Private IP Address Inference in NAT Networks via Off-Path TCP Control-Plane Attack

Suraj Sharma[1][0009-0006-4841-4204], Adityavir Singh[1][0009-0007-6276-817X], and Mahabir Prasad Jhanwar[1][0000-0002-9753-2999]

Ashoka University, Sonipat, Haryana, India
{suraj.sharma, adityavir.singh_phd22, mahavir.jhawar}@ashoka.edu.in

Abstract. NAT is widely assumed to conceal the private IP addresses of internal clients from both off-path attackers within the same LAN and external observers. We show that this assumption does not hold. We present a novel off-path private IP address inference attack against NATed clients engaged in active TCP connections with a remote server. Our attack exploits known NAT behaviors in real-world routers, such as *port preservation*, insufficient *reverse-path validation*, and the absence of *TCP window tracking*, previously shown to enable off-path TCP hijacking. By inferring a NATed client's private IP address, our attack defeats the address-hiding property commonly attributed to NAT, creating a privacy leak that can lead to *deanonymization* and user profiling. We validate the attack on 6 NAT routers and 2 real-world Wi-Fi networks on SSH and HTTP/HTTPS traffic, finding that 5 of the 6 routers and both networks are vulnerable.

Keywords: TCP hijacking · Network Address Translation · off-path attacks · client deanonymization

1 Introduction

Network address translation (NAT) [18,40] has become a ubiquitous and essential feature of modern network deployments. Originally introduced to mitigate the exhaustion of the IPv4 address space, it now underlies most home and enterprise Wi-Fi networks, cellular (4G/5G) infrastructure, cloud platforms, and CDNs, allowing multiple internal clients with private IP addresses to share one or a few globally routable public IP addresses to access the Internet. In addition to conserving IP address space, NAT is often perceived as providing a degree of privacy by concealing internal network structures and private IP addresses from external observers and off-path attackers (who cannot eavesdrop on traffic exchanged between two endpoints) on the same LAN [17, 18, 32, 39].

TCP is the most widely deployed transport layer protocol and underpins a vast majority of today's Internet applications. It is a stateful protocol; endpoints maintain per-connection state, including sequence and acknowledgment numbers, to ensure reliable and ordered data delivery. This reliance on shared connection state has long made TCP an attractive target for hijacking attacks [3, 5–7].

In the traditional off-path TCP hijacking model, both endpoints are assumed to have public IP addresses. The off-path attacker is assumed to know the IP addresses of both endpoints and the server port. The attacker's objective is then to infer connection-specific parameters, including the client's ephemeral source port and the current sequence and acknowledgment numbers. Armed with this information, the attacker can inject forged TCP segments to hijack the victim's TCP connection, i.e., to terminate the connection or poison the plaintext TCP streams. Decades of research and successive rounds of protocol hardening, including source port randomization [24], random initial sequence number generation [15], and TCP challenge-ACK mechanism [42], have substantially raised the bar for executing such attacks.

While NAT mitigates IP address exhaustion and conceals network topology, its shared resource allocation for outbound connections introduces additional attack surfaces [13, 20]. In particular, TCP hijacking in NAT networks exposes attack vectors that differ from those in the traditional off-path TCP hijacking model. In this setting, a client behind a NAT gateway maintains a TCP connection with a remote server. Since the NAT gateway performs source IP address and source port translation for outgoing connections, an off-path attacker can still perform TCP hijacking attacks against NATed clients. To do so, the attacker requires the NAT gateway's public IP address, the translated source port, the target server's IP address and port, and the current sequence and acknowledgment numbers of the victim's TCP connection.

Yang et al. [46] demonstrated that widely deployed NAT behaviors, namely, *port preservation*, insufficient *reverse-path validation*, and the absence of *TCP window tracking*, enable an off-path attacker within the same LAN to recover these parameters and hijack active TCP connections, assuming that one or more internal clients maintain active TCP connections with a target remote server. Their large-scale empirical study of consumer- and enterprise-grade routers, covering 67 router models from 30 vendors and 93 operational Wi-Fi networks, found that 77% of routers and approximately 81% of networks satisfied all three NAT behaviors simultaneously. *Port preservation* is maintained to support NAT traversal and legacy applications that expect predictable external port mappings [17]. *TCP window tracking* is relaxed to reduce processing overhead on constrained gateway hardware and to maintain compatibility with non-standard retransmissions. *Reverse-path validation* is frequently disabled or relaxed to accommodate asymmetric routing in multi-homed and enterprise networks [2].

In this work, we show that the implications of these NAT behaviors extend well beyond TCP hijacking. We ask whether these NAT behaviors also undermine NAT's privacy properties:

Can an attacker exploit these behaviors to infer the private IP addresses of internal clients communicating with a target remote server over TCP, thereby compromising their privacy?

We answer this question in the affirmative. We show that an off-path attacker within the same LAN can exploit these NAT behaviors to infer the private

IP addresses of internal clients participating in active TCP connections, and can lead to *deanonymization* of clients behind the NAT. Private IP address inference can enable fine-grained user profiling by linking private IP addresses to users and tracking the remote services they access over time (see §5.2). We evaluate our attack against 6 commercially available NAT routers in a controlled environment and 2 real-world Wi-Fi networks. 5 of the 6 routers and both real-world networks are vulnerable, demonstrating that the attack is practical against widely deployed NAT implementations.

Contributions. Our main contributions are as follows:

- We devise a novel off-path private IP address inference attack against NATed clients communicating with a target remote server, revealing a previously unrecognized weakness in NAT's privacy assumptions.
- We evaluate 6 commercial routers for NAT behaviors that enable our attack, finding 5 of them to be vulnerable.
- We demonstrate the practicality of our attack in 2 real-world Wi-Fi networks by successfully inferring the private IP addresses of clients communicating over SSH and HTTPS in both networks. This capability can enable *deanonymization* and fine-grained user profiling.
- The proof-of-concept implementation of the attack is at: https://anonymous.4open.science/r/private-ip-address-inference-in-nat-5D9B.

The remainder of the paper is organized as follows: Section §2 covers necessary background relevant to our attack; Section §3 presents the threat model; Section §4 describes the attack procedure; Section §5 presents its evaluation, followed by discussion and countermeasures in Section §6; Section §7 surveys related work, and Section §8 concludes the paper.

2 Preliminaries

In this section, we present the network and protocol preliminaries relevant to our attack. We first provide an overview of TCP, followed by the segment notation used throughout the paper. We then formally describe the TCP state, segment validation conditions, challenge-ACK mechanism. Afterwards, we discuss the relevant NAT mechanisms, including port allocation, *TCP window tracking*, and *reverse-path validation*. Finally, we formally define traditional off-path TCP hijacking and extend the formulation to TCP hijacking in the presence of NAT.

2.1 TCP overview

TCP is a stateful transport layer protocol [9,29]. Each endpoint maintains a comprehensive per-connection state, implemented as a Transmission Control Block (TCB) [9]. This connection state evolves coherently across peers through validated TCP segment exchanges. TCP headers encode the information necessary to drive these state transitions; the segment structure is illustrated in Fig. 1, with the fields relevant to our attack highlighted in gray.

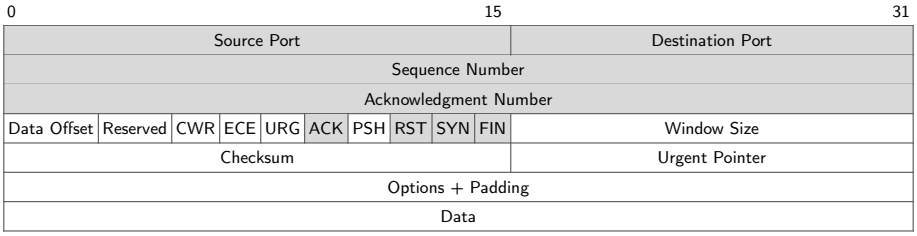

Fig. 1. TCP segment structure

Upon receiving a TCP segment, an endpoint first demultiplexes it to the correct TCB among potentially many active connections, indexed by the 4-tuple: (`Src IP`, `Src Port`, `Dest IP`, `Dest Port`), where `Src IP` and `Src Port` denote the client IP address and port, respectively, and `Dest IP` and `Dest Port` denote the server IP address and port, which uniquely identifies a TCP connection. Operationally, three classes of TCP header fields are central to state management. First, the 4-tuple identifies the relevant connection (`Src IP, Dest IP` in IP header). Second, the control flags - most notably `SYN`, `ACK` - indicate whether a segment participates in connection establishment or steady-state communication, while `RST`, `FIN` participate in connection termination. Third, TCP uses a 32-bit sequence number to identify bytes in the transmitted byte stream. The sequence number carried by a segment denotes the sequence number of its first data byte, with subsequent bytes assigned monotonically increasing sequence numbers modulo 2^{32}. The receiver uses a 32-bit cumulative acknowledgment number to indicate the next sequence number it expects to receive. Fig. 3 illustrates this byte-level numbering and acknowledgment semantics. In Fig. 3, the sender transmits a three byte segment beginning with sequence number 500, thereby carrying bytes 500 through 502. The receiver responds with an acknowledgment number of 503, cumulatively acknowledging all bytes through sequence number 502 and indicating that byte 503 is expected next. In the following, we capture the relevant fields of the TCP header and describe a compact segment notation used throughout the attack description.

