Attacking Connection Tracking Frameworks as used by Virtual Private Networks

Benjamin Mixon-Baca
ASU/Breakpointing Bad
bmixonba@asu.edu

Jeffrey Knockel
Citizen Lab, University of Toronto

Diwen Xue
University of Michigan

Tarun Ayyagari
Arizona State University

Deepak Kapur
University of New Mexico

Roya Ensafi
University of Michigan

Jedidiah R. Crandall
ASU/Breakpointing Bad

ABSTRACT

VPNs (Virtual Private Networks) have become an essential privacy-enhancing technology, particularly for at-risk users such as dissidents, journalists, NGOs, and others vulnerable to targeted threats. While previous research investigating VPN security has focused on cryptographic strength or traffic leakages, there remains a gap in understanding how lower-level primitives fundamental to VPN operations, specifically connection tracking, might undermine the security and privacy that VPNs are intended to provide.

In this paper, we examine the connection tracking frameworks used in common operating systems, identifying a novel exploit primitive that we refer to as the *port shadow*. We use the *port shadow* to build four attacks against VPNs that allow an attacker to intercept and redirect encrypted traffic, de-anonymize a VPN peer, or even portscan a VPN peer behind the VPN server. We build a formal model of modern connection tracking frameworks and identify that the root cause of the port shadow lies in five shared, limited resources. Through bounded model checking, we propose and verify six mitigations in terms of enforcing process isolation. We hope our work leads to more attention on the security aspects of lower-level systems and the implications of integrating them into security-critical applications.

KEYWORDS

computer network, security, VPN, exploit, formal methods

1 INTRODUCTION

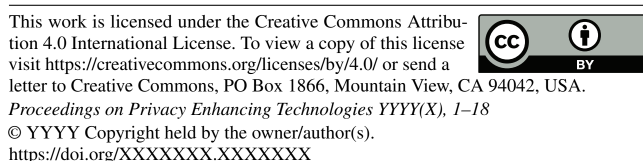
Internet service providers (ISPs) and national governments are increasingly disrupting and manipulating Internet traffic [16, 42, 48]. As a result, the use of virtual private networks (VPNs) has been on the rise, not only among average users who wish to protect their privacy but also, and more notably, among high-risk individuals such as dissidents, activists, or NGO workers, who turn to VPNs to mitigate threats from adversarial network infrastructures that aim to compromise their security. VPN clients encapsulate users' traffic within an encrypted tunnel to the VPN server, which then relays the traffic to its final destination, thus protecting users from potentially untrusted upstream networks.

Although VPNs are not typically thought of as middleboxes, in a privacy context network address translation (NAT) is a key part of VPNs and so both the VPN client and VPN server are de facto middleboxes. Operating systems typically implement NAT and connection tracking in kernel space but completely separate from the kernel's network stack, meaning that none of the actual connection state is available to the connection tracking framework. This creates a situation that conflicts with the end-to-end principle that guided the development of the Internet's core protocols. This end-to-end principle advocates for functionalities such as reliability and security to be implemented on end-hosts. However, applications such as VPNs rely heavily on a stateful connection tracking framework that manages connections as they enter and leave the network stack. Notably, the connection tracking framework is generally a shared resource. It enables efficient management of connections across various kernel threads and processes within the system. Yet, this approach also introduces shared states that, if not properly managed, can pose potential security risks to any applications dependent on the framework.

In this paper, we take the first look at how modern connection tracking frameworks might impact the security and privacy of VPN applications built upon them. Previous research into VPN security has focused on aspects such as protocol integrity, cryptographic strength, or traffic leakages [7, 12, 43, 51]. However, there has been less attention on how underlying systems that facilitate VPN operations, such as connection tracking, might compromise the security offered by VPNs. A key observation that motivated our work is that many of these lower-level systems rely on shared resources. In any system where resources are shared, security is contingent upon how effectively processes are isolated from one another. For connection tracking frameworks, the main concern is the degree of isolation between connections, especially when these are a mix of local and remote connections, or when they are associated with different security notions, as is often the case with VPN applications.

We present a novel exploit primitive that leverages shared resources within connection tracking frameworks as used by many

This work is licensed under the Creative Commons Attribution 4.0 International License. To view a copy of this license visit https://creativecommons.org/licenses/by/4.0/ or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.



Proceedings on Privacy Enhancing Technologies YYYY(X), 1–18
© YYYY Copyright held by the owner/author(s).
https://doi.org/XXXXXXX.XXXXXXX

VPNs and the ability of an attacker to manipulate it to violate process isolation of clients connected to the same VPN server as the attacker. We use the exploit primitive, that we refer to as the *port shadow*, to develop four related attacks against VPNs discovered *via* static and dynamic analysis of real-world implementations (Linux, FreeBSD, OpenVPN, WireGuard, and OpenConnect) and reviewing thirteen VPN and NAT RFCs [17–19, 23, 26, 27, 29, 36, 39]. The attacks challenge the typical security and privacy expectations users may have when using VPNs. For example, in situations where multiple users are connected to the same VPN server, one might expect that the processes of different users should be separate and unable to interfere with each other. However, as shown later in the paper, we demonstrate attacks that violate process isolation, allowing an attacker to intercept a target's encrypted traffic and redirect their traffic, de-anonymize other users connected to the same VPN server, or port scan them behind the VPN server.

To evaluate the practicality of the attacks, we conducted tests in both virtualized and live environments, targeting 58 different configurations of operating systems and connection tracking frameworks. Our evaluation revealed that the attacks were successful in the majority of cases across all tested VPN protocols (OpenVPN, OpenConnect, and WireGuard) using a range of connection tracking frameworks. The broad applicability of the *port shadow* across configurations indicates that the vulnerability lies not within any specific VPN protocol, but rather in the underlying systems that facilitate VPN operations.

Finally, we build a formal model of modern connection tracking frameworks based on our analysis and relevant RFCs. Through this model, we identify that the *port shadow* stems from five shared, limited resources that are fundamental to connection tracking frameworks. Building on this insight, we develop six mitigations and use bounded model checking with a depth of up to 2,000,000 state transitions to verify that non-interference holds. While the mitigations are effective at enforcing process isolation, our analysis also reveals the challenges, trade-offs, and inherent limitations of implementing these mitigations, particularly in the context of VPNs.

Connection tracking frameworks were not initially designed with the same threat model and security considerations as VPNs. The attacks we demonstrate here clearly indicate that the way VPNs rely on connection tracking can have significant implications on the security and privacy that VPNs are intended to provide. While we discuss potential mitigations, we hope that our work leads to a broader, more systematic reevaluation of the security aspects of lower-level primitives and the implications of integrating them into security-critical applications.

The remainder of this paper is structured as follows. § 2 provides background on VPNs, connection tracking frameworks, model checking, the non-interference property, and our threat model. § 3 explains the attacks. The testing environment and case studies are described in § 4, along with evaluation results. § 5 describes the formal model and model checking. We discuss the root causes and propose mitigations in § 6. § 7 covers related work. § 8 covers ethics and the disclosure process. We summarize our insights in § 9.

2 BACKGROUND

In this section, we introduce concepts key to understanding our work: VPNs, connection tracking frameworks, and model checking. Then we outline the threat model under which our attacks have been devised and tested.

2.1 Virtual Private Networks

Virtual private networks (VPNs) serve as an essential tool for at-risk users, such as journalists and dissidents, to safeguard against surveillance and interception from adversarial network infrastructures. The core security features provided by VPNs are IP address obfuscation and encryption. IP obfuscation works by ensuring that when a VPN client sends a packet through the network, it appears to originate from a different IP address than the client's actual IP. Encryption, on the other hand, secures the data in transit and prevents eavesdroppers from identifying the actual servers the client is communicating with. Both IP obfuscation and encryption features in VPNs necessitate address translation, for which most modern VPNs rely on connection tracking frameworks provided by the host system.

When a VPN client initiates a connection to a VPN server, it starts by sending a connection request to the server's listening port. Upon receiving this request, the VPN server allocates an IP address for the client, usually from the private address space [39]. This private IP address, along with specific routing rules, is then sent back to the client. These routing rules ensure that all the client's subsequent packets, except those addressed directly to the VPN server, are first encrypted and sent to the VPN server, which then relays the packets to their final destination.

2.1.1 *VPNs in this work.*

VPNs refer to a broad class of systems that use a variety of protocols and layers of the OSI network model. Shodowsocks is an application layer proxy that we do not consider because it is fundamentally different from the "NAT" VPNs considered in this work. We assume VPNs that work at Layer 3, such as OpenVPN, WireGuard, or OpenConnect. We don not consider IPsec because it is not widely deployed. Because these VPNs perform routing and NAT at the network layer, an attacker can modify shared state in components of the stateful connection tracking framework that facilitate the VPN's NAT and thus its routing behavior. VPNs are often used to either let the client access resources behind the VPN such as those used in corporate networks or they can change the client's IP address to route their traffic with an obfuscated identity, such as, those used for removing geoblocking, protecting against surveillance, or similar threats to at-risk users. We focus on the latter types of VPNs given the higher stakes, though that does not mean the former type of VPN is unaffected. We consider both IPv4 and IPv6 VPNs.

The VPN and connection tracking frameworks they interact with can be configured in many ways. We discuss specific configurations details when describing each attack in § 3 and the implications of their effects to attack success or failure in § 4 and § 6.

2.2 Connection Tracking Framework

When a VPN client sends a packet to the VPN server, the VPN process on the server first decrypts the payload to access the encapsulated packet. The source IP address of the encapsulated packet

is from the private IP space used by the VPN to create its virtual network. From here, the VPN server uses its connection tracking framework to perform address translation and route the decrypted packets to their intended destination. Connection tracking frameworks maintain an internal table, finite in size and shared across all processes, to track connections and support address translation (NAT). The table stores an entry for each connection, recording the originator (*e.g.*, the client), the responder (*e.g.*, a web server), the direction of the last packet sent, source and destination IPs and ports, the layer-4 protocol, *etc.* Additional information, such as expiry, reply status, and eviction eligibility, might also be stored to help manage memory by evicting stale entries. Despite the connection tracking framework's efforts to maintain a consistent connection state that mirrors the end hosts, it is inherently limited as the protocols it tracks, such as IP, TCP, UDP, and ICMP, were originally designed with end-to-end principles in mind. As a result, the connection tracking framework, being an intermediary, cannot always reliably mirror the state of these protocols. This limitation can lead to ambiguities in connection states, such as uncertainty about which host initiated the connection.

Figure 1 depicts a VPN, *N*, routing packets between private and public IPs spaces for two separate UDP processes running between two pairs of hosts, (*A*, *S*) and (*B*, *S*). We describe packets encapsulated and encrypted within the VPN tunnel using the notation $\langle saddr : sport, daddr : dport, * \rangle$ where `saddr` is the source IP address, `sport` is the source port, `daddr` is the destination IP address, `dport` is the destination port, and $*$ is additional information such as a payload, transport layer protocol information, or TCP flags. The outer layer packet in which the VPN packets are encapsulated is implied by this notation. Packets outside of the tunnel use curly braces. We define a connection, per the RFCs [1, 2], as a pair of processes performing interprocess communication *via* transport layer ports and identified by the same 4-tuple that describes a packet. This simplifies logical processes, *e.g.*, kernel threads that track connections, VPN server processes, attack code, *etc.*, to a unified view as a pair of ports and IP addresses with the understanding that IP aliasing and network address translation must be taken into account when identifying the specific machine on which the process runs, and introduce a 5-tuple that includes the protocol number only as necessary.

The following pathological example highlights port and IP collisions between two clients connected to the same VPN server and forms the basis for the port shadow. In steps 1 and 2, *A* and *B* connect to *N*'s VPN process. It assigns them the private IP addresses, 10.0.0.2, and 10.0.0.3, respectively. In step 3, *A* sends the packet $\langle 10.0.0.2 : 1, 4.4.4.4 : 80, GET \rangle$ to *S* at 4.4.4.4. When *N* receives the packet, its connection tracking framework does not find an entry for it in T (the green box), so the packet is passed from the network stack to its VPN process. The VPN process decrypts the payload, and extracts the packet. It then sends the packet to its network stack. Because the packet has a private IP address, the network stack invokes the connection tracking framework to perform NAT, changing the packet's `saddr` from a private IP address to its own globally routable IP address, 2.2.2.2. It creates and inserts the entry, `orig` = $\{10.0.0.2 : 1, 4.4.4.4 : 80\}$, `reply` = $\{4.4.4.4 : 80, 10.0.0.2 : 1\}$, into T. In step 4, *N* routes the packet to *S*. The `orig` and `reply` variables facilitate routing between public and private networks and play a major role in routing outside of the routing table.

