Evaluating Susceptibility of VPN Implementations to DoS Attacks Using Adversarial Testing

Fabio Streun
ETH Zurich
fabio.streun@inf.ethz.ch

Joel Wanner
ETH Zurich
joel.wanner@inf.ethz.ch

Adrian Perrig
ETH Zurich
adrian.perrig@inf.ethz.ch

***Abstract*—Many systems today rely heavily on virtual private network (VPN) technology to connect networks and protect services on the Internet. While prior studies compare the performance of different implementations, they do not consider adversarial settings. To address this gap, we evaluate the resilience of VPN implementations to flooding-based denial-of-service (DoS) attacks.**

We focus on a class of *stateless flooding attacks*, which are particularly threatening because an attacker that operates stealthily by spoofing its IP addresses can perform them. We have implemented various attacks to evaluate the DoS resilience of four widely used VPN solutions and measured their impact on a high-performance server with a 40 Gb/s interface, which has revealed surprising results: An adversary can deny data transfer over an already established WireGuard connection with just 300 Mb/s of attack traffic. When using strongSwan (IPsec), 75 Mb/s of attack traffic is sufficient to block connection establishment. A 100 Mb/s flood overwhelms OpenVPN, denying data transfer through VPN connections and connection establishments. Cisco's AnyConnect VPN solution can be overwhelmed with even less attack traffic: When using IPsec, 50 Mb/s of attack traffic deny connection establishment. When using SSL, 50 Mb/s deny data transfer over already established connections. Furthermore, performance analysis of WireGuard revealed significant inefficiencies in the implementation related to multi-core synchronization. We also found vulnerabilities in the implementations of strongSwan and OpenVPN, which an attacker can easily exploit for highly effective DoS attacks. These findings demonstrate the need for adversarial testing of VPN implementations with respect to DoS resilience.

I. Introduction

In today's Internet ecosystem, enterprise networks are often dispersed over several locations, such as corporate branches, data centers, and infrastructure hosted by cloud providers. VPN systems are an integral part of these setups, serving as the glue that securely connects the different locations by encrypting and authenticating traffic between pairs of endpoints over an untrusted network such as the Internet. Given this role, VPN protocols are as ubiquitous in emerging SD-WAN deployments as they are in more traditional site-to-site connections. Moreover, the recent shift to remote work has led to a surge in VPN use [6], as they are also often deployed as an access control mechanism to protect network segments and services. Since VPN endpoints are usually exposed to the public Internet, they represent an attack surface for various adversaries. Combined with the critical reliance on the endpoints, they are an attractive target for DoS attacks.

All major VPN protocols integrate DoS defenses, such as source-binding cookie mechanisms. However, developing a VPN implementation with strong DoS resilience is challenging. While prior studies analyzed the performance of different VPN implementations [3], [21], their measurements do not consider adversarial settings.

We address this lack of adversarial testing by evaluating the DoS resilience of four widely used state-of-the-art VPNs: WireGuard, strongSwan (IPsec), OpenVPN, and Cisco AnyConnect (IPsec and SSL). This paper describes DoS mitigation mechanisms commonly used by VPNs and gives an overview of how each of the evaluated VPN implementations applies them. To obtain quantitative measurements of each VPN's resilience, we measure the performance of the VPN implementations while performing flooding attacks. Primarily *stateless flooding attacks* attacks are considered, which an adversary with spoofed IP addresses can perform stealthily. Through an in-depth analysis of the VPN implementations, we identify major performance bottlenecks and easily exploitable vulnerabilities. Additionally, we present a flexible framework that simplifies the launching of flooding attacks in a high-performance testbed network. We also look into automating the exploration of flooding attacks, which is challenging due to the huge number of possible attacks and the significant amount of time required to measure an attack's impact.

Overall, we demonstrate that today's standard implementations are highly vulnerable to flooding attacks, even though mitigation mechanisms are in place. Hence, they cannot reliably provide the first layer of DoS protection. Furthermore, our results indicate a significant deficit of adversarial testing.

This paper makes the following contributions:

- We evaluate four state-of-the-art VPN solutions on high-performance hardware, providing quantitative measurements of their resilience under realistic adversarial conditions.
- We analyze the VPN implementations, uncovering significant inefficiencies and multiple vulnerabilities in the implementations. All findings have been reported to the respective developers, resulting in CVE-2021-3568 and multiple patches.
- We present a framework capable of launching flooding attacks in a high-performance testbed network.

Network and Distributed Systems Security (NDSS) Symposium 2022
24-28 April 2022, San Diego, CA, USA
ISBN 1-891562-74-6
https://dx.doi.org/10.14722/ndss.2022.24043
www.ndss-symposium.org

II. Background

A. VPN Protocols

A VPN protocol session between two *peers* comprises two main phases: the handshake and data transfer. During the handshake, the peers authenticate each other, exchange session identifiers, and establish shared keys, which the peers subsequently use to encrypt and authenticate data transferred between them. One peer takes the role of the *initiator*, while the other is called the *responder*. Commonly, authentication is achieved through pre-shared keys, public keys, or certificates, and a variant of the Diffie-Hellman (DH) exchange is performed to establish the shared keys. Data transfer, identified through the session identifiers, is protected by symmetric-key cryptography.

For an adversary attempting to disrupt a VPN connection, the handshake phase is an attractive attack vector: authentication and key establishment rely on CPU-intensive operations, and memory has to be allocated to store information related to the new connection. Hence, an adversary provoking a large number of handshake executions at the server can exhaust the server's resources such that handshake or data transfer packets from legitimate peers cannot be processed and have to be dropped. On the other hand, a data transfer packet requires a valid session identifier, which can be verified efficiently. Furthermore, symmetric-key cryptography requires fewer CPU resources than authentication and key establishment.

B. DoS Mitigation Mechanisms

1) Source Blocking: A primitive technique to mitigate attacks is to filter requests based on their origin. A list of accepted sources (allowlist), a list of denied sources (denylist), or a combination thereof suffices to achieve that.

2) Rate Limiting: Since a DoS attack aims to consume excessive amounts of resources at the victim, the attacker will commonly send requests at higher rates than legitimate peers. This observation can be leveraged to identify and mitigate malicious behavior through rate limiting. VPNs apply this technique often to handshake requests, which are not meant to be sent at a high rate by a single peer. Usually, the IP address serves as an identifier for a peer, but a session ID can also be used if available.

3) Address-Binding Cookies: While various approaches to mitigate IP spoofing are deployed on the Internet [24], it is still widely possible to forge source addresses [15]. Because some adversaries can spoof their IP address, source blocking and rate-limiting mechanisms based on source IP addresses are not always effective. IP-based rate limiting can even be abused for DoS attacks. An adversary using the IP addresses of legitimate peers could exceed their rate limits causing the target to drop their requests.

Address-binding cookies, originally proposed for Photuris [12], provide protocol-level mitigation against spoofing attacks. While adversaries may be able to send packets with arbitrary source addresses, they cannot obtain responses to these packets. A VPN responder can utilize this by responding with a cookie to the initiator, which then repeats the request with the cookie proving control over the used IP address. The cookie is usually a cryptographic hash over the initiator's IP address, using a secret key such that only the responder can compute it. Such a cookie mechanism can be implemented to be stateless and computationally efficient. Since it adds a round trip to the handshake protocol, it should ideally be activated only when the responder suspects to be under attack using spoofed IP addresses.

Ultimately, the combination of rate limits and cookies prevents adversaries from exhausting amounts of resources disproportionately to the number of IP addresses they control.

4) Pre-Shared Keys: Another defense mechanism to protect a server from malicious requests is the use of pre-shared keys. If each request is authenticated with a key shared only among legitimate peers, requests from illegitimate peers can be detected with a lightweight verification step using symmetric cryptography. However, keys shared between multiple peers could be leaked to an adversary. Furthermore, an adversary in possession of previously sent requests could replay them and circumvent such a mechanism in the absence of replay or freshness checks.

5) Priority assignment: Some VPN implementations prioritize authenticated traffic over unauthenticated traffic to protect communication with legitimate peers. This can be achieved by enforcing stricter rate limits for unauthenticated traffic than authenticated traffic. Another approach is to process unauthenticated packets on threads with lower priority than authenticated packets. However, this prioritization can also be problematic since not all legitimate packets are authenticated. If the prioritization is too strict, also legitimate packets are dropped if there is a high load of authenticated traffic.

6) Parallelism: Some VPN implementations rely on a high degree of parallelism to utilize all CPU resources to achieve high performance. This principle can also improve DoS resilience by allowing the system to serve a higher rate of requests. Multi-core systems usually utilize receive-side scaling (RSS), which distributes the processing of received packets among multiple cores. When a packet is received, RSS calculates a hash over some predefined packet header fields, usually IP addresses and TCP/UDP ports. The hash is then used to determine which core will process the packet, and the packet is added to the core's receiver queue. Packets originating from different sources, i.e., packets from different flows, are likely to be processed by different cores. On the other hand, packets of a single flow are always processed by the same core. The hash function usually includes a secret value, such that adversaries cannot know the mapping between the hash fields and the cores. Otherwise, an adversary may deteriorate the performance of a specific flow by overwhelming only a single core.

7) Deferred resource allocation: Often optimizing the processing of legitimate requests also optimizes the processing of malicious requests. However, optimizations, such as pre-computing values and pre-allocating memory, can introduce unnecessary overhead to the processing of malicious requests. In general, deferring resource allocations during packet processing as much as possible minimizes the amount of resources allocated for malicious traffic.

III. Problem Statement

For our evaluation, we consider a VPN setup on the public Internet with one central VPN server and multiple clients. The central server uses a static IP address, which can be considered publicly known. The clients connecting to the server may use arbitrary IP addresses. Hence, the server cannot apply source blocking. We consider an adversary that floods the server with packets to deny all connections to the server or a specific one (e.g., a critical tunnel between two corporate locations).

A. Threat Model

Our threat model considers adversaries with minimal knowledge about the targeted infrastructure: we assume that the adversary knows the VPN protocol used and the location (i.e., IP address and port) at least of the VPN server. An adversary can obtain such information in various ways: insider knowledge, traffic analysis, port scanning, or infiltrating a service (e.g., signing up as a legitimate customer of an SD-WAN service). Further, we assume that the adversary is *off-path*, i.e., it cannot observe communication between the server and a client. Since our evaluation aims to determine the resilience of VPN implementations exposed to the public Internet, it is a realistic assumption that some attackers can spoof their source address [15]. Moreover, we assume that the adversary cannot break the cryptographic primitives used by the VPNs.

Primarily we focus on *stateless flooding attacks*, where an adversary attempts to exhaust a victim's resources by flooding it with packets that do not depend on responses by the victim, hence, stateless. Because the attacks are stateless, they can be performed stealthily with spoofed IP addresses.

B. Attack Impact Metrics

Various metrics exist to describe an attack's impact on a VPN's performance. For our evaluation, we focus mainly on the following two metrics:

Throughput: The maximum rate at which data is transferred over an established VPN connection between two endpoints.

Connection setup time: The time required for two VPN endpoints to complete the handshake, i.e., establish a new connection.

Throughput and connection setup time significantly influence the VPN user experience and are regarded as highly relevant. They are also commonly used in performance evaluations without adversaries. An attack's impact is regarded as significant if the attack significantly reduces throughput or significantly increases connection setup time.

We quantify an implementation's DoS resilience by the amount of attack traffic required to achieve a significant impact. Therefore, we are particularly interested in flooding attacks that block throughput or deny connection establishment with little traffic.