Segment Notation. A TCP segment is represented as follows:

$$\mathsf{Type}^{\mathsf{FLAG}}_{\mathsf{srcIP}\to\mathsf{dstIP}}\left(\begin{array}{|l|l|}\hline \texttt{srcPr: s} & \texttt{desPr: d} \\ \hline \texttt{seqNo: x} & \texttt{ackNo: a} \\ \hline \end{array}\right),$$

where srcIP and dstIP denote the source and destination IP addresses, respectively; FLAG specifies the TCP control flags carried by the segment; s and d denote the TCP source and destination port numbers, respectively; and x and a denote the TCP sequence number and acknowledgment number, respectively. When a field is not applicable, it is denoted by "–". When random values are required, we write $z \xleftarrow{\$} \{0,1\}^{32}$ to denote uniform sampling from the 32-bit sequence number space. When Type = Legit, it denotes a segment whose source IP

address is the actual IP address of the sender. When $\mathsf{Type} = \mathsf{Spoof}$, it denotes a segment whose source IP address is spoofed and does not correspond to the true sender.

TCP State. The TCB corresponding to a connection between two endpoints maintains its identifying 4-tuple and the evolving protocol state. All incoming segments are validated against the corresponding TCB before any state update is performed. We model the full TCP state at an endpoint as the tuple: $S = (C, S_s, S_r, S_c, S_t, S_p)$. The substates S_c, S_t, S_p capture congestion-control variables, timer state, and negotiated connection parameters, respectively. These components are not required for our analysis. Hence, we restrict the TCP state to (C, S_s, S_r).

1. The substate C records the current phase of the connection. During connection establishment, C takes one of $\{\texttt{LISTEN}, \texttt{SYN-SENT}, \texttt{SYN-RECEIVED}\}$; during data transfer, $C = \texttt{ESTABLISHED}$; and during connection termination, C takes one of {`FIN-WAIT-1`, `FIN-WAIT-2`, `CLOSE-WAIT`, `CLOSING`, `LAST-ACK`, `TIME-WAIT`, `CLOSED`} [9].
2. **Send-sequence substate** S_s: It tracks the sequence numbers of data bytes sent to the peer. S_s = (`ISS`, `SND.UNA`, `SND.NXT`, `SND.WND`), where `ISS` is the initial send sequence number; `SND.UNA` is the sequence number of first unacknowledged byte; `SND.NXT` is the next sequence number to be sent and `SND.WND` is the peer-advertised send window. Incoming segments' acknowledgment numbers are matched against S_s to determine the delivery status of transmitted bytes.
3. **Receive-sequence substate** S_r: It tracks the sequence numbers of data bytes expected from the peer. S_r = (`IRS`, `RCV.NXT`, `RCV.WND`), where `IRS` is the initial receive sequence number; `RCV.NXT` is the next expected sequence number and `RCV.WND` is the advertised receive window. Incoming segments' sequence numbers are matched against S_r to enforce in-order and window-constrained delivery.

Fig. 3 illustrates the evolution of S_s and S_r across a segment exchange.

TCP Connection Establishment and Termination. TCP connection state is established via a three-way handshake, as shown in Fig. 2. A graceful TCP connection termination is a four-way exchange of `FIN`/`ACK` segments. One endpoint sends `FIN` and the peer replies with `ACK`; later, the peer sends its own `FIN`, which is acknowledged in return. Each `FIN` consumes one sequence number and the connection reaches `CLOSED` only after both directions are acknowledged.

2.2 TCP Segment Validation and State Update

TCP processes each incoming segment through a set of guarded predicates that determine whether the segment is consistent with the current state S. Formally,

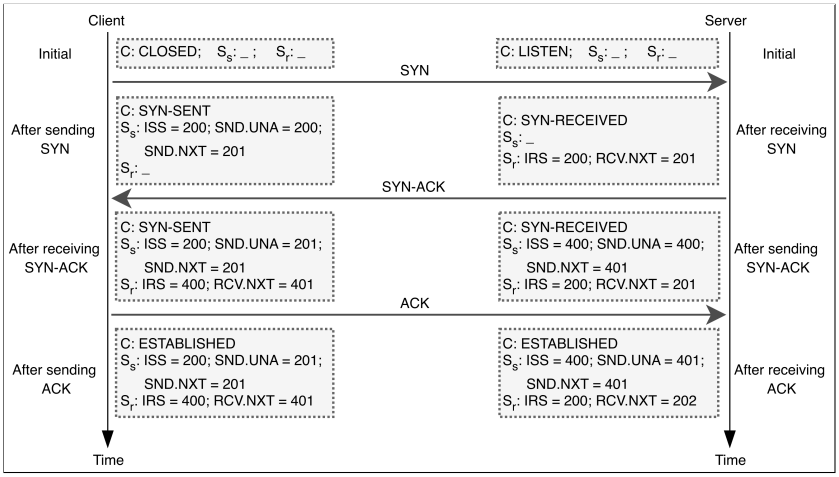

Fig. 2. TCP three-way handshake showing state evolution at the client and server

each predicate is a Boolean function $G_{\mathsf{pred}} : (S, \mathsf{SEG}_{\text{in}}) \longrightarrow \{0, 1\}$, evaluating a pred based on the incoming TCP segment (SEG_{in}) and using some substate of S. Let G_C, G_{S_s} and G_{S_r} be the predicates that validate the control, send-sequence and receive-sequence substates, respectively. A state transition occurs only if all the relevant predicates evaluate to 1.

1. G_C**:** It validates the control flags (`SYN`, `ACK`, `FIN`, `RST`, etc.) in an incoming segment against the control substate C. It evaluates to 1 when the control flags are semantically meaningful with respect to the current phase of the connection. For e.g., $G_c(S, \mathsf{SEG}_{\text{in}}) = 0$ if $(S.C = \texttt{ESTABLISHED}) \wedge (\mathsf{SEG}_{\text{in}}.flag = \texttt{SYN})$ or $(S.C = \texttt{CLOSED}) \wedge (\mathsf{SEG}_{\text{in}}.flag = \texttt{FIN})$.
2. G_{S_s}**:** It validates the acknowledgment number in an incoming segment against the substate S_s. RFC 9293 [9] specifies that $G_{S_s}(S, \mathsf{SEG}_{\text{in}}) = 1$ if:

$$\texttt{SND.UNA} < \mathsf{SEG}_{\text{in}}.\mathsf{ACK} \leq \texttt{SND.NXT} \tag{1}$$

   The interval [`SND.UNA`, `SND.NXT`) represents the data that has been transmitted but remains unacknowledged. An acknowledgment number satisfying Eq. (1) acknowledges previously transmitted data and can advance `SND.UNA`. The sender's outstanding data is constrained by the peer-advertised send window, such that $\texttt{SND.NXT} - \texttt{SND.UNA} \leq \texttt{SND.WND}$. As `SND.UNA` advances upon receiving acknowledgments, the send window moves forward in the 32-bit sequence space.
3. G_{S_r}**:** It validates the sequence number in an incoming segment against the substate S_r. RFC 9293 [9] specifies that $G_{S_r}(S, \mathsf{SEG}_{\text{in}}) = 1$ if:

$$\texttt{RCV.NXT} \leq \mathsf{SEG}_{\text{in}}.\mathsf{SEQ} < \texttt{RCV.NXT} + \texttt{RCV.WND} \tag{2}$$

A segment whose sequence number satisfies Eq. (2) falls within the current receive window and can be accepted. As data is received, `RCV.NXT` advances, and the receive window moves forward in the 32-bit sequence space.