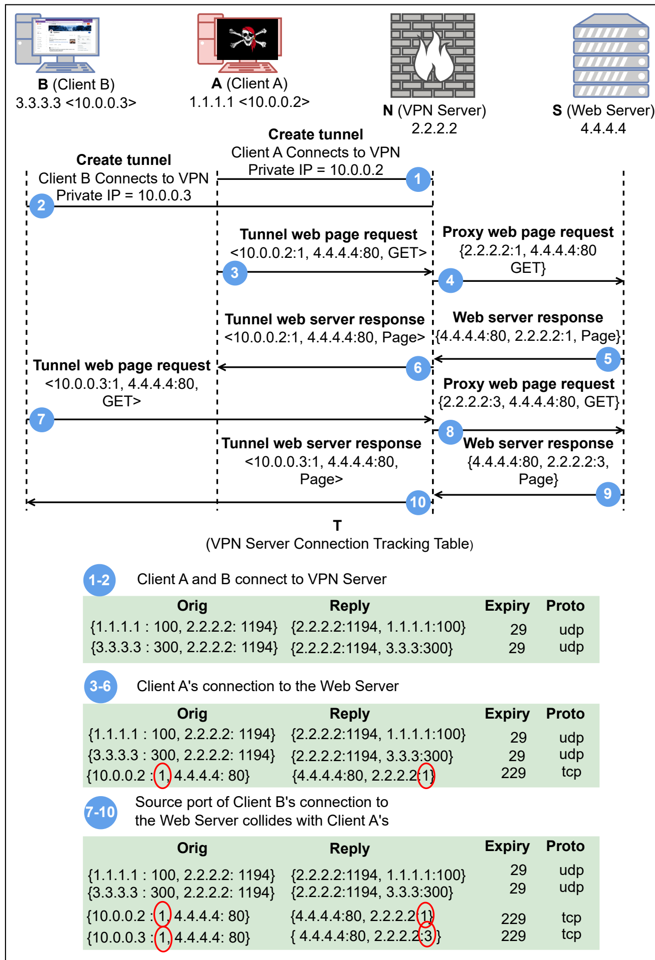
In step 5, *S* replies to *A*'s request by sending the response to *N*. In step 6, after *N* receives *S*'s reply, it checks T for a matching entry. *N*'s connection tracking framework finds the entry's matching `reply` variable from step 3 and performs address translation using the entry's `orig` variable. It also updates `Expiry` (RFC 4787 [29] recommends no less than 2 minutes). Depending on the system, the expiry and other meta-data are used to remove entries either when the number of entries within T exceeds an upper limit, `H` or when its expiry is reached. Its VPN process then encrypts the packet, sets the payload to the encrypted packet in a second packet, and sends the reply back to *A*.

In step 7, *B* sends the packet $\langle 10.0.0.3 : 1, 4.4.4.4 : 80 \rangle$ to *S*. The `sport` of *B*'s packet matches the `sport` of *A*'s packet from step 3. This causes a collision because both hosts use *N*'s public IP to communicate with *S* and select the same port, 1. If *N* uses the same `sport` for *A* and *B*, then *S* cannot differentiate the connections and *N* cannot determine the route back to the correct sender. In step 8, *N* resolves the collision by selecting a new `sport`, 3, creating an entry, and routing the packet to *S*. In step 9, *S* sends a reply back to *N*. In step 10, *N* uses the translation it create for *B* to translate the destination port and IP address to those that *B* is expecting and forwards the response to *B*.

2.2.1 Connection Tracking in Practice. Connection tracking is often implemented as a separate module from the operating system's network stack. This design is intentional and allows the kernel to separate mechanisms, such as routing decisions from policy, *e.g.*, allowing clients from one network to talk to another. The systems we consider in this paper are Ubuntu 20.04 and FreeBSD. We chose these systems because Ubuntu uses Netfilter for its connection tracking. Netfilter is installed on many Linux systems by default and is thus going to be present in many server implementations. It is also present on Android systems. FreeBSD is the basis for many network appliances such as pfsense, and also has overlapping code with the iOS network and connection tracking code. It has four distinct connection tracking frameworks: IPFW, natd, PF, and IPFILTER. We point out in § 3 how the differences between the implementations impact whether the attacks succeed.

2.3 Model Checking

The collision example hints at the possibility that connection tracking frameworks might not enforce process isolation in the context of VPNs, but a concrete answer is absent from the research literature. While exploring whether or not connection tracking frameworks do enforce process isolation we discovered several design flaws that an attacker can exploit. The next step was developing mitigations. Instead of writing kernel code for multiple operating systems that may or may not fix the underlying problem, we built a formal model based on the VPNs, connection tracking frameworks, and relevant RFCs to design and test mitigations. After describing the details of the attacks we discovered in § 3, we describe our model and how we used model checking to verify the mitigations we proposed fix the vulnerabilities. Model checking verifies the consistency of a system description with its formal specification [15] by modeling the system as a set of states and state-transition functions. Correctness is tested by exhaustively searching for states that violate one or more invariants defined using the system description.



Orig	Reply	Expiry	Proto
{1.1.1.1 : 100, 2.2.2.2: 1194}	{2.2.2.2:1194, 1.1.1.1:100}	29	udp
{3.3.3.3 : 300, 2.2.2.2: 1194}	{2.2.2.2:1194, 3.3.3:300}	29	udp

Orig	Reply	Expiry	Proto
{1.1.1.1 : 100, 2.2.2.2: 1194}	{2.2.2.2:1194, 1.1.1.1:100}	29	udp
{3.3.3.3 : 300, 2.2.2.2: 1194}	{2.2.2.2:1194, 3.3.3:300}	29	udp
{10.0.0.2 : 1, 4.4.4.4: 80}	{4.4.4.4:80, 2.2.2.2: 1}	229	tcp

Orig	Reply	Expiry	Proto
{1.1.1.1 : 100, 2.2.2.2: 1194}	{2.2.2.2:1194, 1.1.1.1:100}	29	udp
{3.3.3.3 : 300, 2.2.2.2: 1194}	{2.2.2.2:1194, 3.3.3:300}	29	udp
{10.0.0.2 : 1, 4.4.4.4: 80}	{4.4.4.4:80, 2.2.2.2: 1}	229	tcp
{10.0.0.3 : 1, 4.4.4.4: 80}	{ 4.4.4.4:80, 2.2.2.2: 3}	229	tcp

Figure 1: Source port collision and resolution process for two client's connecting to the same Web Server through the same VPN.

2.3.1 Non-Interference. Goguen and Meseguer proposed a general framework for defining security policies based on the concept of the non-interference property [22]. Non-interference asserts that two processes are non-interfering if the actions of one process have no effect on what the other process can observe in the system. For example, if N's connection tracking framework forwards A's packets to S, then B should not be able to affect A's connection. We chose this framework to explore process isolation of connection tracking frameworks because it is flexible enough for us to consider the flaws of insecure designs while being rigorous enough to verify that mitigations are effective.

2.4 Threat Model

Previous research has considered various on-path and in-path attackers [12, 44, 51] where the attacker is a wifi access point or a machine between the VPN client and critical infrastructure such as the VPN server, DNS server, or DHCP server, or between the VPN server and destination. While such a position can ease the execution of our attacks, such positioning is not strictly required for our model (an adjacent attacker) that considers an attacker connected to the same VPN server as the target. At minimum, our threat model includes four classes of hosts: Server (S, *e.g.* HTTP(S) server or DNS server), Victim (B), Attacker (A and C), and the VPN server (N) bridging the public and private networks. S operates a public service on an open port and only sends packets in response to incoming connection requests. N is *in-path*, capable of manipulating packets traversing through it, and is also *stateful* with a connection tracking table, T, and allocating virtual IPs from a private IP space (*e.g.*, 10.0.0.0/24)[1]. Finally, we assume A possesses reasonable computation and storage capabilities and can send packets with spoofed source IPs from the public network and can connect to the same VPN server as the victim — *i.e.* the attacker is adjacent in the virtual IP space to the victim. While not required, some attacks are significantly easier if the attacker has information about the target, such as their public IP address. A nation state actor can satisfy this assumption as follows. If the client has an app that leaks their PII to a server across the attacker's firewall, their public IP along with PII can be collected. Such situations arise in the wild, *e.g.*, Sogou keyboard [32].

3 ATTACKS

In this section, we introduce the *port shadow*, an exploit primitive that allows an attacker to reroute the packets of other VPN clients by inserting entries into the connection tracking table that cause port and IP collisions with VPN clients routing their traffic through the shared VPN server. The port shadow allows an attacker to position herself *in-path* between an adjacent VPN client and server, deanonymize adjacent client connections, reroute packets from the adjacent VPN clients to the attacker, or port scan a target behind a VPN server without requiring that the attacker be in- or on-path between the VPN server and client, or other privilege location. Previous attacks have demonstrated how an attacker can take over a VPN client's connections by an in-path attacker [44] or remove the tunnel encryption completely [51]. These attacks, however, assume the attacker is in a privileged position, such as being in the routing path between the VPN client and server, or between both the VPN client and DNS server and between the VPN client and a targeted website. In contrast, the port shadow has no such requirements. We only require that the attacker connect to the same VPN server as the target and that the attacker knows the target's public IP. We provide more details about the attack assumptions in § 4.3.

3.1 Adjacent-to-In-Path (ATIP)

The most significant consequence of using NAT frameworks in VPNs is that an attacker (A) can insert routes into the connection tracking table that collide with connections between the VPN server and VPN client (the target). The attacker can use these collisions to redirect a target's (B) packets, including their VPN connection request, to herself and escalate from a virtual adjacent position (*i.e.*, as the target's peer) to an in-path position between the target and the VPN server (N) (*i.e.*, as the target's upstream). Figure 2 depicts how A uses collisions in T to reroute B's VPN connection request to herself and escalating from adjacent to in-path.

[1] N is not strictly required to assign "private" IPs from the RFC 1918 [39] address space.

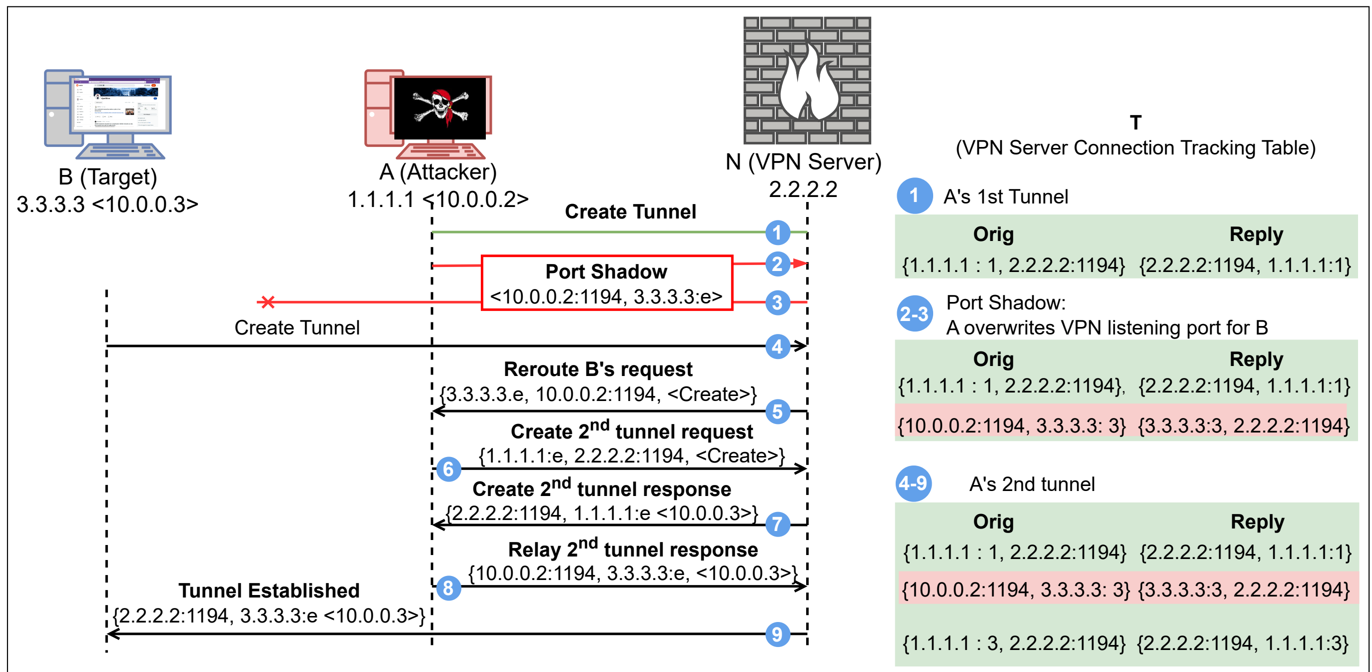


Figure 2: Adjacent-to-in-path attack.