IV. Attack Implementation

We have implemented a framework that enables efficient testing of stateless flooding attacks on a high-performance testbed. Furthermore, we have extended our framework with an attack space exploration algorithm to automatically discover efficient flooding attacks. The source code is publicly available[1].

A. Line-Rate Traffic Generation

In order to carry out flooding attacks, an attack tool must be able to achieve high packet rates while enabling low-layer packet manipulation (e.g., source address spoofing). For this purpose, standard application-layer software sockets are not flexible enough. Alternatively, raw sockets allow low-layer packet manipulation, but the kernel networking stack significantly limits the performance of such an approach. We use the DPDK framework to overcome these limitations. DPDK offers efficient packet processing by bypassing the Linux networking stack and allowing applications in user space to control the networking devices directly [7]. Since this low-level control requires a considerable amount of setup, we have built our implementation with `libmoon` (the underlying framework of MoonGen [5]), which simplifies the setup process and accelerates development [9].

Our framework further encapsulates boilerplate code, which most stateless flooding attack implementations otherwise repeat. Hence, functionalities including basic packet manipulations, rate control, thread management, and NIC offloading are already provided with a command-line interface to control them all. To implement a new flooding attack, a user only needs to prepare the different kinds of packets used in the flood and provide packet field modifications to be performed during the flood. Pcap files containing previously captured packets offer a simple way to prepare packets. This is especially helpful for packets with complex structures. Packet modifications during the flood allow, for instance, to send the same packet with different source IP addresses. In most cases, it is sufficient to define a simple counter, which is increased for each sent packet and is used to modify a particular packet field. Throughout our evaluation, the framework simplified the implementation of numerous flooding attacks. Furthermore, the command-line interface shared by all attacks facilitates the integration into an automated testbed.

B. Attack Space Exploration Algorithm

The space of possible flooding attacks an adversary can perform to disrupt VPN connections is extensive, and manually exploring it is impractical. Therefore, we look into automating the exploration using our flooding attack framework. Because measuring the impact of a single attack requires a considerable amount of time (on the order of seconds), a comprehensive search is impossible for a large attack space. For this reason, we suggest using an optimization algorithm to guide the exploration and find efficient flooding attacks.

A simple attack space consists of flooding attacks in which the same packet is sent repeatedly at a specific rate. Given a list of packets known by the adversary, an attack space is defined, which can be explored automatically, e.g., by steadily increasing the flood's rate until the VPN connection is completely disrupted. This approach reveals which packets impact the VPN's performance more efficiently than others.

[1] https://github.com/fstreun/Flood-Generator

However, a sophisticated adversary will vary the packets during a flooding attack. We thus extend the previously described simple attack space and include flooding attacks in which a packet mix is sent repeatedly at a specific rate. Given a list of n packets known by the adversary, a flood can be defined by vector $\mathbf{x} = (x_1, \ldots, x_n)$, where x_i is the bit rate at which packet i is sent to the victim. As we are primarily interested in finding *efficient* attacks, i.e., attacks with high impact but low cost, we can use optimization algorithms to guide the exploration of the attack space. Using the throughput over a VPN connection $\mathrm{tp}(\mathbf{x})$ during the flood as the impact metric and the flood's bit rate $|\mathbf{x}| = x_1 + \ldots + x_n$ as the cost, we define the following optimization problem:

$$\begin{aligned} \text{minimize} \quad & \mathrm{tp}(\mathbf{x}) + \alpha \cdot |\mathbf{x}| \\ \text{subject to} \quad & x_i \geq 0 \text{ for } i = 1, \ldots, n \\ & |\mathbf{x}| \leq bw \end{aligned}$$

The constant α controls the trade-off between the attack's cost and impact. If α is large, the exploration algorithm favors attacks with low cost, i.e., attacks that require little bandwidth. If α is small, the exploration algorithm favors attacks that minimize the throughput, giving less weight to the required attack rate. The constant bw represents the attacker's maximum achievable bit rate.

We choose to use the differential evolutionary algorithm [25] provided by the Python library SciPy [28] as the optimization algorithm. A differential evolutionary algorithm requires an initial set of candidate solutions to the optimization problem. Over multiple cycles, the candidate solutions are optimized by mixing them and removing the worst ones from the set.

Given a list of packets known by the adversary, our implementation determines first the optimal attack rate for all trivial mixes, i.e., mixes consisting of only one packet. Using the resulting set of trivial floods as the initial set, the differential evolutionary algorithm then explores the space of mixed-packet floods over several cycles and tries to find the most efficient one.

Section VI-E describes the process of applying this mixed-packet flood exploration algorithm to WireGuard, and how it can be helpful in practice. The algorithm did not reveal new interesting attacks for the other VPNs due to multiple reasons. Firstly, it revealed the obvious: floods using initiation packets or tiny packets can be the most efficient ones. Secondly, the list of packets provided was either too small to find a new packet mix that efficiently degrades the throughput, or too big for the algorithm to converge to an efficient packet mix in a reasonable time frame.

V. Evaluation

A. Network Topology

Figure 1 shows our setup consisting of three machines connected over a $40\,\mathrm{Gb/s}$ star topology using a single switch, modeling a VPN connection over the Internet with an off-path adversary. We refer to one endpoint as the *server* and to the other as the *client*. The attacker attempts to disrupt the VPN connection by flooding the server, i.e., the device under test (DUT), with packets.

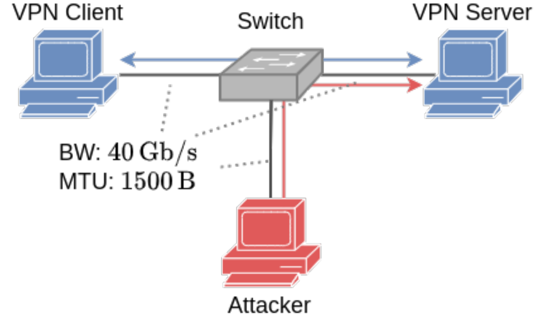


Fig. 1. The network topology used in our evaluation setup. Three high-performance servers are connected through a switch over $40\,\mathrm{Gb/s}$ links.

For the open-source VPN tests, the VPN server is a commodity server running Ubuntu 18.04.4 LTS.[2] The machine is equipped with two 18-core CPUs[3], offering 72 virtual cores with hyper-threading, and a $40\,\mathrm{Gb/s}$ network card.[4] RSS is activated with the maximum number of supported queues on this system, i.e., 64 queues. For RSS, IP addresses and port numbers in a packet are used as hash fields.

The setup for the Cisco AnyConnect tests is slightly different. The VPN server is a Cisco Firepower® 9300 SM-24 security appliance running Cisco Adaptive Security Appliance (ASA) Software.[5] The machine is equipped with two 12-core CPUs[6], offering 48 virtual cores with hyper-threading, and two $40\,\mathrm{Gb/s}$ interfaces. The outside interface is connected to the switch. The inside interface is connected to the machine which serves as the VPN server in the other tests.[7]

Specifications for the client and attacker are provided in Appendix A.

The maximum transmission unit (MTU) on the network interfaces is set to $1500\,\mathrm{B}$ to simulate real-world limitations of the public Internet.

B. Attack Impact Measurements

While the VPN server is the DUT, the client machine measures the attack impact according to the metrics defined in Section III-B.

Throughput measurement: `iperf` over a single TLS connection is used to measure the maximum achievable throughput from the client over the already established VPN connection. The measurement is taken over a duration of 10 seconds, during which the attack is performed.

Connection setup time measurement: The Linux `time` command in combination with `ping` is used to obtain the connection setup time by measuring the time required by the VPN client to connect to the VPN server and perform the first packet exchange over the VPN connection. Attacks are performed over a duration of 25 seconds to allow for multiple connection establishment tries during the attack.

[2] GNU/Linux 5.10.11-051011-generic x86_64
[3] Intel® Xeon® CPU E5-2695 v4, 2.10 GHz
[4] Intel® Ethernet Controller XL710 for 40 GbE QSFP+
[5] Version 9.16(2)7 (SSP Operating System Version 2.10(1.159))
[6] Intel® Xeon® CPU E5 series, 2.20 GHz
[7] The additional machine is required to measure throughput over the ASA.

The measurement continues for 60 more seconds to see if the client is able to connect after the attack has terminated.

Impact measurements are taken across five runs. The plots in the following sections show mean and standard deviation values across these runs. To eliminate carry-over effects across runs, the VPN applications running on client and server machines are always restarted before each run.

To further analyze an attack's impact on the DUT, we collect metrics about packets dropped by the network interface controller (NIC) and CPU utilization, e.g., with `ethtool` and `perf` sampling, respectively.

C. Attack Traffic Flow Configurations

Since we test the VPN implementations on multi-core machines with RSS, different attack traffic flow configurations must be accounted for. We consider the following three attack traffic flow configurations:

single-flow (sf): The attacker uses a single source IP address and port such that all attack packets are mapped to the same receiver core. With high probability, the client's packets are mapped to another core.

multi-flow (mf): The attacker uses multiple source IP addresses or source port numbers to distribute the attack packets among all receiver cores. This configuration is likely favorable for an attacker trying to disrupt all connections to the server.

client-flow (cf): The attacker uses the same IP address and port as the client such that attack packets and the client's packets are mapped to the same receiver core. This configuration can be favorable for an attacker targeting a specific connection. However, the attacker must, firstly, know the client's IP address and the port used, and secondly, spoof its address with the client's address.

D. Overview of Evaluation

For each VPNs we evaluate its resilience against flooding attacks using initiation requests. Initiation request floods are commonly known, and all evaluated VPN protocols and implementations have mechanisms to mitigate their impact. Additionally, we also evaluate other stateless flooding attacks, which were found manually or with the help of our attack space exploration algorithm.

Table I provides an overview of the evaluation results. The table contains the lowest attack rate that achieves the given impact using the most efficient attack found for each VPN. We indicate two different types of impact thresholds: (a) throughput reaches less than 5% of the throughput achieved in the absence of attacks, and (b) connection establishment is denied during the duration of the flooding attack, i.e., 25 seconds. These thresholds provide a quick point of comparison between the different implementations. The overview shows that for all VPNs, a couple of hundred Mb/s of attack traffic suffice to either practically deny throughput or completely block connection establishment.

For WireGuard, our attack space exploration algorithm discovered a mixed-packet flood, which completely disrupts throughput with $300\,\mathrm{Mb/s}$ in the client-flow configuration.

TABLE I. AN OVERVIEW OF THE EVALUATION RESULTS. THE VALUES SHOW THE ATTACK RATES AND THE FLOW CONFIGURATIONS REQUIRED TO ACHIEVE THE GIVEN ATTACK IMPACT.