State Transition. The state transition function δ based on the current state S and an incoming segment SEG_{in} is defined as:

$$\delta(S, \mathsf{SEG}_{\text{in}}) = \begin{cases} (S', \mathsf{SEG}_{\text{out}}) & \text{if} \quad G_C \wedge G_{S_s} \wedge G_{S_r} = 1 \\ (S, \mathsf{SEG}_{\text{out}}) & \text{otherwise.} \end{cases} \tag{3}$$

where $\mathsf{SEG}_{\text{out}}$ denotes the output segment[1]. $\mathsf{SEG}_{\text{out}} \in \{\emptyset$, `ACK`, `RST`, `SYN-ACK`, `FIN-ACK`, `RST-ACK`, `PSH-ACK`, etc.$\}$. Fig. 3 illustrates a segment exchange in which $G_C \wedge G_{S_s} \wedge G_{S_r} = 1$. Consequently, the segments are accepted by the receiver and sender, and the state $S = (C, S_s, S_r)$ is updated.

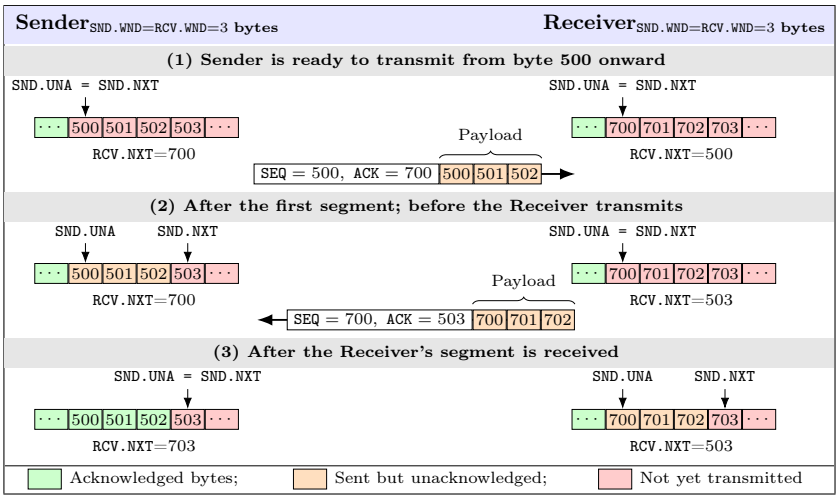

Fig. 3. TCP byte-level sequence numbering and cumulative acknowledgment semantics

2.3 Challenge-ACK Mechanism

RFC 5961 [42] introduces modifications to TCP segment processing to mitigate blind in-window `RST`, `SYN`, and data-injection attacks. As per TCP segment processing rules defined in RFC 793 [29], an off-path attacker who correctly guesses the 4-tuple and an in-window sequence number (i.e., $\texttt{RCV.NXT} \leq \mathsf{SEG}_{\text{in}}.\texttt{SEQ} < \texttt{RCV.NXT} + \texttt{RCV.WND}$) can forge `RST` or `SYN` segments to terminate an established

[1] When $\mathsf{SEG}_{in} = \emptyset$, the transition $\delta(S) = (S', \mathsf{SEG}_{out})$ models locally triggered events (e.g., application sends, retransmissions, or timer-driven transmissions).

connection or inject malicious data into the TCP stream. RFC 5961 therefore recommends stricter validation of such segments. For brevity, we describe the case relevant to our attack, namely, spoofed `SYN` injection in an established connection.

Prior to RFC 5961, if a TCP endpoint received a `SYN` segment in `ESTABLISHED` state and the sequence number of the received segment was outside the current receive window (i.e., $\mathsf{SEG}_{\text{in}}.\mathsf{SEQ} < \texttt{RCV.NXT} \vee \mathsf{SEG}_{\text{in}}.\mathsf{SEQ} \geq \texttt{RCV.NXT} + \texttt{RCV.WND}$), the segment was discarded. If the sequence number was within the receive window, however, the `SYN` was processed as it matched the 4-tuple of the `ESTABLISHED` connection. The receiver therefore interpreted the segment as an attempt by the peer to reestablish the connection after loss of previous connection and responded with a `RST` segment, thereby terminating the established connection at both endpoints.

RFC 5961 modifies the processing of `SYN` segments received by an endpoint in `ESTABLISHED` state. The receiver of such `SYN` segment MUST send a `challenge-ACK` (`c-ACK`), irrespective of its sequence number. Hence,

$$\mathsf{SEG}_{\text{out}} = \texttt{c-ACK}\,(\texttt{SEQ} = \texttt{SND.NXT}, \texttt{ACK} = \texttt{RCV.NXT}, flag = \texttt{ACK})\,.$$

A legitimate peer that has actually lost its connection state will respond to this `c-ACK` with a `RST` holding correct sequence and acknowledgment numbers copied from the `c-ACK`, thereby confirming that the previous connection state has been lost. A spoofed `SYN`, in contrast, causes only the additional `c-ACK` to be sent to the TCP peer and does not alter the established connection [42].

Formally, for a TCP connection in `ESTABLISHED` state, an incoming `SYN` segment with the correct 4-tuple and an in-window sequence number,

$$G_C(S, \mathsf{SEG}_{\text{in}}) = 0 \quad \text{if} \quad (S.C = \texttt{ESTABLISHED}) \wedge (\mathsf{SEG}_{\text{in}}.flag = \texttt{SYN}),$$

the corresponding state transition is $\delta(S, \mathsf{SEG}_{\text{in}}) = (S, \mathsf{SEG}_{\text{out}})$, where $\mathsf{SEG}_{\text{out}} = \texttt{c-ACK}\,(\texttt{SEQ} = \texttt{SND.NXT}, \texttt{ACK} = \texttt{RCV.NXT}, flag = \texttt{ACK})\,.$

2.4 Network Address Translation

Network address translation (NAT) enables multiple private hosts [30] to share one or a few public IP addresses for accessing the Internet by rewriting the source IP address and port of outbound segments and maintaining the corresponding entry in a translation table [17, 40, 41]. When reply traffic arrives, the NAT consults the translation table and forwards the packets to the corresponding internal hosts by rewriting the destination IP addresses and ports.

For TCP, when an internal client initiates a connection to a remote server by sending a `SYN` segment, the NAT creates an entry in the translation table containing the client's private IP address and source port, the translated source port, the destination IP address and port, and associated connection state and timeout value [17]. As subsequent packets are exchanged, the NAT uses this entry to translate and forward traffic between the internal client and the remote server. As a stateful middlebox, the NAT gateway tracks the connection state

throughout the active flow and uses state-dependent timeout values to manage its resources. For example, upon observing a `RST` segment, a NAT transitions the connection to `CLOSE` state and retain the corresponding entry for a short timeout before deleting it. Most routers use `CLOSE` state timeout of 1-10 seconds [46].

Port Allocation Strategies. External port allocation is not standardized across NAT implementations [17], and NAT devices differ in how they select external source ports for outbound TCP connections [20]. Under the widely deployed *port preservation* strategy, the NAT attempts to preserve the internal host's original ephemeral source port whenever possible. When a collision occurs, i.e., another internal host selects the same source port for a connection to the same destination IP address and destination port, the NAT selects another unused port. Other port allocation strategies include *random* and *per-destination sequential allocation* [12, 20, 46].

TCP Window Tracking in NAT. TCP window tracking is a mechanism that validates the sequence and acknowledgment numbers of received segments against the connection state maintained by the endpoints, as specified in Eq. (1) and Eq. (2). As a stateful middlebox that maintains connection state for flows originating from internal hosts behind it, a NAT gateway should ideally perform similar validation to ensure that only segments consistent with the endpoint's TCP state are forwarded. However, since TCP was designed as an end-to-end protocol without considering middleboxes, many NAT devices do not implement TCP window tracking, primarily due to performance constraints on resource-limited hardware and compatibility considerations. Linux-based router firmware controls this behavior through explicit configuration options: OpenWrt has the option `nf_conntrack_tcp_no_window_check` and AsusWrt has the option `ip_conntrack_tcp_be_liberal`, both of which relax TCP sequence number validation and are often enabled by default. Many commercial routers exhibit similar permissive behavior. In our analysis, OpenWrt 21.02 enables this option by default. Consequently, routers based on these operating systems may also operate without strict TCP window tracking. For example, the GL.iNet AX1300, which is based on OpenWrt 21.02, does not implement window tracking.

Reverse Path Validation in NAT. This mechanism checks whether the source IP address of an incoming packet is reachable through the interface on which the packet was received, thereby preventing IP address spoofing [2, 14]. In Linux systems, this functionality is supported through the `rp_filter` mechanism [25]. However, prior work [46] has demonstrated that many NAT routers either disable reverse-path validation or configure it loosely. As a result, the NAT gateway accepts and forwards packets carrying spoofed source addresses.