In (1), A establishes a VPN connection to N and is assigned the private IP address 10.0.0.2. Next (2), A sends packets of the form $\langle 10.0.0.2 : 1194, 3.3.3.3 : e\rangle$ (*i.e.*, port shadow) through its tunnel to B's public IP address, with e denoting ephemeral ports, and (3) creates entries in the tracking table T. A sends a packet to each ephemeral port to ensure that when B attempts a connection request to N, the ephemeral port chosen by B will collide with an entry in the tracking table. 1194 is a typical port for a VPN to listen on.

(4) B sends a VPN connection request to N. The request collides with A's previously created entry from (2-3). Next (5), when N receives B's packet, it uses the colliding entry to translate the packet back to 10.0.0.2 instead of passing it to the VPN process. After A decrypts and extracts the payload containing B's VPN connection request, it changes the source IP address to 1.1.1.1 and sends it to N directly as if establishing a *second* VPN connection (6)[2]. Next, N completes A's second VPN connection and sends a response back to A along with a new private IP, 10.0.0.3 (7). Upon receiving this response, A forwards it through A's *first* VPN tunnel (8). Finally, N decrypts, decapsulates, NATs, and forwards the packet back to B (9). When B receives the packet, it believes N is following the VPN connection establishment protocol. At this point, A has successfully positioned herself in-path between B and N. As a result, any packets sent by B will be routed through A's *first* tunnel and then relayed to N, which proxies these packets as part of a *second* tunnel. A can use this position to perform subsequent injection attacks, such as those covered by Tolley et al. [44].

The root of the port shadow, detailed in § 6, lies in the shared resources, VPN public IP, port space used by every host connected to or routing packets through the VPN server, the connection tracking table, and the contention for these resources between VPN clients. In addition to modifying the network topology, an attacker can leverage the port shadow to infer the existence of connections between the VPN server and other IPs and redirect forwarded ports on the VPN server from the intended client to herself. The following subsections cover these tertiary consequences.

3.1.1 *Connection Inference.*

A can leverage colliding ports and IPs to learn whether B is currently connected to N. This is possible because when B establishes a VPN connection to N, an entry is created for the connection in T, as shown in Figure 8 of Appendix A.1. If A sends a packet to B as she typically would in an ATIP attack, and the destination port collides with the ephemeral port B is using for the VPN connection, the resulting port collision will force N to select a new port for the `reply` direction corresponding to the entry of A's packet. A can test this by spoofing a packet from B to N that should match the colliding packet. If A receives the packet, then no collision occurred because the source port was not changed and A infers that no connection exists; otherwise, it indicates a connection exists.

3.1.2 *Port-forward Overwrite.*

A can also use collisions to overwrite forwarded ports on the VPN server to redirect incoming packets to herself. This is achieved by A sending outgoing packets with a source port that matches the forwarded port, as in the ATIP and connection inference attacks. N creates an entry in T with a destination port in the `reply` direction that collides with the forwarded port. As a result, N will incorrectly route any incoming packets destined to the forwarded port to A instead of B. While this attack requires that A know the forwarded port, she can easily discover it by using a

[2]Note that A cannot decrypt B's encryption, only the encryption N's VPN process applied to the packet for its own VPN connection.

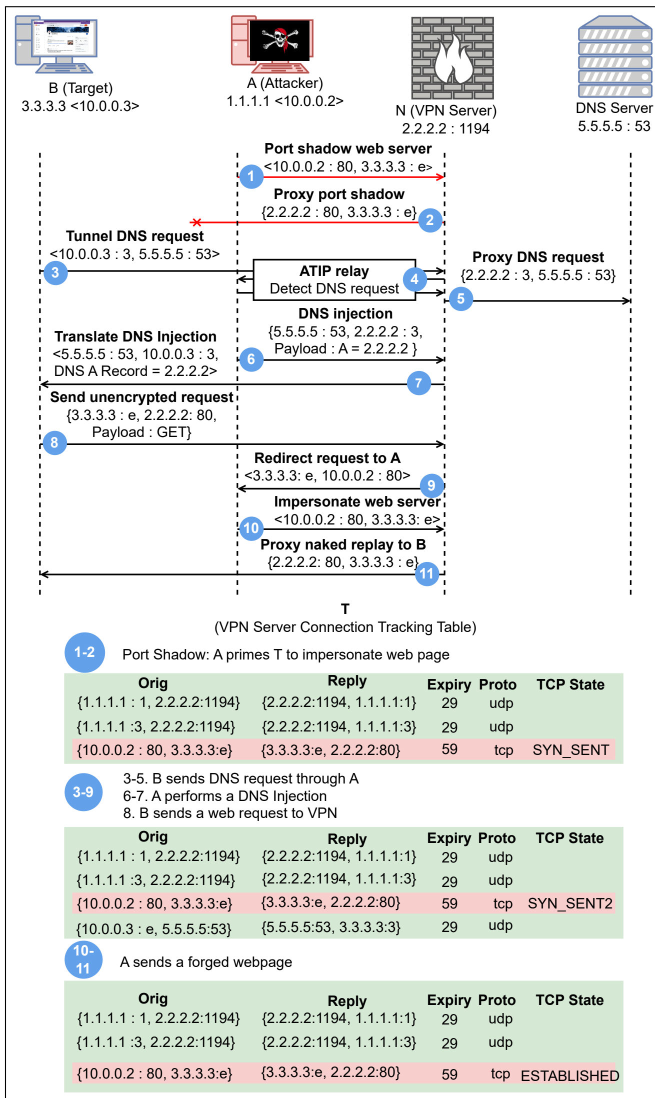


Figure 3: Decapsulation used to redirect B to a forged webpage served by A through N.

technique similar to the connection inference attack to identify port collisions.

3.2 Decapsulation

When applications send packets from a client with an established VPN connection, these packets are encrypted by the VPN software before being routed to the VPN server and then to their final destination IP. Layer 3 VPNs achieve this by assigning each host a private IP when they first connect to the VPN. VPN clients ensure traffic is routed through the VPN tunnel by modifying the routing rules on the client so that all packets destined for public IP addresses are routed first to the VPN client software for encryption, and then passed to the client's network stack, where the encrypted packet is placed in the payload of a second packet and sent to the connected VPN server. To avoid routing loops, the routing rules are set so that all packets destined for the VPN server are sent directly to it, bypassing the VPN client's encryption. Appelbaum et al. [12] discusses these issues with VPNs of 2012 and this was subsequently used by Perfect Privacy [7] to reveal the victim's real IP if port forwarding is in use. More recently Xue et al. [51] used similar methods to remove encryption between a VPN client and target (non-)server. Aside from the limitation of their threat model, their attack–specifically their "ServerIP" attack–can only perform decryption against one target IP address at a time because they replace the VPN IP address with the target server's when performing DNS injection. We demonstrate how an attacker can combine this routing weakness with IP and port collisions in the connection tracking table that permits the attacker deanonymize multiple servers by replacing server IP addresses the client wishes to communicate with to the VPN server IP and using the collisions in T to relay packets between the VPN client and target (non-VPN) server. An attacker can exploit this to make the target send packets outside of the VPN tunnel in two ways: first, by injecting DNS requests made by the target and replacing the actual server's IP with the IP of the VPN server and then relaying the packets between the target server and client and second, by targeting popular peer-to-peer applications like BitTorrent.

Figure 3 demonstrates how an attacker can combine port and IP collisions she insert into T, the ATIP attack, and routing weakness to replace the actual server IP with the VPN server IP, remove the VPN tunnel encryption and break anonymity. Assume that A has executed an ATIP against B and is now in-path. First, A prepares T to reroute B's web page request by sending a sequence of packets to B, which are forwarded by N (1-2). This creates an opening for incoming packets from B to be routed to A (the port shadow). Note that the packets are TTL-limited so that they do not trigger responses from B, such as TCP RST packets that would close the port shadow created. When B sends a DNS request (3), the request gets relayed by A (4). Being in-path, A can detect DNS packets by their size. Next, N proxies B's legitimate DNS request to the DNS server, which also creates an opening to route the DNS response (5). A injects a DNS A record, to the VPN server, mapping the queried domain name to the VPN server IP (6). In turn, the forged DNS response gets routed by N to B's private IP at 10.0.0.3 (7). Receiving the forged A record, B sends a web request, without VPN encryption, directly to the VPN server (8). The VPN server then relays the request to A's private IP 10.0.0.2 (9) because of the port shadow created by collision between the request packet and entries from step (1). Finally, A sends a forged web page through her tunnel to B (10), which is relayed unencrypted by N (11). Our example assumes that the target visited an unencrypted website that the attacker impersonated. If the site is HTTPS, then A needs a compromised certificate to impersonate the site. Additionally, instead of impersonating the website through the VPN, A can relay the packets to eavesdrop on B's packets because they were sent outside the VPN tunnel.

If A targets a peer-to-peer protocol like BitTorrent, then A simply needs to connect to the same VPN server as B then connect to the peer who is using the VPN server IP address as their real IP address. When B's p2p software responds to A's requests, the p2p traffic is sent to N directly, outside the VPN tunnel.

3.3 Eviction Reroute

The next vulnerability combines IP and port collisions in T with that attacker's ability to evict entries from T. This allows an attacker to redirect ingress packets intended for the target to the attacker *after* the client has connected to the VPN server. This attack is possible because the mappings between private IP addresses and the VPN clients managed by the VPN server are mutable. An attacker can exploit the NAT framework to overwrite existing mappings in the connection tracking table. Figure 4 illustrates how A can abuse mutable nature of connection mappings within the connection tracking table to replace a target's entries with her own, thus rerouting incoming packets that were meant for the target to herself instead.

Assume initially that T is empty. B and A connect to N and receive the private IPs 10.0.0.3 and 10.0.0.2, respectively (1-2). N adds two entries to T, increasing the number of entries to two. A then sends $H-4$ packets, $\langle 10.0.0.2 : e, x.x.x.x : x \rangle$, through N (3), where the destination IP and port are arbitrary. The entries are initially evictable (`Assured`= *False*) because N has not seen packets in both directions (`seen_reply`= *False*). Next, A spoofs responses to those packets to N (4) updating `seen_reply` and `Assured` variables, making the entries ineligible for eviction. T now contains $H-2$ entries that cannot be evicted prior to their expiry. Next, B sends an outgoing packet (*e.g.*, a DNS request) to a public server (5), creating a connection tracking entry in T and increasing the number of entries to $H-1$ (6). B's entry can be evicted (`Assured`= *False*) because packets have not been seen in both directions (`seen_reply`= *False*). After this, A sends the Hth packet to induce evictions from T (7), resulting in the entry corresponding to B's DNS request being evicted (8). A builds the port shadow by replacing B's entry from (5-6) by sending the packet $\langle 10.0.0.2 : 3, 4.4.4.4 : 53 \rangle$ to the DNS server. Finally, the DNS server sends the response to B's DNS request from step 5 (11), and N uses A's port shadow entry from step 9 to route the ingress DNS response intended for B to A instead (12). While this attack does require a significant volume of packets, it highlights how design flaws in connection tracking frameworks have downstream consequences to VPN security.

