FUZZVPN: Finding Vulnerabilities in OpenVPN

Anqi Chen
Northeastern University

Cristina Nita-Rotaru
Northeastern University

Abstract

OpenVPN is one of the most widely used VPN protocols, allowing for a connection to be securely proxied through another computer. Due to the protocol's critical role in securing communications, it is essential that OpenVPN remains robust against attacks. Previous work has discovered vulnerabilities in OpenVPN, revealing its susceptibility to denial of service, the potential for flow fingerprinting, and the risk of VPN protection being bypassed through operating system exploits or TCP connection hijacking.

In this work, we take a systematic approach to finding attacks by inferring the protocol's specification. We study OpenVPN configured with both the UDP and TCP variants. Given that no standard exists and specification is sparse, we first construct a detailed message sequence chart of the protocol handshake under the UDP and TCP modes, respectively. We use this information to perform systematic adversarial testing with malformed configurations, replay attacks, denial-of-service, resilience to acknowledgments-related attacks, and packet value modifications based on protocol semantics. We found several new attacks: two new denial-of-service attacks due to the replay of control and acknowledgment packets, the incorrect handling of input validation for 17 protocol configuration options, a scenario where due to an inconsistent view of the state of the connection, the server sends data prematurely to the client causing the client to ignore it, and a scenario where a malicious client configured with weaker authentication can degrade the performance of a victim client configured with stronger authentication.

1 Introduction

A Virtual Private Network (VPN) is a protocol that ensures privacy, security, and anonymity for users by masking their IP address and encrypting their internet traffic. This is achieved with the help of a VPN server that acts as an intermediary between the user's device and the internet. The communication between the user's device and the VPN server is encrypted and, for all the internet communication, the IP of the device is replaced with the IP address of the VPN server.

VPNs are used by individuals, businesses, and organizations to improve their online security and privacy. Individuals use VPNs for personal privacy and to prevent cyberattacks and data breaches. Business organizations employ VPNs to secure business communications and data transfers, when employees work remotely. Universities use VPNs to allow students and faculty to securely access internal network resources from off-campus locations. A user study with around 1000 Americans found that 95% of adults are now familiar with the technology, and 46% use VPNs [23]. A 2024 survey by Statista showed around 23.1% of internet users worldwide used a VPN [25].

Several VPN services are available such as ExpressVPN [28], NordVPN [34], Surfshark [42], or AnyConnect [2]. Many such VPN services use public protocols such as OpenVPN [36], WireGuard [9], Internet Key Exchange Version 2 (IKEv2) [13], and Layer 2 tunneling protocol (L2TP) [46]. [1] [2] Other VPN services use proprietary protocols such as the Secure Socket Tunneling Protocol (SSTP) [1] owned by Microsoft and the two protocols used by AnyConnect [2] owned by Cisco. Decentralized VPNs like Orchid [7], Mysterium [33], Boring [18], DeeperNetwork [27], and HOPR Network [30] aim to provide privacy and security benefits by using a distributed network of nodes maintained by individual users or independent operators.

Among the public VPN protocols, OpenVPN is one of the most popular; the 2024 user study showed that the top 2 popular VPN software are NordVPN and Proton VPN, which both support OpenVPN as one of the recommended underlying protocols [23]. Surprisingly, while highly popular, the OpenVPN protocol, unlike many other secure protocols, has not been standardized yet by the IETF. A work-in-progress RFC draft [41] is available but lacks many details such as a protocol message sequence chart, or description of the configura-

[1] Both IKEv2 and L2TP are often paired with IPsec.

[2] Older VPN protocols like Point-to-Point Tunneling Protocol (PPTP) [54] are notorious for security problems and are less used nowadays.

tion parameters. Most of the protocol description is scattered in several documents included in the OpenVPN repository [36]. Our target in this paper is the open-source OpenVPN project [36], which is known as the OpenVPN protocol [40], with over 50 million downloads. There are other commercial products by the OpenVPN company (the company behind the open source community project that provides commercial services and products), including the Access Server and 5 OpenVPN Connect Client implementations for different OS platforms (Windows, iOS, Android, Linux, MacOS).

OpenVPN operates over either TCP or UDP, supports many encryption algorithms including post-quantum protocols, provides perfect forward secrecy, and runs on many platforms, including Linux, Windows, macOS, iOS, Android, and FreeBSD. OpenVPN is open source with 68 versions [35], the most recent version being 2.6.14 released in April 2025.

As with all Internet protocols, security is a major concern for OpenVPN. Previous work analyzing the security of OpenVPN includes evaluating denial of service susceptibility [26], fingerprinting the VPN network flow [51], exploiting operating system vulnerabilities to bypass the VPN [52], and hijacking the TCP connection protected by VPN [45]. An older version of OpenVPN, version 2.4.0, was audited by Quarkslab, and the report [19] identified among other issues, two denial of service vulnerabilities and insecure configuration options. Some memory bugs were found by a project [49] testing OpenVPN with libFuzzer [31]. Finally, two works [20,50] that analyzed security and privacy deployment issues of several VPN software found a misconfigured “kill-switch" feature leading to traffic leakage for the OpenVPN Access Server [20] and several bugs in various OpenVPN client applications [50].

None of the aforementioned works with the exception of the version 2.4.0 Quarkslab security audit [19], which is seven years old, studied the OpenVPN protocol itself in depth. Several changes were made to the OpenVPN protocol since version 2.4.0, including adding protection to the denial of service attack found in [26].

In this work, we study the security of the OpenVPN protocol’s implementation, 2.6.12, by inferring the protocol’s specification. While OpenVPN uses TLS for part of its connection establishment, it has its own handshake protocol that includes more than just TLS messages. Since no RFC exists for the OpenVPN protocol and its description is sparse and incomplete, we first created a detailed Message Sequence Chart (MSC) of the handshake protocol. We focused on the OpenVPN over both the UDP and TCP variants, and we used both passive and active learning approaches to construct the specification. We first examined the captured packet trace of a normal connection in Wireshark to understand the OpenVPN packet structure and get a high-level knowledge of the packet sequence for a normal OpenVPN connection establishment. Once we learned the packet format, we employed a Man-in-the-Middle setup to actively control the message-sending progress, monitor the client and the server to understand the message retransmission behavior, and learn the content of TLS ciphertext messages from the verbose logs of the OpenVPN execution.

Next, we examine three main classes of protocol fuzzers: mutation-based (e.g., AFLNET [47]), generation-based (e.g., PEACH [29]), and proxy-based (e.g., BLEEM [53]). AFLNET [47] relies on reusing the initial seed, which suits plaintext protocols (e.g. FTP, RTSP) but fails when faced with OpenVPN’s session-specific encryption. PEACH [29] requires detailed protocol packet structures and state models, and requires code instrumentation to place the fuzzer in an appropriate location, difficult to achieve for OpenVPN due to its reliance on external cryptographic libraries. BLEEM [53] does not have its code publicly available. Thus, none of these protocol fuzzers are directly applicable to OpenVPN. These limitations motivated our design of FUZZVPN, a proxy-based, open-source fuzzer that not only detects memory bugs, but also supports protocol-level vulnerability discovery such as replay and denial-of-service attacks, as well as configuration file fuzzing.

Equipped with the MSC, we created FUZZVPN to perform systematic adversarial testing with malformed configurations, replay attacks, resilience to acknowledgment-related attacks, denial-of-service injection, and packet value modifications based on protocol semantics. First, we confirmed that the attack from [26] was fixed in version 2.6.12 under the UDP mode, however, the attack is still possible for the TCP mode. We also found several new denial-of-service attacks due to the replay of control and acknowledgment packets, incorrect handling of input validation in malformed configurations, a scenario where due to an inconsistent view of the state of the connection the server sends data prematurely causing the client to ignore it, and a scenario where a malicious client configured with weaker authentication can degrade the performance of a victim client configured with stronger authentication.

We are also the first to document and study acknowledgment packets in the OpenVPN handshake design. While OpenVPN supports protection of the control and acknowledgment packets through pre-configured keys, a recent paper [51] found 180,858 OpenVPN endpoints without such protection enabled. There is no description of the design goals for the acknowledgment mechanisms other than “**`P_ACK_V1`** – Acknowledgement for control channel packets received" on the official website of OpenVPN [40] and the OpenVPN development website [39]. The Quarkslab technical report [19] analyzing OpenVPN 2.4.0 mentions “It is notable that acknowledgment can either be done by a dedicated **`P_ACK_V1`** packet or by including it in P_CONTROL packets.". Our study found that actually both acknowledgment methods are provided at the same time. This redundancy made the protocol resilient to some of the attacks we tried against the control packets or acknowledgment packets when OpenVPN was configured in the UDP mode, however, we found several attacks

that closed the connection in the TCP mode.

Our contributions are summarized as follows:

- We created a detailed MSC of the OpenVPN handshake protocol in UDP and TCP mode respectively, using passive and active learning techniques.
- We created FUZZVPN which injects runtime attacks that consider protocol semantics such as content and order of messages in the protocol sequence. In addition to protocol-based attacks, FUZZVPN also considers malformed configurations. We run about 1000 scenarios that took about 5.5 hours for the UDP mode and about 6 hours for the TCP mode.
- We show that a previous denial of service attack involving the replay of one type of control packet, was fixed for OpenVPN 2.6.12 under the UDP mode, but the attack is still possible when OpenVPN is configured with the TCP mode.
- We found that the replay protection the *tls-auth* mode claims to activate worked in the UDP mode but not in the TCP mode.
- We analyzed OpenVPN's acknowledgment mechanisms and found that while the UDP mode is resilient to the attacks we conducted, that is not the case for TCP.
- We found several new attacks: 2 are denial of service through replaying of control or acknowledgment packets, 17 are improper handling of input option value in a malformed configuration, one where an inconsistent state between client and server results in the loss of data, and a scenario where a malicious client configured with weaker authentication can degrade the performance of a victim client configured with stronger authentication.

2 Background

In this section we provide a high-level description of the OpenVPN design, including specification and code availability. We then overview previous work on OpenVPN security.

2.1 OpenVPN Overview

OpenVPN is a popular VPN protocol that is designed to transport network traffic with confidentiality, authenticity, and integrity. When OpenVPN is enabled on a computer, a secure tunnel is established with the VPN server, and the network traffic can be encapsulated inside OpenVPN data packets in an encrypted form and forwarded through the tunnel.

Common VPN protocols (aside from IKEv2/IPsec) share some similarities in architecture: they require (1) a mechanism to process and encapsulate the traffic flowing through the VPN (typically through a NIC virtual interface), and (2) a method to ensure the desired traffic is correctly directed through the virtual interface such that encryption and decryption can be handled by the VPN software (typically through modifying the routing rules). After both the VPN client side and the server side have enabled the virtual interface and updated their routing rules, the VPN tunnel can forward encrypted traffic through securely.

