TECHNISCHE HOCHSCHULE LÜBECK

Bachelorarbeit

Prototypische Implementierung einer portablen Client-Bibliothek für das OpenVPN-Protokoll

Vorgelegt von: Jan Torben Hopp

Fachbereich: Elektrotechnik und Informatik

Studiengang: Informatik / Softwaretechnik

Erstprüfer: Prof. Dr. rer. nat. Sebastian Berndt

Ausgabedatum: 15. Mai 2025

Abgabedatum: 15. August 2025

Aufgabenstellung

Der IT-Security-Hersteller Securepoint GmbH hat eine eigene OpenVPN-Client-Bibliothek entwickelt, die in C geschrieben ist. Diese kommt in Securepoints VPN-Clients für Windows, Android und iOS zum Einsatz. In Stabilitätsberichten der Clients hat sich gezeigt, dass die interne OpenVPN-Client-Bibliothek eine häufige Ursache für Abstürze war.

Das Ziel dieser Bachelorarbeit ist die prototypische Umsetzung einer portablen Client-Bibliothek für das OpenVPN-Protokoll. Teil der Arbeit ist auch die Untersuchung der in dieser Hinsicht relevanten Teile des Protokolls.

Der Prototyp soll eine bessere Speichersicherheit und Laufzeit-Stabilität bieten und potenziell die intern entwickelte OpenVPN-Client-Bibliothek in C langfristig ersetzen.

Dazu soll:

- eine Anforderungsanalyse an eine solche OpenVPN-Client-Bibliothek erfolgen, indem:
  - die bestehende Softwarelösung untersucht wird
  - die Funktionsweise des OpenVPN-Protokolls in relevanten Teilen untersucht wird
- ein Prototyp einer neuen OpenVPN-Client-Bibliothek entwickelt werden
- die Eignung des Prototyps als Alternative zur bestehenden Software bewertet werden

TECHNISCHE HOCHSCHULE LÜBECK

Eigenständigkeitserklärung

Declaration of Originality

Hopp, Jan Torben

Name, Vorname
Last name, first name

365208

Matrikelnummer
Matriculation number

Ich versichere hiermit, dass ich die vorliegende
I hereby declare that this

☐ **Hausarbeit** term paper ☒ **Bachelorarbeit** bachelor's thesis ☐ **Masterarbeit** master's thesis

mit dem Titel
with the title

Prototypische Implementierung einer portablen Client-Bibliothek für das OpenVPN-Protokoll

eigenständig und ohne unerlaubte fremde Hilfe angefertigt habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel verwendet und Entlehnungen aus anderen Arbeiten kenntlich gemacht. Für den Fall, dass die Arbeit zusätzlich elektronisch und/ oder digital eingereicht wird, erkläre ich, dass die schriftliche und die elektronische und/ oder digitale Form identisch sind. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

is my own original work and any assistance from third parties has been acknowledged. I have clearly indicated and acknowledged all sources and resources as well as any borrowings from other works. In case of an additional electronic and/or digital submission of this work, I declare that the written form and the electronic and/or digital form are identical. This work has not previously been submitted either in the same or in a similar form to another examination office.

Ich bin damit einverstanden, dass die vorliegende Hausarbeit/ Bachelorarbeit/ Masterarbeit für Veröffentlichungen, Ausstellungen und Wettbewerbe des Fachbereiches verwendet und Dritten zur Einsichtnahme vorgelegt werden kann.

I agree that this work can be used for publishing, exhibition or competition purposes and can be inspected by third parties.

☒ **ja** Yes ☐ **nein** no ☐ **es liegt ein Sperrvermerk bis _____vor** there is an embargo period until

Ort, Datum
Place, Date

Unterschrift
Signature

Inhaltsverzeichnis

1. Einleitung

Um zu verstehen, wie die Themenwahl dieser Bachelorarbeit zustande kam, wird im Folgenden die zu lösende Problemstellung geschildert. Zunächst möchte ich die Ausgangslage beschreiben, dann die Probleme erläutern und schließlich den Lösungsansatz und dessen Umsetzung zusammenfassen.