3.4 Port Scan

The eviction reroute attack is enabled by two factors: the private IP space shared by all clients, and the mutable mapping between a private IP and its assigned logical host. A third aspect to consider is the direction in which packets travel across the VPN. An attacker can exploit these factors to port scan a host *behind* the VPN server. Specifically, when a VPN client disconnects, its previously assigned private IP can be reallocated to another client. If the attacker has placed entries in T and disconnects from N, but those entries remain active in T, then any incoming packets to the VPN that match these stale entries will be rerouted to the newly connected VPN client, which will respond based on the status of the destination port on the current client. Figure 5 illustrates an example where the attacker can connect to an SSH server running on a client behind VPN.

Assume initially that A and B are not connected to N and that B has an SSH server listening on port 22. First, A connects to N and receives private IP address 10.0.0.2 (1), creating an entry in T. Next, A sends a TCP `SYN` packet to a second machine she controls, $A2$ (2), creating a second connection tracking entry into T (3). The

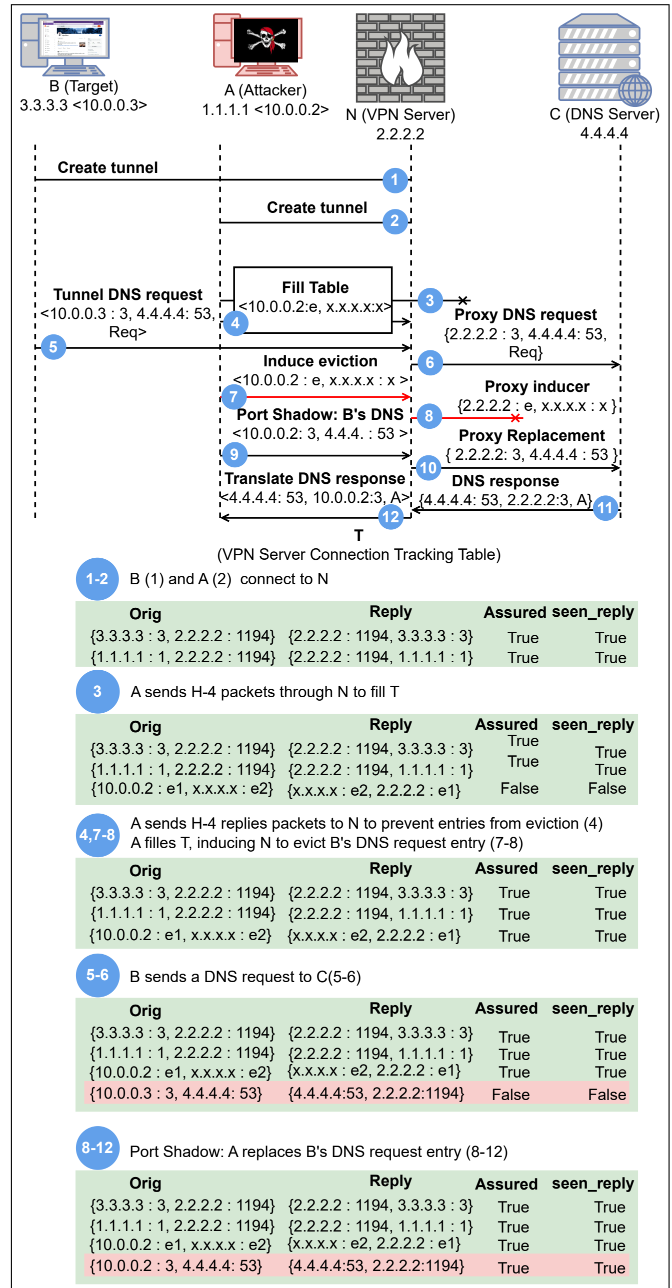


Figure 4: Eviction reroute attack.

entry is in the `SYN_SENT` state. A disconnects from N (4). When A disconnects, the entry for A's VPN connection is eventually removed by the garbage collector.

B connects to N and is assigned the same private IP address (10.0.0.2) that A previously had (5). Next, $A2$ sends a *second* TCP `SYN` to N to initiate a TCP 3-way handshake with B [3], and N uses the entry A created in step 2 to route the `SYN` packet to B (6-7). N also

[3]If A had sent a packet with different TCP flags, then the incoming packet would not be routed to B. This is because only specific sequences of TCP packets are valid according

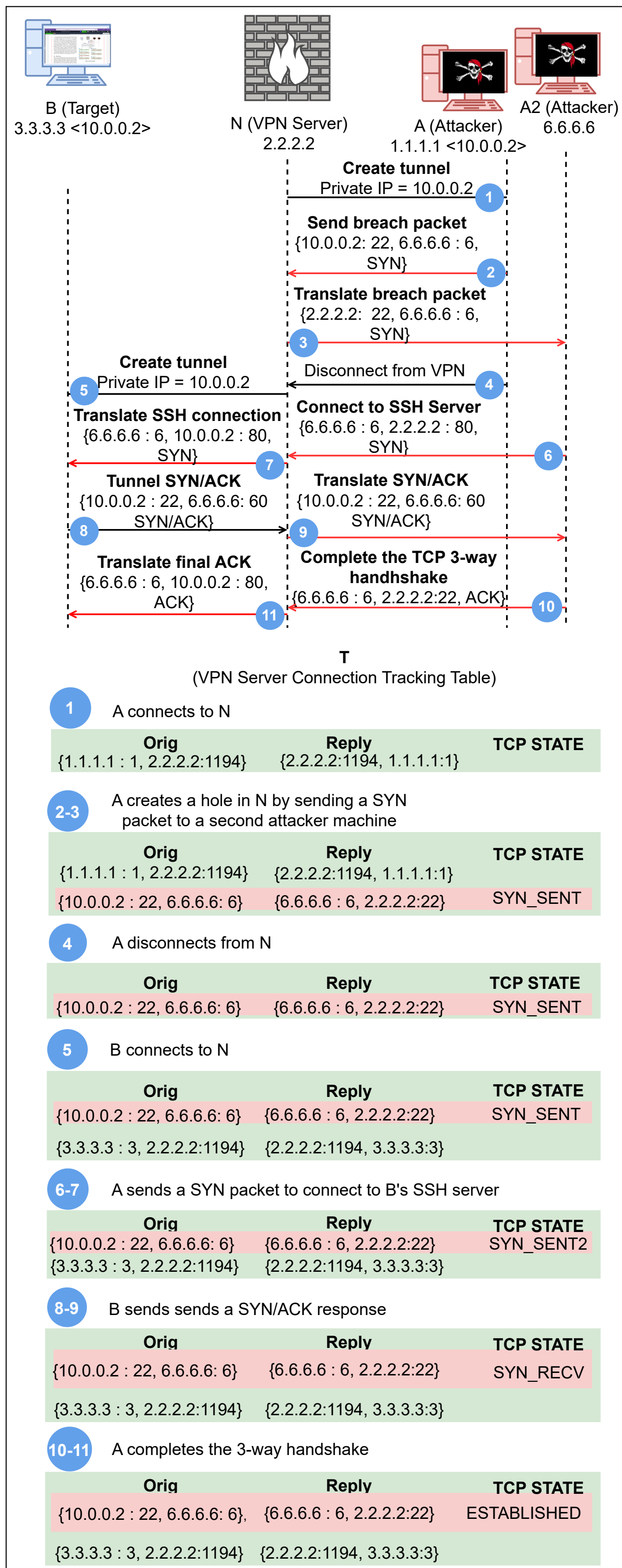


Figure 5: Port scan attack.

updates the entry's TCP state to SYN_SENT2. The SYN_SENT2 state is used by TCP for SIMULTANEOUS_OPEN, which is an alternative to the TCP 3-way handshake that two TCP end-points can use to establish a connection. *B* responds to the SYN packet with a SYN/ACK packet, and *N* updates the source port to its public IP address and forwards it to *A*2 (8-9). Finally, *A* sends the final ACK packet completing the TCP 3-way handshake, which *N* uses the translation to forward to *B* at 10.0.0.2 (10-11). At this point, *A* has successfully confirmed that *B* is running an SSH server. *A* can use similar methods to perform SYN scans, connect scans, UDP scans, and various other scans against hosts behind the VPN server.

4 CASE STUDIES

We demonstrate the efficacy of our attacks in case-studies in both virtualized and live test environments. We first describe the platforms tested and then present the experimental results. Finally, we describe additional analyses we performed to test assumptions the attacker must satisfy for successful exploitation.

4.1 Testing Environments

We evaluated our attacks in both virtualized and live test environments. Our virtualized environment consists of nine machines. All machines are Ubuntu Linux 20.04 except one VPN server that runs FreeBSD 14. We use Vagrant with VirtualBox to configure and provision the machines. We target three popular VPN implementations: OpenVPN, WireGuard, and OpenConnect deployed on Ubuntu Linux running Netfilter, FreeBSD running IPFW, natd, IPFILTER, or IPF. We did not target application layer VPNs like ShadowSocks because their address translation is not tightly coupled with the operating system's connection tracking framework.

Our live environment consists of OpenVPN or WireGuard running on an Ubuntu Linux 20.04 server. The attacker machine is a desktop running Ubuntu Linux 20.04 and OpenVPN or WireGuard. We used two sets of targets, one consisting of Android and Apple mobile devices and one consisting of a server deployed to a Vultr virtual server running Ubuntu Linux 20.04. All machines are in different geographic regions.

4.1.1 Configuration. Figure 6 depicts this environment. For the Ubuntu VPN server, we tested both IPv4 and IPv6. For the Ubuntu OpenVPN clients, we tested with and without network namespaces using namespaced. Our VPN configuration is based on tutorials that we googled, such as, from Digital Ocean [3, 4, 10], the official OpenVPN, WireGuard, and OpenConnect documentation [6, 9, 11], and additional documentation [30]. We discussed our configuration environment during disclosure and received no actionable information about how to make it more like actual deployments. Our impression based on our research is that system administrators are supposed to know what needs hardening and act accordingly. This is made difficult given that the Netfilter, IPFW, natd IPFILTER, and PF documentation does not discuss explicitly behavior in the presence of IP obfuscating VPNs. Between this and a lack of explicit and consistent information in the RFCs regarding both NAT and VPNs, it is no surprise the risk of IP/port collisions has gone undocumented.

to the TCP protocol and the connection tracking framework detects this and does not forward the packets.

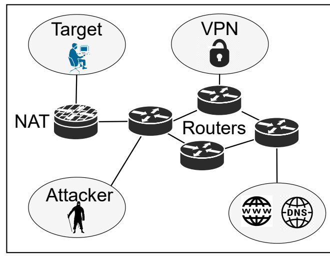


Figure 6: Full virtualized testing environment.

4.2 Attack Results

We now describe our attack results which are summarized in Table 1. We describe the different operating systems used and details about success rate, timing, and configurations where relevant.

4.2.1 Adjacent-to-in-path. We tested this attack in both the virtualized and live environments against OpenVPN, OpenConnect, and WireGuard. The attack was successful in both environments against all three VPNs on Linux and Netfilter and against IPv4 and IPv6. We tested FreeBSD in the virtualized environment and found the same implementation-specific behaviors that we found in the connection inference attack also allow the ATIP attack for IPFW and natd but not PF or IPFILTER. Once blind in-path, we executed an HTTP injection to serve the client a custom web page when they made an HTTP request to the web server. The injection only works for HTTP but not HTTPS due to encryption. Next, we injected a DNS response by inferring the client's DNS request through packet size. The DNS response had a CNAME/A record combination, pointing to an attacker-controlled server and tricked the client into connecting to this server instead of to the legitimate one.

We tested the amount of time it takes for the client to establish a connection to the VPN server with and without ATIP. The victim takes 6.1 seconds to connect to the VPN server without ATIP and 16.6 seconds with it. It takes the attacker on average 5 seconds to fill the 27,850 ephemeral IP addresses and the success rate is close to 100% in the control environment and in the network environment - it is directly proportional to the percent of the ephemeral port space the attacker fills. Subsequent packets have an average RTT delay of 1000ms RTT added against OpenVPN and 100ms for WireGuard.