	$\leq 5\%$ throughput		setup time $\geq 25\,\mathrm{s}$	
WireGuard	$300\,\mathrm{Mb/s}$	(cf)	$6\,\mathrm{Gb/s}$	(cf)
strongSwan (IPsec)	$15\,\mathrm{Gb/s}$	(mf)	$75\,\mathrm{Mb/s}$	(mf)
OpenVPN	$50\,\mathrm{Mb/s}$	(cf)	$100\,\mathrm{Mb/s}$	(mf)
Cisco (IPsec)	$300\,\mathrm{Mb/s}$	(mf)	$50\,\mathrm{Mb/s}$	(mf)
Cisco (SSL)	$50\,\mathrm{Mb/s}$	(mf)	$200\,\mathrm{Mb/s}$	(mf)

With the multi-flow configuration, the packet mix requires roughly $700\,\mathrm{Mb/s}$. For strongSwan, flooding the server over multiple flows with initiation requests at a rate of $75\,\mathrm{Mb/s}$ suffice to consistently deny connection establishment due to a found vulnerability in the implementation. OpenVPN can also be overwhelmed with an initiation packet flood. With the client-flow configuration, roughly $50\,\mathrm{Mb/s}$ of attack traffic practically deny the throughput between the client and the server. Using multiple flows, $75\,\mathrm{Mb/s}$ are required. When flooding Cisco's IPsec implementation with initiation requests, $50\,\mathrm{Mb/s}$ suffice to block connection establishment. Cisco's SSL-based VPN is easiest overwhelmed with minimal-sized UDP packets. Roughly $50\,\mathrm{Mb/s}$ are sufficient to deny throughput.

The following sections on WireGuard, strongSwan, OpenVPN, and Cisco describe the different attacks in detail and provide an analysis of their impact.

VI. WIREGUARD

WireGuard is a relatively new VPN protocol that differentiates itself from competing solutions through its simplicity, high performance, and modern design [3]. WireGuard is "cryptographically opinionated", i.e., it only offers a single cipher suite, and keeps configuration options to a minimum. This approach allows its protocol specification and source code to be significantly less complex than other VPNs, which commonly offer multiple cipher suites and a plethora of configuration options. This minimalist approach ultimately reduces the risk of implementation bugs and possible misconfigurations that lead to vulnerable deployments.

In a WireGuard setup, each peer has a unique key pair for identification and authentication. For two peers to establish a connection, both peers have to know the public key of the other peer. The handshake is an instance of the *IK* pattern defined by the Noise protocol framework [19] and relies on DH operations for authentication and key-derivation. Under normal circumstances, the handshake requires one initiation request and one response to complete, as shown in Fig. 3. As its transport layer, WireGuard uses UDP for all packets.

The implementation published for the Linux operating system is a kernel module and was included in the kernel tree with the Linux 5.6 release.[8] To increase performance, the processing is distributed among all available CPU cores (Fig. 2). When a packet is received, RSS assigns it to a core, on which it is then processed by the kernel stack and classified by WireGuard as a handshake or data packet. Handshake packets are queued to the handshake list and assigned to

[8]Besides the kernel module, also a cross-platform Go implementation exists.

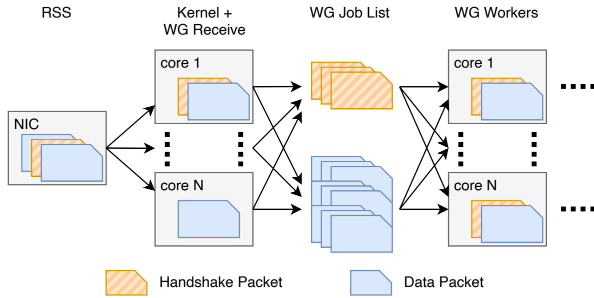


Fig. 2. WireGuard's packet processing pipeline: Packets are initially processed on the receiving core, which distributes work among dedicated workers spawned on all cores. Multiple queues are used for incoming packets: a handshake list and a data list for each established connection.

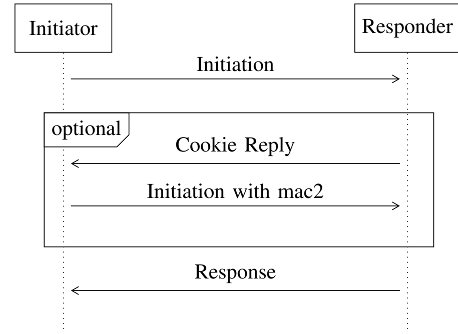


Fig. 3. WireGuard's handshake, including a cookie mechanism that is activated if the responder is under load.

handshake worker threads, which are spawned on all available cores. The handshake worker then (a) dequeues the packet, (b) validates the cryptographic message authentication codes (MACs) (described later), (c) processes it according to the Noise protocol framework, and finally (d) stores the connection state in memory. The processing of initiation and response packets only differ in Step (c).

A. DoS Defense Mechanisms

WireGuard follows the principle that memory allocation or modification is only performed for authenticated peers. Hence, a per-peer rate limit for handshakes is applied to mitigate memory exhaustion attacks.

As a first line of defense against CPU exhaustion attacks, WireGuard leverages the peers' public keys as pre-shared keys. The initiation packet contains a keyed-hash MAC (HMAC) called `mac1`, which is the hash over all WireGuard fields (up to the `mac1` field) using the receiver's public key as the hash key. The responder drops any incoming initiation packets without a valid `mac1`, ensuring that only initiation packets from peers with knowledge of the public key are processed.

However, if the victim's public key is known to the attacker, the `mac1` does not provide any protection. Furthermore, an attacker in possession of a previously sent initiation packet can pass this check by replaying the packet.

As a second line of defense, WireGuard uses a novel cookie mechanism and IP-based rate limiting. If the responder receives an initiation packet with a valid `mac1`, it may respond with a cookie, which is encrypted with the responder's public key. The initiator decrypts the cookie, uses it to compute a second HMAC called `mac2`, and repeats the request. Since the peers never exchange the cookie in clear, an attacker without knowledge of the responder's public key cannot create packets with valid `mac2` even if possessing an initiation request with a valid `mac1`.

Furthermore, the implementation assigns low thread priority to handshake workers such that a flood of handshake messages would not monopolize the CPU and prevent previously established connections from being served. To mitigate algorithmic complexity attacks, it uses fast hash tables for peer lookup and secret keys for the hash functions.

B. Setup and Configuration

For the WireGuard evaluation, we use the newest version provided for Ubuntu at the time of writing.[9] The client and the server are configured to use the default WireGuard port (51820). Otherwise, the client would use a random port, complicating the evaluation of client-flow flooding attacks. Besides that, WireGuard does not offer much more configuration options. The cookie mechanism is implemented to activate as soon as the handshake packet list exceeds half of its capacity. Tate limits are also hard-coded values.

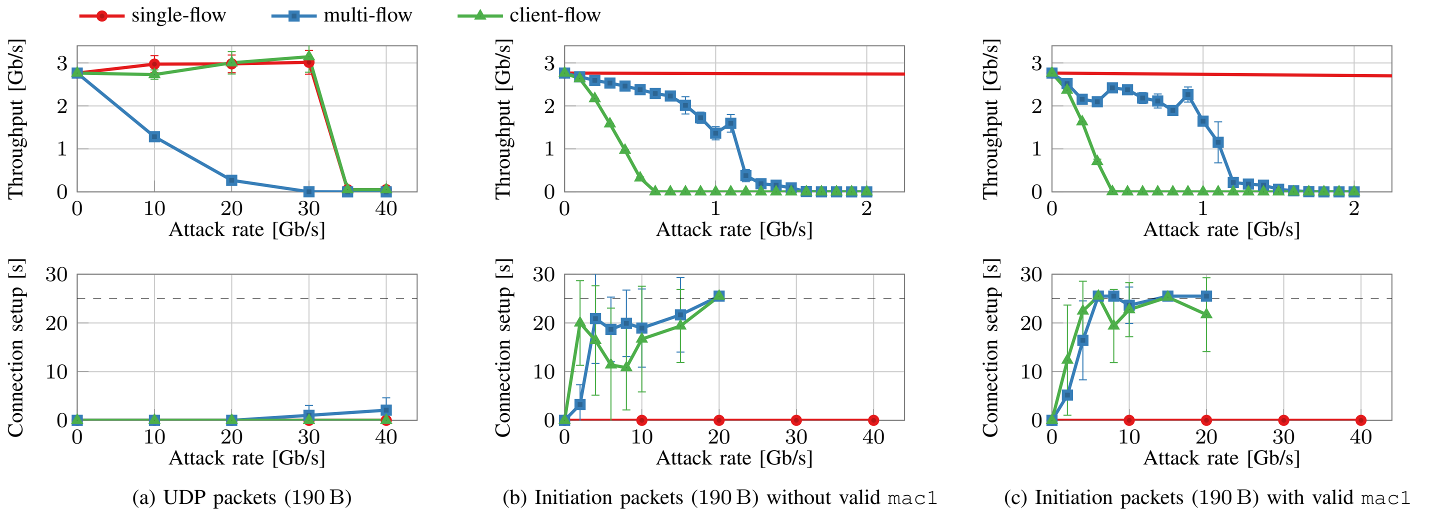
Measuring the VPN's performance under normal circumstances reveals that a maximum throughput of roughly $2.8\,\mathrm{Gb/s}$ can be achieved on the system under evaluation.

C. Flooding Attacks

To evaluate the effectiveness of WireGuard's DoS mitigation mechanisms against initiation packet floods, we first establish a baseline. As a baseline, representing a theoretical upper limit on performance under attack, we flood the server with UDP packets of the same size as a WireGuard initiation packet, i.e., $190\,\mathrm{B}$, but with a payload of only zeros (Fig. 4a). If the flood is performed with the single-flow or the client-flow configuration, the throughput is not negatively affected up to an attack rate of $30\,\mathrm{Gb/s}$, at which point the network is nearing saturation. However, if the flood spreads multiple flows, we observe that WireGuard's throughput steadily decreases as the attack rate increases. Connection setup time is not significantly affected by UDP floods up to rates of $30\,\mathrm{Gb/s}$ independent of the flow configuration. For higher rates, the connection establishment sometimes requires one retransmission, i.e., succeeds after $5\,\mathrm{s}$.

Flooding the server with initiation requests drastically impacts WireGuard's performance, even if the packets do not contain valid `mac1` fields, as shown in Fig. 4b. **If the attack traffic spreads over multiple flows, the achievable throughput collapses at attack bandwidths above $700\,\mathrm{Mb/s}$ and drops to zero for $1.6\,\mathrm{Gb/s}$.** If the client's source IP and port number are used, the attack is even more effective: with just $600\,\mathrm{Mb/s}$, the attack already stops practically all legitimate traffic through the VPN connection. To consistently deny connection establishment, still, roughly $20\,\mathrm{Gb/s}$ of attack

[9] 1.0.20200513-1 18.04.2



(a) UDP packets (190 B) (b) Initiation packets (190 B) without valid `mac1` (c) Initiation packets (190 B) with valid `mac1`

Fig. 4. Performance achieved by WireGuard under flooding attacks using different types of packets.

traffic are required. Again, the flood's impact with the single-flow configuration is insignificant.

The flood becomes slightly more effective if the initiation packets contain a valid `mac1` field, as shown in Fig. 4c. Connection establishment is consistently denied with roughly 6 Gb/s of attack traffic with the multi-flow or the client-flow configuration.

We have also evaluated the impact of initiation request floods with valid `mac1` and `mac2` fields (Appendix B). However, with valid `mac2` fields, the flood is not significantly more effective than without.

D. System Instrumentation

During single-flow and client-flow floods, the overall CPU utilization of the server never exceeds 25%, independent of the attack rate. Since RSS assigns all attack packets to the same core, only a single receiver thread performs the initial processing and manages their distribution among the different handshake workers. For attack rates above 500 Mb/s, this thread exhausts the entire capacity of the core it is running on. This causes packets assigned to this core to be dropped sometimes, while packets mapped to other cores are still processed. Therefore, client-flow attacks are devastating, while single-flow attacks have a negligible impact. Profiling reveals that the overloaded receiver thread spends roughly 50% of its runtime assigning packets to particular workers and waking them up.

