Reverse_path_forwarding.

[42] Universal plug and play. http://en.wikipedia.org/wiki/Universal_Plug_and_Play/.

[43] Useful statistics for web designers. http://www.tvidesign.co.uk/blog/useful-statistics-for-web-designers.aspx.

[44] Verizon - our attention needed: Re-configure your email settings to send email. http://www22.verizon.com/ResidentialHelp/HighSpeed/General+Support/Top+Questions/QuestionsOne/124274.htm.

[45] Whois ip address/domain name lookup. http://cqcounter.com/whois/.

[46] Comet (programming). http://en.wikipedia.org/wiki/Comet_(programming), Retrieved on 03/04/2012.

[47] TCP hijacking video demo. http://youtu.be/T65lQtgUJ2Y, Retrieved on 03/04/2012.

[48] Access Point Name. http://en.wikipedia.org/wiki/Access_Point_Name, Retrived on 03/04/2012.

[49] 3rd Generation Partnership Project. 3GPP TS 25.133: “Requirements for support of radio resource management (FDD)”.

[50] 3rd Generation Partnership Project. 3GPP TS 25.331: "RRC Protocol Specification".

[51] P. Bahl and V. N. Padmanabhan. Radar: an in-building rf-based user location and tracking system. In *Proc. Int. Conf. Computer Communications*, 2000.

[52] F. Baker and P. Savola. Ingress filtering for multihomed networks. In *RFC 3704*, 2004.

[53] Mahesh Balakrishnan, Iqbal Mohomed, and Venugopalan Ramasubramanian. Where's That Phone?: Geolocating IP Addresses on 3G Networks. In *Proceedings of IMC*, 2009.

[54] Nilanjan Banerjee, Sharad Agarwal, Paramvir Bahl, Ranveer Chandra, Alec Wolman, , and Mark Corner. Virtual Compass: relative positioning to sense mobile social interactions. In *IEEE Pervasive Computing*, 2010.

[55] Ethan K. Bassett, Harsha V. Madhyastha, Vijay K. Adhikari, Colin Scott, Justine Sherry, Peter Van Wesep, Thomas Anderson, and Arvind Krishnamurthy. Reverse traceroute. In *Proc. of NSDI*, 2010.

[56] Robert Beverly, Arthur Berger, Young Hyun, and k claffy. Understanding the efficacy of deployed internet source address validation filtering. In *In Proc. of IMC*, 2009.

[57] Robert Beverly, Arthur Berger, Young Hyun, and k claffy. Understanding the Efficacy of Deployed Internet Source Address Validation Filtering. In *Proc. Internet Measurement Conference*, 2009.

[58] R. Bonica, D. Gan, D. Tappan, and C. Pignataro. Extended ICMP to Support Multi-Part Messages. `http://tools.ietf.org/html/rfc4884`.

[59] Andrew Bortz and Dan Boneh. Exposing private information by timing web applications. In *WWW*, 2007.

[60] Shuo Chen, Rui Wang, Xiaofeng Wang, and Kehuan Zhang. Side-channel leaks in web applications: a reality today, a challenge tomorrow. In *Proc. of IEEE Security and Privacy*, 2010.

[61] Y. Chen, Y. Chawathe, A. LaMarca, and J. Krumm. Accuracy characterization for metropolitan-scale wi-fi localization. In *Proc. Int. Conf. Mobile Systems, Applications And Services*, 2005.

[62] Cisco. Cisco ASA 5500 Series Configuration Guide using the CLI, 8.2. `http://www.cisco.com/en/US/docs/security/asa/asa82/configuration/guide/conns_tcpnorm.html`.

[63] Ionut Constandache, Xuan Bao, Martin Azizyan, and Romit Roy Choudhury. Did You See Bob?: Human Localization using Mobile Phones. In *Proc. Int. Conf. Mobile Computing and Networking*, 2010.

[64] William Enck, Patrick Traynor, Patrick McDaniel, and Thomas La Porta. Exploiting open functionality in SMS-capable cellular networks. In *Proceedings of the 12th ACM conference on Computer and communications security*, CCS, 2005.

[65] Roya Ensafi, Jong Chun Park, Deepak Kapur, and Jedidiah R. Crandall. Idle port scanning and non-interference analysis of network protocol stacks using model checking. In *Proc. of USENIX Security Symposium*, 2010.

[66] Laura Falk, Atul Prakash, and Kevin Borders. Analyzing websites for user-visible security design flaws. In *Proc. of Usable privacy and security*, 2008.

[67] Adrienne Porter Felt, Helen J. Wang, Alexander Moshchuk, Steven Hanna, and Erika Chin. Permission re-delegation: attacks and defenses. In *Proc. of USENIX Security Symposium*, 2011.