Table 1: Comparison between VPN protocols

Protocol	Virtual Interface	Transport Layer	Open-sourced
OpenVPN	TUN or TAP	TCP or UDP	Yes
Wireguard	TUN	UDP	Yes
IKEv2/IPsec	implicit[1]	IKEv2 over UDP	some[3]
L2TP/IPsec	PPP	L2TP over UDP	some [2]
SSTP	PPP	TLS over TCP	No [4]

OpenVPN interaction with the OS. OpenVPN can be used to forward both the link layer (layer 2 of the OSI model) and network layer (layer 3 of the OSI model) traffic in the encrypted tunnel. For the link layer traffic (Ethernet), a TAP device (layer 2 virtual network device) will be created; while for the network layer (IP), a TUN device (layer 3 virtual network device) will be created. Wireguard protects only layer 3 (IP) packets and needs to create a virtual interface TUN device. IKEv2/IPsec protects only IP-level packets and may not create an explicit virtual interface since the functionality is integrated into the operating system network stack. L2TP/IPsec and SSTP need to create a Point-to-Point Protocol (PPP) virtual network interface and protect only IP-level traffic. Table 1 shows a comparison between the different VPN protocols.

The OpenVPN protocol is logically divided in two channels differentiated based on an `opcode` included in the first byte of each packet. The first channel is the control channel which carries session initialization, TLS handshake ciphers negotiation, key renegotiation, and acknowledgement traffic. The second channel is the data channel, which carries application data in the form of encrypted Ethernet frames or IP packets depending on how OpenVPN is configured. Data flows through the data channel once the session is established, including the symmetric keys used for encryption and integrity. The `opcode` values have changed over time as the OpenVPN evolved, and we provide a list of them in Table 3.

Data channel protection. The most important component of OpenVPN is the connection establishment procedure, which is based on TLS. While older OpenVPN versions have supported both a pre-shared static key and a TLS mode, the newer versions of OpenVPN have deprecated the static key mode and made the TLS mode mandatory. OpenVPN packets support both UDP and TCP, but UDP is recommended for efficiency.

Control channel protection. In addition to the handshake

[3] Some implementations are open source and some are proprietary.

[4] SSTP is initially designed by Microsoft and proprietary.

messages, OpenVPN also implements its own acknowledgment mechanism. There are three options to protect the integrity and privacy of the OpenVPN control channel packets:

i. *tls-auth*: This mode protects the integrity of the OpenVPN control channel packets with a MAC calculated using a static key shared among all clients and the OpenVPN server.

ii. *tls-crypt*: This mode protects the integrity and privacy of the OpenVPN control channel packets with a static key shared among all clients and the OpenVPN server.

iii. *tls-crypt-v2*: This mode is similar to *tls-crypt*, but each client has its own static key shared with the OpenVPN server.

OpenVPN users can configure the service with any of the above protections or none of them. While it might seem obvious that *tls-auth* should always be enabled, and a previous security audit [19] recommended that *tls-crypt* or *tls-auth* should always be used, the recent work fingerprinting OpenVPN [51] found 180,858 OpenVPN endpoints with *tls-auth* disabled in their experiments with the Censys.io [12] database.

Configuration. One important aspect of OpenVPN design is the configuration file, which provides the instructions and settings necessary for secure and efficient communication. It allows for a high degree of customization, enabling users to tailor their VPN setup to their specific needs. Proper configuration is essential to ensure that VPN operates securely and effectively. Examples of information that can be configured include: the IP address or hostname of the VPN server, port number for the VPN connection, protocol (TCP or UDP), encryption and authentication algorithms, location of the user's certificate, and private key files.

Code and specification. OpenVPN is available in an open-source GitHub repository [36] and as a proprietary implementation called OpenVPN AccessServer [38]. OpenVPN relies on OpenSSL (or mbedTLS) for TLS functionality. There is no standard specification of how the protocol works other than a work-in-progress OpenVPN RFC [41] that lacks many details and appears unfinished. Most of the protocol description is scattered in several documents included in the OpenVPN repository [36]. Another source that has a few more details is the Quarkslab audit report of an older OpenVPN version (2.4.0.) [19]. The report mentions that they relied solely on the code since no written protocol specification was available.

2.2 Security of OpenVPN

Several attacks were previously found against OpenVPN, and we categorize them below. We list the most representative attacks in Table 2 in Appendix A.

Denial of service (DoS): These attacks aim to overwhelm a VPN server with a flood of traffic or cause a server crash by several means, rendering it unable to process legitimate user requests and effectively disrupting service. The work in [26] found that flooding the server with packets of type `P_CONTROL_HARD_RESET_CLIENT_V2` (this type identifies the first packet ever sent by a client to establish a connection to the server) can deny data transmission and connection establishment for other clients. The Quarkslab technical report analyzed an older version of OpenVPN 2.4.0 and found several vulnerabilities including DoS caused by assert triggered and DoS caused by exhaustion of packet identifiers [19].

VPN traffic fingerprinting: In traffic fingerprinting, attackers analyze encrypted VPN traffic and identify patterns, potentially determining which VPN protocol is being used or even the nature of the data being transmitted. The work in [51] showed how to fingerprint the network flow of OpenVPN based on some key observations of fixed patterns of OpenVPN traffic.

Man-in-the-Middle (MitM) attacks: In a MitM attack, an attacker intercepts and potentially alters the communication between the user, the VPN server, and the application server that the user wants to visit with VPN protection. The work in [45] showed that TCP connections forwarded through a VPN tunnel can be hijacked by connection inference and sending spoofed packets.

Operating system exploits: An attacker can exploit vulnerabilities in the operating system of the VPN client or server, to bypass VPN protection. The work in [52] showed that an attack against the VPN client could leak traffic in plaintext outside the VPN tunnel by abusing the routing table exceptions and spoofing the packet-sending address.

Memory bugs: Memory bugs may arise due to errors in the VPN software implementation. The work in [49] fuzzed the OpenVPN source code by intercepting system calls execution and feeding a fuzzing corpus to the target functions with the help of libFuzzer [31] and uncovered several memory-related vulnerabilities.

Replay attacks: In a replay attack, an attacker intercepts and retransmits valid data packets. If a VPN does not implement proper replay protection mechanisms, this can lead to unauthorized actions being carried out, potentially leading to denial of service. The work in [26] found that flooding by replaying a particular type of packet can deny data transmission and connection establishment.

Older attacks: There are many attacks on deprecated versions of OpenVPN. The OpenVPN CVE website [3] lists over 50 CVEs of OpenVPN. Several of them are related to the cryptographic mechanisms (some due to vulnerabilities in OpenSSL) or memory corruptions.

3 Learning OpenVPN Message Sequence Chart of Connection Establishment

In this section, we describe our method for learning the details of the OpenVPN connection establishment in both the UDP

and TCP modes. We observed the packet exchange behavior and verbose logs of OpenVPN, read the source code of the open-source implementation [36], and read scattered information about OpenVPN in peer-reviewed papers or technical blogs. Below we describe our approach, and then describe the learned Message Sequence Chart (MSC) for OpenVPN.

3.1 Our Approach

Our learning was done in two phases. During the first phase, we passively observed the protocol to gain information about the packets, while in the second phase, we actively interacted with the protocol to gain a deeper understanding of its behavior.

Passive Learning. We started by seeking to understand the OpenVPN packet structure and the high-level message sequence for a normal OpenVPN connection establishment. The packet structure knowledge helps write the parsing code for OpenVPN packets, which facilitates later active learning. We do this by observing the captured packet trace of a normal connection in Wireshark. While information about some of the fields in the OpenVPN control packets was available in some documentation, they were not complete, and some referred to older versions. The notation we use below to denote different fields are the names as reported by Wireshark.

We display and explain the packet header structure of the OpenVPN layer for UDP and TCP mode in Fig. 4 and Fig. 5 in Appendix B. The most important field is *Opcode* denoting the OpenVPN packet type. The control packets fall in three categories: (1) packets that correspond to connection management (starting a connection (**`P_CONTROL_HARD_RESET_CLIENT_V2`** and **`P_CONTROL_HARD_RESET_SERVER_V2`**), moving to a different key (**`P_CONTROL_SOFT_RESET_V1`**)), (2) TLS related control packets (**`P_CONTROL_V1`**), and (3) acknowledgment packets (**`P_ACK_V1`**).There is only one *Opcode* for data packets (**`P_DATA_V2`**). Table 3 in Appendix B shows a summary of the main fields of the OpenVPN header, including *Opcode* name, value, and description.

Active Learning. To understand the message-sending state transition in-depth and to learn the content of ciphertext TLS messages from the verbose logs of OpenVPN corresponding to specific steps in the connection establishment, we employ a MitM setup as shown in Fig. 1, supporting both the UDP mode and TCP mode.

The setup allows us to actively control the message-sending progress and monitor the behavior of both client and server by looking at what messages are being sent and the verbose logs of OpenVPN. We can stop the execution of the protocol after a particular message or sequence of messages was sent and observe the response from the server and the corresponding logs from the OpenVPN executable. In Fig. 1, the "Connection progress control parameter i" means the total number of control channel messages we allow the *LearnVPN* program to forward to the client and server, e.g., when $i = 1$, the *LearnVPN* program will only allow one message to be forwarded, which is the first message from the client.

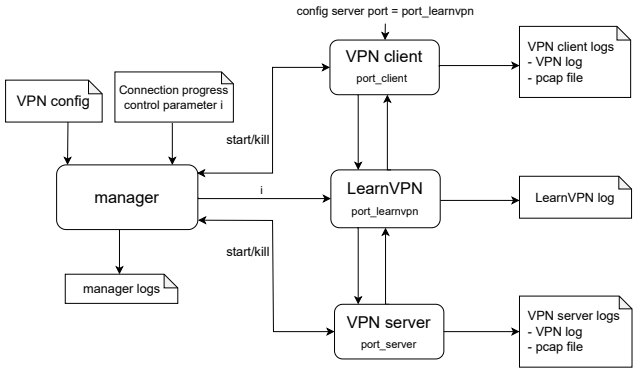

Figure 1: The LearnVPN system diagram

For the UDP mode, we find that 16 messages are required for the connection to complete (M_1 to M_{16} in Fig. 2). We also found that under certain network conditions, two more messages are sent: a control and an acknowledgment message, and we display them with dashed lines (M_{17} and M_{18} in Fig. 2). We describe their role in Section 3.2. For the TCP mode, 14 messages are needed normally for the connection success while under certain network conditions, two more messages (M_{15} and M_{16} in Fig. 6) are needed.

3.2 OpenVPN Connection Establishment

Using the methodology described in the previous section, we created the MSC of OpenVPN connection establishment (default TLS configuration) for both UDP and TCP. Below we focus on the UDP mode (shown in Fig. 2) which is the recommended deployment, and we include the MSC description for the TCP mode in Appendix C.