4.2.2 Port-forward Overwrite. We conducted this attack in the virtualized environment against Linux and FreeBSD systems, targeting OpenVPN, WireGuard, and OpenConnect. The attack was successful in both settings against OpenVPN, WireGuard, and OpenConnect on Linux running Netfilter and against IPv4 and IPv6.

It failed against FreeBSD with PF and IPFILTER and caused a DoS against IPFW and natd. IPFILTER fails because it maintains two separate lists for connection entries, one for outgoing and incoming rules. The rule that implements port forward is stored as an entry in the incoming table so the attacker's outgoing packet does not overwrite the incoming rule. For IPFW, the forwarding rule is stored separately from the table, as with Netfilter, however, this creates a conflict with the table's dynamic rule, which prevents IPFW from correctly routing the packet.

4.2.3 Connection Inference. We confirmed this attack succeeds in both the virtualized and the live environments against OpenVPN, WireGuard, and OpenConnect only in the virtualized environment and against IPv4 and IPv6. For FreeBSD it was successful against IPFW and natd, similar to Linux running Netfilter. The success rate is over 90%. However, when testing on FreeBSD running IPFILTER and PF, the source port of outgoing packets was randomized by default and restricted to the range $49152, 65535$. The attack can still be executed but the VPN server's listening port must be in this range and enough packets must be sent for a collision between the selected source port and the listening port to occur. OpenVPN's default listening port is 1194, but both OpenVPN and WireGuard can be configured with ports within the range, making the attack possible.

4.2.4 Decapsulation. We tested two versions of the decapsulation attack in the virtualized and live environments against OpenVPN. First, we tested whether an attacker can perform the DNS injection to redirect a target from a normal web page to the VPN server IP. We successfully injected a DNS A record with the VPN server IP to the target. The traffic between the target and VPN server was sent unencrypted but the packets sent from the VPN server to the attacker were encrypted. Next, we tested an attacker targeting a peer-to-peer application. We selected BitTorrent as the target application and used qBittorent on both the target and attacker. The target provided a seed file pointing to an mp4 stored on its hard drive. When the attacker downloaded the mp4, the packets sent from the target to the VPN server were sent unencrypted. When executing the attack, we fill the 28231 ephemeral ports defined by `net.ipv4.ip_local_port_range` in under 10 seconds. Upon the target sending the request to the VPN server, the success rate is over 90%. The attack was successful against IPv4 and IPv6.

4.2.5 Eviction Reroute. We tested eviction rerouting in the virtualized environment on both Ubuntu running OpenVPN, WireGuard, and OpenConnect and FreeBSD running the same OpenVPN. First, the attacker executed ATIP. Next, the attacker filled the connection tracking table such that evictions could be triggered by a small number of additional packets. When the client made a DNS request, the attacker inserted entries that matched the `reply` direction of the DNS request and triggered evictions. The DNS response was then routed to the attacker. The attack was successful against IPv4 and IPv6. We are able to fill a 262144 entry table in less than 5 minutes. This attack took on average 30 minutes before a single successful attempt and the success rate was 20% on average. This is because the attack requires very specific timings to succeed and the attacker cannot precisely control which entries are evicted. If an attacker can more precisely control the ports assigned to the target's source port in the reply direction, it may be possible to increase the success rate and/or decrease the time to successful execution.

We were unsuccessful against FreeBSD configurations. IPFW/natd's NAT implementation maintains two separate lists for managing entries, one for connections with packets in only one direction and one for both directions. Once packets reach the latter list, they remain until the expiry. Unlike Linux, the entries cannot be evicted early, even if the table becomes full. Because of this, an attacker cannot force entries to be removed. IPFILTER removes entries when the table becomes full, but it will not remove entries unless they are

System Details			Attack						
OS	Connection Tracking Framework	VPN	ATIP	Connection Inference	Port Forward Overwrite	Decapsulation	Eviction Reroute	Port Scan	Evaluation Environment(s)
Ubuntu 20.04–22.04	Netfilter	OpenVPN	■	■	■	■	■	■	Virtualized, live
		OpenConnect	■	■	■	■	■	■	Virtualized
		WireGuard	■	■	■	■	■	●	Virtualized, live
FreeBSD 14	IPFW	OpenVPN	■†	■†	▲	◇	●	■	Virtualized
		WireGuard	■†	■†	▲	◇	●	●	Virtualized
FreeBSD 14	natd	OpenVPN	■†	■†	▲	◇	●	■	Virtualized
		WireGuard	■†	■†	▲	◇	●	●	Virtualized
FreeBSD 14	IPFILTER	OpenVPN	■†	■†	●	◇	●	●	Virtualized
		WireGuard	■†	■†	●	◇	●	●	Virtualized
FreeBSD 14	PF	OpenVPN	■‡	■‡	●	◇	●	●	Virtualized
		WireGuard	■‡	■‡	●	◇	●	●	Virtualized

Table 1: Case-study Results. ■ indicates a specific VPN configuration (row) is vulnerable to a specific attack (column). ● indicates the attack does not work for the specific configuration. ▲ indicates a DoS instead of rerouting. ◇ indicates the attack does not apply to that configuration. † indicates the platform can be configured to use "Port Preservation" as the source port assignment behavior as described in RFC 4787 §4.2.1 [29]. ‡ indicates that the system uses "Random Assignment" for source port assignment behavior as described in RFC 4787 §4.2.1 [29].

"inactive". Inactive can range anywhere from not being used for more than 4 days to as little as 30 seconds.

4.2.6 Port Scan. Two members of the research team, that we refer to as the attacker and target, tested the port scan attack in the live environment. The attacker connected to the VPN, created entries in the table, and disconnected following the steps outlined in § 3.4. The target then connected to the same VPN and was assigned the same private IP address previously assigned to the attacker. The port scanning successfully revealed services running on Apple and Android devices. The two researchers were in different locations, and the results were consistent in the virtual environment. The IP version made no difference to attack success. We verify port scanning is also possible against IPFW and natd in FreeBSD in the virtualized environment. We are able to fill the 28231 ephemeral ports in under 10 seconds. We can achieve a success rate of over 90% if either the target waits for 10 minutes after the attacker connects and disconnects from the VPN server or the attacker consumes all of the private IPs by repeatedly connecting and disconnecting from the VPN server, in which case the target is assigned the attacker's previously assigned private IP address.

We were unable to perform port scanning against PF and IPFILTER. The default behavior randomizes source ports which means we cannot scan specific services. Furthermore, the ranges are restricted to above port 1024 so the privileged ports cannot be accessed by default. We were also unable to get sequences of packets, such as outgoing `SYN`, incoming `ACK` for PF and this appears to be the result of the logic used for transitioning entries between TCP states.

4.3 Attack Assumption Tests

The attacks require three key assumptions: The attacker knows the target's public IP address; the attacker knows the VPN server IP; and, the VPN server's entry and exit IP addresses are the same. The following sections provide results for our analysis of these assumptions and while we believe that a well-resourced attacker (*e.g.*, state-sponsored) could easily satisfy them, our analysis provides more concrete justifications for each case.

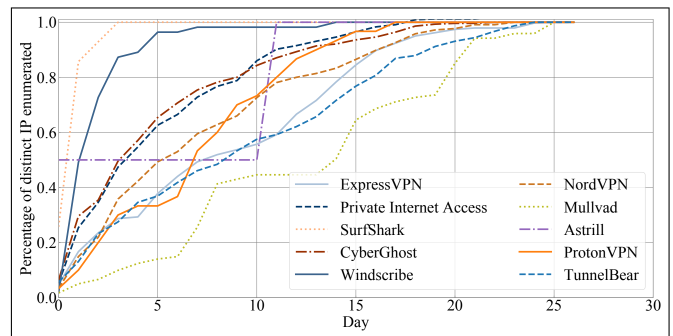


Figure 7: Variance in the VPN server selection when connected in default setting and from the same location.

4.3.1 Target IP. Most of the attacks assume that the attacker knows the target's public IP address. An adversary analyzing traffic at their national border may be able to harvest this information if the client has applications installed on their device that transmit sensitive or identifiable information that the attacker can correlate to the target's public IP address. Researchers have shown how applications commonly leak sensitive information about users [32, 33, 49]. Leaks occur directly due to normal software operation, by third-party advertising network behavior, through crash reporting, or by periodically sending telemetry back to a server in the application's country of origin. Adversaries sitting on the server-side or operating on-path as a firewall can passively acquire the target's public IP, even if they use a VPN or proxies. For example, Ramesh et al. found 26 VPN providers leak the client's public IP during tunnel failure [43]. An intelligence service can combine these information leaks to collect the target's public IP address.

4.3.2 *VPN server IP.*
To launch the ATIP, decapsulation, and eviction reroute attacks, the attacker needs to know the VPN server IP to which the target connects. In addition to collecting this information at the national border when the target connects to VPN services, the target location may also permit an attacker to learn the VPN server IP. We observe that knowing the name of the VPN provider and target's geographic location can ease this requirement, as VPN providers often select servers for clients to connect to based on geographic location to optimize performance. We examine the feasibility of the attacker knowing the VPN server IP given the name of the VPN provider and target's geolocation by evaluating the variance in server selection when a person connects 1) in the default way and 2) from the same location. Specifically, we connect through a VPN provider using the default setting[4] every 20 minutes and log the VPN server IP upon a successful connection. We repeat this process for 28 days with the 10 most popular VPN providers [45]. We collected 13,925 valid test results. As shown in Figure 7, even for popular VPN providers with large selections of servers, an adversary can feasibly guess the target's VPN IP by enumeration. For 7 out of 10 VPN providers, we were able to enumerate, in 12 days, over 80% of all servers we would see over the entire testing period of 28 days, suggesting that VPN server selections may not vary much over time. With the server variance of the top 10 VPNs being $min = 2, avg = 184, max = 383$ when connected from the same location an intelligence service could enumerate the target's VPN server IPs.

4.3.3 *VPN entry IP and exit IP.*
ATIP, connection inference, port scan, and port-forward overwrite attacks require that the VPN servers targeted do not deploy a "multi-hop" architecture. Crucial to these attacks is the requirement that the VPN's entry IP must be the same as the exit IP of the decapsulated packets. We analyzed measurement data collected from the VPNalyzer project [43] to determine how prevalent this deployment configuration is. VPNalyzer collects the exit IP of a VPN connection through IP lookup and the entry IP via client-side packet capture. Out of 569 valid measurements, 313 of them from 54 unique VPN providers suggest that the entry and exit share the same IP. Furthermore, we tested the top 10 VPNs and found half of them use this configuration [45]. An attacker could target the offending VPN providers and successfully execute ATIP or one of the other attacks covered.

5 FORMAL MODEL

We built a formal model in TLA+ [35] based on our analysis and relevant RFCs to explore the root cause of the vulnerabilities, implement, and validate proposed mitigations. Our analysis focuses on the degree to which connection tracking frameworks enforce process isolation. We test process isolation by reframing the problem in terms of a non-interference property and use model checking to verify whether or not non-interference holds. The model implements the network topology depicted in Figure 9. We implement two models, which include the VPN server, N, and three hosts, A and B, C, in the "public" network. The first design is consistent with the vulnerabilities we identified and we verify that non-interference is violated in those cases. After identifying root causes, we then develop and implement mitigations. Finally, we verify that non-interference holds in the fixed cases.

N can execute four actions, *Connect*, *Disconnect*, *PublicToPrivate*, and *PrivateToPublic*. A and B are identified by host marker variables, $H1$ and $H2$, associated with hosts A and B, respectively and are independent of the IP addresses. They are moved between the *FreeHosts* and *UsedHosts* lists based on whether *Connect* or *Disconnect* is called. The model saves the variables $A - Marker = H1$ and $B - Marker = H2$ which facilitate tracking information flow between hosts. The model tracks two possible private IP addresses, A' and B', in the *FreeIPs* and *UsedIPs* variables. The model stores associations between host marker and private IP using the *Connections* map. Connections are tracked in T. Each entry stores the outgoing packet 4-tuple information in the *origin* variable, the incoming packet's 4-tuple information in the *reply* variable, and the host marker in the *host_marker* variable which is set to the host marker of the host who created the entry. § A includes the model pseudo-code.