During multi-flow attacks, RSS distributes the packets among multiple cores. Hence, multiple receiver threads perform the initial processing of the attack packets. For attack rates above 1.5 Gb/s, the receiver threads and handshake workers require 45% and 54%, respectively, of all CPU resources, hence, completely exhausting system resources. Surprisingly, **approximately 90% of CPU time is spent in spinlocks to add/remove packets to/from the handshake packet queue.** This observation clearly identifies multi-core synchronization as the main bottleneck.

TABLE II. The most efficient mixed-packet flood found by the exploration algorithm. For each packet, the relative frequency, packet rate (kp/s), and bit rate (Mb/s) are listed.

Packet type (size)	Frequency [%]	kp/s	Mb/s
Handshake Initiation (190 B)	15.6	116.3	176.7
Handshake Response (134 B)	28.1	209.3	224.3
Handshake Cookie Reply (106 B)	53.1	395.3	335.2
Transport Data (170 B)	0.0	0.0	0.0
Transport Data (170 B)	0.0	0.0	0.0
Transport Data Keepalive (74 B)	3.1	23.3	13.8

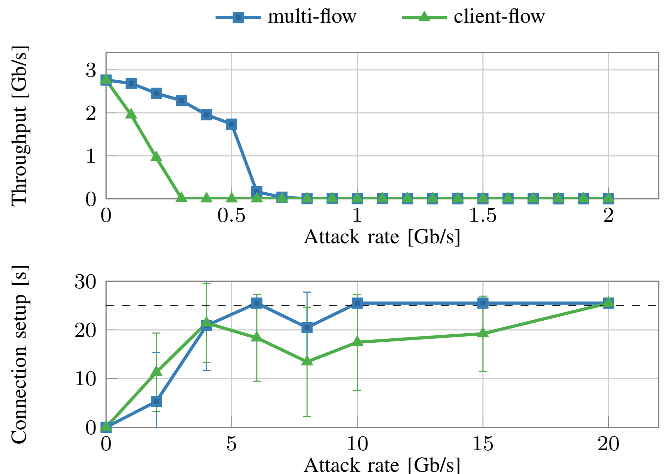


Fig. 5. Performance of WireGuard under a mixed-packet flood found by our exploration algorithm.

E. Attack Space Exploration

We apply the automated attack space exploration algorithm described in Section IV-B. We consider an adversary with minimal knowledge of the setup, which performs flooding attacks using multiple different IP addresses. We provide a list of 6 generic WireGuard packets to the algorithm, containing all packet types as defined by the protocol specification. The attack cost weight α is set to 0.001, such that the search prioritizes attacks with high impact.

After 58 evolution steps and a runtime of approximately 1.5 hours, **the exploration algorithm discovers a packet mix that completely disrupts the throughput with just 750 Mb/s.**

The mix, shown in Table II, clearly favors handshake packets over data packets. Out of the data packets, only the keepalive packet, the smallest possible data packet, is represented in the mix. Furthermore, the share of each handshake packet (including the cookie packet) in the mix correlates with the packet's size. Hence, this distribution indicates that the handshake packet rate is a major factor for an efficient flooding attack. The measurements in Fig. 5 show that the mixed-packet flood is significantly more effective than the initiation packet flood. If the attack is performed on the client's flow, roughly 300 Mb/s are sufficient to deny throughput. Even though the exploration algorithm only tries to optimize the flood's efficiency regarding the throughput, the found mix also denies connection establishment with a lower attack rate than the previously evaluated attacks. Roughly 6 Gb/s suffice to block new connections from being established.

Applying the gained insights, we also evaluate a flood based solely on the cookie response packet (Appendix B). Even though the cookie flood achieves a higher handshake rate than the mixed-packed flood for the same bit rate, it is slightly less efficient.

F. Discussion of WireGuard Results

The design of WireGuard appears to be sound against DoS attacks: the protocol cleverly leverages shared public keys as an additional layer of defense if they are kept secret. A sophisticated cookie mechanism and rate limiting protect against spoofing attacks. Furthermore, the multi-threaded implementation aims to utilize all available computation resources, increasing the number of requests, including malicious ones, it can process.

However, our evaluation has exposed significant weaknesses in the implementation that negate the DoS resilience properties of the design. Our flooding attacks cause WireGuard to spend most of its processing time in spinlocks, indicating that CPU resources are wasted due to suboptimal thread synchronization. This has a severe impact on the data processing of legitimate connections. Interestingly, flooding the server with handshake packets affects data transfer more than connection establishment, even though WireGuard tries to prioritize the processing of data packets.

We have applied the attack space exploration algorithm to find an efficient flooding attack beyond manual analysis. The found packet mix also helped identify a significant bottleneck in the implementation, exploited through a high handshake packet rate.

VII. STRONGSWAN: IPSEC

Internet Protocol Security (IPsec) is a secure network protocol suite that is widely used in VPN solutions. Multiple IETF working groups have continually updated the IPsec standards and published them as RFCs [8]. The relevant pieces of the IPsec protocol suite for our analysis are Internet Key Exchange Version 2 (IKEv2) [13] for the handshake, and Encapsulating Security Payload (ESP) [14] for the data transport.

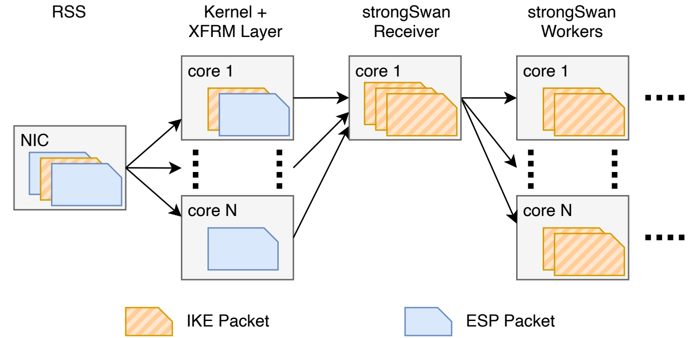


Fig. 6. strongSwan's packet processing pipeline: while ESP packets are directly processed by the kernel, IKE packets are first inspected the receiver thread and then distributed among other workers.

An IKEv2 exchange consists of two phases:

a) `IKE_SA_INIT`*:* The first phase establishes a secure channel between the initiator and responder and consists of one request and one response. Both peers provide a randomly generated security parameter index (SPI) to identify the connection, and symmetric keys are established using the DH key exchange. SPIs, keys, and other information related to the connection are stored in an IKE security association (SA). Since the peers have not authenticated each other yet, the IKE SA is also called *half-open*. Any subsequent packet is identified with the corresponding SPI, and its payload is encrypted and integrity-protected using the established key.

b) `IKE_SA_AUTH`*:* Mutual authentication is performed in the second phase of the IKEv2 protocol, e.g., by using pre-shared keys or certificates. The `IKE_SA_AUTH` phase could complete in one round trip but may require more.

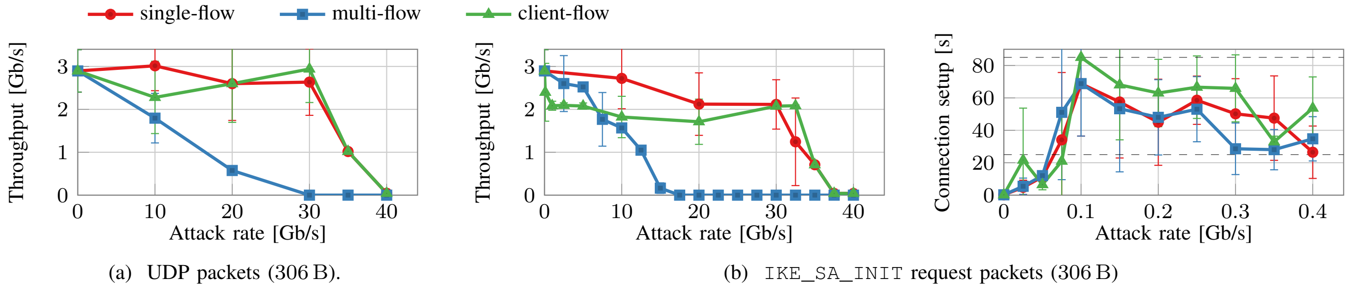
IKEv2 uses UDP as its transport layer. ESP functions directly on top of the IP layer. However, ESP can also be encapsulated in UDP packets to enable the traversal of NAT devices, which we activate for our evaluation. While the `IKE_SA_INIT` exchange uses port 500, the `IKE_SA_AUTH` exchange switches to port 4500, which the UDP-encapsulated ESP packets subsequently use.

A widely used open-source VPN solution based on IPsec is strongSwan [27], which provides an implementation of the IKEv2 protocol. The implementation consists of a user-space daemon that handles the IKE packets and distributes the processing of them among multiple threads. For the encryption and authentication of data packets (i.e., ESP packets), the IPsec stack implementation in the operating system's kernel is responsible (called *XFRM* in Linux). Figure 6 shows a simplified overview of the packet processing pipeline.

A. DoS Defense Mechanisms

Numerous RFCs have been published that update the IKE protocol or recommend best practices for implementation and configuration to increase DoS resilience. RFC 8019 [18] is particularly interesting, as it summarizes possible DoS attacks and mitigation mechanisms.

Since during the `IKE_SA_INIT` phase, a DH exchange is performed, and memory is allocated, an adversary may exhaust



(a) UDP packets (306 B).

(b) IKE_SA_INIT request packets (306 B)

Fig. 7. Performance achieved by strongSwan under flooding attacks using different types of packets.

a server's resources by flooding it with IKE_SA_INIT requests. Such an attack does not require any special knowledge from the adversary, as no authentication takes place during this phase of the protocol.

CPU resources required in the IKE_SA_INIT phase can be significantly decreased by deferring the key derivation into the IKE_SA_AUTH phase. This is possible because the IKE_SA_INIT response requires only the DH public key, which can efficiently be generated. The symmetric DH keys are at the earliest required for the decryption of the payload in the following IKE_SA_AUTH request.

To mitigate memory exhaustion attacks, it is recommended to limit the creation of half-open IKE SAs for a single source IP. Since an adversary capable of spoofing its sources address can circumvent such IP-based limits, IKEv2 makes use of a cookie mechanism, which activates if the number of half-open IKE SAs exceeds a certain threshold. When receiving an initiation request, the responder does not create a new half-open IKE SA but instead responds with a cookie. The initiator then repeats its request but with the cookie added to its payload, proving control over the used IP address. The responder creates half-open IKE SAs only for requests with a valid cookie and if the limits allow it.

B. Setup and Configuration

For our IPsec setup with strongSwan[10], we use IKEv2 as the initiation protocol with a certificate-based authentication method. A recommended cipher set[11] is chosen [26], which utilizes the dedicated hardware accelerators on the machines. Moreover, strongSwan is configured to spawn a thread for each logical core, i.e., 72 threads in total, for IKE packet processing. The *cookie threshold* is set to 32, causing the cookie mechanism to activate if the number of half-open IKE SAs exceeds this threshold. Additionally, a maximum of 128 concurrent half-open IKE SAs are allowed for a single IP before the address is blocked from creating more. Default values are used for all remaining configuration options.

In absence of any attacks, the client achieves a throughput of roughly 2.9 Gb/s, which is sightly more than WireGuard achieves.