[68] P. Ferguson and D. Senie. Network ingress filtering: Defeating denial of service attacks which employ ip source address spoofing. In *RFC 2827*, 2000.

[69] H. Galeana, R. Ferrus, and J. Olmos. Transport capacity estimations for over-provisioned utran ip-based networks. In *WCNC*, 2007.

[70] F. Gont and S. Bellovin. Defending Against Sequence Number Attacks. RFC 6528, 2012.

[71] Marco Gruteser, Dirk Grunwalddepartment, and Computer Science. Anonymous usage of location-based services through spatial and temporal cloaking. In *Proc. of Mobisys*, 2003.

[72] B. Gueye, A. Ziviani, M. Crovella, and S. Fdida. Constraint-based Geolocation of Internet Hosts. 2004.

[73] Saikat Guha and Paul Francis. Characterization and Measurement of TCP Traversal through NATs and Firewalls. In *Proc. Internet Measurement Conference*, 2005.

[74] Harri Holma and Antti Toskala. HSDPA/HSUPA for UMTS: High Speed Radio Access for Mobile Communications. John Wiley and Sons, Inc., 2006.

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

[76] Laurent Joncheray. A simple active attack against tcp. In *Proc. of USENIX Security Symposium*, 1995.

[77] Juniper. Stateful Inspection Firewalls. `http://www.abchost.sk/download/204-4/juniper-stateful-inspection-firewall.pdf`.

[78] Heikki Kaaranen, Ari Ahtiainen, Lauri Laitinen, Siamak Naghian, and Valtteri Niemi. UMTS Networks: Architecture, Mobility and Services (2nd Edition). John Wiley and Sons, Inc., 2005.

[79] Seny Kamara, Sonia Fahmy, Eugene Schultz, Florian Kerschbaum, and Michael Frantzen. Analysis of Vulnerabilities in Internet Firewalls. In *"Computers & Security"*, 2003.

[80] Patrick P. C. Lee, Tian Bu, and Thomas Woo. On the Detection of Signaling DoS Attacks on 3G Wireless Networks. In *Proc. Int. Conf. Computer Communications*, 2007.

[81] G. LEECH, P. RAYSON, and A. WILSON. Procfs Analysis. `http://www.nsa.gov/research/_files/selinux/papers/slinux/node57.shtml`.

[82] Z. Morley Mao, Lili Qiu, Jia Wang, and Yin Zhang. On as-level path inference. In *In Proc. of SIGMETRICS*, 2005.

[83] Olivier De Mey, Laurent Schumacher, and Xavier Dubois. Optimum Number of RLC Retransmissions for Best TCP Performance in UTRAN. In *Proceedings of Annual IEEE International Symposium on Personal Indoor and Mobile Radio Communications (PIMRC)*, 2005.

[84] Neil Spring and Ratul Mahajan and David Wetherall. Measuring ISP Topologies with Rocketfuel. In *Proc. ACM SIGCOMM*, 2002.

[85] D. Niculescu and B. Nath. VOR base stations for indoor 802.11 positioning. In *Proc. Int. Conf. Mobile Computing and Networking*, 2004.

[86] Tom Olzak. Dns cache poisoning, 2006.

[87] P. Ferguson and D. Senie. Network ingress filtering: Defeating denial of service attacks which employ ip source address spoofing. IETF RFC 2267.

[88] V.N. Padmanabhan and L. Subramanian. An Investigation of Geographic Mapping Techniques for Internet Hosts. 2001.

[89] Check Point. What's New for FireWall-1/TCP Sequence Checking. `http://www.checkpoint.com/ngupgrade/whatsnew/products/features/tcpseqcheck.html`.

[90] Feng Qian, Zhaoguang Wang, Alexandre Gerber, Z. Morley Mao, Subhabrata Sen, and Oliver Spatscheck. Characterizing Radio Resource Allocation for 3G Networks. In *Proc. Internet Measurement Conference*, 2010.

[91] Feng Qian, Zhaoguang Wang, Alexandre Gerber, Z. Morley Mao, Subhabrata Sen, and Oliver Spatscheck. TOP: Tail Optimization Protocol for Cellular Radio Resource Allocation. In *Proceedings of the ICNP*, 2010.

[92] Feng Qian, Zhaoguang Wang, Alexandre Gerber, Zhuoqing Morley Mao, Subhabrata Sen, and Oliver Spatscheck. Characterizing radio resource allocation for 3G networks. In *Proc. Internet Measurement Conference*, 2010.