Connect creates an association between private IP and host marker by removing an IP from the *FreeIPs* list and storing it in the *UsedIPs* list. The host marker is moved from the *FreeHosts* to *UsedHosts* in the same way. The IP-*host_marker* association is stored in the *Connections* map. The model uses the host marker to track information flow by identifying when a host receives a packet that should go to a different host.

Disconnect removes the private IP-*host_marker* association by deleting the mapping from the *Connection* map, removing the IP from *UsedIPs* and storing it in *FreeIPs* and removing the *host_marker* from *UsedHosts* and storing it in *FreeHosts*.

PrivateToPublic models outgoing NAT behavior. It generates an outgoing packet, updates T based on the packet's fields, and places a response in *SendQueue*. T's entries store *orig* and *reply* information and a *host_marker* set to the packet's *host_marker*. Evictions are modeled here because only outgoing packets can create entries and a FIFO eviction policy is applied.

PublicToPrivate models public-to-private NAT by popping a packet from *SendQueue* and either using T to test whether the packet is routable or checks for a connection to its listening port. It performs a correctness check to see whether the packet's *host_marker* matches the entry's *host_marker*. If it does then non-interference holds, otherwise, the response packet will be routed to a host other than origin who elicited the incoming response.

5.1 Correctness

We define a correct connection tracking framework as one that enforces process isolation. We define a non-interference property using two invariants based on the two host markers, $A - Marker$ and $B - Marker$ as follows:

$$A - Marker = H1 \wedge B - Marker = H2$$

The invariant fails if either A or B's marker ever changes, indicating incorrect routing. This violates process isolation because one host will receive packets that should be routed to a different host.

5.1.1 *Model Checking.*
We perform bounded model checking using up to 2,000,000 state transitions to test whether the non-interference property holds. We verify that it does not in the first model and that it does in the second model.

[4] Or "recommended", "smart location", or "USA"/"New York" if a location needs to be specified.

Figure 9 in the § A.2 shows when non-interference is violated for the ATIP, connection inference, and port-forwarding overwrite attacks. A sends a packet $\langle A' : L, B : b \rangle$ to B and N adds an entry to T. B then sends $\{B : b, N : L\}$ to establish a VPN connection. N routes B's packet to A' instead of processing it directly because *PublicToPrivate* processes the packet before the code to associate it with a private IP. This causes $A - Marker$ to change from $H1$ to $H2$. This highlights two shared resources, the port space in the "public" IP address space, and the public IP address used by A' and B. If we select a new source port for A' when it attempts to use an allocated port, then $A - Marker$ remains unchanged and non-interference holds.

For the eviction reroute attack, B' sends a packet to S and expects a response as usual. A' triggers an eviction in T, causing B''s translation to be evicted before S sends its response. When N does receive the response it incorrectly routes it to A' instead of to B'. This attack highlights the shared public and private IP address space. Because A' replaces B''s entry, the mapping between hosts changes. The host-marker in the entry is copied to the incoming packet which ultimately changes $A - Marker$, violating non-interference. Evictions are necessary to prevent the table from filling up and to allow new connections to be created. If evictions do not occur, then a denial-of-service will happen. If we limit the number of connections per-host, then the eviction does not happen and non-interference holds.

For the port scanning attack, when A' sends packets to C in step 1, N creates a connection tracking entry. In step 2, A' disconnects, then B connects to N in step 3, and is assigned A's private IP address, A', in step 4. When C sends packets to N matching the entry created in step 1, they are routed to A', which maps to B. Because A, not B, created the mapping that permits this routing, B's host-marker is changed, violating non-interference. Removing entries from T after a host disconnects mitigates this attack.

Decapsulation occurs because B''s routing subsystem forwards the outgoing packet directly to N instead of first performing encapsulation. This violates process isolation because connections within the tunnel are separate from connections in the public Internet. We mitigate this attack by forcing packets to be processed by the encapsulation code first.

6 MITIGATIONS

Table 2 summarizes the six attack scenarios presented (rows), their corresponding root causes (columns two-five), and mitigations (column six; *M1*, *M2*, *M3*, *M4*, *M5*, *M6*). The vulnerabilities identified demonstrate how connection tracking frameworks are ineffective at enforcing process isolation for VPNs. The primary factors we identified are the following five shared, limited resources within connection tracking frameworks: The ephemeral port space on the public side of the network; the private IP addresses assigned to VPN clients in the private network; the connection tracking table; the public IP address of the VPN server; and, the specific 5-tuple used to track a connection (source and destination IPs and ports, and transport layer protocol). The fifth shared resource is subtle, because this tuple, which defines a process, is a consequence of the VPN's shared public IP address and ports. It would not otherwise be a shared resource leading to a failure of process isolation except for the fact that every connection has a direction based on who initiated the connection. Connections in opposite directions for the same tuple contend for this resource, leading to *Directional Ambiguity*. We now discuss the five mitigations we implemented and tested in our model.

***M1*. Allocated Port Restriction.** Ports associated with services on the VPN can be overwritten if the connection tracking framework does not coordinate with the operating system. Process isolation is violated because a host can receive packets for connections it did not initiate. Using a rule such as, `# iptables -t nat -A POSTROUTING -p udp -o enp0s8 –sport 1194 -j SNAT –to-source 192.168.2.254:32768-60999`, mitigates ATIP. A systematic fix involves properly coordinating port allocation between the network stack in the kernel and the connection tracking framework mitigates this issue, though this would break modularity. It is not sufficient to mitigate connection inference though because an attacker can still infer ports in use. Placing a range bound on the source ports, as done in PF and IPFILTER, partially resolves this problem unless the listening/forwarded port is in the range. In the case of IPFILTER in FreeBSD, the default behavior is to restrict the source port space to well outside the privileged port space (10,000). They also randomized the source port, unlike IPFW and natd, making it more secure by default. It is possible to redefine the port space to include the ephemeral range and disable randomized source port selection but the admin would have to go out of their way to do this.

***M2*. Static Private IP Assignment.** Port scanning occurs because the private IP address can be assigned to two different hosts over time. If T contains stale entries and a host is assigned a private IP associated with one of them, any incoming packets matching the entry will be routed to that host, violating process isolation. Statically assigning private IP addresses mitigates this attack. In practice, OpenVPN's `ifconfig-pool-persist` option can aid in mitigation. This is not a complete mitigation though because the number of private IP addresses is finite, so it is likely the connection tracking framework will have to use a hash function or similar random process to manage and assign private IP addresses which introduces other security implications that are out of scope for this work. Port scanning will not work if the VPN strictly controls the association between private IP addresses and logical hosts. For WireGuard because the private IP address is configured in the client's script, an attacker may be able to enumerate the private IP address more easily than if they are assigned dynamically as in OpenVPN or OpenConnect. Using namespaces, which mitigates the decapsulation attack, does not mitigate the port scan attack, though it is restricted for TCP as the incoming attacker packet illicits an RST from the target.

***M3*. Per-host Connection Limit.** Eviction rerouting is possible when the attacker can fill the table either to capacity or to when evictions begin. This violates process isolation because one host can affect how the connection tracking framework routes packets to others. Limiting the number of concurrent connections per host makes these attacks more expensive but an attacker with enough IP addresses or VPN accounts may still succeed.

***M4*. Orphan Entry Flush.** Port scanning occurs because entries in T can persist after the host who created them disconnects from the VPN. Removing a host's entries after they disconnect from the VPN mitigates this attack because there is no route back to the host. This can be accomplished by running the command `$ sudo conntrack -D –src=PRIVATE_IP` after a client disconnects.

	Shared Public Port	Shared Private IP Address	Shared Table	Shared Public IP Address	Directional Ambiguity	Mitigation
ATIP	✓		✓	✓		*M1*, *M6*
Connection Inference	✓		✓	✓		*M1*, *M6*
Port Forward Overwrite	✓		✓	✓		*M1*, *M6*
Decapsulation				✓		*M5*
Eviction Reroute		✓	✓	✓		*M3*
Port Scan		✓		✓	✓	*M2*, *M4*

Table 2: Root causes for each attack. ✓indicates the specified root cause (column) contributes to the specified attack (row). The last column represents one of the six mitigations: *M1* Allocated port restriction, *M2* Static Private IP Assignment, *M3* Per-host Connection Limit, *M4* Orphan Entry Flush, *M5* Routing Precedent, and *M6* Public IP Management.

***M5*. Routing Precedent.** When the client's network stack routes packets, it consults the routing table to determine where to forward the packet. If the route is very specific, as is the case with many VPNs, the routing code is invoked instead of passing the packet to the VPN process. This violates process isolation because the inner packet represents a distinct connection. Sending outgoing packets to the VPN process first eliminates this attack. Using network namespaces, such as namespaced-openvpn [5], mitigates this attack.

***M6*. Public IP Management** The port shadow is possible when the entry and exit IP addresses of the VPN server are the same. RFC [26] briefly mentions problems with public and private IP and port management in NAT, but we did not encounter any "best practices" during this work. This includes during the disclosure processes where, during discussions, no specific changes to our configuration were suggested to us. To fix this, instead of using a simple `MASQUARADE` rule in iptables, we configure a `SNAT` rule for all VPN packets leaving the network to have an IP of a second network interface on the VPN server. `# iptables -t nat -A POSTROUTING -o enp0s8 -s 10.0.0.0/8 -j SNAT -to-source 192.168.1.133`. This mitigates the ATIP attack and connection inference attacks.

7 RELATED WORK

Previous work has investigated VPN security, focusing mostly on the encryption, protocol, software, and commercial ecosystem. Applebaum et al. [12] studied VPNs from an adversary's perspective and pointed out that cryptographic security must be considered in the presence of routing. They observe the security critical nature of maintaining a secure routing table. They do not mention the connection tracking framework as used by most modern VPNs which reinforces the subtle issues arising from retrofitting systems with different threat models. While they consider multiple attack scenarios, they do not consider a hostile, adjacent VPN client which, as we have shown, is a significant threat. Researchers have also looked into VPN apps, finding evidence of traffic redirection, manipulation, and privacy leaks in VPN apps on mobile devices [28, 31, 40, 43, 52].

NAT security researchers have passively estimated the number of hosts behind a NAT using either IPIDs [13], ephemeral ports [38], or clock skews [34]. Gilad et al. [20] showed that IP fragmentation can be exploited to launch DoS or interception attacks against victims behind NAT. Qian et al. [41] presented multiple attacks on network firewalls with varying threat models, including malware on the victim's machine.

To demonstrate our ATIP attack, we employed the attack presented by Tolley et al. [44]. They demonstrated that *blind in/on-path* attackers can learn the IP a host behind a VPN is communicating with and hijack connections supposedly protected by the tunnel. Our ATIP attack enables an attacker to escalate to blind in-path from an initially off-path position. Research that violates process isolation in a purely off-path manner [14, 21, 24, 25, 37] is both inspiration for and complementary to our own work.

Xue et al.'s "tunnelcrack" attack [51] and the Perfect Privacy's Port Fail attack [7] are related to Appelbaum et al.'s attack against routing rules [12] that causes the VPN client to send packets directly to the VPN server. This is similar to our decapsulation attack. However, the difference between our attack and theirs is that they can only target one server for which traffic is sent unencrypted while our decapsulation attack can target multiple servers.

Researchers have found process isolation weaknesses, such as injecting untrusted data from the data plane to the control plane in SDNs[46, 47, 50]. By contrast, our attacks focus on the broad impact that design flaws between VPNs and stateful connection tracking have on process isolation.

8 ETHICAL DISCLOSURE

We followed a conservative ethical disclosure process aligned with US-CERT. We notified OpenVPN 45 days before disclosing our initially successful exploits against it. We notified WireGuard initially of our OpenVPN findings at the same time and indicated they might be vulnerable. We confirmed the attacks worked against WireGuard and that eviction reroute works against OpenVPN and WireGuard prior to submission of this work and notified them of this result. This work received CVE-2021-3773 [8]. We did not develop proof-of-concept exploits for other VPN server implementations (*e.g.*, strongSwan or SoftEther) because they are not used by major VPN vendors and the problems we identified are fundamental to how connection tracking and network address translation are defined in RFCs and not any specific VPN server implementation's code.