C. Flooding Attacks

As a comparison to the initiation flood on WireGuard described in Section VI-C, we evaluate an equivalent attack on IPsec, i.e., we evaluate the impact of an IKE_SA_INIT request flood on strongSwan. For this attack, we use an IKE_SA_INIT request packet of the same structure as used by a legitimate peer.

To establish a baseline, we first evaluate floods consisting of generic UDP packets of the same size, i.e., 306 B, but with only zeroes as payload. The results are depicted in Fig. 7a. For the single-flow and client-flow configurations, the measurements show significant variance up until 30 Gb/s, beyond which the throughput falls sharply. If the UDP flood is performed over multiple flows, the impact is stronger: the maximum throughput between the peers is inversely proportional to the attack rate. Connection setup time is not affected by any of the UDP floods.

The impact of the IKE_SA_INIT request floods are shown in Fig. 7b. The single-flow attack degrades the connection's throughput only slightly more than the baseline UDP flood. If the attack is performed over multiple flows, throughput decreases steadily with increasing attack rate, similar to the baseline. Attack rates above 15 Gb/s largely deny data transfer between the peers. If the attacker uses the client's address for the traffic, the attack starts to impact the connection at much lower rates as with an unrelated single flow: at an attack bandwidth of 1 Gb/s, throughput is reduced by roughly 30%.

On the other hand, connection setup time is drastically affected by the attack, independent of the attack traffic configuration: **with just 75 Mb/s, the attack inhibits the client from connecting to the server during its full duration.** For the attack to be effective, the IKE_SA_INIT requests must use a high number of different initiator SPIs to pass a retransmission filter implemented in strongSwan.

D. Discussion of strongSwan Results

1) Throughput: As expected, single-flow attacks affect the throughput the least among all considered flow constellations. Since the IKE_SA_INIT requests and data transport packets use different port numbers, RSS assigns them to different cores with high probability. Hence, the client-flow attack configuration is not expected to affect the throughput significantly more than the single-flow constellation. Nevertheless, the measurements show a surprising drop of roughly 30% for attack rates above 1 Gb/s, for which the root cause is unclear. As the multi-flow attack configuration requires the resources of many cores, it also slows down the processing of legitimate traffic. perf sampling during the multi-flow attack with 20 Gb/s reveals that the Linux kernel uses 65% of the

[10] Version 5.9.2

[11] aes128gcm16-prfsha256-ecp256

total CPU capacity to process received packets. Another 12% is used by strongSwan to process IKE packets. However, we have not identified a specific bottleneck that causes the throughput to drop to zero.

2) Connection Setup Time: Overall, the results show that strongSwan's implementation of the IKE handshake is not resilient against floods of initiation requests. There are multiple potential reasons for this. While strongSwan implements some recommended DoS mitigation mechanisms (such as rate limiting in combination with the cookie mechanism), others (such as deferred key derivation) are missing. Furthermore, even though the implementation is multi-threaded, the initial processing of all IKE packets, including the cookie validation, is performed by a single receiver thread. Hence, even initiation requests with invalid cookies can overwhelm this single thread, causing legitimate IKE packets to be dropped. Measurements reveal that the server sends at a maximum of about 40 000 cookie replies, indicating that at most 40 000 `IKE_SA_INIT` request packets per second can be processed. This limit is reached with a 98 Mb/s flood, partially explaining the client's inability to connect to the server during larger attacks.

E. Discovered Vulnerabilities

The strongSwan implementation was created years before some of the DoS mitigation recommendations were published, such as with RFC 8019, and has since not been updated to follow them all. For instance, the implementation calculates the symmetric DH key during the `IKE_SA_INIT` phase instead of deferring the calculation to the `IKE_SA_AUTH` phase, increasing computational resources wasted for malicious `IKE_SA_INIT` requests. However, we have not found an attack that exposes this potential implementation weakness, possibly due to the large number of IKE workers, which perform this computation.

On the other hand, we discovered that recommendations given in RFC 7296 regarding the cookie mechanism were interpreted in a way that resulted in a mechanism that can be exploitable for DoS attacks. The standard recommends that "the responder should change the value of the secret frequently, especially if under attack". strongSwan accomplishes this by changing the cookie secret after using it for 10 000 cookie responses.[12] Hence, at a rate of roughly 98 Mb/s, the `IKE_SA_INIT` packet flood transmits about 40 000 requests to the server, causing the secret to change four times per second. This has the effect that the server often rejects the client's `IKE_SA_INIT` requests due to outdated cookies. A fix to this vulnerability, and potentially the correct interpretation of the recommendation, is to update the cookie secret periodically after some seconds, e.g., every 30 s. A significant improvement can be observed with this fix: the `IKE_SA_INIT` packet flood requires more than 300 Mb/s instead of 100 Mb/s to deny connection establishment consistently. For rates above 300 Mb/s, the receiver thread, which validates the cookie values of all requests, seems to be the main bottleneck since it uses the entire capacity of its core.

Another bug in the implementation causes the client to unexpectedly abort connection establishment after receiving five different cookie responses, which is likely to happen if the cookie secret changes often. This explains why connection establishment sometimes does not succeed even after the attack has stopped, resulting in a measured connection setup time of 85 s.

[12]This value is hard-coded and not configurable.

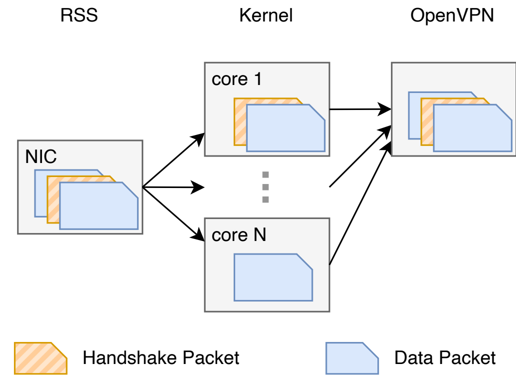


Fig. 8. OpenVPN's packet processing pipeline: all packets are processed by a single thread.

Appendix D provides measurements of strongSwan with the cookie mechanisms fixed and a description of an attack in which an attacker with spoofing capabilities can exploit IP-based limits even if the configurations suggest differently.

VIII. OpenVPN

OpenVPN is a VPN solution designed to be easily portable and deployable on different platforms [11]. OpenVPN uses the TLS protocol for its handshake to authenticate peers and securely derive key material. The back-end mainly relies on the OpenSSL library for cryptographic algorithms. A wide variety of cipher suites are offered, such that users can select one of them depending on the given hardware capabilities and security requirements. Supported authentication methods include pre-shared keys, username/password, and certificates.

On Linux, OpenVPN is a single threaded implementation running in user space (as depicted in Fig. 8). Compared to other VPN implementations, which run in kernel space and use multiple cores, this approach is expected to achieve weaker performance.

OpenVPN supports both UDP and TCP as underlying transport protocols. UDP is the recommended choice and also used in our evaluation. Since the TLS handshake assumes a reliable transport underlay, OpenVPN implements a custom reliable transport layer on top of UDP. The first packet sent by a client to establish a new connection is the `P_CONTROL_HARD_RESET_CLIENT_V2` packet, which contains a random session identifier used by the client to identify the connection. On reception, the server responds with a similar packet containing the client's session identifier and a new session identifier used by the server. During the subsequent exchanges, authentication and key establishment take place.

A. Defense Mechanisms

OpenVPN offers three options to protect its TLS handshake against flooding attacks: `tls-auth`, `tls-crypt`, and `tls-`

crypt-v2. These options all provide mechanisms to authenticate each packet using pre-shared keys.

The tls-auth option relies on a single key shared between all legitimate peers and uses it to calculate an HMAC for each TLS packet. With tls-crypt, the TLS packets are also encrypted with the same shared key, improving privacy, e.g., by hiding the used certificates. However, none of these options scale well in setups with many peers: if a single peer is compromised, the secret key must be changed on all clients and servers to restore the security properties.

tls-crypt-v2 offers a solution to this scaling problem. With this option, each peer receives a unique secret key and a version of its key encrypted with the server's secret key, which is called the *wrapped client key*. The client uses its secret key for encryption and authentication of the TLS messages, just like with tls-crypt, but also sends the wrapped client key in the first packet. The receiving server first decrypts the wrapped client key and then decrypts the received TLS message. The wrapped client key can also contain metadata, such as the client's identifier or expiration date, facilitating key management. Note that the server only needs to know its key and does not need to store a list of client keys. This method scales better because it allows the revocation of single keys using a revocation list.

Since none of the TLS protection mechanisms perform a freshness check on the packet, an adversary can replay a previously sent TLS packet to pass it. However, a replay protection mechanism ensures that the same request is only processed once over a certain period. Hence, an attacker constantly replaying the same packet should not be able to exhaust the server's resources.

B. Setup and Configuration

For our OpenVPN setup, we use the most recent version available at the time of writing.[13] The peers authenticate each other using certificates over TLS 1.3 with the ciphers AES 128 GCM and SHA256. For encryption of the data, AES 128 GCM is used as well.

Under normal conditions, the peers achieve a throughput of about 600 Mb/s. By increasing the MTU of OpenVPN's virtual network interface to 60 000 B and enabling IP fragmentation on the physical network interface, a throughput of up to 4 Gb/s is achieved. This performance gain is likely due to OpenSSL performing much better on larger packets. However, IP fragmentation also causes issues, such as low tolerance to packet loss, which could reduce DoS resilience. Therefore, we do not consider this approach in our evaluation.

C. Flooding Attacks

At first, we evaluate OpenVPN's performance without any TLS protection activated while flooding the server with P_CONTROL_HARD_RESET_CLIENT_V2 packets. The measurements in Fig. 9 show that the server can barely process legitimate data packets when flooded at a rate of 50 Mb/s, which corresponds to slightly more than 100 000 packets per second. **Starting at an attack rate of 100 Mb/s, the flood completely denies data transmission and connection establishment.** With the client-flow configuration, connection establishment often never succeeds, even after the attack has stopped, resulting in a measured connection time of 85 s.

[13] Version 2.5.1 (git:release/2.5/f186691b32e68362) with OpenSSL 1.1.1

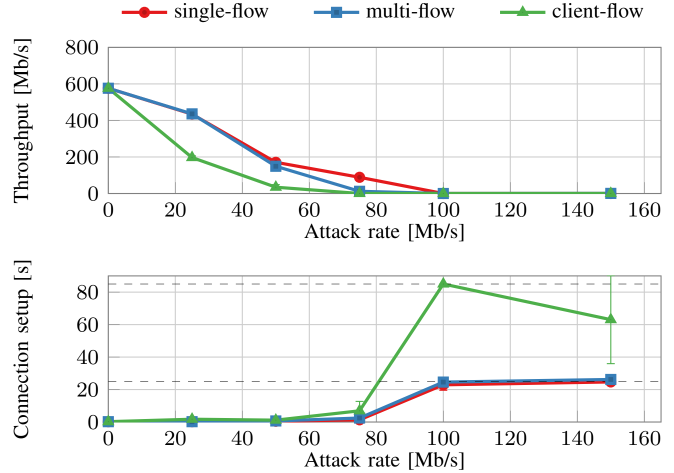


Fig. 9. Performance of OpenVPN under flooding attacks using initiation packets (60 B). TLS protection mechanisms are not enabled.

OpenVPN's TLS protection mechanisms seem to offer an efficient way to detect illegitimate TLS packets, assuming the adversary does not possess a legitimate key. However, these protections do not affect the impact of an initiation packet flood significantly: the server is still overloaded by a 50 Mb/s flood of invalid initiation packets, independently of the mechanism used. Detailed measurements are presented in Appendix C.