The handshake protocol aims to establish the data channel key that will be used for transporting application data securely inside the VPN tunnel (we denote it with K_{data}). During the handshake, two other keys are established. The first one, denoted as K_{hs}, is used for protecting the rest of the TLS handshake messages after its establishment. The second one, denoted as K_{app}, is established as the TLS application record data key.

We use the following notation in Fig. 2. We use sid_c to denote *Session ID*, sid_s for the *Remote Session ID*, *MID_array* to denote *Message Packet-ID Array*, and *MID* to denote *Message Packet-ID*. For simplicity, we omit in the MSC description the field *Message Packet-ID Array Length*.

Starting a new connection. When a client wants to start a new connection with a server, it sends a message of type **`P_CONTROL_HARD_RESET_CLIENT_V2`** packet and waits for a response with the type

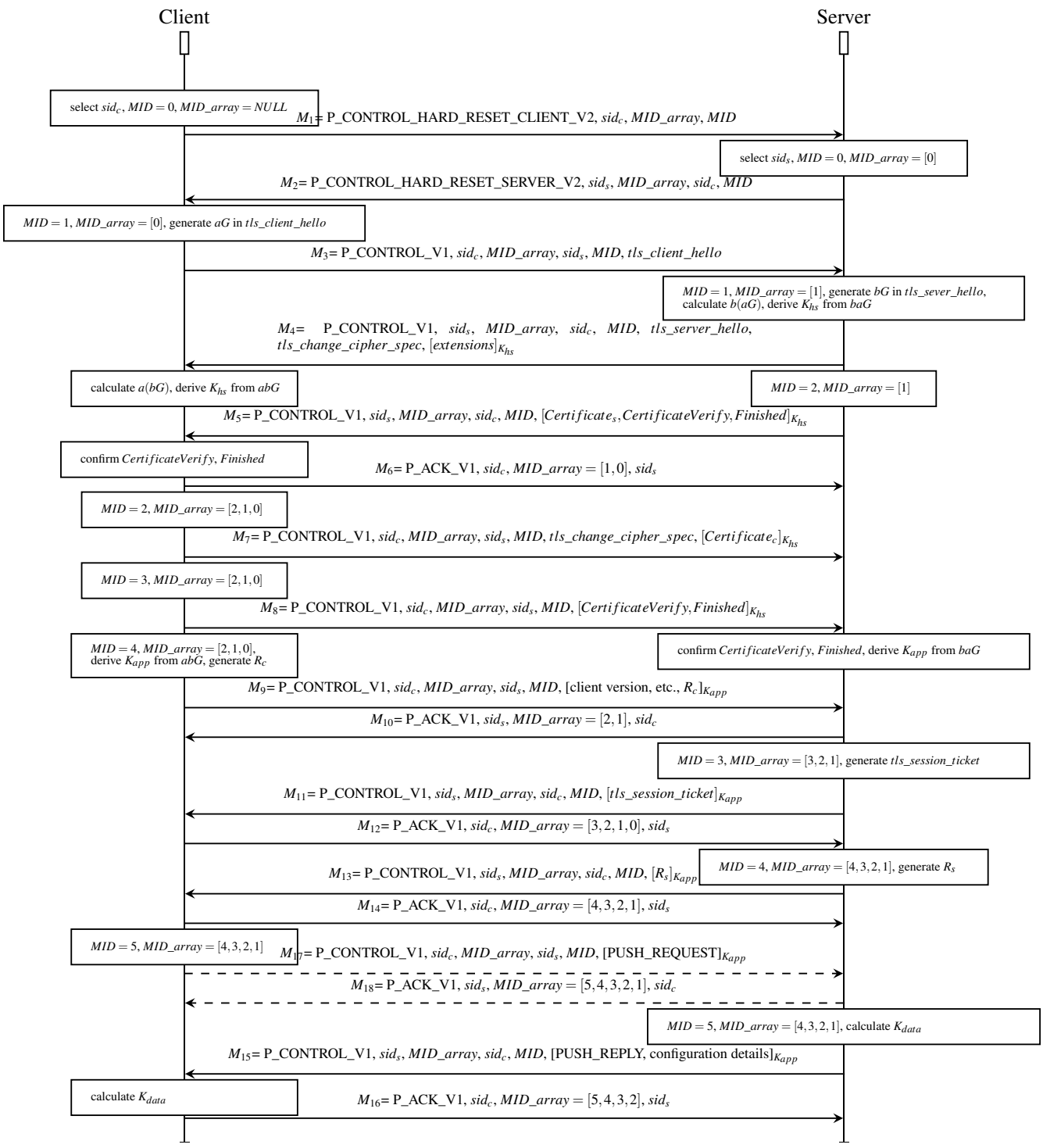

Figure 2: MSC of OpenVPN handshake (UDP mode, default TLS configuration), some fields omitted for simplicity.

P_CONTROL_HARD_RESET_SERVER_V2 packet from the server (messages M_1 and M_2 in Fig. 2). The goal of these two messages is to exchange the session identifiers at both ends, including sid_s, the *Session ID* of the VPN server, and sid_c, the *Session ID* of the client. After this exchange, the TLS key establishment phase starts.

TLS key establishment. The goal of this phase is to establish a secure TLS channel, which is then used to protect the data channel key generation phase. OpenVPN relies on the OpenSSL or mbedTLS library to implement the TLS functionality. Below we describe the message flow and omitting explicit acknowledgments. The default algorithm used for TLS handshake is *0x25519 ECDH (Elliptic Curve Diffie Hellman)* [37] [14].

First, the client sends a *TLS Client Hello* record, which includes the client-side public key aG in the *Key_Share* record, i.e., the client-side ECDH contribution that will be used later to compute the TLS pre-master secret (message M_3 in Fig. 2). Then the server will reply with 2 OpenVPN packets including several TLS records: *TLS Server Hello* including the server-side public key bG in the *Key_Share* record, *Change Cipher Spec* implying that the messages afterward will be ciphertext, and some application data records including *extensions*, server's certificate ($Certificate_s$), *CertificateVerify*, and *Finished* (messages M_4 and M_5 in Fig. 2). Note that after obtaining aG from the client, the server can now compute the ECDH secret as $b(aG)$, i.e. the TLS pre-master secret, serving as the basis for generating the K_{hs} and K_{app} keys to protect TLS and data channel key generation messages.

Upon receiving messages M_4 and M_5 from the server, the client can also compute the same ECDH shared secret as $a(bG)$ and derive various further keys. The client will reply with 3 OpenVPN packets including several TLS records including a *Change Cipher Spec* record and some application data records including client certificate ($Certificate_c$), *CertificateVerify*, *Finished* (M_7, M_8 in Fig. 2). This concludes the TLS phase for the client. The server then sends an OpenVPN packet where the TLS application data records include the *TLS session ticket* (M_{11} in Fig. 2). This concludes the TLS phase for the server.

Data channel key generation phase. The client starts the data channel key generation phase with the message M_9 where, along with the client version, he sends the client-side random material R_c. On the server side, this is message M_{13}, which contains the random contribution of the server R_s. The deprecated method uses both R_c and R_s to compute the data channel key, while now OpenVPN uses the TLS key exporter [21] method, and R_c and R_s are sent but not used, just for backward compatibility. Both R_c and R_s are encrypted with K_{app}, the key that was negotiated via TLS.

After receiving M_{13} and M_{15} from the server, the client can compute the data channel keys. After the data channel key is established for both sides, the data channel OpenVPN packets which start with an *Opcode* of **P_DATA_V2** can be transported through the VPN channel. The VPN channel consists of virtual network interfaces launched on both sides and the routing policy updated to direct certain traffic through the VPN channel. The virtual interface helps to exchange traffic between kernel space and user space for encryption or decryption by VPN software. The updated routing policy ensures the traffic desired to be protected is directed through the VPN tunnel.

A normal connection establishment takes only 16 messages to complete in the UDP mode. When the connection is tampered with or due to other network factors the client cannot receive M_{15} in time, the client will send one more **P_CONTROL_V1** packet M_{17} to push the server to reply (i.e. push the server to send M_{15}). The server explicitly acknowledges this request (M_{17}) with M_{18}. We show M_{17} and M_{18} with dashed lines in Fig. 2.

Reliability. OpenVPN implements an explicit acknowledgment mechanism as it does not always operate over TCP. All OpenVPN acknowledgment messages have the *Opcode* **P_ACK_V1**. There are four acknowledgment packets sent from the client and one acknowledgment packet sent from the server to acknowledge the message receiving progress explicitly and to ensure reliability in the UDP mode.

All the OpenVPN control channel messages, except the **P_ACK_V1** messages, have the *MID* field in the OpenVPN header. We marked the *MID* value for each message in Fig. 2. Besides, all the control channel messages including the **P_ACK_V1** messages have the *MID_array* field in the OpenVPN header. We also marked the values of *MID_array* field for all the messages in Fig. 2. Reliability can be done by checking the *MID_array* on **P_CONTROL_V1** control channel messages, and explicitly on **P_ACK_V1** messages. For both of them, the value of *MID_array* is the array of current received messages' *MID*'s, and by checking which messages have been received, a side can find out which messages have not been received and retransmit them.

4 FUZZVPN Design

In this section, we describe the design of our fuzzing system. We first describe our design goals and give a high-level overview, then we describe the attack strategies supported by our system and finally present what information is captured with each experiment to analyze each of them and determine if it is a vulnerability that can be exploited by an attacker.

4.1 Design Goals and High-level Overview

FUZZVPN aims to test the OpenVPN open-sourced implementation from several aspects to find security vulnerabilities. We want our testing platform to achieve several goals.

Platform independent: Our testing framework should not be limited by the implementation language of the VPN software or the operating system that the VPN runs on.

Protocol-logic aware: Our testing framework should be able to test protocols-logic aware scenarios, i.e., to not only change or inject packets but also impact the normal protocol message flow by delaying, dropping, reordering packets, etc., while tracking the states of the involved parties.

Not invasive: Ideally, we would like the platform to not change the code running on both the client and server side.

Test both the client and the server: Finally, we would like our platform to allow for testing both the server and client implementation by being able to interact with both the client and the server implementations.

With these goals in mind, we chose a design based on the one we used to learn the MSC of the protocol and inspired by [53]. Fig. 3 shows a diagram of our testing setup. The `manager` module automates all the experiments and logs results, taking as inputs VPN configuration files and the supported fuzzing strategies. The `manager` can start and kill the client, server, and FUZZVPN program. The *fuzzing strategy* on the arrow in Fig. 3 acts as an input parameter of FUZZVPN which will apply the actual fuzzing actions.