[93] Zhiyun Qian and Z. Morley Mao. Off-Path TCP Sequence Number Inference Attack – How Firewall Middleboxes Reduce Security. In *Proc. of IEEE Security and Privacy*, 2012.

[94] Zhiyun Qian and Z. Morley Mao. Off-path tcp sequence number inference attack – how firewall middleboxes reduce security. In *Proc. of IEEE Security and Privacy*, 2012.

[95] Zhiyun Qian, Z. Morley Mao, Yinglian Xie, and Fang Yu. Investigation of Triangular Spamming: A Stealthy and Efficient Spamming Technique. In *Proc. of IEEE Security and Privacy*, 2010.

[96] Zhiyun Qian, Z. Morley Mao, Yinglian Xie, and Fang Yu. Investigation of triangular spamming: a stealthy and efficient spamming technique. In *In Proc. of IEEE Security and Privacy*, 2010.

[97] Zhiyun Qian, Zhaoguang Wang, Qiang Xu, Z. Morley Mao, Ming Zhang, and Yi-Min Wang. You Can Run, but You Can't Hide: Exposing Network Location for Targeted DoS Attacks in 3G Networks. In *Proc. of NDSS*, 2010.

[98] Taghrid Samak, Adel El-Atawy, and Ehab Al-Shaer. Firecracker: A framework for inferring firewall policy using smart probing. In *In the Proceedings of the fifteenth IEEE International Conference on Network Protocols (ICNP)*, 2007.

[99] Jeremy Serror, Hui Zang, and Jean C. Bolot. Impact of Paging Channel Overloads or Attacks on a Cellular Network. In *WiSe*, 2006.

[100] Sushant Sinha, Michael Bailey, and Farnam Jahanian. Shades of Grey: On the Effectiveness of Reputation-based "Blacklists". In *Malware 2008*, 2008.

[101] Dawn Xiaodong Song, David Wagner, and Xuqing Tian. Timing analysis of keystrokes and timing attacks on ssh. In *Proc. of USENIX Security Symposium*, 2001.

[102] Spamhaus policy block list (PBL). `http://www.spamhaus.org/pbl/`, Jan 2007.

[103] Brett Stone-Gross, Marco Cova, Lorenzo Cavallaro, Bob Gilbert, Martin Szydlowski, Richard Kemmerer, Christopher Kruegel, and Giovanni Vigna. Your botnet is my botnet: Analysis of a botnet takeover. In *In Proc. of CCS*, 2009.

[104] Patrick Traynor, Michael Lin, Machigar Ongtang, Vikhyath Rao, Trent Jaeger, Patrick McDaniel, and Thomas La Porta. On Cellular Botnets: Measuring the Impact of Malicious Devices on a Cellular Network Core. In *Proceedings of the 16th ACM conference on Computer and communications security*, CCS, 2009.

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

[106] Shobha Venkataraman, Subhabrata Sen, Oliver Spatscheck, Patrick Haffner, and Dawn Song. Exploiting network structure for proactive spam mitigation. In *In Proc. of USENIX Security Symposium*, 2007.

[107] Vodafone, RIM, Huawei, and AT&T. UE "Fast Dormancy" behavior. 3GPP discussion and decision notes R2-075251, 2007.

[108] Martin Vuagnoux and Sylvain Pasini. Compromising electromagnetic emanations of wired and wireless keyboards. In *Proc. of USENIX Security Symposium*, 2009.

[109] Guohui Wang, Bo Zhang, and T. S. Eugene Ng. Towards network triangle inequality violation aware distributed systems. In *In Proc. of IMC*, 2007.

[110] Haining Wang, Cheng Jin, and Kang G. Shin. Defense against spoofed ip traffic using hop-count filtering. *IEEE/ACM Trans. Netw.*, 2007.

[111] Zhaoguang Wang, Zhiyun Qian, Qiang Xu, Z. Morley Mao, and Ming Zhang. An Untold Stody of Middleboxes in Cellular Networks. In *SIGCOMM*, 2011.

[112] Paul A. Watson. Slipping in the window: Tcp reset attacks. In *CanSecWest*, 2004.

[113] Nicholas Weaver, Robin Sommer, and Vern Paxson. Detecting forged tcp reset packets. In *In Proc. of NDSS*, 2009.

[114] Qiang Xu, Junxian Huang, Zhaoguang Wang, Feng Qian, Alexandre Gerber, and Zhuoqing Morley Mao. Cellular data network infrastructure characterization and implication on mobile content placement. In Proc. of SIGMETRICS, 2011.

[115] Kehuan Zhang and Xiaofeng Wang. Peeping tom in the neighborhood: Keystroke eavesdropping on multi-user systems. In *Proc. of USENIX Security Symposium*, 2009.