1.1. Ausgangslage

Die Securepoint GmbH stellt für ihre Kunden Virtual Private Network (VPN)-Anwendungen für Android, iOS, macOS und Windows bereit, die das OpenVPN-Protokoll unterstützen (Securepoint GmbH, 2025). Um das Protokoll nicht auf jeder Platform erneut zu implementieren, greifen diese Anwendungen auf eine intern entwickelte, portable Software-Bibliothek zurück, die die clientseitige Logik des OpenVPN-Protokolls implementiert.
Der Grund für die interne Reimplementierung ist, dass die offizielle OpenVPN-Software unter der GNU General Public License Version 2 (GNU GPLv2) steht (OpenVPN Inc. and contributors, 2025). Die GNU GPLv2 ist eine Open-Source-Lizenz, unterliegt jedoch dem Copyleft-Prinzip. Das bedeutet, dass Software, die auf GNU GPLv2-lizenzierte Komponenten nutzt, ebenfalls unter der GPLv2 veröffentlicht werden muss. Securepoints VPN-Clients stehen aber nicht unter einer Copyleft-Lizenz, weswegen die Verwendung der offiziellen OpenVPN-Software legal nicht möglich ist.
Die interne OpenVPN-Client-Bibliothek ist in der Programmiersprache C geschrieben und unterstützt einen großen Teil der für Clients relevanten Konfigurationsoptionen ([Firmen-interne Quelle], Bastian Kummer, 2018).
Die Wahl von C ermöglicht eine performante Implementierung und die Unterstützung aller notwendigen Plattformen, birgt allerdings auch erhebliche Risiken. Bei C muss die Speicherverwaltung manuell erfolgen, was ein beträchtliches Fehlerpotenzial birgt: Es kann zu Speicherlecks oder Sicherheitslücken führen und Zugriffe außerhalb des erlaubten Speicherbereichs werden nicht erkannt, sondern führen zum Programmabsturz (Devietti et al., 2008; Turner, 2014).

1.2. Probleme

In der Vergangenheit sind verschiedene Probleme im Zusammenhang mit der OpenVPN-Client-Library aufgetreten, die hier kurz erklärt werden sollen.

1.2.1. Abstürze durch Verletzung der Speicherrechte

Im Sommer 2024 traten einige Abstürze in der Android-App auf, die in der Google Play Console gemeldet wurden. Dies geschah nach der Einführung des Features `tls-auth` ([Firmen-interne Quelle], 2024), welches OpenVPN-Servern das Filtern aller Pakete ohne gültige Signatur erlaubt (OpenVPN Inc., 2025a). Dort kam es zu einem Speicherzugriff außerhalb des erlaubten Speicherbereichs in der Funktion `buffer_write`. Das führte dazu, dass das Android-Betriebssystem den Prozess des VPN-Clients sofort beendet hat:

```
[split_config.arm64_v8a.apk!libopenvpn-lib.so] string.h - buffer_write
SIGSEGV

*** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***
pid: 0, tid: 26098 >>> de.securepoint.ms.agent <<<

backtrace:
  #00  pc 0x000000000007f3e0  /apex/com.android.runtime/lib64/bionic/libc.so (memcpy+96)
  #01  pc 0x000000000000f160  /data/app/de.securepoint.ms.agent-
nwypKaqNhfaEftLHuuY2Ag==/split_config.arm64_v8a.apk!libopenvpn-lib.so (buffer_write+60)
(BuildId: 903fa70d9d98011772c6d33dd46e74f683d22e2c)
```

Listing 1: Backtrace eines Absturzes in `buffer_write` durch einen Segmentation Fault unter Android ([Firmen-interne Quelle], 2024)

Modernere Programmiersprachen erlauben viele der in C möglichen Abstürze gar nicht mehr. Sie vermeiden mit anderen Speichermodellen ganze Kategorien von Fehlern. Rust beispielsweise validiert alle Speicherreferenzen mit dem Compiler, sodass ihre sichere Nutzung bei Laufzeit des Programms garantiert werden kann (The Rust Project Contributors, 2025a). Go löst das Problem des Speichermanagements, indem ein Garbage Collector nachweislich ungenutzte Referenzen erkennt und ihre Speicherbereiche wieder freigibt (Donovan & Kernighan, 2016). Beide Fälle demonstrieren, dass moderne Programmiersprachen Speichermanagement vereinfachen und somit auch Fehler darin zumindest reduzieren können.