D. Detected Vulnerability

Throughput our evaluation of OpenVPN, we discovered that an adversary capable of spoofing its address and using the client's address can completely block the client from communicating with the server with just a single attack packet. OpenVPN acknowledged the vulnerability, and CVE-2021-3568 was registered to track the issue. Appendix D3 describes the attack in detail.

IX. Cisco AnyConnect: IPsec and SSL

Cisco AnyConnect [10] is a proprietary VPN solution and offers multiple different deployment options. We evaluate remote access VPN with IPsec and SSL. The VPN client, which connects to the VPN server, uses the Cisco AnyConnect Secure Mobility Client.[14] The server implementation, as well as the client implementation, are closed-source. Therefore, we primarily relied on application guides and deployment recommendations to gain an understanding of how they work.

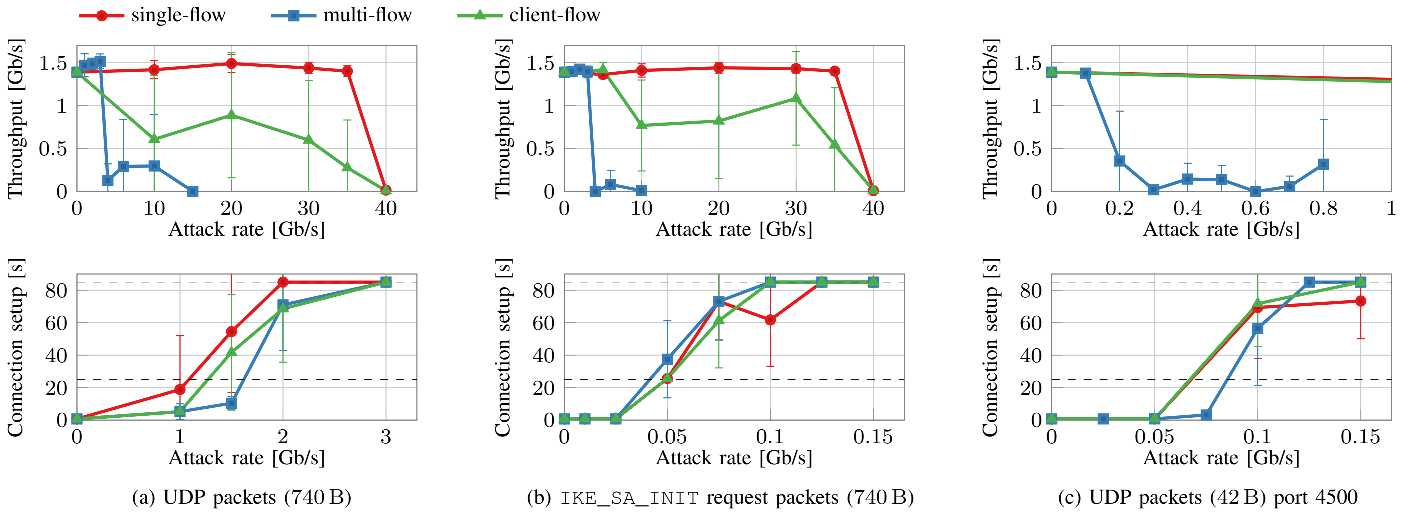
In the following, we evaluate Cisco's DoS resilience for the two different setups: IPsec and SSL.

A. IPsec – Setup and Configuration

For the IPsec setup, the Cisco ASA software is configured to use IKEv2 and ESP with the same cipher set[15] as for

[14] Version 4.10.02086

[15] aes128gcm16-prfsha256-ecp256



(a) UDP packets (740 B) (b) `IKE_SA_INIT` request packets (740 B) (c) UDP packets (42 B) port 4500

Fig. 10. Performance achieved by Cisco AnyConnect (IPsec) under flooding attacks using different types of packets.

strongSwan. The crypto engine accelerator bias is set to IPsec to increase the available hardware resources required for our setup. We set the limits for IKE SAs and half-open IKE SAs to 1000 and 100, respectively. The cookie threshold is set to 50, i.e., the cookie mechanism activates if more than 50 half-open IKE SAs exist. For the remaining configuration options, defaults are used whenever possible.

Under normal circumstances, the client measures a maximal throughput of roughly 1.5 Gb/s, almost half as much as with strongSwan. Most likely, the throughput is limited by the client's AnyConnect application, which uses only a single core.

B. IPsec – Flooding Attacks

Like for strongSwan, we analyze the effect of flooding the VPN server with `IKE_SA_INIT` requests of the same structure as the client uses (740 B). As a baseline, we flood the server with generic UDP packets of the same size addressed to the IKE port 500. The measurements in Fig. 10a show that the flood significantly affects the throughput when performed over multiple flows. With roughly 4 Gb/s of attack traffic, the throughput drops below 10%. On the other hand, connection establishment is denied with roughly 2 Gb/s independent of the flow constellation. The logs reveal that the generic UDP packets are rejected due to an unknown IKE version number, potentially by one of the server's first checks when receiving a potential IKE packet.

Figure 10b shows the impact measurements of `IKE_SA_INIT` request floods. The impact on the throughput is only slightly higher compared to the baseline. However, **just 50 Mb/s of attack traffic, i.e., roughly 8500 `IKE_SA_INIT` packets per second, suffice to deny connection establishment consistently**. Like for strongSwan, using a high number of different initiator SPIs is crucial for the attack to be effective.

We suspect that a single thread processes all IKE packets, while ESP packets are processed on multiple cores. This would explain the flood's low impact on the throughput of already established connections. Further analysis reveals that the overall CPU utilization rises above 80% when the multi-flow flood exceeds 3.5 Gb/s, i.e., roughly 590 000 packets per second. For lower rates, the utilization does not even exceed 10%. This sudden increase might be caused by the synchronization required between the multiple receiving cores and the IKE thread, which becomes infeasible for high packet rates. We confirm this hypothesis by flooding the server with minimal-sized UDP packets, i.e., UDP packets with no payload. Roughly 400 Mb/s of minimal-sized UDP packets spread over multiple flows and addressed to the IKE port 500 are sufficient to deny throughput. When the packets are addressed to the ESP port 4500, i.e., the data transfer port, already 300 Mb/s are sufficient, as shown in Fig. 10c. Blocking connection establishment requires roughly 100 Mb/s, which is significantly more than the `IKE_SA_INIT` packet flood requires.

C. SSL – Setup and Configuration

When choosing SSL, a protocol developed by Cisco is used. The handshake is based on the HTTPS (HTTP over TLS) standard and offers various authentication methods. For the data transfer, the VPN establishes a DTLS connection, such that UDP is used as the transport layer. Overall, the number of round trips required to set up the VPN connection is significantly higher compared to the other evaluated protocols.

Because the TLS and the DTLS connections perform handshakes, which require expensive cryptographic operations and likely allocate some memory, both offer an attack surface. Since for TLS, the lightweight TCP handshake precedes all other exchanges, SYN cookies provide protection from spoofing attacks [4] allowing an IP-based rate limiting to mitigate resource exhaustion attacks. For the DTLS connection, also a cookie mechanism protects the server from spoofing attacks [23]. In contrast to the other analyzed protocols, the DTLS specification recommends enabling the cookie exchange by default and not just when a DoS attack is suspected.

For the Cisco AnyConnect SSL setup, the protocols TLS 1.2 and DTLS 1.2 are used, with a cipher suite[16] similar to the

[16] ECDHE-RSA-AES128-GCM-SHA256

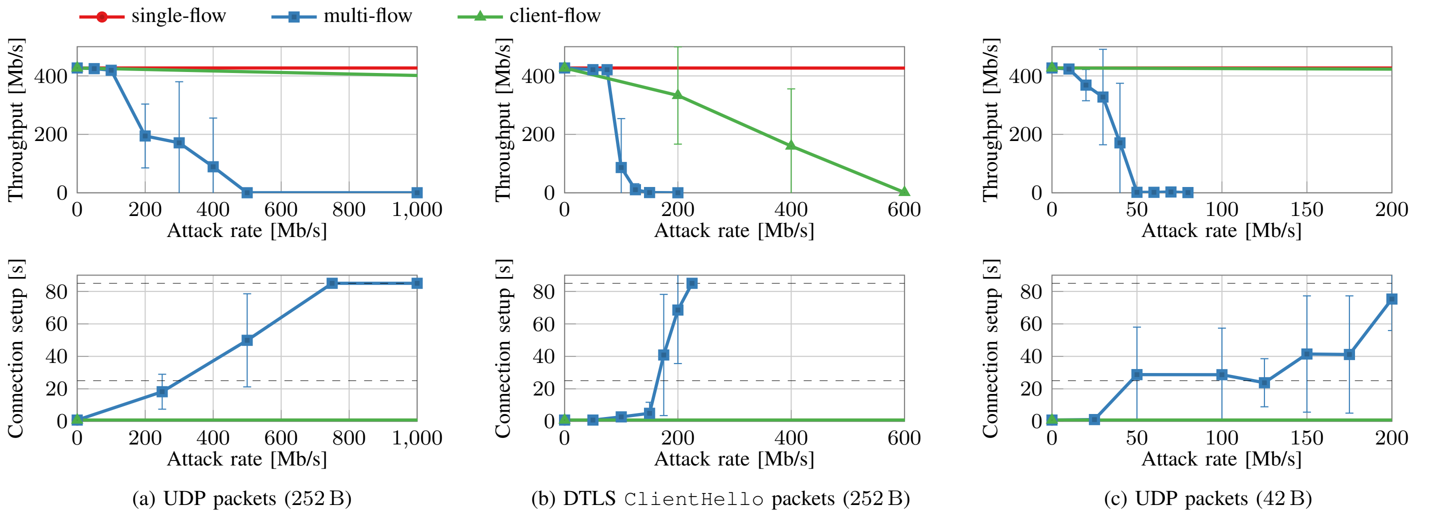


Fig. 11. Performance achieved by Cisco AnyConnect (SSL) under flooding attacks using different packets.

IPsec setup. The crypto engine accelerator bias is set to SSL.

D. SSL – Flooding Attacks

We evaluate the resilience of Cisco's implementation against a flood of DTLS initiation packets, i.e., `ClientHello` packets. To establish a baseline, we flood the server with generic UDP packets of the same size (252 B) and addressed to the same port (443). The measurements shown in Fig. 11a reveal that the flood is the most effective if performed over multiple flows. Only about $500\,\mathrm{Mb/s}$ suffice to overwhelm the server, completely disrupting throughput and denying connection establishment during the 25 seconds of the flooding attack. The overall CPU utilization matches the multi-flow attack's impact on the throughput. For attack rates above $200\,\mathrm{Mb/s}$, the CPU utilization quickly rises above 65%.

Flooding the server with DTLS `ClientHello` packets degrades the VPN's performance even more efficiently, as shown in Fig. 11b. **Roughly $125\,\mathrm{Mb/s}$ already suffice to overwhelm the server, causing the overall CPU utilization to exceed 90% and the throughput to drop to 0.** Connection establishment is denied with just $200\,\mathrm{Mb/s}$.

Just as for Cisco's IPsec implementation, we evaluate floods consisting of minimal-sized UDP packets, i.e., 42 B packets. The measurements shown in Fig. 11c reveal that $40\,\mathrm{Mb/s}$ are sufficient to block an already established connection from communicating over the VPN. At $40\,\mathrm{Mb/s}$ also the overall CPU utilization suddenly rises to 90%. However, to consistently deny connection establishment, roughly $200\,\mathrm{Mb/s}$ are required. With attack rates between $50\,\mathrm{Mb/s}$ and $200\,\mathrm{Mb/s}$, the establishment succeeds 50% of the time during the attack.