9 CONCLUSION

Connection tracking frameworks were not originally designed with the same threat model as VPNs, and the consequences of their use in VPNs was an open question before this work. Our attacks were motivated by this knowledge gap. The four attacks we developed and the case studies demonstrate how an attacker can use simple port and IP collisions to build a powerful exploit primitive with broad security implications for VPNs.

Using our first model, we identified five shared, limited resources that contribute to violations of process isolation. We validated this by testing a non-interference property based on how information flows between hosts through the connection tracking framework. We used these insights to design and validate six mitigations in the second model. We found that the mitigations address most, but not all, of the weaknesses. This analysis revealed the challenges, trade-offs, and fundamental limitations that connection tracking frameworks have when enforcing process isolation in VPNs.

ACKNOWLEDGMENTS

The authors would like to thank Esther Rodriguez for her work reverse engineering android apps that gave us insights about attacker capabilities. We thank Beau Kujath and William Tolley for the many discussions about VPN configuration, attacks, and these NATs, as well as for helping with the ethical disclosure process. We would also like to thank Antonio Espinoza for his feedback during the drafting process. This material is based upon work supported by the National Science Foundation under Grant Numbers CNS-2237552, CNS-2141512, CNS-2007741, and CNS-2141547; and, the Defense Advanced Research Projects Agency (DARPA) under Agreement HR00112190127, and the Open Technology Fund's Information Controls Fellowship Program and Internet Freedom Fund. Finally, we would like to thank the anonymous reviewers for their helpful feedback.

REFERENCES

[1] 1980. User Datagram Protocol. RFC 768. https://doi.org/10.17487/RFC0768
[2] 1981. Transmission Control Protocol. RFC 793. https://doi.org/10.17487/RFC0793
[3] 2020. How to set up and configure an Openvpn server on Ubuntu 20-04. https://www.digitalocean.com/community/tutorials/how-to-set-up-and-configure-an-openvpn-server-on-ubuntu-20-04
[4] 2024. How To Set Up WireGuard on Ubuntu 20.04. https://www.digitalocean.com/community/tutorials/how-to-set-up-wireguard-on-ubuntu-20-04
[5] 2024. namespaced-openvpn. https://github.com/slingamn/namespaced-openvpn
[6] 2024. OpenVPN How-To: Quick Start. https://openvpn.net/community-resources/how-to/#openvpn-quickstart
[7] 2024. Port Fail. https://www.perfect-privacy.com/en/blog/ip-leak-vulnerability-affecting-vpn-providers-with-port-forwarding
[8] 2024. Port Shadow. https://nvd.nist.gov/vuln/detail/CVE-2021-3773
[9] 2024. Recipes for Openconnect VPN. https://ocserv.openconnect-vpn.net/recipes.html
[10] 2024. Set Up OpenConnect VPN Server (ocserv) on Ubuntu 20.04 with Let's Encrypt. https://www.linuxbabe.com/ubuntu/openconnect-vpn-server-ocserv-ubuntu-20-04-lets-encrypt
[11] 2024. WireGuard: Quick Start. https://www.wireguard.com/quickstart/
[12] Jacob Appelbaum, Marsh Ray, Karl Koscher, and Ian Finder. 2012. vpwns: Virtual pwned networks. In *2nd USENIX Workshop on Free and Open Communications on the Internet. USENIX Association*.
[13] Steven M Bellovin. 2002. A technique for counting NATted hosts. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet Measurment*. 267–272.
[14] Yue Cao, Zhiyun Qian, Zhongjie Wang, Tuan Dao, Srikanth V Krishnamurthy, and Lisa M Marvel. 2018. Off-Path TCP Exploits of the Challenge ACK Global Rate Limit. *IEEE/ACM Transactions on Networking* 26, 2 (2018), 765–778.
[15] Edmund Clarke, Orna Grumberg, and Doron Peled. 2001. *Model Checking*. MIT Press.
[16] Zakir Durumeric, Zane Ma, Drew Springall, Richard Barnes, Nick Sullivan, Elie Bursztein, Michael Bailey, J. Halderman, and Vern Paxson. 2017. The Security Impact of HTTPS Interception. https://doi.org/10.14722/ndss.2017.23456
[17] Kjeld Borch Egevang and Paul Francis. 1994. The IP Network Address Translator (NAT). RFC 1631. https://doi.org/10.17487/RFC1631
[18] Kjeld Borch Egevang and Pyda Srisuresh. 2001. Traditional IP Network Address Translator (Traditional NAT). RFC 3022. https://doi.org/10.17487/RFC3022
[19] Bryan Ford, Saikat Guha, Kaushik Biswas, Senthil Sivakumar, and Pyda Srisuresh. 2008. NAT Behavioral Requirements for TCP. RFC 5382. https://doi.org/10.17487/RFC5382
[20] Yossi Gilad and Amir Herzberg. 2013. Fragmentation Considered Vulnerable. *ACM Trans. Inf. Syst. Secur.* 15, 4, Article 16 (apr 2013), 31 pages. https://doi.org/10.1145/2445566.2445568
[21] Yossi Gilad, Amir Herzberg, and Haya Shulman. 2014. Off-Path Hacking: The Illusion of Challenge-Response Authentication. *IEEE Security & Privacy* 12, 5 (2014), 68–77. https://doi.org/10.1109/MSP.2013.130
[22] J. A. Goguen and J. Meseguer. 1982. Security Policies and Security Models. In *1982 IEEE Symposium on Security and Privacy*. 11–11. https://doi.org/10.1109/SP.1982.10014
[23] Tony L. Hain. 2000. Architectural Implications of NAT. RFC 2993. https://doi.org/10.17487/RFC2993
[24] Amir Herzberg and Haya Shulman. 2013. Fragmentation Considered Poisonous, or: One-domain-to-rule-them-all.org. In *2013 IEEE Conference on Communications and Network Security (CNS)*. 224–232. https://doi.org/10.1109/CNS.2013.6682711
[25] Amir Herzberg and Haya Shulman. 2013. Socket Overloading for Fun and Cache-Poisoning. In *Proceedings of the 29th Annual Computer Security Applications Conference* (New Orleans, Louisiana, USA) *(ACSAC '13)*. Association for Computing Machinery, New York, NY, USA, 189–198. https://doi.org/10.1145/2523649.2523662
[26] Matt Holdrege and Pyda Srisuresh. 1999. IP Network Address Translator (NAT) Terminology and Considerations. RFC 2663. https://doi.org/10.17487/RFC2663
[27] Matt Holdrege and Pyda Srisuresh. 2001. Protocol Complications with the IP Network Address Translator. RFC 3027. https://doi.org/10.17487/RFC3027
[28] Muhammad Ikram, Narseo Vallina-Rodriguez, Suranga Seneviratne, Mohamed Ali Kaafar, and Vern Paxson. 2016. An analysis of the privacy and security risks of android vpn permission-enabled apps. In *Proceedings of the 2016 internet measurement conference*. 349–364.
[29] Cullen Jennings and Francois Audet. 2007. Network Address Translation (NAT) Behavioral Requirements for Unicast UDP. RFC 4787. https://doi.org/10.17487/RFC4787
[30] Jan Just Keijser. 2011. *OpenVPN 2 Cookbook*. Packt.
[31] Mohammad Taha Khan, Joe DeBlasio, Geoffrey M Voelker, Alex C Snoeren, Chris Kanich, and Narseo Vallina-Rodriguez. 2018. An empirical analysis of the commercial vpn ecosystem. In *Proceedings of the Internet Measurement Conference 2018*. 443–456.
[32] Jeffrey Knockel, Zoë Reichert, and Mona Wang. 2023. *"Please do not make it public": Vulnerabilities in Sogou Keyboard encryption expose keypresses to network eavesdropping*. Technical Report.
[33] Jeffrey Knockel, Adam Senft, and Ronald Deibert. 2016. Privacy and Security Issues in BAT Web Browsers. In *6th USENIX Workshop on Free and Open Communications on the Internet (FOCI 16)*.
[34] Tadayoshi Kohno, Andre Broido, and Kimberly C Claffy. 2005. Remote physical device fingerprinting. *IEEE Transactions on Dependable and Secure Computing* 2, 2 (2005), 93–108.
[35] Leslie Lamport. 1994. The Temporal Logic of Actions. *ACM Trans. Program. Lang. Syst.* 16, 3 (May 1994), 872–923. https://doi.org/10.1145/177492.177726
[36] Dr. Arthur Y. Lin, Andrew G. Malis, Dr. Juha Heinanen, Bryan Gleeson, and Dr. Grenville Armitage. 2000. A Framework for IP Based Virtual Private Networks. RFC 2764. https://doi.org/10.17487/RFC2764
[37] Bill Marczak and John Scott-Railton. 2016. *The million dollar dissident: NSO group's iPhone zero-days used against a UAE human rights defender*. Technical Report. Citizen Lab.
[38] Sophon Mongkolluksamee, Kensuke Fukuda, and Panita Pongpaibool. 2012. Counting NATted hosts by observing TCP/IP field behaviors. In *2012 IEEE International Conference on Communications (ICC)*. IEEE, 1265–1270.
[39] Robert Moskowitz, Daniel Karrenberg, Yakov Rekhter, Eliot Lear, and Geert Jan de Groot. 1996. Address Allocation for Private Internets. RFC 1918. https://doi.org/10.17487/RFC1918
[40] Vasile C. Perta, Marco V. Barbera, Gareth Tyson, Hamed Haddadi, and Alessandro Mei. 2015. A Glance through the VPN Looking Glass: IPv6 Leakage and DNS Hijacking in Commercial VPN clients. *Proceedings on Privacy Enhancing Technologies* 2015, 1 (2015), 77–91. https://doi.org/10.1515/popets-2015-0006
[41] Zhiyun Qian and Z Morley Mao. 2012. Off-path TCP sequence number inference attack-how firewall middleboxes reduce security. In *2012 IEEE Symposium on Security and Privacy*. IEEE, 347–361.
[42] Ram Sundara Raman, Leonid Evdokimov, Eric Wurstrow, J Alex Halderman, and Roya Ensafi. 2020. Investigating large scale HTTPS interception in Kazakhstan. In *Proceedings of the ACM Internet Measurement Conference*. 125–132.
[43] Reethika Ramesh, Leonid Evdokimov, Diwen Xue, and Roya Ensafi. 2022. VPNalyzer: Systematic Investigation of the VPN Ecosystem. In *Network and Distributed System Security*.
[44] William J. Tolley, Beau Kujath, Mohammad Taha Khan, Narseo Vallina-Rodriguez, and Jedidiah R. Crandall. 2021. Blind In/On-Path Attacks and Applications to VPNs. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association. https://www.usenix.org/conference/usenixsecurity21/presentation/tolley
[45] top10 2023. Top10VPN: VPN Reviews. https://www.top10vpn.com/.
[46] Benjamin E Ujcich, Samuel Jero, Anne Edmundson, Qi Wang, Richard Skowyra, James Landry, Adam Bates, William H Sanders, Cristina Nita-Rotaru, and Hamed Okhravi. 2018. Cross-app poisoning in software-defined networking. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*. 648–663.
[47] Benjamin E Ujcich, Samuel Jero, Richard Skowyra, Steven R Gomez, Adam Bates, William H Sanders, and Hamed Okhravi. 2020. Automated discovery of cross-plane event-based vulnerabilities in software-defined networking. In *Network and Distributed System Security Symposium*.
[48] ValdikSS. 2023. Encrypted traffic interception on Hetzner and Linode targeting the largest Russian XMPP (Jabber) messaging service. (2023).
[49] Eline Vanrykel, Gunes Acar, Michael Herrmann, and Claudia Diaz. 2017. Leaky birds: Exploiting mobile application traffic for surveillance. In *Financial Cryptography and Data Security: 20th International Conference, FC 2016, Christ Church,*

Barbados, February 22–26, 2016, Revised Selected Papers 20. Springer, 367–384.
[50] Feng Xiao, Jinquan Zhang, Jianwei Huang, Guofei Gu, Dinghao Wu, and Peng Liu. 2020. Unexpected data dependency creation and chaining: A new attack to SDN. In *2020 IEEE symposium on security and privacy (SP)*. IEEE, 1512–1526.
[51] Nian Xue, Yashaswi Malla, Zihang Xia, Christina Pöpper, and Mathy Vanhoef. 2023. Bypassing Tunnels: Leaking VPN Client Traffic by Abusing Routing Tables. In *32nd USENIX Security Symposium (USENIX Security 23)*. 5719–5736.
[52] Qi Zhang, Juanru Li, Yuanyuan Zhang, Hui Wang, and Dawu Gu. 2017. Oh-Pwn-VPN! security analysis of OpenVPN-based Android apps. In *International Conference on Cryptology and Network Security*. Springer, 373–389.