FUZZVPN is a UDP (or TCP) proxy we implemented with the help of twisted library [44], which can handle UDP (or TCP) packets sent to the corresponding port that the proxy is bound to, apply fuzzing actions on the intercepted packets, and forward packets to a desired destination if needed.

As shown in Fig. 3, we configured the VPN client to run on *port_client*, and let it connect (i.e. send the packets for completing the connection) to FUZZVPN which runs on *port_fuzzvpn*. Meanwhile, we configure the VPN server to run on *port_server*. We let FUZZVPN serve as a MitM proxy, forwarding packets from the client to the server, thus the server will believe a client is running on *port_fuzzvpn* and reply packets to it. Upon receiving the packets from the server, FUZZVPN can also apply the fuzzing strategy and then forward packets to the client. The only difference between UDP mode and TCP mode is using the UDP ports (and a UDP proxy) or TCP ports (and a TCP proxy).

We compiled the open-sourced OpenVPN implementation with AddressSanitizer (ASan) [24] and Undefined Behavior Sanitizer (UBSan) [32] to check for memory bugs and unexpected behavior during program execution (ASan, UBSan log output in Fig. 3). We also use tcpdump [43] to intercept the packets on the client and server side, which generates the pcap file output in Fig. 3.

4.2 Attack Strategies Supported

Our system supports several testing strategies such as (1) malformed configuration files; (2) replay packets; (3) field-level modification; (4) reordering packets; and (5) acknowledgment-related attacks.

Malformed configuration file testing. We craft malformed client and server configuration files with illegal values or formats, etc. to feed the OpenVPN program and monitor its

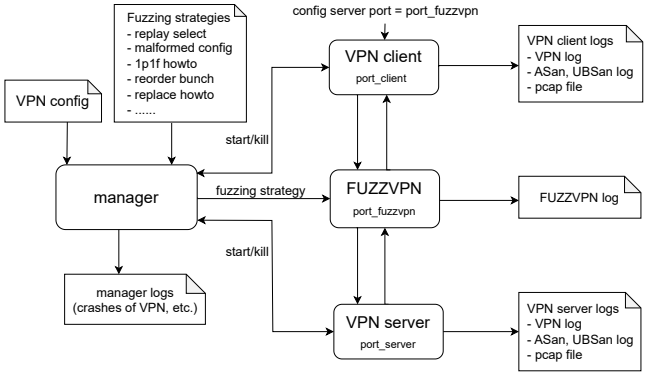

Figure 3: The VPN fuzzing system diagram

reaction. We also tried inconsistent settings at the client and server that can potentially result in downgrade attacks. We designed different fuzzing methods depending on the specific configuration option type and semantics. For options with numeric values (e.g. “port", “lport", "keepalive", etc.), we changed them to zero and a very large number (currently we use 2^{70}) that overflows 64 bits. For all the options with string values, we insert a NULL character after the first character of the string. We created a script to generate all the malformed configuration files and test them with the OpenVPN program. In total, we generated 242 malformed configuration files (half are for the UDP mode and the rest are for the TCP mode), including both for the server and client.

Denial of service with replay packets. Inspired by [26], which only focused on initiation packet flooding, we designed a more comprehensive way to test the resilience of OpenVPN against replay attacks with different-typed packets, especially **`P_CONTROL_V1`** and **`P_ACK_V1`** packets.

Field value modification. The field-level modification and reordering strategies are controlled by the fuzzing parameters specified by the experiment manager. The experiment manager specifies one set of fuzzing experiments by a set of parameters: (1) *fuzzing_way*, which implies the selected fuzzing strategy, (2) detailed parameters.

When $fuzzing\_way = 1p1f$, where *1p1f* means 1-packet-1-field, we will change a packet's selected field with a certain method, so the following parameters will be *pkt*, $field$ and *howto*, which means the selected packet, field, and how to change the field value.

For fields that have a limited number of valid values (e.g. *Opcode*, *MID*, etc.), *howto* could be (1) *rand_vali*, which means randomly choosing a valid value to assign the field; (2) *rand_any*, which means assigning a random value; and (3) *rand_zero*, which means assigning zero value for the field.

For other fields that have a specific numeric value (e.g. *Session ID*, *Remote Session ID*, *MID_array Length*, etc.), *howto* could be (1) *rand_any* or (2) *rand_zero*, or values of other

legitimate sessions from other clients.

Reordering of a sequence of control packets. When $fuzzing\_way = reorder$, it means we will reorder several messages from one side, then the following parameter will be *bunch*, meaning the selected set of messages from one side, which could be $s1$, $c1$, and $s2$, with $s1$ denoting the first set from the server (i.e., M_4 and M_5 in Fig. 2), $c1$ denoting the first set from the client (i.e., M_7, M_8, and M_9 in Fig. 2) and $s2$ denoting the second set from the server (i.e., M_{11}, M_{13}, and M_{15} in Fig. 2). For the TCP mode, $s1$ denotes the M_4 and M_6 in Fig. 6 which include 3 OpenVPN packets, $c1$ denotes the M_8 in Fig. 6 which includes 3 OpenVPN packets, and $s2$ denotes the M_{10} in Fig. 6 which includes two OpenVPN packets.

Acknowledgement-related attacks. We are interested in fields with important semantics, for example, the value of *MID_array* contributes to the acknowledgment mechanism, so we designed some specific strategies related to the meaning of this field. We support the following actions:

1. Remove one element out of the array: by removing elements from the array we control what message should not be acknowledged.
2. Replace one element of the array: by replacing an element of the array we acknowledge packets are not meaningful at that time in the protocol message sequence.
3. Replace with an earlier **`P_ACK_V1`** packet: by replacing a **`P_ACK_V1`** packet from the client with an earlier **`P_ACK_V1`** packet we reacknowledge old packets and not send the acknowledgement for the recent packets.
4. Replace sid_c and sid_s with the other client's: by replacing the sid_c and sid_s values of a **`P_ACK_V1`** packet (from client to server) we inject acknowledgement messages corresponding to other session ids.
5. Drop **`P_ACK_V1`** packets: by dropping some or all the **`P_ACK_V1`** packets during the handshake of OpenVPN we test the robustness of the handshake to such attacks.

4.3 Analyzing the Behavior

For the malformed configuration file testing, we feed them to the OpenVPN program, monitor whether it reports any warnings or errors, and decide if it is expected behavior.

For the replay attacks, we monitor the logs from both client and server to see if any defense is triggered or any detection of the replay traffic. Meanwhile, we monitor connection failure messages in the logs of the OpenVPN executable and measure the attack effect on a legitimate user's VPN data channel throughput.

For the other attacks such as changing the field values and reordering, we monitor the program statuses of both the client and server to see if any unexpected program crashes happen. To help analyze the vulnerable case, for each fuzzing experiment, we collect (1) client and server execution logs (provided by OpenVPN); (2) client and server stderr (mostly ASan and UBSan output); (3) client and server-side packet traffic captured (by tcpdump [43]); (4) the FUZZVPN program output.

We also search the logs for any indication of a successful connection. OpenVPN logs indicate client-side and server-side connection success respectively through the following messages: S_c="Initialization Sequence Completed"; and S_s="Peer Connection Initiated with [Client Address]".

5 Results

In this section, we present the results of our evaluation. We first describe the methodology we used, then we show that a previous attack has been fixed for the UDP mode but it is still possible for the TCP mode (the findings about previous attacks are discussed in Appendix E.1.). Then we describe several new vulnerabilities we found. Finally, we investigate the resilience to ACK-related denial of service attack scenarios and we found that while the UDP mode protocol is resilient to them, the TCP mode is not. The sanitizers did not report any memory bugs in our experiments.

5.1 Methodology

Platform. Our experiments are done on a machine equipped with two Intel Xeon Silver 4114 CPUs (each with 10 physical cores and Hyper-Threading enabled, providing a total of 40 logical processors), x86_64 architecture, 188 GB of physical memory, and Ubuntu 24.04 operating system. We created a docker image where we added the open-source OpenVPN repository and the required configuration files, and we used the image to create a docker container, mapping the path of our fuzzing source code to a directory inside the docker container. The docker version we used in our experiments is 27.1.1. We ran the OpenVPN version 2.6.12. We will open-source our Dockerfile, docker image, and all the relevant source code. Our code is available at `https://doi.org/10.5281/zenodo.15476514`

FUZZVPN setup. As we mentioned in Section 4.1, we configured FUZZVPN as a MitM by changing the ports where the VPN client and VPN server will run and connect. We select $port\_client = 40000$, $port\_fuzzvpn = 50000$, and $port\_server = 1194$ (the default port number OpenVPN usually uses.) The ports are used for both UDP and TCP configuration modes of OpenVPN. In Fig. 3, we created a script to manage all the fuzzing experiments, feeding the fuzzing strategy parameters to FUZZVPN program, as explained in Section 4.2. For fuzzing strategies that change the content or sending order of legitimate packets, we run each experiment for 60 seconds, during which either a crash happens or the `manager` will kill all the running programs and start a new experiment.

Automated and manual analysis. To get a detailed execution log of both OpenVPN client and server, we added a

configuration option "−− verb 9" when we ran the OpenVPN program, which we found to be verbose enough for our investigation [37]. We also collected logs of ASan and UBSan stderr messages to check memory and undefined behavior bugs. Finally, we collected logs of FUZZVPN and used tcpdump [43] to capture the packets on both the client and server sides. We analyzed the logs both automatically and manually to confirm a suspected vulnerability. We created a script to analyze the logs by searching for errors and warnings, as well as key sentences signaling the connection success in the execution logs of both the client and server.

Attack scenarios. Overall we tried about 1000 scenarios and our fuzzing experiments took about 5.5 hours for the UDP mode and 6 hours for the TCP mode. The analysis took more time because it also involved code inspection to confirm the experimental findings.

5.2 New Denial-of-Service Attacks

In this section we discuss several new vulnerabilities we discovered with the OpenVPN 2.6.12 version implementation. They are denial of service attacks by replay packets, thus they can not be defended only with cryptographic mechanisms. We note that in the UDP mode, when the *tls-auth option* is enabled, some rate-limiting is implemented but the defense is not very effective; when *tls-auth option* is not enabled, no rate-limiting is implemented. In the TCP mode, even when the *tls-auth option* is enabled, we do not observe any replay protection being effective. It might seem obvious that *tls-auth* should always be enabled, however, several works like the recent work in fingerprinting OpenVPN [51], found 180,858 OpenVPN endpoints with *tls-auth* disabled in their experiments with the Censys.io [12] database.

We found attacks with two type of packets: `P_CONTROL_V1` and `P_ACK_V1`. The attacks we found can prevent the client's connection whose packets we replay in the attack. Further investigation also shows that the replay attacks also harm the server's availability and decrease the other legitimate user's VPN data channel throughput. Below we describe the impact of the attacks, and then describe experiments conducted in more realistic settings in Google Cloud.