2.5 Off-Path TCP Hijacking

Traditional off-path TCP hijacking can be viewed as a *state inference* problem. An off-path attacker seeking to hijack a connection between two endpoints must infer sufficient components of the connection's TCB, comprising the connection 4-tuple and state S. The inferred state enables the attacker to construct spoofed segments satisfying $G_C \wedge G_{S_s} \wedge G_{S_r} = 1$, thereby inducing valid state transitions of the form $\delta(S, \mathsf{SEG}_{\text{in}}) = (S', \mathsf{SEG}_{\text{out}})$ (see Eq. 3). Successful state inference allows the attacker to terminate the connection or inject malicious data into plaintext stream. We first formalize the minimal subset of the TCB required for traditional off-path TCP hijacking and then extend the formulation to NAT networks.

Minimal Hijacking State ($\Pi(S)$). We define the minimal hijacking state required by an off-path attacker as:

$$\Pi(S) = \{\texttt{Src Port}, S_r.\texttt{RCV.NXT}, S_s.\texttt{SND.NXT}\}\,.$$

Here, `Src Port` denotes the client's ephemeral source port, S_r.`RCV.NXT` denotes the next sequence number expected by the target endpoint, and S_s.`SND.NXT` denotes the next sequence number the target endpoint is prepared to send. The client IP address (`Src IP`), server IP address (`Dest IP`), and server port (`Dest Port`) are typically known [3, 5, 6, 11, 43]. An attacker that successfully infers $\Pi(S)$ can craft spoofed segments satisfying δ (see Eq. 3), at the target endpoint.

In a NAT network, a client communicates with a remote server through a NAT gateway, which translates the client's private IP address and ephemeral source port before forwarding segments to the server. From server's perspective, the connection is identified by the NAT gateway's public IP address and translated source port, while the client's private IP address and original source port remain hidden behind the NAT.

Minimal Hijacking State in NAT ($\Pi_{\mathsf{NAT}}(S)$). We define the minimal hijacking state in a NAT network as:

$$\Pi_{\mathsf{NAT}}(S) = \{\mathsf{IP}_{\mathsf{pub}}, p_t, S_r.\texttt{RCV.NXT}, S_s.\texttt{SND.NXT}\}\,.$$

Here, $\mathsf{IP}_{\mathsf{pub}}$ denotes the public IP address of the NAT gateway and p_t denotes the translated source port associated with the TCP connection. The remaining components, S_r.`RCV.NXT` and S_s.`SND.NXT`, represent the relevant TCP receive- and send-sequence substate values of the target endpoint. Unlike $\Pi(S)$, the minimal hijacking state in NAT, $\Pi_{\mathsf{NAT}}(S)$ is independent of the client's private IP address. Existing off-path TCP hijacking attacks therefore operate on the public NAT mapping and do not require identification of the internal client associated with the connection [12, 13, 46]. In this work, we go beyond $\Pi_{\mathsf{NAT}}(S)$ and investigate whether an off-path attacker can additionally infer the private IP address of the internal client associated with a NAT mapping.

3 Threat Model of our Private IP Inference Attack

In this section, we describe the threat model for our off-path private IP address inference attack, as shown in Fig. 4. Our threat model comprises four entities: (i) a set of clients in a NATed LAN accessing TCP-based services (e.g., video streaming, SSH, etc.) from a remote server; (ii) a target remote server providing TCP-based services on a fixed IP address and port (e.g., HTTPS on port 443, SSH on port 22, HTTP on port 80); (iii) a NAT gateway that maintains active NAT mappings for the clients and multiplexes their TCP connections over a shared public IP address; and (iv) an off-path attacker located within the same LAN as the clients. In the following, we formally present the system model, attacker capabilities, and assumptions about NAT under which our attack operates.

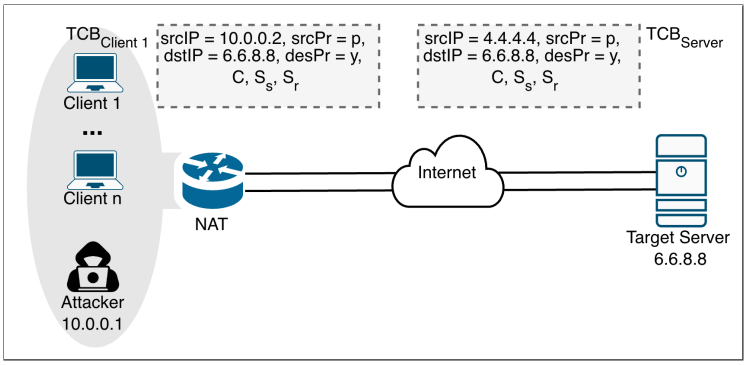

Fig. 4. The threat model

System Model. We consider a NATed LAN consisting of n client machines with private IP addresses $\mathsf{ip}_1, \ldots, \mathsf{ip}_n$ within the $10.0.0.0/k$ range [30]. The gateway exposes a single public IP address (4.4.4.4) to external hosts. Each client establishes a TCP connection to the server (holding public IP address 6.6.8.8) using a locally chosen ephemeral source port; we denote these ports by $p_1, p_2, \ldots, p_n \mid p_i \neq p_j$ for all $i \neq j$[2]. The server listens on a fixed destination port y. We refer to the set $\{p_j\}_{j=1}^{n}$ as the *victim ports* and to the corresponding set of client IP addresses $\{\mathsf{ip}_j\}_{j=1}^{n}$ as the *victim client IP addresses*. The clients and the server maintain a TCB for each TCP connection. The gateway maintains NAT translation table entries for each active TCP connection.

[2] Modern OSes randomly select ephemeral source ports, making port collisions among clients in the LAN highly unlikely. In the event of a collision, our attack can infer the private IP address of the client whose source port is preserved by the NAT, while the private IP address corresponding to the translated port cannot be determined; we therefore exclude such cases from our analysis.

Attacker Model. We assume an attacker located within the same LAN as the victim clients (i.e., within the same organization such as university network, or an open Wi-Fi network, at an airport, coffee shop, etc.) is assigned a private IP address (say, 10.0.0.1). The attacker is off-path with respect to the TCP connections between the victim clients and the server: it cannot observe, intercept, or modify segments exchanged between any $\{\mathsf{ip}_j\}_{j=1}^{n}$ and 6.6.8.8. The attacker knows the candidate IP addresses in the LAN, which can be determined from its own network configurations.

NAT Assumptions. Our attack assumes three behaviors commonly observed in widely deployed NAT gateways: the absence of *TCP window tracking*, insufficient *reverse-path validation*, and *port preservation*. The n concurrent TCP connections cause the NAT gateway to create and maintain n corresponding entries in its translation table. Under preserved allocation, the NAT entries take the form:

LAN Side	**WAN Side**
$\mathsf{ip}_1 : p_1$	$4.4.4.4 : p_1$
$\vdots$	$\vdots$
$\mathsf{ip}_n : p_n$	$4.4.4.4 : p_n$

The attacker aims to infer the private IP addresses of clients communicating with the target remote server, thereby determining the complete mapping $\{(\mathsf{ip}_j, p_j)\}_{j=1}^{n}$.

4 Private IP Inference Attack

We now describe our attack for inferring the private IP addresses of NATed TCP clients communicating with a target remote server. The attack proceeds in two stages. First, as a prerequisite, the attacker infers the NAT gateway's public IP address and the set of active source ports associated with the victim connections on the WAN side of the gateway, denoted by $\{p_j\}_{j=1}^{n}$. Under the assumed *port preservation* behavior, each port p_j is preserved across the NAT mapping and therefore identifies the corresponding connection on both the LAN and WAN sides of the gateway. We adopt the source-port inference methodology of Yang et al. [46] for this prerequisite. A complete description of this procedure is provided in (§A).

With the set of active source ports $\{p_j\}_{j=1}^{n}$, the attacker proceeds to its main objective: inferring the private IP address associated with each victim port. Specifically, the attacker seeks to infer the complete set $\{(\mathsf{ip}_j, p_j)\}_{j=1}^{n}$.

4.1 Inferring Victim Client IP Addresses $\{\mathsf{ip}_j\}_{j=1}^{n}$

Let $\mathcal{T}$ be the set of candidate IP addresses determined by the attacker (say, through its network configuration) such that $\{\mathsf{ip}_j\}_{j=1}^{n} \subseteq \mathcal{T}$. In the worst-case scenario, $\mathcal{T}$ may contain up to all 2^{24} private IP addresses in a block such as

10.0.0.0/8. The attacker processes the elements of $\mathcal{T}$ one at a time. For a fixed candidate IP address $\mathsf{ip} \in \mathcal{T}$, the attacker iterates over the set of victim ports $\{p_j\}_{j=1}^{n}$, testing each ordered pair (ip, p_j) in turn. The goal of this inner iteration is to determine whether ip coincides with one of the victim client IP addresses. Concretely, for a given $\mathsf{ip} \in \mathcal{T}$ and for each p_j, $1 \le j \le n$, the attacker executes the procedure described below to test whether $\mathsf{ip} = \mathsf{ip}_j$.