A APPENDIX

A.1 Attacks

Figure 8, described in § 3, depicts N's connection tracking table when B connects to N before A. The porting collision causes N to select a new source port for A. A exploits the fact that the port space is a shared resource in this context to infer the existence of the (B, N) VPN connection. Other connections between N and other hosts can be inferred using a similar process.

VPN Connection Tracking Table

	Orig	Reply
1. B's VPN Connection	{3.3.3.3 : 300, 2.2.2.2: 1194}	{2.2.2.2:1194, 3.3.3:300}
2. A's VPN Connection	{1.1.1.1 : 100, 2.2.2.2: 1194}	{2.2.2.2:1194, 1.1.1.1:100}
3. A's entry that collides with B's VPN connection	{10.0.0.2 : 1194, 3.3.3.3: 300}	{ 3.3.3.3:111, 2.2.2.2:1194 }

Figure 8: N's connection tracking table for the connection inference. 1. B's VPN connection, 2. A's VPN connection. 3. A's packet that collides with B's VPN connection.

A.2 Formal Model

Figure 9 depicts the model state before and after the ATIP attack is run.

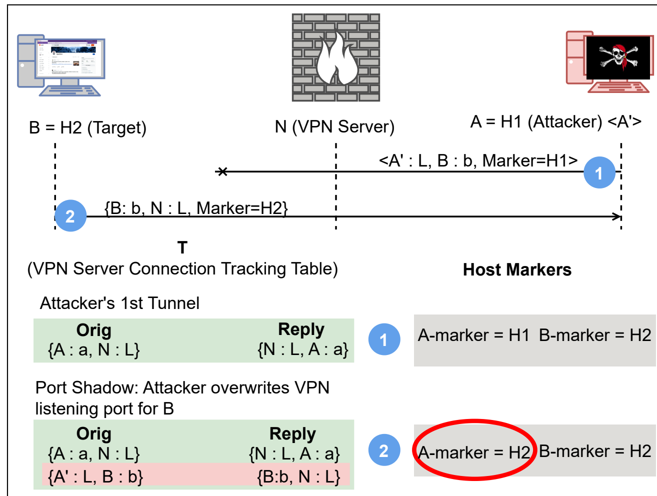


Figure 9: Adjacent-to-in-path attack represented in the formal model.

A.2.1 *Model Pseudocode.*

The following code listings are our model's pseudocode. The code was originally written in PlusCal, a wrapper for the TLA+ modeling language. The first four functions are the pseudo code for the vulnerable VPN scenario. The second four functions are pseudocode for the fixed versions of the same functions.

Algorithm 1 Connect(host, ip)

```
if LenFreeIPs > 0 ∧ LenFreeHosts > 0 then
    FreeIPs := SelectSeq(FreeIPs,
            LAMBDA e: (e≠ip));
    UsedIPs := Append(UsedIPs, ip);
    FreeHosts := SelectSeq(FreeHosts,
            LAMBDA e: (e≠host));
    UsedHosts := Append(UsedHosts, host);
    Connections := Append(Connections, << ip, host >>);
end if
return
```

Algorithm 2 Disconnect(host, ip)

```
if LenUsedIPs > 0 then
    Connections := SelectSeq(Connections,
            LAMBDA c: (Head(c)≠ip));
    UsedIPs := SelectSeq(UsedIPs, LAMBDA c: (c≠ip));
    FreeIPs := Append(FreeIPs, ip);
    FreeHosts := Append(FreeHosts, host);
    UsedHosts := SelectSeq(UsedHosts,
            LAMBDA a: (a≠host));
end if
return
```

Algorithm 3 PrivateToPublic(depth)

```
call EventSequence(depth);
if LenConnections > 0 then
    sourcePort ∈ Ports
    conn ∈ Connections
    destPort ∈ Ports
    daddr ∈ hosts
    hostMarker := Head(Tail(conn))
    pkt := [saddr → Head(conn), sport → sourcePort,
        daddr → daddr, dport → destPort,
        host_marker → hostMarker]
    entry := [host_marker → hostMarker,
        orig → [saddr → pkt.saddr, sport → pkt.sport,
            daddr → pkt.daddr, dport → pkt.dport],
        reply → [saddr → pkt.daddr, sport → pkt.dport,
            daddr → N, dport → pkt.sport ]];
    otherEntry := SelectSeq(T,
        LAMBDA k: k.reply.saddr=pkt.daddr ∧
            k.reply.sport=pkt.dport ∧
            k.reply.daddr=N ∧ k.reply.dport=pkt.sport);
    if LenotherEntry > 0 then
        T := SelectSeq(T,
            LAMBDA e: ¬(e.reply.saddr=pkt.daddr ∧
                e.reply.sport=pkt.dport ∧
                e.reply.daddr=N ∧ e.reply.dport=pkt.sport) );
    end if
    T := Append(T, entry);
    pkt := [saddr →pkt.daddr, sport → pkt.dport,
        daddr → N, dport → pkt.sport,
        host_marker → hostMarker];
    SendQueue := Append(SendQueue, pkt);
    return
end if
```

Algorithm 4 PublicToPrivate()

```
if LenSendQueue > 0 then
    pkt := Head(SendQueue);
    SendQueue := Tail(SendQueue);
    if LenT > 0 then
        entry := SelectSeq(T, LAMBDA e:
                e.reply.saddr=pkt.saddr ∧
                e.reply.sport=pkt.sport ∧
                e.reply.daddr=pkt.daddr ∧
                e.reply.dport=pkt.dport);
        if Lenentry <= 0 then
            assert(FALSE);
        end if
        entry := Head(entry);
        if entry.host_marker= pkt.host_marker then
            assert(entry.host_marker/=pkt.host_marker);
        end if
        conn := SelectSeq(Connections, LAMBDA c:
                entry.orig.saddr = Head(c));
        if Lenconn > 0 then
            conn := Head(conn);
            hostMarker := conn[2];
            assert( hostMarker = entry.host_marker);
        end if
    end if
end if
return
```

Algorithm 5 ConnectFixed(depth)

```
call EventSequence(depth);
if LenFreeHosts > 0 then
    host_idx := DOMAIN FreeHosts;
    hidx := CHOOSE h ∈ host_idx : TRUE;
    host := FreeHosts[hidx];
    FreeHosts := SelectSeq(FreeHosts,
            LAMBDA a: a ≠ host);
    UsedHosts := Append(UsedHosts, host);
    port_idx := DOMAIN Ports;
    pidx := CHOOSE p ∈ port_idx : TRUE;
    pkt := [ saddr → host, sport → Ports[pidx],
            daddr → N, dport → N,
            cmd → CmdConnect,
            host_marker → host];
    SendQueue := Append(SendQueue, pkt);
end if
return;
```

Algorithm 6 DisconnectFixed(depth)

```
call EventSequence(depth);
if LenConnections > 0 then
    connDomain := DOMAIN Connections
    cidx := CHOOSE c ∈ connDomain : TRUE
    conn := Connections[cidx]
    ip := conn[1]
    host := conn[2]
    Connections := SelectSeq(Connections,
            LAMBDA cc: Head(cc)≠ip)
    UsedIPs := SelectSeq(UsedIPs,
            LAMBDA ccc: ccc≠ip)
    FreeIPs := Append(FreeIPs, ip)
    disconnectPurgeOrphans1: T := SelectSeq(T,
            LAMBDA e: e.orig.saddr ≠ ip)
    disconnectPurgeOrphans2: T := SelectSeq(T,
            LAMBDA e: e.orig.saddr ≠ host)
    if host = H1 then PortMap1 := <<>>
    else PortMap2 := <<>>
    end if
    FreeHosts := Append(FreeHosts, host)
    UsedHosts := SelectSeq(UsedHosts,
            LAMBDA m: m ≠ host)
end if
return
```

Algorithm 7 PublicToPrivateFixed(depth)

```
call EventSequence(depth);
if LenSendQueue > 0 then
    pkt := Head(SendQueue);
    SendQueue := Tail(SendQueue);
    entry := SelectSeq(T,
        LAMBDA e: e.reply.saddr=pkt.saddr ∧
            e.reply.sport=pkt.sport ∧
            e.reply.daddr=pkt.daddr ∧
            e.reply.dport=pkt.dport)
    if Lenentry <= 0 then
        if pkt.dport = N then
            if LenFreeIPs > 0 then
                ip ∈ FreeIPs
                FreeIPs := SelectSeq(FreeIPs, LAMBDA d: d ≠
ip)
                UsedIPs := Append(UsedIPs, ip)
                host := pkt.saddr
                Connections := Append(Connections,
<< ip, host >>)
            end if
        end if
    else
        entry := Head(entry);
        if entry.host_marker≠pkt.host_marker then
            (* EvictionReroute *)
            if pkt.host_marker = H1 then Marker1 := en-
try.host_marker
            else Marker2 := entry.host_marker
            end if
        end if
        (* PortScan *)
        conn := SelectSeq(Connections,
            LAMBDA e: entry.orig.saddr = Head(e));
        if Lenconn > 0 then
            conn := Head(conn);
            hostMarker := conn[2];
            if hostMarker = H1 then Marker1 := en-
try.host_marker;
            else Marker2 := entry.host_marker;
            end if
        end if
    end if
end if
return ;
```

Algorithm 8 PrivateToPublicFixed(depth)

```
call EventSequence(depth);
if LenConnections > 0 then
    sourcePort ∈ Ports
    conn ∈ Connections
    destPort ∈ Ports
    daddr ∈ DOMAIN hosts
    hostMarker := Head(Tail(conn))
    if hostMarker=H1 then
        if LenPortMap1 >= MaxPorts then return;
        else PortMap1 := Append(PortMap1, sourcePort)
        end if
    else
        if LenPortMap2 >= MaxPorts then return
        else PortMap2 := Append(PortMap2, sourcePort)
        end if
    end if
    if sourcePort=N then
        if hostMarker=H1 then sourcePort := EP1
        else sourcePort := EP2
        end if
    end if
    pkt := [saddr→Head(conn), sport→sourcePort,
              daddr→ daddr, dport→destPort,
              host_marker→ hostMarker]
    entry := [host_marker→hostMarker,
           orig→[saddr→pkt.saddr, sport → pkt.sport,
             daddr→pkt.daddr, dport → pkt.dport],
           reply → [saddr → pkt.daddr, sport → pkt.dport,
             daddr → N, dport → pkt.sport ]];
    otherEntry := SelectSeq(T, LAMBDA k:
            k.reply.saddr=pkt.daddr ∧ k.reply.sport=pkt.dport∧
k.reply.daddr=N
             ∧ k.reply.dport=pkt.sport ∧ k.hostMarker ≠ host-
Marker);
    if LenotherEntry > 0 then
        if LenExtraPorts > 0 then
            new_sport := Head(ExtraPorts);
            ExtraPorts := Tail(ExtraPorts);
            entry.reply.dport := new_sport
            pkt.sport := new_sport
        else PortSpaceFull := TRUE
        end if
    end if
    T:= Append(T, entry)
    if LenT >= MaxTableSize then TableFull := TRUE
    end if
    pkt := [saddr →pkt.daddr, sport → pkt.dport,
             daddr → N, dport → pkt.sport,
             host_marker → hostMarker]
    SendQueue := Append(SendQueue, pkt)
end if
return
```