1) Denial of service with replay of `P_CONTROL_V1` packets. `P_CONTROL_V1` is the most common packet type among all the control channel packets, 9 packets with this type are exchanged between the client and server to complete the VPN handshake in the UDP mode and 10 for the TCP mode. We found denial of service where an attacker observing a client connecting to a server, replays packets from this very connection to the server, making the connection establishment fail.

We found that when *tls-auth* mode is not used, for the UDP mode, flooding the server by replaying 10,000,000 `P_CONTROL_V1` packets (around the rate of 25 MB/second) can block the normal connection attempt from the client, and the server does not detect the replay attack, only reporting the error message "TLS key negotiation failed to occur within 60 seconds (check your network connectivity)". For the TCP mode, sending the same number of packets resulting in an attack rate of 16 MB/s showed similar results.

Our code inspection further revealed that when *tls-auth* mode is enabled in the UDP mode, a replay protection is activated. Specifically, a field called *Replay-Packet-ID* in the OpenVPN packet header and a sliding window replay check algorithm are added. The sliding window replay check algorithm takes two input parameters "size" n and "time" t, which can be configured with the option "–replay-window n t", and the default values are $n = 64$ and $t = 15$. We confirmed in experiments when *tls-auth* is used, the replay check functionality takes effect for the UDP mode, but not for the TCP mode. The denial of service worked for both UDP and TCP modes.

Finally, we note that the official manual of OpenVPN 2.6 [37] seems to imply that replay protection is provided only for the data channel packets. A more detailed explanation of how control packets are treated is needed.

2) Denial of service with replay of `P_ACK_V1` packets. `P_ACK_V1` packets are another common type of packets serving as explicit acknowledgments in both UDP and TCP modes.

When no *tls-auth* mode is used, for the UDP mode, flooding the server with 10,000,000 replayed `P_ACK_V1` packets (i.e. around the rate of 218 KB/second) caused similar results as flooding with `P_CONTROL_V1` packets, the normal connection attempt from the client is blocked with error message "TLS Error: TLS key negotiation failed to occur within 60 seconds (check your network connectivity)" in the logs of both the client and the server. For the TCP mode, sending the same number of packets around the rate 1.6 MB/s showed similar results. When *tls-auth* mode is used, we found similar behavior as in the above replay attack, i.e., UDP mode generated warnings but did not prevent the attack while TCP mode even did not generate warnings.

3) Attack evaluation in Google Cloud. To better evaluate the effect of the above replay attacks, we also tested them in more realistic network conditions in a Google Cloud environment. We set up 3 VMs as the OpenVPN server and two clients: the VMs are e2-standard-2 x86-64 machines with 8 GB memory and the OS Ubuntu 20.04, the server VM is located in us-west4-b zone while the two client VMs are in us-central1-a zone. We launched the replay attacks with packets from Client1 and observed the effects on the OpenVPN data channel throughput of Client2. We used the *iperf3* tool to measure the throughput. We summarize the results in Table 4 and explain two scenarios when the TCP mode was configured. We present a detailed description of attack scenarios with OpenVPN configured in the UDP mode in Appendix E.2.

In the TCP mode with no *tls-auth*, the replay of 10,000,000

P_CONTROL_V1 packets at the rate of 212 MB/s can almost block Client2's data channel traffic (throughput is less than 1 MB/s in comparison to a throughput of around 168 MB/s achieved when no attack takes place). We looked into the server logs and found that the replay attack can cause the server "Fatal TLS error (check_tls_errors_co), restarting", which might explain why Client2's data channel got almost blocked. Replay of 10,000,000 **P_ACK_V1** packets, which are shorter packets, at the rate of 50 MB/s also caused a serious decrease in throughput from 160 MB/s to 30 MB/s.

The replay protection of *tls-auth* seems to not work in TCP. In TCP with *tls-auth*, sending 10,000,000 **P_CONTROL_V1** packets at the rate of around 60 MB/s can almost block Client2's data channel to be less than 1 MB/s, compared with the normal throughput 116 MB/s. Replay of 10,000,000 **P_ACK_V1** packets, which are shorter packets, at the rate of 10MB/s also caused a serious decrease in throughput from 116 MB/s to 30 MB/s.

5.3 Improper Input Validation for Configurations

OpenVPN supports complex configuration files. We tested numerous malformed configurations and report our findings below.

Port validation. We noticed that when we provided a server (or client) configuration file with a port number (e.g. 70000) exceeding the valid port number range (from 0 to 65535 on Linux Systems), the OpenVPN implementation does not report any error, whether under the UDP or TCP mode, but instead just chooses another valid port for usage without informing the user. We also tested the OpenVPN Connect client software (Mac version) which correctly reported the error of an invalid port number and did not make the change. Our communication with the OpenVPN team confirmed the port validation vulnerability. The root cause lies in `glibc`: when passing a numeric string as port/service to the `getaddrinfo` function representing a number larger than 65535, the function will not complain and will simply extract the lowest 16 bits of the converted number. A patch is now under development. The MAC version is a different code base and that is why the error was not present.

Options validation. We observed that whether under the TCP or UDP mode, the OpenVPN implementation applied stricter checking rules for some configuration options, and less strict checking for other options. For example, when we assign values that should obviously be rejected to some options with a strict validation rule, like the option "replay-window", the log will report "Options error: replay-window window size parameter (-1) must be between 0 and 65536"; however, for some other options like "hand-window" no warnings are provided. One interesting case is the option "max-clients", where when we assign 0 to it, the program will crash with a "fatal error" since it triggers an assertion failure in the source code; however, when we assign 2^{70}, the program will report "Options error: –max-clients must be at least 1". In Appendix E.3, we provide more details on similar scenarios in Table 5.

5.4 Server Prematurely Sends Data

We found two attacks that cause an inconsistent view of the client and server concerning the state of the connection. We confirmed both scenarios through code and logs inspection.

For the first attack, we notice that the OpenVPN server will start sending DATA_V2 packets immediately after sending the last **P_CONTROL_V1** packet M_{15} to the client. If we drop M_{15}, we will observe the client sending several **P_CONTROL_V1** packets (i.e. M_{17} [PUSH_REQUEST] packets) trying to push the server to send M_{15}. If we drop the M_{17} packets from the client as well as the reply packets from the server, then the client will face connection failure while the server can keep sending **P_DATA_V2** packets. The root cause of server-side premature data sending is that the server believes it has connected successfully too early before confirming receiving the last **P_ACK_V1** packet M_{16} from the client. Also, by examining the server logs, we observe the server logs the connection success sentence S_s indicating connection success with the client even before it sends out M_{15}. The client does not think the connection succeeded and did not log its connection success sentence S_c and by looking at its verbose log, we confirm that the client's connection attempt is blocked at this point.

The second variant is more concerning. We created an attack scenario where the attacker drops M_{15} packets up to a threshold and then resumes sending it to allow the client to successfully finish the handshake. In this case, the client is not aware that it actually dropped data that was sent prematurely by the server. While applications can implement some additional checks, this behavior is not specified anywhere and can be problematic for applications that depend on initial data not being lost and are not aware of this behavior.

For the TCP mode, we found similar behavior. Refer to the MSC in Fig. 6 in Appendix C, this means the server will start sending data immediately after it sends out M_{13} without confirmation of receiving M_{14}. If we drop the M_{13} and possible packets from the client afterward, then the server ends up thinking the connection is a success and starts sending data channel packets while the client is trapped in connection failure. Also, if we resume the sending of the packet after a period of time, then the connection can still be successful, although the client misses some data packets from the server.

5.5 ACK-related Attacks

There is almost no documentation about the design and intended behavior for acknowledgments for the OpenVPN over UDP. The official website of OpenVPN [40] as well as the OpenVPN development website [39] only says "**P_ACK_V1**

– Acknowledgement for control channel packets received.". The older technical report [19] that analyzed OpenVPN 2.4.0 only says "It is notable that acknowledgment can either be done by a dedicated **`P_ACK_V1`** packet or by including it in P_CONTROL packets.".

Unlike TLS for example, which operates over TCP and assumes reliable delivery of the packets, the OpenVPN handshake, while using TLS, has to implement its own acknowledgment mechanism. Previous works overlooked the functionality of the ACK messages (i.e., **`P_ACK_V1`** packets) used during the OpenVPN handshake, for example, in the work [8], all the ACK messages are omitted in the inferred OpenVPN session. In this section we describe several fuzzing experiments specific to the ACK messages and uncovered that the protocol is resilient against many ACK attacks under the UDP mode but not under the TCP mode.

The most important field of a **`P_ACK_V1`** packet related to the acknowledgment progress is the field called *Message Packet-ID Array*, as we explained in Section 3.2. When no *tls-auth* mode is used, i.e., no HMAC protection, we can arbitrarily change the field of OpenVPN packets and the altered packet may still get processed. We performed all the strategies listed in Section 4.2 about acknowledgments.

UDP mode. The strategies we tried could not stop the OpenVPN handshake from succeess in the UDP mode. We managed to slow it down by causing packets to be retransmitted as expected. We looked into how the acknowledgment mechanism works during the OpenVPN handshake, which is a design of the specific protocol to provide more reliability when OpenVPN packets are transported over the unreliable UDP protocol. We found that not only the ACK messages can explicitly contribute to the acknowledgment processing and state transition advancement of the protocol, but the **`P_CONTROL_V1`** packets can help in that, too, partially because they also have a *Message Packet-ID Array* field in the header. This factor may explain why OpenVPN behaves so robustly under the above ACK attacks. We do not know if this was a deliberate design or just a configuration artifact.

TCP mode. For the TCP mode, OpenVPN still uses the ACK mechanism (see the MSC in Appendix C for details). Unlike UDP mode's resilience to the acknowledgment-related attack strategies in 4.2, we found that for the TCP mode, there are several successful attacks that prevent the connection from finishing. They are: (1) changing the *Message Packet-ID Array* of M_12 (or M_11 or M_7 or M_5) in *rand_zero* or *rand_vali* ways; (2) changing the *sid* (or sid_r) field of any ACK messages in *rand_zero* or *rand_any* ways; (3) changing the *sid* and sid_r fields of the ACK messages from client to server to be values of the other current legitimate client's.

5.6 Performance Attacks

We also found another scenario in the TCP mode, where a malicious client connecting to a server with a weaker authentication setting can create a performance degradation for a victim client configured with stronger authentication. Specifically, when we configure the server in TCP *tls-auth* mode, we can launch a performance attack by letting a client connect with no *tls-auth* configured, and cause a decrease in data channel throughput for a victim user configured with *tls-auth* from 180 MB/s to around 120 MB/s (in the Google Cloud setup described in Section 5.2). The server generates log error messages "TLS Error: cannot locate HMAC in incoming packet from [IP: port address]... Fatal TLS error (check_tls_errors_co), restarting". In comparison, for the UDP mode, we observe that the server only generated the log message "TLS Error: cannot locate HMAC in incoming packet from [IP: port address]" without the "restarting" error message while the data channel throughput of the legitimate client is not affected.