E. Cisco AnyConnect Discussion

Cisco's implementations of IPsec and SSL do not offer strong DoS resilience. In both cases, the evaluated attacks are the most efficient when performed over multiple flows. This could be due to multiple reasons. We suspect that the implementations require substantial coordination between the receiving cores because they all might forward certain packets to the same thread for part of the processing. Further, thread detection and firewall rules may also add some overhead.

While the floods consisting of minimal-sized UDP packets can easily be detected and blocked, can their evaluation still provide relevant information. Firstly, they are not blocked by default, even though they are clearly illegitimate VPN packets. Secondly, even if they were blocked, the smallest legitimate packet is likely not much bigger. Moreover, the impact of small UDP packets confirms the suspicion that packet processing, which is minimal for these blank packets, is not the primary cause for the high CPU utilization. Rather packet management, potentially with a high concurrency overhead, is likely to be the main cause.

X. Discussion

Our evaluation has revealed significant limitations in the DoS resilience of popular VPN solutions and exposed multiple related implementation flaws. In the following subsections, we discuss root causes for the performance issues identified in our evaluation and propose possible mitigation techniques. Because Cisco's implementation is closed-source, the following discussion mainly focuses on the open-source VPNs.

A. Concurrency Aspects

As expected, the multi-threaded implementations of WireGuard and strongSwan (IPsec) perform in the absence of attacks significantly better than the single-threaded implementation of OpenVPN. However, our evaluation reveals that this does not directly translate to better DoS resilience. Even though WireGuard and strongSwan have access to all computational resources on the machine, they do not optimally use them to fend off flooding attacks.

WireGuard distributes processing among all available cores. However, when flooded with handshake packets, most of the processing resources are spent on inter-thread communication, i.e., in spinlocks. This highlights a well-known challenge in parallel programming, where a shared data structure becomes a bottleneck and nullifies the benefits of multi-core processing. On the other hand, strongSwan performs some initial

processing, such as the cookie verification, before distributing the work among other cores. This approach is favorable if under attack the inter-thread communication requires more computational resources than the mechanism mitigating the attack. However, since in strongSwan only a single thread performs the initial processing, an attack can easily overwhelm the pipeline, causing legitimate requests to be dropped.

While utilizing as many available resources as possible to handle handshake packets might increase the DoS resilience of the handshake process, it can decrease the DoS resilience of the data traffic process. This inevitable trade-off between allocating resources for handshake packet processing and data packet processing is evident in WireGuard. Even though WireGuard assigns lower thread priority to handshake workers than to data packet workers, our evaluation reveals that this prioritization is insufficient to protect communication with already authenticated clients.

For single-threaded VPN implementations like OpenVPN, running multiple instances combined with a load balancer to distribute traffic evenly across the processes can improve overall performance. This approach has been shown to scale linearly with the number of instances, as long as enough physical cores are available on the system [21]. However, an attacker knowing the mapping performed by the load balancer might still be able to target individual connections effectively.

B. Network-Layer Mitigation

a) Port randomization: We observed that for WireGuard, client-flow attacks are more impactful than multi-flow attacks when targeting a specific connection. However, this requires the adversary to direct its attack traffic to the same receiving core on the server as the targeted client. When RSS includes the source port in the hash calculation and clients connect with random ports, an off-path adversary cannot easily target a specific connection. All evaluated VPN implementations offer client port randomization options. However, it may still be possible for the attacker to provoke a collision on the client's RSS queue through a side-channel guessing attack.

b) DDoS defenses: Our results for multi-flow configurations have shown that attacks using many source IP addresses are consistently successful. While cookie mechanisms theoretically mitigate spoofing attacks, they are ineffective against adversaries controlling many IP addresses, e.g., a large botnet. For IKEv2, an addition to the cookie mechanism has been proposed that defends against such an attacker model [18]. If activated, initiation requests are required to contain a valid cookie value and a solution to a *client puzzle*, which requires a moderate amount of resources to compute. Hence, in addition to controlling the used IP, an attacker would need to expend computational resources for every request, vastly increasing the cost of a high-rate packet flood. At the time of this writing, the client puzzle mechanism is implemented neither by strongSwan nor Cisco. However, such a mechanism would not mitigate the impact of our evaluated attacks, as they expose other bottlenecks.

C. Specification and Implementation

IPsec is a highly complex protocol. Specifications, as well as implementation and configuration recommendations, are described across multiple RFCs. Our analysis of strongSwan has revealed that the implementation does not follow all of the recommendations for DoS mitigation. Furthermore, also misinterpretations lead to vulnerabilities as we have found. These rarely documented deviations from the standard make it very challenging to deploy and configure a DoS-resilient VPN, even for an administrator with a thorough understanding of the specifications. WireGuard does not suffer from the same issues, as it was designed and implemented in conjunction with simplicity as a core goal.

D. Adversarial Testing

We envision developers integrating adversarial performance measurements such as ours into the development process. Implementations should be optimized with respect to legitimate traffic and malicious traffic. To increase the incentive for developers, performance evaluations should also include metrics, which reflect, e.g., the DoS resilience. A high-bandwidth testbed is beneficial for such evaluations. Still, as our results have shown, many attacks already show their impact at traffic rates below $1\,\mathrm{Gb/s}$, easily achievable on inexpensive commodity hardware. Moreover, the attacks are not specific to one network topology and can also be deployed on cloud-based testbeds. Our adversarial testing framework can be applied to new protocols and modified to perform other flooding attacks. Even providing just a simple list of generic protocol packets to the exploration algorithm can reveal unexpected results, as we have observed in our analysis of WireGuard.

XI. Related Work

This section reviews existing performance studies of VPN implementations, provides an overview of the literature on DoS attacks and defenses on VPNs, and relates our approach to standard software testing techniques.

a) VPN performance evaluation: In a recent evaluation, Pudelko et al. [21] compare the performance of three open-source VPN implementations, all of which are also covered by our study. Their findings are consistent with our baseline measurements (i.e., with no attack traffic) and also highlight shortcomings related to multi-core synchronization. The original WireGuard paper also includes a performance comparison against IPsec and OpenVPN [3], although the developers concede that these measurements are relatively dated [29]. Our results complement these evaluations, as they do not consider malicious traffic and therefore provide no indication about the DoS resilience of different implementations.

b) DoS attacks on VPN: To the best of our knowledge, limited literature exists about practical DoS attacks on VPNs. The Deviation attack on IKEv2 is a recent attack [17] that has been implemented against strongSwan. The attack builds on a vulnerability in the IKE handshake, which was discovered in 1999 but has been considered hard to exploit in practice [16]. The Deviation attack can only be performed by an attacker capable of intercepting legitimate packets, which assumes a much stronger position than our off-path attacker model.

c) Software testing: Our attack space exploration algorithm aims to find worst-case traffic patterns that consume resources at the server. This approach is related to fuzzing, a common software testing technique that explores a large

domain of possible inputs to provoke edge cases and reveal implementation bugs [20]. This technique has been applied to OpenVPN to find logical flaws in the state machine of implementations [2]. Like our approach, some fuzzing tools also apply evolutionary algorithms to explore a given black-box application's input space effectively [22]. The adversarial testing tool Frankencerts, which tests SSL/TLS implementations, makes use of synthetic certificates that are automatically crafted from multiple real certificates and randomly mutated to explore the input space more efficiently [1]. However, fuzzing algorithms are often designed for a setting in which candidate inputs can be tested in rapid succession. This is not given in our setting since measuring the impact of a flooding attack strategy is time-consuming.

XII. Conclusion

Our evaluation, on a real setup, shows that state-of-the-art VPN solutions are vulnerable to well-orchestrated flooding DoS attacks. This has important implications for real-world deployments as various infrastructures rely on a functioning VPN. Critical site-to-site connections may use several redundant VPN tunnels between different endpoints to ensure high availability. However, our results show that as many as 10 endpoints could be brought down entirely using a few gigabits of bandwidth if the adversary can determine their locations.

These results highlight that rigorous adversarial testing is crucial for creating more DoS-resilient VPN implementations. Our framework represents an important step toward making adversarial testing more accessible.

In future work, it would be valuable to complement the evaluation with other adversary models, such as *on-path adversaries* and *insider adversaries*. Moreover, the automated attack space exploration would benefit from additional extensions, e.g., by a combination with traditional fuzzing techniques that enable a higher degree of automation such that the framework can be applied more quickly to new protocols.

Ethical Considerations

We have carefully followed conventions for responsible disclosure. More than three months prior to publication, we reported all our findings according to the respective security report policies of the studied VPN projects.

Acknowledgment

We thank the anonymous reviewers for their insightful feedback and suggestions. Furthermore, we gratefully acknowledge the support from ETH Zurich, and from the Zurich Information Security and Privacy Center (ZISC).

References

[1] Chad Brubaker, Suman Jana, Baishakhi Ray, Sarfraz Khurshid, and Vitaly Shmatikov. Using frankencerts for automated adversarial testing of certificate validation in SSL/TLS implementations. In *IEEE Symposium on Security and Privacy*, November 2014.

[2] Lesly-Ann Daniel, Erik Poll, and Joeri de Ruiter. Inferring OpenVPN state machines using protocol state fuzzing. In *IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, 2018.

[3] Jason A. Donenfeld. WireGuard: Next generation kernel network tunnel. In *Network and Distributed System Security Symposium (NDSS)*, 2017.

[4] Wesley Eddy. TCP SYN flooding attacks and common mitigations. RFC 4987, RFC Editor, 2007.

[5] Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer, Florian Wohlfart, and Georg Carle. MoonGen: A Scriptable High-Speed Packet Generator. In *Internet Measurement Conference 2015*, October 2015.

[6] Anja Feldmann, Oliver Gasser, Franziska Lichtblau, Enric Pujol, Ingmar Poese, Christoph Dietzel, Daniel Wagner, Matthias Wichtlhuber, Juan Tapiador, Narseo Vallina-Rodriguez, Oliver Hohlfeld, and Georgios Smaragdakis. The lockdown effect: Implications of the COVID-19 pandemic on Internet traffic. In *Internet Measurement Conference (IMC)*, 2020.

[7] Linux Foundation. Data plane development kit (DPDK), 2015.

[8] Sheila Frankel and Suresh Krishnan. IP security (IPsec) and internet key exchange (IKE) document roadmap. RFC 6071, RFC Editor, 2011.

[9] Sebastian Gallenmuller, Dominik Scholz, Florian Wohlfart, Quirin Scheitle, Paul Emmerich, and Georg Carle. High-performance packet processing and measurements. In *International Conference on Communication Systems & Networks (COMSNETS)*, pages 1–8, 2018.

[10] Cisco Systems Inc. Cisco AnyConnect secure mobility client. [Online]. Available: https://www.cisco.com/c/en/us/products/security/anyconnect-secure-mobility-client.

[11] OpenVPN Inc. OpenVPN. [Online]. Available: https://openvpn.net/.

[12] Phil R. Karn and William A. Simpson. Photuris: session-key management protocol. RFC 2522, RFC Editor, 1999.

[13] Charlie Kaufman, Paul E. Hoffman, Yoav Nir, Pasi Eronen, and Tero Kivinen. Internet key exchange protocol version 2 (IKEv2). RFC 7296, RFC Editor, 2014.