- If the test succeeds for some p_j, the attacker concludes that ip is a victim client IP address, records the mapping (ip_j, p_j), and immediately moves on to the next candidate IP address in $\mathcal{T}$.
- If the test fails for all $p_j \in \{p_1, \ldots, p_n\}$, then ip does not correspond to any victim client and is discarded.

By repeating this iteration over all $\mathsf{ip} \in \mathcal{T}$ in the outer loop and all p_j in the inner loop, the attacker eventually recovers the complete mapping $\{(\mathsf{ip}_j, p_j)\}_{j=1}^{n}$. Therefore, for a fixed candidate IP address $\mathsf{ip} \in \mathcal{T}$, the attacker sequentially selects each $p_j \in \{p_1, \ldots, p_n\}$ and initiates a test for the pair (ip, p_j) by sending the following spoofed `RST-ACK` segment towards the server:

$$\mathsf{Spoof}^{\texttt{RST-ACK}}_{\mathsf{ip}\to 6.6.8.8}\left(\begin{array}{|c|c|}\hline \texttt{srcPr: } p_j & \texttt{desPr: } y \\ \hline \texttt{seqNo: arb} & \texttt{ackNo: arb} \\ \hline \end{array}\right)$$

The sequence and acknowledgment numbers are arbitrary values in the range $[0, 2^{32}-1]$. To ensure these segments never reach the server to avoid overwhelming the server or triggering NIDS alerts [26], the attacker sets a low `TTL` (2 in our experiments), causing the segments to be dropped by some intermediate router.

When this spoofed `RST-ACK` segment reaches the gateway, its behavior depends on the presence of an existing mapping. If no mapping exists for (ip, p_j), the NAT simply discards the segment. However, if $\mathsf{ip} = \mathsf{ip}_j$, meaning a mapping is present, the NAT removes the mapping after its `CLOSE` state timeout period (see §2.4). Since many NAT devices disable *TCP window tracking*, they evict such mappings even when the `RST-ACK` contains incorrect sequence and acknowledgment numbers.

Once the attacker has sent the `RST-ACK` segment and waited for the NAT `CLOSE` timeout, it sends the following `SYN` segment towards the server:

$$\mathsf{Legit}^{\texttt{SYN}}_{10.0.0.1\to 6.6.8.8}\left(\begin{array}{|c|c|}\hline \texttt{srcPr: } p_j & \texttt{desPr: } y \\ \hline \texttt{seqNo: } z \xleftarrow{\$} \{0,1\}^{32} & \texttt{ackNo: } - \\ \hline \end{array}\right)$$

When this `SYN` reaches the NAT gateway, the NAT device behaves based on whether the earlier spoofed `RST-ACK` segment evicted an entry.

Case 1: $\mathsf{ip} \neq \mathsf{ip}_j$ (No mapping was evicted). Since the port inference technique (see §A) confirmed the existence of an active port p_j and corresponding WAN side entry $(4.4.4.4, p_j)$, the `SYN` segment causes a port collision. Consequently, the NAT selects an unused WAN side port p'_j and creates the entry: $\boxed{10.0.0.1 : p_j}\boxed{4.4.4.4 : p'_j}$. The gateway then forwards:

$$\mathsf{Legit}^{\texttt{SYN}}_{4.4.4.4\to 6.6.8.8}\left(\begin{array}{|c|c|}\hline \texttt{srcPr: } p'_j & \texttt{desPr: } y \\ \hline \texttt{seqNo: } z & \texttt{ackNo: } - \\ \hline \end{array}\right)$$

Since the server maintains n active connections from 4.4.4.4 at source ports $\{p_j\}_{j=1}^{n}$, treats this SYN as a new connection request from $(4.4.4.4, p'_j)$ and replies with a SYN-ACK. The NAT forwards this SYN-ACK to the attacker. The arrival of this segment informs the attacker that $\mathsf{ip} \neq \mathsf{ip}_j$. The attacker then terminates this accidental new connection by sending a RST to the server.

Case 2: $\mathsf{ip} = \mathsf{ip}_j$ (Mapping was evicted earlier). Since the previous spoofed RST-ACK evicted the entry: $\boxed{\mathsf{ip} = \mathsf{ip}_j : p_j \mid 4.4.4.4 : p_j}$, the SYN segment causes the NAT to create the following entry: $\boxed{10.0.0.1 : p_j \mid 4.4.4.4 : p_j}$. When the translated SYN arrives at the server, the server already has an established TCP connection at $(4.4.4.4 : p_j, 6.6.8.8 : y)$. The server treats this SYN as an unexpected segment for an ESTABLISHED connection and responds with a c-ACK. Formally, the received SYN segment violates G_C (see §2.2), i.e.,

$$G_C(S, \mathsf{SEG}_{\text{in}}) = 0 \text{ when } (S.C = \texttt{ESTABLISHED}) \wedge (\mathsf{SEG}_{\text{in}}.flag = \texttt{SYN})$$

Thus, no state transition occurs and $\delta(S, \mathsf{SEG}_{\text{in}}) = (S, \mathsf{SEG}_{\text{out}})$, where $\mathsf{SEG}_{\text{out}}$ is a c-ACK segment:

$$\mathsf{Legit}^{\texttt{ACK}}_{6.6.8.8 \to 4.4.4.4} \left(\begin{array}{|c|c|} \hline \texttt{srcPr: } y & \texttt{desPr: } p_j \\ \hline \texttt{seqNo: SND.NXT} & \texttt{ackNo: RCV.NXT} \\ \hline \end{array} \right)$$

The c-ACK carries the server's current SND.NXT and RCV.NXT values, as stored in its TCB for that connection. These values define the server's authoritative send/receive substates (see §2.1) and are used to validate subsequent segments on the flow.

Upon receiving the c-ACK, the NAT gateway forwards the translated segment to the attacker:

$$\mathsf{Legit}^{\texttt{ACK}}_{6.6.8.8 \to 10.0.0.1} \left(\begin{array}{|c|c|} \hline \texttt{srcPr: } y & \texttt{desPr: } p_j \\ \hline \texttt{seqNo: SND.NXT} & \texttt{ackNo: RCV.NXT} \\ \hline \end{array} \right)$$

This c-ACK confirms the attacker that $(\mathsf{ip}, p_j) = (\mathsf{ip}_j, p_j)$, thereby identifying one element of the true mapping $\{(\mathsf{ip}_j, p_j)\}_{j=1}^{n}$, and identifying the private IP address of the victim. In addition, the attacker obtains $(\texttt{RCV.NXT}, \texttt{SND.NXT})$, which together with the valid 4-tuple forms $\mathit{\Pi}_{\mathsf{NAT}}(S)$, as defined in (§2.5). The attacker may terminate the victim connection by sending a RST using the server-consistent sequence and acknowledgment numbers obtained from the c-ACK.

5 Evaluation

We evaluate our attack on NAT-enabled Wi-Fi routers in a controlled environment and on two real-world Wi-Fi networks, and discuss practical considerations for the attack in real-world settings.

Ethical Considerations. All experiments were conducted in an ethical and responsible manner. We conducted two types of experiments: (1) *Controlled experiments:* We performed a controlled evaluation of our attack in an isolated

network laboratory. We tested NAT routers for the vulnerabilities exploited by our attack and evaluated the effectiveness of the attack under controlled conditions. (2) *Real-world experiments:* We evaluated the attack in two Wi-Fi networks, with permission from the respective network administrators. All client devices involved in these experiments were under our control, and no uninvolved users or systems were targeted or affected. For the TCP-based services (SSH and HTTPS), we used a remote server hosted on a commercial cloud platform under our control. Furthermore, we configured the TTL values of the `RST` packets generated during the attack to limit their propagation beyond the network and minimize unnecessary bandwidth consumption or load on the remote server.

5.1 Controlled Evaluation

In the following, we present our router analysis of the vulnerabilities exploited by our attack, evaluate its success rate and time cost, and assess its impact on SSH and HTTP connections. To evaluate the attack, we deployed a /28 network in which the victim clients and the attacker resided on the same LAN. The victim clients established SSH and HTTP connections to a server outside the LAN. For each inferred source port, we tested each candidate IP address sequentially to determine whether the corresponding client is communicating with the server.