The attacks in Section 5.2, 5.4 and 5.5 require an on-path attacker that can intercept, modify, drop and forward packets. On-path attackers are realistic and well-documented in a range of practical scenarios. For example, adversaries in public Wi-Fi networks can perform ARP spoofing to intercept and manipulate VPN traffic. In enterprise settings, a malicious insider may control internal routing to position themselves in the communication path. At the ISP or nation-state level, middleboxes equipped with deep packet inspection can observe and tamper with encrypted tunnels. Even in cloud environments, misconfigured virtual networks may inadvertently expose packet flows to neighboring tenants. Since VPNs are actually designed to be used over completely untrusted networks, and as such, packet loss, retransmissions, and mangling are expected and fully part of the threat model.

6 Related Work

In this section, we discuss related work other than related work for OpenVPN which we reviewed in Section 2.2. First, we discuss attacks that were found against VPNs, then we discuss the formal analysis conducted to prove the security properties of VPNs. Finally, we discuss general-purpose network protocol fuzzers, even if none of them were applied to VPNs: they tend to think of the network protocols as normal software and don't explore the DoS replay attacks, configuration validation testing, or protocol execution soundness.

Security of VPNs. Several works studied the security of VPNs, focusing on denial of service, fingerprinting, connection hijacking, or misconfigurations. The work in [20] creates a tool to locate security misconfigurations and privacy leakages. The work in [26] uncovers several DoS vulnerabilities. The work in [45] proposed in/on-path attacks to hijack the TCP connections forwarded through VPN tunnel by inferring virtual IP, connection timing, and SEQ/ACK. The work in [52] manipulated the routing exceptions added to the routing table to make the victim send arbitrary traffic in plaintext outside the VPN tunnel. The work in [50] evaluated thousands

of universities' academic VPN setups and uncovered some common configuration vulnerabilities. The work in [15] used Zero-shot machine learning for website fingerprinting, i.e., identify which website a user is visiting over an encrypted tunnel. The work in [6] uncovers some client-side L2TP/IPsec configuration vulnerabilities that can be exploited to strip off traffic encryption or bypass the VPN server authentication.

Formal analysis of VPNs. To the best of our knowledge, the only VPN for which formal analysis was conducted is WireGuard. A work-in-progress paper [10] by the WireGuard company applied TAMARIN to verify the security properties of the key exchange of the WireGuard protocol. The work in [22] proposed a unified formal symbolic model of WireGuard protocol using automatic cryptographic protocol verifiers SAPIC+, PROVERIF, and TAMARIN, and found a flaw of the anonymity of the communications. The work in [11] builds a description of WireGuard's key exchange phase and then proves the security of WireGuard's key exchange protocol under standard cryptographic assumptions.

Fuzzing on OpenVPN that did not find any vulnerabilities. There are also two efforts to find vulnerabilities in OpenVPN by looking at the protocol state machine. The work in [8] applies the protocol state fuzzing techniques to infer the state machine of OpenVPN, using black-box fuzzing with the *LearnLib* library. The work in [48] works on understanding how OpenVPN works on a coarse level and attempted some fuzzing by corrupting client-side messages. Their message sequence chart is not as detailed as the one we created, they overlooked details of ciphertext TLS messages' content, data key negotiation details, and the acknowledgment messages. Both works infer models that are less detailed than ours and were not able to find any vulnerabilities.

Fuzzers for network protocols in general. Several fuzzers were proposed for network protocols. Most of them focus on memory-related bugs, without exploring the replay DoS replay attacks, configuration validation testing, or protocol execution soundness checks, and can not be directly applied to OpenVPN.

Generation-based fuzzers like PEACH [29] require the user to manually specify the model logic of protocol implementation, which is error-prone and there is no RFC for OpenVPN describing the protocol logic details.

When no specified model is given, previous works proposed different ways to estimate the approximate states, track state transitions, and design feedback to guide fuzzing mutations. AFLNET [47] uses the server's response codes to identify states and retain those variations of original exchanged messages that can increase either code or state coverage rate. However, that is not enough to understand the content of the ciphertext TLS messages. Also, many messages during the OpenVPN handshake use the same **`P_CONTROL_V1`** type. StateAFL [17] hashes memory layout during the compiling process to detect unique states, construct a state machine at runtime, and prioritize the inputs which can increase new coverage in the state machine.

BLEEM [53] emphasized sequence-level mutations apart from the packet-field level, but the authors did not open-source the work. It designs a data structure to record the state transitions, called SSTG (System State Tracking Graph), and the state is a pair like $C(a)|S(b)$, implying that after the Server sent the abstract packet b, the Client sent packet a. Note that the "abstract packet" retains the enumeration-typed information, e.g., $Initial[CRYPTO,ACK] + Handshake[CRYPTO]$, which is similar to SGFUZZ [5]. When exploring paths to generate the sequence, newly added state transitions will be preferred to be chosen, which is like prioritizing the inputs that can produce new states in STT in SGFUZZ [5].

CHATAFL [16] uses publicly available RFCs and LLMs to learn message formats, based on which input packets are generated and guide the fuzzing of the target protocol. The approach cannot be applied to OpenVPN as there is no official RFC: only a work-in-progress draft [41].

DY fuzzing [4] proposes a new approach to fuzzing cryptographic protocols, considers the set of abstract DY executions of the DY attacker as possible test cases, and uses a novel mutation-based fuzzer to explore this set. The work tested 3 popular TLS implementations, resulting in the discovery of four novel vulnerabilities.

7 Conclusion

In this work, we took a systematic approach to find attacks in OpenVPN. We first constructed a detailed message sequence chart of the handshake protocol under the UDP and TCP modes, respectively. We used this information to perform systematic adversarial testing with malformed configurations, replay attacks, denial-of-service, resilience to acknowledgments-related attacks, and packet value modifications based on packet and protocol semantics. We found several new attacks: two new denial-of-service attacks due to the replay of control and acknowledgment packets, the incorrect handling of input validation for 17 protocol configuration options, a scenario where due to an inconsistent view of the state of the connection, the server sends data prematurely to the client causing the client to ignore it, and a scenario where a malicious client configured with weaker authentication can degrade the performance of a victim client configured with stronger authentication.

Ethics Considerations

We have informed the OpenVPN developers about our evaluation, and they have studied some of our findings while others are still under investigation. For example, the improper validation of the port number in the configuration file was identified to be a problem in the `getaddrinfo` function in `glibc`, and a patch is now under development by the `glibc` team.

References

[1] Secure socket tunneling protocol (sstp). https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-sstp/c50ed240-56f3-4309-8e0c-1644898f0ea8.

[2] Cisco anyconnect client. https://www.cisco.com/c/en/us/support/security/anyconnect-secure-mobility-client-v4-x/model.html, [n. d.].

[3] Openvpn : Security vulnerabilities, cves. https://www.cvedetails.com/vulnerability-list/vendor_id-3278/Openvpn.html, [n. d.].

[4] Max Ammann, Lucca Hirschi, and Steve Kremer. DY fuzzing: Formal dolev-yao models meet cryptographic protocol fuzz testing. Cryptology ePrint Archive, 2023.

[5] Jinsheng Ba, Marcel Böhme, Zahra Mirzamomen, and Abhik Roychoudhury. Stateful greybox fuzzing. In *31st USENIX Security Symposium (USENIX Security 22)*, Boston, MA, August 2022. USENIX Association.

[6] Thanh Bui, Siddharth Rao, Markku Antikainen, and Tuomas Aura. Client-side vulnerabilities in commercial vpns. In *Secure IT Systems*. Springer International Publishing, 2019.

[7] Jake S. Cannell, Justin Sheek, Jay Freeman, Greg Hazel, Jennifer Rodriguez-Mueller, Eric J. R. Hou, and Brian J. Fox. Orchid: A decentralized network routing market. 2019.

[8] Lesly-Ann Daniel, Erik Poll, and Joeri de Ruiter. Inferring OpenVPN State Machines Using Protocol State Fuzzing. In *2018 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. IEEE, 2018.

[9] Jason A. Donenfeld. Wireguard: Next generation kernel network tunnel. In *Proceedings of the 2017 Network and Distributed System Security Symposium. NDSS'17.*, 2017.

[10] Jason A Donenfeld. Formal Verification of the WireGuard Protocol, [n. d.].

[11] Benjamin Dowling and Kenneth G. Paterson. A cryptographic analysis of the wireguard protocol. In Bart Preneel and Frederik Vercauteren, editors, *Applied Cryptography and Network Security*, pages 3–21, Cham, 2018. Springer International Publishing.

[12] Zakir Durumeric, David Adrian, Ariana Mirian, Michael Bailey, and J. Alex Halderman. A search engine backed by internet-wide scanning. CCS '15, page 542–553, New York, NY, USA, 2015. Association for Computing Machinery.

[13] Pasi Eronen, Yoav Nir, Paul E. Hoffman, and Charlie Kaufman. Internet Key Exchange Protocol Version 2 (IKEv2). RFC 5996, September 2010.

[14] Adam Langley, Mike Hamburg, and Sean Turner. Elliptic Curves for Security. RFC 7748, January 2016.

[15] Ding LI, Chunxiang GU, and Yuefei ZHU. Gene fingerprinting: Cracking encrypted tunnel with zero-shot learning. *IEICE Transactions on Information and Systems*, E105.D, 06 2022.

[16] Ruijie Meng, Martin Mirchev, Marcel Bohme, and Abhik Roychoudhury. Large language model guided protocol fuzzing. In *NDSS*, 2024.

[17] Roberto Natella. StateAFL: fuzzing for stateful network servers. *Empirical Software Engineering*, 27(7):191, 2022.

[18] Boring protocol team. Boring protocol: A decentralized vpn on solana. https://github.com/boringprotocol, [n. d.].

[19] Quarkslab. Openvpn 2.4.0 security assessment. https://ostif.org/wp-content/uploads/2017/05/OpenVPN1.2final.pdf, 2017.

[20] Reethika Ramesh, Leonid Evdokimov, Diwen Xue, and Roya Ensafi. VPNalyzer: Systematic Investigation of the VPN Ecosystem. In *Proceedings 2022 Network and Distributed System Security Symposium*, 2022.

[21] Eric Rescorla. Keying Material Exporters for Transport Layer Security (TLS). RFC 5705, March 2010.

[22] Sylvain Ruhault, Pascal Lafourcade, and Dhekra Mahmoud. A Unified Symbolic Analysis of WireGuard. In *Proceedings 2024 Network and Distributed System Security Symposium*, San Diego, CA, USA, 2024. Internet Society.