[14] S. Kent. IP encapsulating security payload (ESP). RFC 4303, RFC Editor, 2005.

[15] Franziska Lichtblau, Florian Streibelt, Thorben Krüger, Philipp Richter, and Anja Feldmann. Detection, classification, and analysis of inter-domain traffic with spoofed source IP addresses. In *Internet Measurement Conference (IMC)*. ACM, 2017.

[16] C. Meadows. Analysis of the internet key exchange protocol using the NRL protocol analyzer. In *IEEE Symposium on Security and Privacy*, 1999.

[17] Tristan Ninet, Axel Legay, Romaric Maillard, Louis-Marie Traonouez, and Olivier Zendra. The deviation attack: A novel denial-of-service attack against IKEv2. In *IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, 2019.

[18] Y. Nir and V. Smyslov. Protecting internet key exchange protocol version 2 (IKEv2) implementations from distributed denial-of-service attacks. RFC 8019, RFC Editor, 2016.

[19] Trevor Perrin. The Noise protocol framework. [Online]. Available: https://noiseprotocol.org, 2018.

[20] Theofilos Petsios, Jason Zhao, Angelos D. Keromytis, and Suman Jana. SlowFuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017.

[21] M. Pudelko, P. Emmerich, S. Gallenmüller, and G. Carle. Performance analysis of VPN gateways. In *IFIP Networking Conference*, 2020.

[22] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. VUzzer: Application-aware evolutionary fuzzing. In *Network and Distributed System Security Symposium (NDSS)*, 2017.

[23] Eric Rescorla and Nagendra Modadugu. Datagram Transport Layer Security Version 1.2. RFC 6347, RFC Editor, 2012.

[24] Kotikalapudi Sriram and Douglas C. Montgomery. Resilient interdomain traffic exchange: BGP security and DDoS mitigation. Technical report, NIST, 2019.

[25] Rainer Storn and Kenneth Price. Differential evolution – a simple and efficient heuristic for global optimization over continuous spaces. *Journal of global optimization*, 11(4):341–359, 1997.

[26] strongSwan. strongSwan. [Online]. Available: https://www.strongswan.org/.

[27] strongSwan. strongSwan security recommendations. [Online]. Available: https://wiki.strongswan.org/projects/strongswan/wiki/SecurityRecommendations.

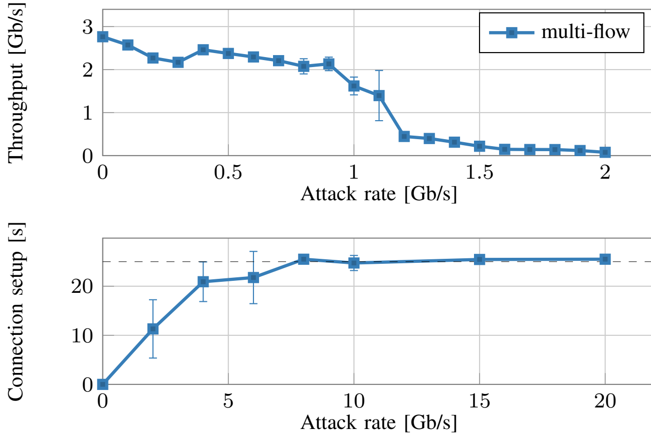


Fig. 12. Performance of WireGuard while flooding it with initiation packets containing valid `mac1` and `mac2` fields.

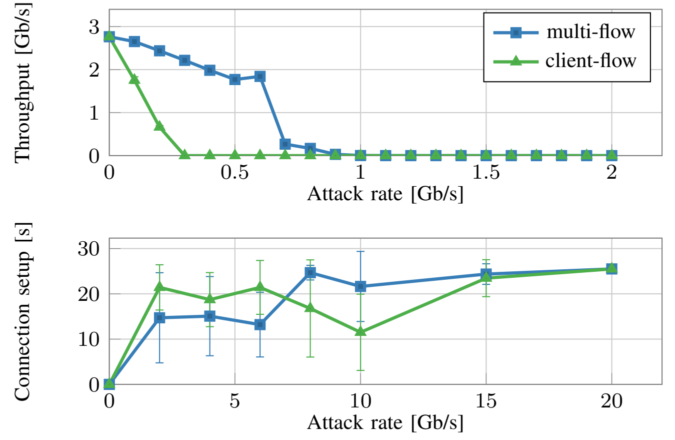


Fig. 13. Performance of WireGuard while flooding it with cookie response packets.

[28] The SciPy community. scipy.optimize.differential_evolution – SciPy v1.7.1 Manual. [Online]. Available: https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.differential_evolution.html.

[29] WireGuard. WireGuard: fast, modern, secure VPN tunnel. [Online]. Available: https://www.wireguard.com/.

APPENDIX

A. Additional Specifications of Testbed

a) Client: The VPN client is a commodity server machine running Ubuntu 18.04.4 LTS[17] with two 16-core CPUs[18] at 2.80 GHz (64 virtual cores using hyper-threading) and a network card[19] that supports RSS and traffic up to 100 Gb/s.

b) Attacker: The attacker has the same specifications as the VPN server used for the open-source tests.

B. Additional Attacks on WireGuard

Figure 12 shows WireGuard's performance when flooded with initiation requests containing valid `mac1` and valid `mac2` fields. Compared to the flood consisting of initiation packets containing only a valid `mac1`, the impact on WireGuard's performance is not significantly different. Since the cookie mechanism is only activated if the floods are performed over multiple flows (above 350 Mb/s, i.e., 230 000 initiation requests per second), the other configurations are not considered.

Figure 12 shows WireGuard's performance while being flooded with cookie responses. The cookie response packet is with 106 B the smallest of the three WireGuard handshake packets. Even though the cookie flood achieves a higher handshake rate than the mixed-packed flood for the same bit rate, it is marginally less efficient.

C. Additional Attacks on OpenVPN

Figure 14 shows the impact of the TLS protection options in OpenVPN. None of them show any significant improvement in resisting a flood of initiation packets; if anything, the options even appear to degrade the DoS resilience of the server slightly (especially `tls-crypt`).

[17]GNU/Linux 5.4.0 x86_64

[18]Intel® Xeon® Gold 6242 CPU

[19]Mellanox MT27800 ConnectX-5

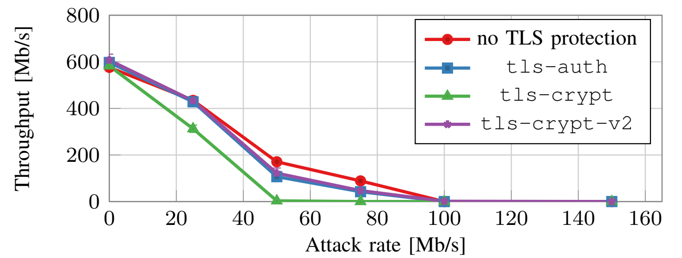


Fig. 14. Throughput comparison of OpenVPN with the different TLS protection mechanisms enabled under a single-flow flooding attack using generic initiation packets.

D. Security Vulnerabilities and Fixes

1) strongSwan – Cookie Reuse Limit: Based on our recommendations, the developers of strongSwan implemented a patched version of their cookie mechanism, which updates the cookie secret every 30 s. Figure 15 shows the impact of `IKE_SA_INIT` packet floods on the patched version. A significant improvement can be observed for attack rates between 75 Mb/s and 300 Mb/s.

2) strongSwan – High Number of Half-Open IKE SAs: Even with the cookie threshold set to a specific value, an attacker spoofing its IP address can create far more half-open IKE SAs on the server. The reason for this behavior is a known race condition in the implementation. When receiving an `IKE_SA_INIT` request, strongSwan checks if the cookie threshold or the IP limit is reached. If the number of half-open IKE SAs is below the threshold and the limit, the packet

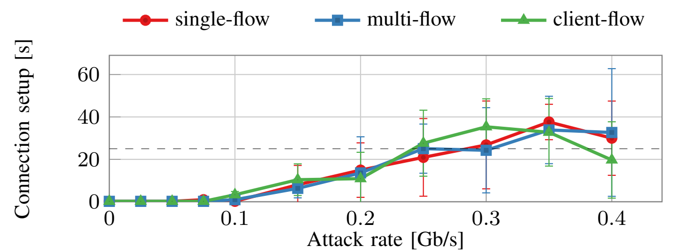


Fig. 15. Connection setup time of strongSwan with the cookie mechanism fix applied while being flooded with `IKE_SA_INIT` packets (306 B).

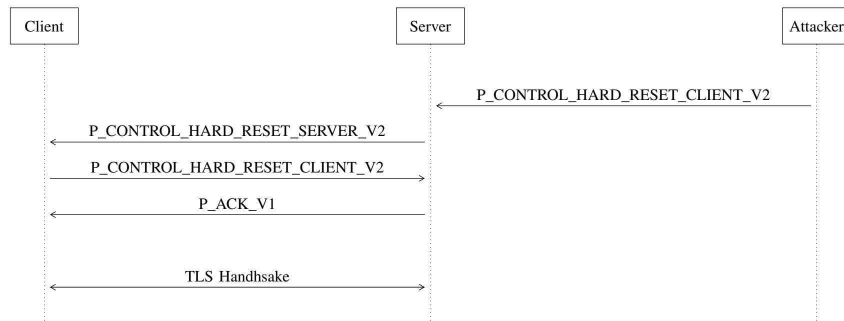


Fig. 16. Packet exchange of the forced key renegotiation attack on OpenVPN.

is added to the job queue and processed by another thread, creating a new half-open IKE SA for the request. Since the thresholds are only checked initially and hundreds of jobs can be queued, an attacker spoofing its IP address can create more half-open IKE SAs than the cookie threshold might suggest. An attacker using a legitimate client's IP address can even exceed the number of half-open IKE SAs allowed for one IP address, causing requests from the client to be dropped. This attack also demonstrates the importance of setting the IP limit higher than the cookie threshold, which was not done by default at the time of writing.

3) OpenVPN – Forced Key Renegotiation: Our evaluation of OpenVPN revealed that an adversary capable of spoofing a legitimate client's address could trigger a TLS renegotiation. By injecting a single spoofed initiation packet, the server believes the client requests a key renegotiation and responds to it. The client receives the packet, accepts it, assumes the server demands a key renegotiation, and responds accordingly. Consequently, the client and server perform an unnecessary TLS renegotiation, as shown in Fig. 16.

While a forced key renegotiation is already problematic, the attack causes a more significant problem because the two peers do not complete the handshake properly. This results in a complete DoS for the connection, which persists even through a restart of the client. Only after restarting OpenVPN on the server, the client can establish a connection again. The TLS-protection mechanisms `tls-auth`, `tls-crypt`, and `tls-crypt-v2` offer only limited protection since an attacker in possession of the secret key or a previously sent initiation packet can circumvent them.

We would expect the client to prevent the attack by dropping the initiation requests since it contains an unknown client session ID. Interestingly, when receiving the server's packet, the client reports an error in the logs due to an unexpected acknowledgment of a packet, which is actually the packet sent by the attacker. Nevertheless, it proceeds with the handshake.

While TLS is also used in many applications other than OpenVPN, we do not expect them to be vulnerable. The bug we have exposed seems not to be in the TLS implementation itself but rather in the custom OpenVPN reliable layer on top of UDP and how TLS is integrated into it. According to the developers and the still reserved CVE, fixing this bug is not trivial.