Experimental Setup. The experimental setup involved following devices:

1. *NAT Gateway:* We evaluated 6 NAT-enabled Wi-Fi routers from different vendors. See Table 1 for details.
2. *Target Server:* We configured a server, running Ubuntu 26.04 (kernel 7.0, OpenSSH 10.2), providing SSH and HTTP services to connecting clients.
3. *Victim Clients:* Five victim clients running heterogeneous operating systems - Ubuntu 24.04, MacOS Tahoe 26.6, Android 16, and Windows 10/11 were deployed within the LAN. Our attack is independent of the client OS.
4. *Attacker:* The attacker machine (MacOS Tahoe 26.6) crafts spoofed TCP segments to infer the source ports of victim connections, evict NAT mappings, and ultimately infer the private IP addresses of victim clients.

Routers' Behavior. Table 1 summarizes the observed NAT behaviors across the 6 evaluated routers. Five routers - `Linksys E5600`, `D-Link M30`, `GL.iNet A1300`, `TP-Link Archer C6` and `Ubiqcom UB5021` - employ *port preservation* and disable both *TCP window tracking* and *reverse-path validation*, making them fully vulnerable to our attack. `Netgate 1541` lacks *port preservation* and enforces *reverse-path validation*, but disables *TCP window tracking*, and thus not vulnerable to our attack. The `CLOSE` state timeout of NAT also affects attack duration as shorter timeouts reduce overall attack time by removing the NAT entry after the `RST-ACK` packet. The `TP-Link` router used a 1 s `CLOSE` timeout, while `Linksys`, `D-Link`, and `GL.iNet` used 10 s.

Table 1. Observed NAT behaviors in evaluated routers; **TO**: NAT `CLOSE` Timeout; **PP**: Port Preservation; **WT**: TCP Window Tracking; **RPV**: Reverse-Path Validation. (✓): present; (✗): absent

Vendor	Model	TO (s)	PP	No WT	No RPV	Vuln.
Linksys	E5600	10	✓	✓	✓	✓
TP-Link	Archer C6	1	✓	✓	✓	✓
D-Link	M30	10	✓	✓	✓	✓
GL.iNet	GL-A1300	10	✓	✓	✓	✓
Ubiqcom	UB5021-GVW-D	10	✓	✓	✓	✓
Netgate	1541	90	✗	✓	✗	✗

Attack Performance. The attacker first inferred the ephemeral source ports of all active TCP connections using the methodology described in (§A). Across 20 trials, the attacker successfully inferred the private IP addresses of all clients with active SSH sessions. The inference time scaled approximately linearly with the NAT `CLOSE` timeout (see Table 2), requiring an average of 22 s with a 1 s timeout and 65 s with a 10 s timeout for five victim clients. We further evaluated the attack using an HTTP server with five clients behind a NAT, each periodically fetching data at randomized 5-20 s intervals over a 5-minute period. With a 1 s `CLOSE` timeout, the attacker correctly identified the private IP address in approximately 24 s in all 20 trials. With a 10 s timeout, the average inference time increased to approximately 70 s, with a success rate of 80%. The lower success rate is primarily due to client traffic during the `CLOSE` interval, which refreshes the NAT mapping and prevents its eviction.

The time required to test a candidate private IP address consists of two components - T_1 and T_2. T_1 is the time required for the NAT mapping to transition to the `CLOSE` state after receiving a spoofed `RST-ACK`, and T_2 is the time required to observe the target server's response to a subsequent `SYN` probe. For a subnet containing n candidate clients and m inferred source ports, sequentially testing every port for each candidate requires $(n-1) \cdot m \cdot (T_1 + T_2)$ s in the worst case, where the target client is evaluated last. The attacker can eliminate the dependence on m by testing all inferred source ports in parallel for each candidate IP address, reducing the worst-case time to $(n-1) \cdot (T_1 + T_2)$ s. Thus, for a /28 subnet with 13 candidate addresses, the expected worst-case time is $13 \cdot (1+1) = 26$ s for routers with a 1 s `CLOSE` timeout and $13 \cdot (10+1) = 143$ s for routers with a 10 s timeout, assuming $T_2 = 1$ s, i.e., the network latency between the attacker and the server. The attack can be further accelerated by partitioning the candidate IP addresses among k attackers, yielding an approximately $1/k$ reduction in inference time.

Table 2. Controlled evaluation results in a /28 subnet; TO: NAT `CLOSE` state timeout

TO (s)	Protocol	# Victims	IP Inferred	Success	Avg. T(s)
10	SSH	5	5/5	100%	65
	HTTP	5	4/5	80%	70
1	SSH	5	5/5	100%	22
	HTTP	5	5/5	100%	24

5.2 Real-World Experiments

To evaluate the attack under realistic network conditions, we conducted experiments on two real-world Wi-Fi networks configured with WPA2/WPA3-Personal security. The networks used /20 and /24 private address spaces, accommodating up to 4094 and 254 hosts, respectively. In both networks, the NAT gateway exposed a single public IP address to the Internet. Unlike our controlled experiments, where the number of clients and candidate IP addresses was small enough to permit sequential testing, the larger address spaces in these networks make exhaustive testing impractical. Therefore, our real-world experiments focus on determining whether an attacker can, given a candidate private IP address, infer whether the corresponding client is communicating with the target remote server under realistic network conditions.

Experimental Setup. The attacker and victim clients shared the same Wi-Fi LAN. For SSH experiments, the victim clients established connections to a commercial cloud server running Ubuntu 24.04 (Linux kernel 6.14.0). For HTTPS experiments, victim clients accessed a popular online banking website (www.anonymous.com[3] over TLS 1.3.

Impact on SSH and HTTPS Sessions. We established multiple concurrent SSH sessions from victim clients to the cloud server, with the NAT gateway creating per-flow mappings for the connections. Across all trials, the attacker correctly inferred the private IP addresses of the clients communicating with the server.

We tested the attack on HTTPS sessions, with victim clients continuously accessing an online banking website over TLS 1.3. Across all trials, the attacker correctly identified the private IP addresses of the clients communicating with the server. A representative trace of attack impact on victim side is shown in Fig. 5. During the attack, the victim received an ICMP `TTL Exceeded` message (Type 11, Code 0) after the attacker spoofed the victim's IP address in a low-`TTL` `RST-ACK` segment. The segment expired en route, generating the ICMP message while causing the NAT gateway to move the corresponding mapping to the `CLOSE` state. If the mapping entry persisted within the timeout window, the ICMP message was forwarded to the victim. The client discarded this unsolicited `ICMP`

[3] The website name is anonymized for ethical considerations.

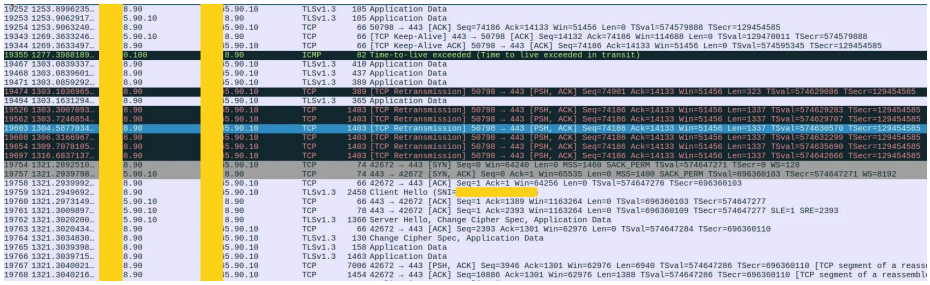

Fig. 5. Attack impact observed by the victim client on an active HTTPS session

error. After the attacker took over the mapping by sending a `SYN`, confirmed by a `c-ACK`, subsequent segments from the victim on the hijacked port were dropped, causing TCP retransmissions. The browser eventually timed out and reconnected using a new ephemeral port, initiating a fresh TLS 1.3 handshake [31].

Real-World Considerations. The nature of the attack differs between small networks and large-subnet real-world deployments. In a small network with a limited address space, an attacker can sequentially test all candidate IP addresses and evaluate the attack across the entire address space. In a larger network, however, exhaustively testing all possible addresses is considerably more expensive. Consequently, a realistic attacker would typically require some prior knowledge of the target's private IP address. For example, a malicious off-path adversary residing on the same LAN could obtain a high-value private IP address (say, an admin machine) through out-of-band information, such as by observing DHCP messages using Wireshark and correlating private IP addresses with DHCP hostnames to identify high-value targets. The attacker could then determine whether the target client is communicating with a particular remote service. The attacker can periodically test the target address by sending spoofed segments and observing the server's response (see §4.1). Automating these tests at regular intervals could allow the attacker to construct a long-term communication profile of the target client.

CGNAT Deployments: Carrier-Grade NAT (CGNAT) is widely deployed by ISPs and mobile carriers to conserve public IPv4 addresses, introducing a second translation layer beyond the home or enterprise router [27]. Our attack also extends to such multi-tier NAT deployments, where the first-hop home router translates to a private WAN side address that is further translated by a carrier-operated NAT. The attacker first learns the home router's WAN side address using existing techniques [46], then runs the attack exactly as before, but only against the first NAT. Thus the attack's success in CGNAT settings reduces to the same home router conditions as the single-NAT case; we leave validation against a live CGNAT deployment to future work.