[23] Security.org. 2024 vpn trends, statistics, and consumer opinions. https://www.security.org/resources/vpn-consumer-report-annual/, 2024.

[24] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. Addresssanitizer: a fast address sanity checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, USA, 2012. USENIX Association.

[25] statista. Usage of virtual private networks (vpn) worldwide as of 3rd quarter 2024, by country. https://www.statista.com/statistics/1382869/use-of-virtual-private-networks-vpn-by-country/, 2024.

[26] Fabio Streun, Joel Wanner, and Adrian Perrig. Evaluating Susceptibility of VPN Implementations to DoS Attacks Using Adversarial Testing. In *Proceedings 2022 Network and Distributed System Security Symposium*, 2022.

[27] Deepernetwork team. Deeper network decentralized vpn. https://shop.deeper.network/pages/decentralized-vpn, [n. d.].

[28] Express VPN team. Expressvpn. https://www.expressvpn.com/, [n. d.].

[29] Gitlab team. Peach fuzzing platform. https://gitlab.com/gitlab-org/security-products/protocol-fuzzer-ce, [n. d.].

[30] HOPRnet team. Hopr: Blockchain data protection and privacy. https://hoprnet.org/, [n. d.].

[31] LLVM team. libfuzzer – a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html, [n. d.].

[32] LLVM/Clang team. Clang 15.0.0 documentation, undefined behavior sanitizer. https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html, [n. d.].

[33] Mysterium team. Mysterium vpn. https://www.mysteriumvpn.com/, [n. d.].

[34] NordVPN team. Nordvpn. https://nordvpn.com/, [n. d.].

[35] OpenVPN team. Community supported openvpn versions. https://community.openvpn.net/openvpn/wiki/SupportedVersions, [n. d.].

[36] OpenVPN team. The open-sourced openvpn github repository. https://github.com/OpenVPN/openvpn/blob/master/, [n. d.].

[37] OpenVPN team. Openvpn 2.6 manual. https://openvpn.net/community-resources/reference-manual-for-openvpn-2-6/, [n. d.].

[38] OpenVPN team. Openvpn access server. https://openvpn.net/access-server/, [n. d.].

[39] OpenVPN team. Openvpn development website. https://build.openvpn.net/doxygen/network_protocol.html, [n. d.].

[40] OpenVPN team. Openvpn protocol. https://openvpn.net/community/, [n. d.].

[41] OpenVPN team. Work-in-progress github repository: building rfc for openvpn. https://github.com/OpenVPN/openvpn-rfc, [n. d.].

[42] Surfshark team. Surfshark. https://surfshark.com/, [n. d.].

[43] Tcpdump team. Tcpdump documentation. https://www.tcpdump.org/, [n. d.].

[44] Twisted team. Twisted: An event-driven networking engine. https://twisted.org/, [n. d.].

[45] William J. Tolley, Beau Kujath, Mohammad Taha Khan, Narseo Vallina-Rodriguez, and Jedidiah R. Crandall. Blind In/On-Path attacks and applications to VPNs. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, August 2021.

[46] Andrew J. Valencia, Glen Zorn, William Palter, Gurdeep-Singh Pall, Mark Townsley, and Allan Rubens. Layer Two Tunneling Protocol "L2TP". RFC 2661, August 1999.

[47] Abhik Roychoudhury Van-Thuan Pham, Marcel Bohme. AFLNET: A Greybox Fuzzer for Network Protocols. In *International Conference on Software Testing, Validation and Verification (ICSTVV)*. IEEE, 2020.

[48] Sven van Valburg, Erik Poll, and Joeri de Ruiter. Master thesis computer science: Fuzzing openvpn. 2018.

[49] Guido Vranken. Openvpn fuzzing with libfuzzer repository. https://github.com/guidovranken/openvpn/tree/fuzzing, 2017.

[50] Ka Lok Wu, Man Hong Hue, Ngai Man Poon, Kin Man Leung, Wai Yin Po, Kin Ting Wong, Sze Ho Hui, and Sze Yiu Chau. Back to school: On the (In)Security of academic VPNs. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 5737–5754, Anaheim, CA, August 2023. USENIX Association.

[51] Diwen Xue, Reethika Ramesh, Arham Jain, Michalis Kallitsis, J. Alex Halderman, Jedidiah R. Crandall, and Roya Ensafi. OpenVPN is open to VPN fingerprinting. In *31st USENIX Security Symposium (USENIX Security 22)*, Boston, MA, August 2022.

[52] Nian Xue, Yashaswi Malla, Zihang Xia, Christina Pöpper, and Mathy Vanhoef. Bypassing tunnels: Leaking VPN client traffic by abusing routing tables. In *32nd USENIX Security Symposium (USENIX Security 23)*, Anaheim, CA, August 2023.

[53] Feilong Zuo et al. Zhengxiong Luo, Junze Yu. BLEEM: Packet Sequence Oriented Fuzzing for Protocol Implementations. In *USENIX Security Symposium (USENIX Security)*, 2023.

[54] Glen Zorn, Gurdeep-Singh Pall, and Kory Hamzeh. Point-to-Point Tunneling Protocol (PPTP). RFC 2637, July 1999.

A Attacks against OpenVPN

Several attacks were previously found against OpenVPN, and we list the most representative attacks in Table 2.

B OpenVPN Opcodes and Other Fields

Table 3 shows a summary of the *Opcode* name, value, and description. Fig. 4 and Fig. 5 display the packet header structure of the OpenVPN layer in UDP and TCP mode respectively.

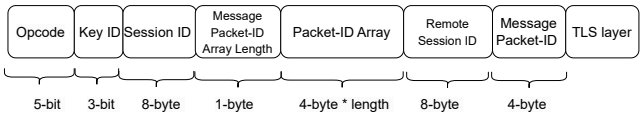

Figure 4: UDP mode OpenVPN control packet structure

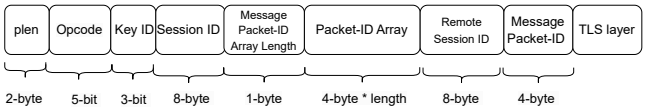

Figure 5: TCP mode OpenVPN control packet structure

In the UDP mode, the header of OpenVPN control packets starts with a 5-bit *Opcode* denoting the OpenVPN packet type, and a 3-bit *Key ID* denoting the corresponding data channel key's identifier. In the TCP mode, there is a 2-byte field *plen* before the *Opcode* field, which denotes the length of the following OpenVPN layer content. *Session ID* is the session identifier created by the source side for the VPN connection attempt, e.g., if the message is sent from the client, then it is a random number created by the client . There is another field *Remote Session ID* in the OpenVPN header, which is the counterpart of *Session ID*, i.e., the session identifier created by the destination part. Note that the first message from the client does not have that field since the *Remote Session ID* is unknown at this time. *Message Packet-ID Array Length* and *Message Packet-ID Array* keep track of the number of packets in the array and the array of received messages from the other party. These packet identifiers and arrays are used by the acknowledgment mechanism of OpenVPN, which we describe in Section 3.2. Because several packets have the same type **`P_CONTROL_V1`**, *Message Packet-ID* is used to uniquely identify each type of packet with an order number in the protocol logical sequence of packets. Note that the value remains the same if the packet is retransmitted, as it is associated with the logical order and not the number of packets sent. The value is used by the *Message Packet-ID Array* field present on all packets which includes from 0 to 8 elements. All the OpenVPN control channel messages except the **`P_ACK_V1`** messages have this field in the OpenVPN header.

The OpenVPN control packet header encapsulates either a blank value or the TLS layer records, e.g., *Client Hello*, *Server Hello*, *Change Cipher Spec*, and *Application Data*. We describe them in detail in Section 3.2.

If *tls-auth* is enabled, two additional fields are present in the OpenVPN header: *HMAC* and *Replay-Packet-ID*. *HMAC* helps protect the integrity of control channel packets, while *Replay-Packet-ID* is used to detect that the packet is a replay.

C OpenVPN Handshake MSC (TCP mode)

We provide the constructed MSC of the handshake in the TCP mode in Fig. 6. The main difference between the UDP and TCP modes is that the TCP mode will send several OpenVPN messages inside one TCP packet, such as M_6, M_8 and M_{10}, while in the UDP mode, one UDP packet only includes one OpenVPN packet. Compared to M_5 in the UDP mode, M_6 in the TCP mode includes two OpenVPN packets, although the information they carry should be similar to M_5 in the UDP mode. One side effect of encapsulating several OpenVPN messages in one TCP packet is that the ACK messages differ a bit: now in the TCP mode, one more **`P_ACK_V1`** is generated, and the *MID_array* values are different.

D The Inferred State Machines

Client and server finite state machines. Since some messages can be sent in parallel, we also provide the inferred finite state machine (FSM) of both the OpenVPN server and client (Fig. 7 and Fig. 8). We use the notation "condition (receiving a message) / action (sending a message)" on the arrow from one state to the other to denote the condition and action effect of a state transition, M_i symbols correspond to the messages in Fig. 2.

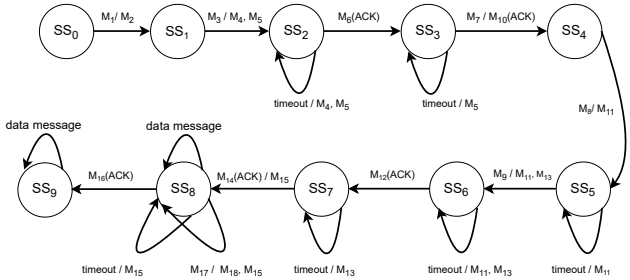

Figure 7: Inferred FSM of OpenVPN Server (UDP mode)

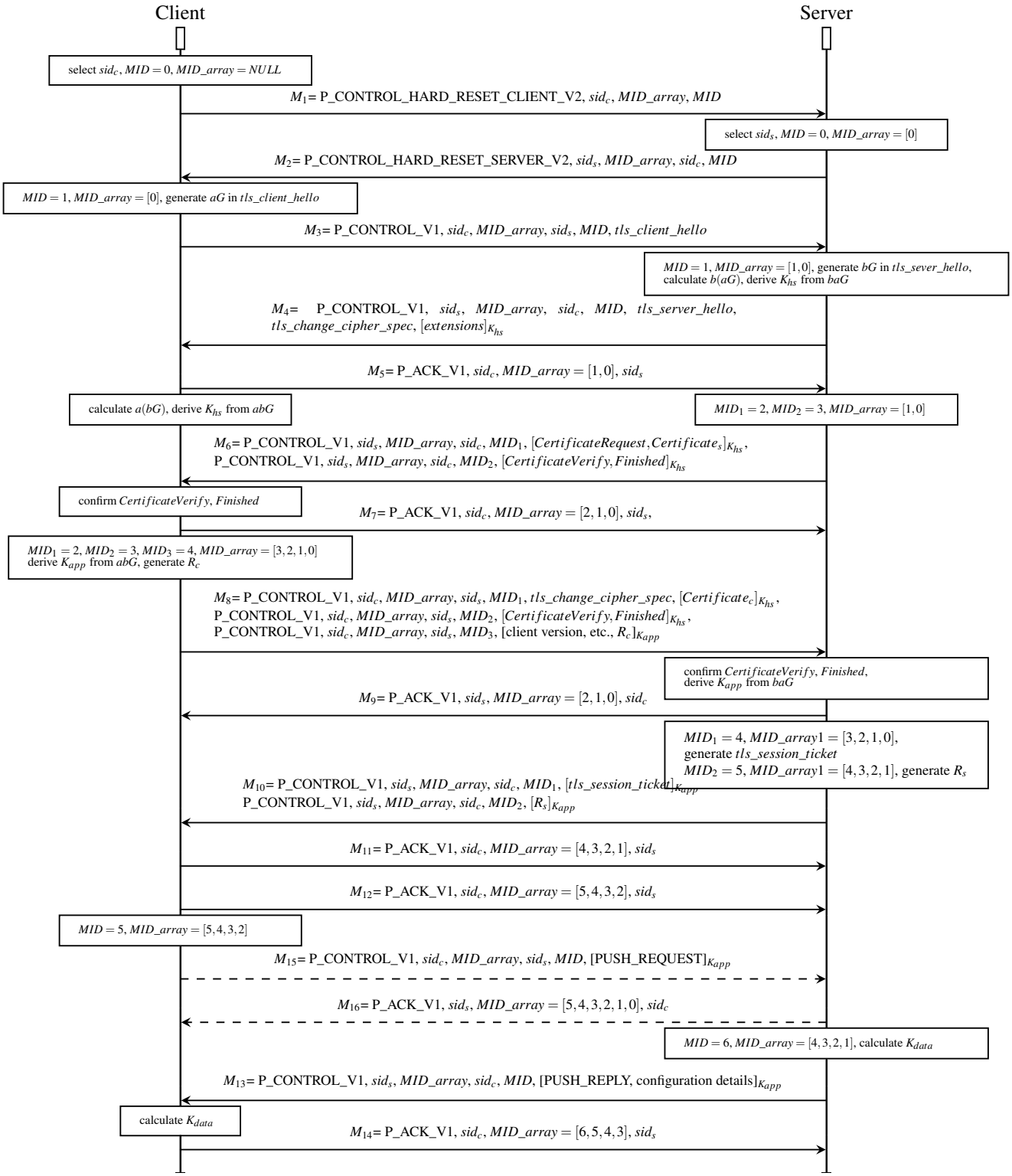

Figure 6: MSC of OpenVPN handshake (TCP mode, default TLS configuration), some fields omitted for simplicity.

Table 2: Representative Vulnerabilities of OpenVPN

Category	Details	Vulnerable Versions	Fixed or Not
DoS	Flooding with **`P_CONTROL_HARD_RESET_CLIENT_V2`** packets to the server can deny data transmission and connection establishment [26]	2.5.1	Yes (confirmed in testing and release notes of v2.6.)
DoS	Authenticated remote attacker sending a certificate with an embedded NULL character can cause a server crash (CVE-2017-7522) [49]	before 2.4.3 and before 2.3.17	Yes
DoS	Authenticated client sending 2^{32} packets can cause the exhaustion of Packet Identifiers thus the server crash (CVE-2017-7479) [19]	before 2.3.15 and before 2.4.2	Yes
VPN Traffic Fingerprinting	OpenVPN flows can be identified based on protocol features like Opcode byte pattern, uniform ACK packet size, and active probing getting the server's response [51]	all	No
MitM Attacks	TCP connections forwarded through a VPN tunnel can be hijacked by connection inference and sending spoofed packets [45]	all	partially fixed by certain OS (e.g. MacOS)
Operating System Exploits	Manipulating the routing policy can lead to traffic sent in plaintext bypassing the VPN encryption tunnel [52]	all	partially fixed by certain OS (e.g. Android)
Memory bugs	Stack buffer overflow, data leakage [49]	before 2.4.3 and before 2.3.17	Yes

Table 3: OpenVPN *Opcode* Information

Opcode Name	Value	Description
P_CONTROL_HARD_RESET_CLIENT_V2	7	Client hard reset
P_CONTROL_HARD_RESET_SERVER_V2	8	Server response
P_CONTROL_V1	4	Control packet
P_ACK_V1	5	Acknowledgement packet
P_DATA_V2	9	Data packet
P_CONTROL_SOFT_RESET_V1	3	Soft reset, a graceful transition from the old to new key
P_CONTROL_HARD_RESET_CLIENT_V3	10	Hard reset with client-specific *tls-crypt* key
P_CONTROL_HARD_RESET_CLIENT_V1	1	Deprecated
P_CONTROL_HARD_RESET_SERVER_V1	2	Deprecated
P_DATA_V1	6	Deprecated
P_CONTROL_WKC_V1	11	Deprecated

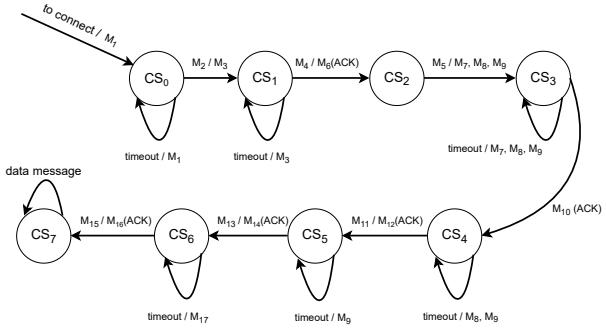

Figure 8: Inferred FSM of OpenVPN client (UDP mode)

E Additional Results

E.1 Previous Attack not Fixed for TCP Mode

We first investigate the status of a previous DoS attack. The work in [26] showed that flooding the server with **`P_CONTROL_HARD_RESET_CLIENT_V2`** packets (M_1) for OpenVPN (version 2.5.1), will easily cause a decrease in throughput and prevent legitimate connection attempts. Recall that M_1 is the first packet sent from the client to hard reset a VPN connection with the server (see Fig. 2).

During our experiments, we found that while the attack has been fixed when OpenVPN is configured in the UDP mode, the attack is still possible in the TCP mode. In the UDP mode the aforementioned vulnerability was patched with a replay limit check of 100 initial connection attempts per 10 seconds: this patch is enabled and works regardless if OpenVPN was configured with the *tls-auth* option. Recall that the *tls-auth* configuration option protects the integrity of control packets (including M_1) with an HMAC based on a static secret pre-shared key between the client and the server.

In the TCP mode, the flooding degrades the availability of the server. Specifically, the OpenVPN data channel throughput of a legitimate victim user can be significantly decreased (e.g. without *tls-auth* and attack sending rate 15 MB/s: from 220 MB/s to 110 MB/s, with *tls-auth* and attack sending rate 12 MB/s: from 220 MB/s to 115 MB/s) and the attack can trigger the server logging a state "Fatal TLS error (check_tls_errors_co), restarting".

E.2 Attack Result Google Cloud

We summarize the results of replay attack evaluation in Google Cloud in Table 4.

We present results for the DOS attacks with **`P_CONTROL_V1`** and **`P_CONTROL_V1`** packets conducted in more realistic network conditions in Google Cloud, for OpenVPN configured with UDP mode.

In UDP mode without *tls-auth*, sending 10,000,000 **`P_CONTROL_V1`** packets at the rate of around 17 MB/s can almost block Client2's data channel, compared with the normal throughput of 32 MB/s. Replay of 10,000,000 **`P_ACK_V1`** packets, which are shorter packets, at the rate of 1.5 MB/s can decrease Client2's data channel throughput to 12 MB/s.

In UDP mode with *tls-auth* enabled, sending 10,000,000 **`P_CONTROL_V1`** packets at the rate of around 20 MB/s can almost block Client2's data channel, compared with 18 MB/s without attack. While the server will log "TLS Error: incoming packet authentication failed from [IP:port address]. Authenticate/Decrypt packet error: bad packet ID (may be a replay)", it didn't do anything to ban such a replay attack. Replay of 10,000,000 **`P_ACK_V1`** packets, which are shorter packets, at the rate of 5 MB/s can almost block Client2's data channel throughput with similar logs on the server side.

E.3 Table of the Configuration Options with Improper Validation

We provide the table of the configuration options with improper validation for an input value of zero or over-large(2^{70}) in Table 5.

Table 4: Replay attack effects on the data channel throughput of the other legitimate client.

Mode	*tls-auth*	Replay Packet Type	Attack Sending rate (MB/s)	Throughput without attack (MB/s)	Throughput under attack (MB/s)
UDP	N	P_CONTROL_V1	17	32	<1
UDP	N	P_ACK_V1	1.5	32	12
UDP	Y	P_CONTROL_V1	20	18	<1
UDP	Y	P_ACK_V1	5	20	<1
TCP	N	P_CONTROL_V1	212	168	<1
TCP	N	P_ACK_V1	50	160	30
TCP	Y	P_CONTROL_V1	60	116	<1
TCP	Y	P_ACK_V1	10	116	30

Table 5: Configuration options with improper validation of a zero and/or over-large (2^{70}) input value

Configuration Option Name	Server or Client Configuration	Meaning
hand-window	both	Data channel key exchange must finalize within n seconds of handshake initiation by any peer (default=60)
max-routes-per-client	Server	Allow a maximum of n internal routes per client
mute	both	Log at most n consecutive messages in the same category
nice	both	Change process priority (>0 = lower, <0 = higher)
ping-exit	both	Exit if n seconds pass without reception of remote ping
ping	both	Ping remote once every n seconds over TCP/UDP port
ping-restart	both	Restart if n seconds pass without reception of remote ping
reneg-bytes	both	Renegotiate data chan. key after n bytes sent and recvd
reneg-pkts	both	Renegotiate data chan. key after n packets sent and recvd
reneg-sec max [min]	both	Renegotiate data chan. key after at most max (default=3600) seconds
tls-timeout	both	Packet retransmit timeout on TLS control channel if no ACK from remote within n seconds (default=2)
tran-window	both	Transition window – old key can live this many seconds after new key renegotiation begins (default=3600)
tun-mtu-extra	both	Assume that tun/tap device might return as many as n bytes more than the tun-mtu size on read (default TUN=0 TAP=32)
verb	both	Set output verbosity to n (default=1), ...,:6 to 11 – debug messages of increasing verbosity
script-security level	both	Where level can be 0–..., 1–...,2–:..., 3 – allow password to be passed to scripts via env
max-clients	Server	Allow a maximum of n simultaneously connected clients