6 Discussion & Countermeasures

6.1 Attack Limitations

1. The attack fails if a victim client communicates with the server after the attacker has sent a `RST-ACK` and before the `CLOSE` state timeout expires, since such segments refresh the NAT mapping. A subsequent `SYN` from the attacker then causes a port collision, causing the server to respond with a `SYN-ACK` and resulting in a false negative (see §4.1). However, most routers use short `CLOSE` state timeouts (see Table 1), making this condition uncommon in practice.
2. The inference time grows linearly with subnet size and `CLOSE` state timeout. Larger address spaces increase runtime but do not affect correctness. In real-world networks, an attacker may first identify high-value private IP addresses via network scanning (e.g., observing DHCP traffic in Wireshark), then probe those addresses to build their profile against internet services.
3. DHCP assigns IP addresses for finite lease periods [8]. However, organizations often use relatively long lease durations to improve network stability. In the real-world Wi-Fi network used in our evaluation, we observed lease durations of 5 days for one SSID and 30 days for another, reflecting different administrative policies across user groups. Shorter lease durations can limit long-term profiling based solely on IP addresses, thereby constraining sustained *deanonymization*, although short-term identification remains feasible. Moreover, DHCP implementations (e.g., ISC Kea DHCP [23]) may reassign the same IP address to a client after lease expiry. This behavior, commonly referred to as "lease affinity", occurs when the DHCP server retains expired lease records for a configurable hold time and preferentially reassigns the same address to a returning client, provided that the address has not been assigned to another device in the interim.

6.2 Countermeasures

1. NAT devices should implement *TCP window tracking*. Validating sequence and acknowledgment numbers before modifying translation table entries prevents entry eviction via spoofed `RST-ACK`, thereby preventing IP inference.
2. The attacker requires the set of external source ports allocated by the NAT device for the victim connections as a prerequisite for the attack (see §4). These source ports can be inferred due to the *port preservation* strategy employed by NAT devices (see §A). To prevent this inference, NAT devices should employ *random selection* for allocating external ports, thereby disrupting the attack at its outset.
3. To perform the attack, the attacker spoofs the victim's IP address while using its own legitimately associated MAC address (see §4.1). The attack succeeds because the network does not verify that the source IP address of a packet is bound to the MAC address of the sender. This check is typically enforced by the access point or switch on the LAN, rather than by the NAT function

itself, although these functions are often integrated into the same device. Enforcing this IP-MAC binding, for example, through DHCP snooping with IP Source Guard, causes packets with spoofed source IP addresses to be dropped, thereby preventing IP inference. *Reverse-path validation* should also be enabled to discard packets with spoofed external source IP addresses arriving on the LAN, thereby preventing port inference [2].

7 Related Work

Traffic Analysis and Fingerprinting. A separate line of work compromises user privacy by classifying encrypted traffic to identify the applications, websites, or DNS queries a user accesses, typically using statistical or learning-based models trained on traffic fingerprints [1, 4, 10, 19, 21, 28, 33–38, 44]. These techniques require the attacker to observe the target's traffic at a network vantage point and to match the observed traffic against a corpus of fingerprints in order to determine which of many possible destinations the target is communicating with, or, as in Wang et al. [45], by sniffing encrypted Wi-Fi frames on the shared wireless channel. Our attack assumes a fixed, attacker-specified target server and infers which internal client, if any, is communicating with that server. This inference is drawn solely from the NAT gateway's and server's control-plane responses to spoofed segments, without observing the victim's traffic, without a fingerprint database, and without dependence on the victim generating any particular traffic pattern.

Attacks in NAT Networks. NAT introduces distinct side channels that enable novel attacks in Wi-Fi networks [12, 13, 46]. Feng et al. [13] demonstrated a remote DoS attack against clients in NAT networks. Yang et al. [46] demonstrated that *port preservation*, lack of *TCP window tracking*, and insufficient *reverse-path validation* in NAT allow an off-path attacker within the LAN to detect and hijack internal TCP connections by manipulating the shared NAT table with spoofed segments. Feng et al. [12] exploited a path-MTU-based side channel in IP-address-sharing scenarios to terminate a victim's TCP connection. These attacks have predominantly focused on terminating a victim's connection, assuming one or more victims maintain a TCP connection with the target. In contrast, our work examines the privacy aspect of NATed clients and answers the question of which specific victim client is communicating with a target remote server.

8 Conclusion

We show that common NAT behaviors in Wi-Fi routers - *port preservation*, insufficient *reverse-path validation*, and lack of *TCP window tracking* - allow an off-path attacker to infer the private IP addresses of clients communicating with

a target server. By combining NAT state manipulation with the TCP challenge-ACK mechanism, our attack exposes a practical privacy leak in NATed environments, demonstrated against SSH and HTTP/HTTPS traffic on commercial routers and real-world Wi-Fi networks. We propose three countermeasures to mitigate the attack.

References

1. Bahramali, A., Houmansadr, A., Soltani, R., Goeckel, D., Towsley, D.: Practical Traffic Analysis Attacks on Secure Messaging Applications. In: NDSS (2020)
2. Baker, F., Savola, P.: Ingress Filtering for Multihomed Networks. RFC 3704, IETF (2004)
3. Bellovin, S.M.: Security problems in the TCP/IP protocol suite. SIGCOMM Comput. Commun. Rev. (1989)
4. Bushart, J., Rossow, C.: Padding Ain't Enough: Assessing the Privacy Guarantees of Encrypted DNS. In: USENIX FOCI (2019)
5. Cao, Y., Qian, Z., Wang, Z., Dao, T., Krishnamurthy, S.V., Marvel, L.M.: Off-Path TCP Exploits: Global Rate Limit Considered Dangerous. In: USENIX Security (2016)
6. Cao, Y., Wang, Z., Qian, Z., Song, C., Krishnamurthy, S.V., Yu, P.: Principled Unearthing of TCP Side Channel Vulnerabilities. In: ACM SIGSAC CCS (2019)
7. Chen, W., Qian, Z.: Off-path TCP exploit: how wireless routers can jeopardize your secrets. In: USENIX Security (2018)
8. Droms, R.: Dynamic Host Configuration Protocol. RFC 2131, IETF (1997)
9. Eddy, W.: Transmission Control Protocol (TCP). RFC 9293, IETF (2022)
10. van Ede, T., Bortolameotti, R., Continella, A., Ren, J., Dubois, D.J., Lindorfer, M., Choffnes, D.R., van Steen, M., Peter, A.: FlowPrint: Semi-Supervised Mobile-App Fingerprinting on Encrypted Network Traffic. In: NDSS (2020)
11. Feng, X., Li, Q., Sun, K., Fu, C., Xu, K.: Off-Path TCP Hijacking Attacks via the Side Channel of Downgraded IPID. IEEE/ACM Trans. Netw. (2021)
12. Feng, X., Li, Z., Li, Q., Wang, Z., Sun, K., Xu, K.: Off-Path TCP Exploits: PMTUD Breaks TCP Connection Isolation in IP Address Sharing Scenarios. In: ACM SIGSAC CCS (2025)
13. Feng, X., Yang, Y., Li, Q., Zhan, X., Sun, K., Wang, Z., Wang, A., Du, G., Xu, K.: ReDAN: An Empirical Study on Remote DoS Attacks against NAT Networks. In: NDSS (2024)
14. Ferguson, P., Senie, D.: Network Ingress Filtering: Defeating Denial of Service Attacks which employ IP Source Address Spoofing. RFC 2827, IETF (2000)
15. Gont, F., Bellovin, S.M.: Defending against Sequence Number Attacks. RFC 6528, IETF (2012)
16. Goodchild, B.J., Chiu, Y.C., Hansen, R., Lua, H., Calder, M., Luckie, M., Lloyd, W., Choffnes, D., Katz-Bassett, E.: The record route option is an option! In: ACM SIGCOMM IMC (2017)
17. Guha, S., Biswas, K., Ford, B., Sivakumar, S., Srisuresh, P.: NAT Behavioral Requirements for TCP. RFC 5382, IETF (2008)
18. Hain, T.L.: Architectural Implications of NAT. RFC 2993, IETF (2000)
19. Hayes, J., Danezis, G.: k-Fingerprinting: A Robust Scalable Website Fingerprinting Technique. In: USENIX Security (2016)
20. Herzberg, A., Shulman, H.: Security of Patched DNS. In: ESORICS (2012)

21. Houser, R., Li, Z., Cotton, C., Wang, H.: An Investigation on Information Leakage of DNS over TLS. In: ACM CoNEXT (2019)
22. ifconfig.co: ifconfig.co (2026), https://ifconfig.co/, accessed: 2026-04-15
23. Internet Systems Consortium (ISC): Kea administrator reference manual: 11. lease expiration. https://kea.readthedocs.io/en/stable/arm/lease-expiration.html (2026), section 11.4, "Configuring Lease Affinity." Accessed: 2026-09-08
24. Larsen, M., Gont, F.: Recommendations for Transport-Protocol Port Randomization. RFC 6056, IETF (2011)
25. Linux Kernel Developers: rp_filter (2025), https://sysctl-explorer.net/net/ipv4/rp_filter/
26. Mahajan, V., Peddoju, S.K.: Deployment of Intrusion Detection System in Cloud: A Performance-Based Study. In: IEEE Trustcom/BigDataSE/ICESS (2017)
27. Perreault, S., Yamagata, I., Miyakawa, S., Nakagawa, A., Ashida, H.: Common Requirements for Carrier-Grade NATs (CGNs). RFC 6888, IETF (2013)
28. Petagna, E., Laurenza, G., Ciccotelli, C., Querzoni, L.: Peel the Onion: Recognition of Android Apps Behind the Tor Network. In: ISPEC (2019)
29. Postel, J.: Transmission Control Protocol. RFC 793, IETF (1981)
30. Rekhter, Y., Moskowitz, R., Karrenberg, D., de Groot, G.J., Lear, E.: Address Allocation for Private Internets. RFC 1918, IETF (1996)
31. Rescorla, E.: The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, IETF (2018)
32. Richter, P., Wohlfart, F., Vallina-Rodriguez, N., Allman, M., Bush, R., Feldmann, A., Kreibich, C., Weaver, N., Paxson, V.: A Multi-perspective Analysis of Carrier-Grade NAT Deployment. In: ACM IMC (2016)
33. Rimmer, V., Preuveneers, D., Juarez, M., Van Goethem, T., Joosen, W.: Automated Website Fingerprinting through Deep Learning. In: NDSS (2018)
34. Shen, M., Liu, Y., Zhu, L., Du, X., Hu, J.: Fine-Grained Webpage Fingerprinting Using Only Packet Length Information of Encrypted Traffic. IEEE Transactions on Information Forensics and Security (2021)
35. Shen, M., Zhang, J., Zhu, L., Xu, K., Du, X.: Accurate Decentralized Application Identification via Encrypted Traffic Analysis Using Graph Neural Networks. IEEE Transactions on Information Forensics and Security (2021)
36. Shen, M., Zhang, J., Chen, S., Liu, Y., Zhu, L.: Machine Learning Classification on Traffic of Secondary Encryption. In: 2019 IEEE GLOBECOM (2019)
37. Shulman, H.: Pretty Bad Privacy: Pitfalls of DNS Encryption. In: WPES (2014)
38. Siby, S., Juarez, M., Díaz, C., Vallina-Rodriguez, N., Troncoso, C.: Encrypted DNS -> Privacy? A Traffic Analysis Perspective. In: NDSS (2020)
39. Smith, M., Hunt, R.: Network security using NAT and NAPT. In: IEEE ICON (2002)
40. Srisuresh, P., Egevang, K.: Traditional IP Network Address Translator (Traditional NAT). RFC 3022, IETF (2001)
41. Srisuresh, P., Holdrege, M.: IP Network Address Translator (NAT) Terminology and Considerations. RFC 2663, IETF (1999)
42. Stewart, R.R., Dalal, M., Ramaiah, A.: Improving TCP's Robustness to Blind In-Window Attacks. RFC 5961, IETF (2010)
43. de Vivo, M., de Vivo, G.O., Koeneke, R., Isern, G.: Internet vulnerabilities related to TCP/IP and T/TCP. SIGCOMM Comput. Commun. Rev. (1999)
44. Wang, T., Cai, X., Nithyanand, R., Johnson, R., Goldberg, I.: Effective Attacks and Provable Defenses for Website Fingerprinting. In: USENIX Security (2014)

45. Wang, Z., Feng, X., Li, Q., Sun, K., Yang, Y., Li, M., Du, G., Xu, K., Wu, J.: Off-Path TCP Hijacking in Wi-Fi Networks: A Packet-Size Side Channel Attack. In: NDSS (2025)
46. Yang, Y., Feng, X., Li, Q., Sun, K., Wang, Z., Xu, K.: Exploiting Sequence Number Leakage: TCP Hijacking in NAT-Enabled Wi-Fi Networks. In: NDSS (2024)

A Inferring Victim Ports $\{p_j\}_{j=1}^{n}$

Before proceeding to infer the ephemeral ports used by the victim clients to establish TCP connections, the attacker must first determine the public IP address of the NAT gateway. This can be achieved in many different ways [16, 22].

Afterwards, the off-path attacker iterates over each candidate source port $r \in \{32768, 32769, \ldots, 65535\}$ and determines whether r belongs to the set $\{p_1, \ldots, p_n\}$. This search space corresponds to the ephemeral source port range typically used by TCP implementations for outgoing connections[4]. By exhaustively testing this range, the attacker eventually recovers the complete set of victim ports, i.e., $\{p_j\}_{j=1}^{n}$.

For a selected port r, the attacker sends the following legitimate `SYN` segment to the server:

$$\mathsf{Legit}^{\texttt{SYN}}_{10.0.0.1\to 6.6.8.8}\left(\begin{array}{|c|c|}\hline \texttt{srcPr: } r & \texttt{desPr: } y \\ \hline \texttt{seqNo: } z \xleftarrow{\$} \{0,1\}^{32} & \texttt{ackNo: } - \\ \hline \end{array}\right)$$

Upon receiving this segment, the NAT gateway router behaves differently depending on whether r matches a victim port, i.e., $r \in \{p_j\}_{j=1}^{n}$ or not.

Case 1: $r = p_j$, for some j in $1 \le j \le n$. Because the port is already in use, the NAT router cannot preserve it. It allocates a new external port $r' \neq r$ and inserts the NAT table entry: $\boxed{10.0.0.1 : r \mid 4.4.4.4 : r'}$ and forwards:

$$\mathsf{Legit}^{\texttt{SYN}}_{4.4.4.4\to 6.6.8.8}\left(\begin{array}{|c|c|}\hline \texttt{srcPr: } r' & \texttt{desPr: } y \\ \hline \texttt{seqNo: } z & \texttt{ackNo: } - \\ \hline \end{array}\right)$$

Case 2: $r \neq p_j$ for all $1 \le j \le n$. Since the port is unused, the NAT preserves the port and inserts the entry: $\boxed{10.0.0.1 : r \mid 4.4.4.4 : r}$. It then forwards:

$$\mathsf{Legit}^{\texttt{SYN}}_{4.4.4.4\to 6.6.8.8}\left(\begin{array}{|c|c|}\hline \texttt{srcPr: } r & \texttt{desPr: } y \\ \hline \texttt{seqNo: } z & \texttt{ackNo: } - \\ \hline \end{array}\right)$$

To avoid overwhelming the server or triggering NIDS alerts [26], the attacker prevents these `SYN` segments from reaching the server by setting a very low `TTL` (2 in our experiments), ensuring the segments are dropped by some intermediate router.

[4] In practice, ephemeral ports for a TCP connection typically range from 32768-61000 for Linux systems and 49152-65535 for Windows systems [13].

Next, the attacker sends the following spoofed SYN-ACK segment to the public IP address of the NAT gateway:

$$\mathsf{Spoof}^{\texttt{SYN-ACK}}_{6.6.8.8 \to 4.4.4.4} \left(\begin{array}{|c|c|} \hline \texttt{srcPr}\colon y & \texttt{desPr}\colon r \\ \hline \texttt{seqNo}\colon u \xleftarrow{\$} \{0,1\}^{32} & \texttt{ackNo}\colon z+1 \\ \hline \end{array} \right)$$

The gateway accepts this spoofed segment due to insufficient *reverse-path validation*. The router then behaves differently depending on the earlier case:

Case 1: ($r = p_j$): The NAT router forwards the SYN-ACK segment to the internal host with $\mathsf{srcIP} = \mathsf{ip}_j$. The attacker therefore receives no response, leaking the fact that r is indeed a victim port.

Case 2: ($r \notin \{p_1, \ldots, p_n\}$): The NAT router forwards the segment back to the attacker, confirming that r is not a victim port.

Through this process, the attacker can test all 2^{15} possible ports and recover the complete set $\{p_j\}_{j=1}^{n}$.