1.2.2. Performance-Probleme durch suboptimale Integration

Die Integration der C-Bibliothek in die Clients für verschiedene Plattformen gestaltet sich aufwendig. Dies führt häufig dazu, dass Entwickler funktionierende, aber schwer nachvollziehbare und qualitativ nicht optimale Implementierungen beibehalten. So auch in diesem Beispiel:

Vor drei Jahren ist aufgefallen, dass Uploads über die VPN-Verbindung unter iOS deutlich langsamer waren als auf anderen Plattformen. Die Ursache dieses Fehlers war die Tatsache, dass die OpenVPN-Library unter iOS nicht direkt mit dem Socket interagiert hat, wie es auf den anderen Plattformen der Fall ist.
Stattdessen wurde in einem Read-Loop aktiv gewartet, ob es Pakete zu verschicken gab. Wenn es ein Paket für den Upload gab, wurde es an die C-Bibliothek gereicht, die jedes Paket als Pointer wieder an die iOS-App zurück übergeben hat. Die Daten des Pointers wurden in einen Apple-Datentyp `Data` kopiert (Apple Inc., 2025a). Diese Kopie wurde dann für den Upload über Apples Netzwerk-API in den Socket geschrieben. ([Firmen-interne Quelle], 2022a; 2022b).

Die Lösung des Problems war die Nutzung der Implementierung für FreeBSD, die für interne Zwecke bereits unterstützt wurde ([Firmen-interne Quelle], 2022b). Dies war ohne viele Anpassungen möglich, da Apples XNU-Kernel seine Ursprünge zum Mach-Kernel von der Carnegie Mellon University und zu FreeBSD zurückverfolgen kann (Apple Inc., 2025b).

Moderne Sprachen bieten für die Integration von Libraries in z. B. Apps für Android und iOS oftmals Tooling, das diese Bindings automatisch generieren kann. Das erleichtert die Integration und verhindert Probleme wie das das oben beschriebene. Es gibt beispielsweise `gomobile` für die Programmiersprache Go, womit Bindings zu einer Go-Library für die Sprachen der jeweiligen Plattformen generiert werden können. Für iOS können Objective-C-Bindings erzeugt werden und für Android Bindings für Java (Google LLC, 2025a).
`gomobile` wurde bei Securepoint bereits erfolgreich eingesetzt, um eine DNS-over-HTTPS-App für Android zu realisieren ([Firmen-interne Quelle], Securepoint GmbH, 2025).

1.2.3. Build-Konfiguration und Deklaration von Abhängigkeiten

Es gibt zudem noch ein weiteres Problem: C hat kein Standard-Build-System. Abhängigkeiten können oft nicht zentral in einer Datei angegeben werden, wie es bei modernen Programmiersprachen wie Go mit `go.mod`-Dateien (Google LLC, 2025b) oder Rust mit `Cargo.toml`-Dateien (Rust contributors, 2025) der Fall ist.

Es gibt zwar Lösungen mit `Makefile` oder `CMakeLists.txt`, allerdings ist deren Konfiguration und Einsatz meiner Erfahrung nach aufwendiger, da man dort z. B. noch den Compiler und Linker festlegen muss.
Außerdem ist dort keine Verwaltung von externen versionierten Abhängigkeiten integriert. Es wird erwartet, dass man weiß, wie man die Abhängigkeiten manuell so installiert, dass sie gefunden und genutzt werden können. Es werden tendenziell kleinere Abhängigkeiten direkt ins Projekt kopiert, was dann einen – meines Erachtens nach – komplexeren Update-Prozess nach sich zieht (Ronin, 2016). Wenn man komfortablere Lösungen moderner Programmiersprachen gewohnt ist, sind einem

diese Lösungen wahrscheinlich zwangsweise suspekt. Die Programmiersprache Go zum Beispiel wurde dazu entwickelt, Probleme eines C++-Projekts zu lösen, darunter auch die aufwendige Verwaltung von Software-Abhängigkeiten, die Entwicklern Zeit kostet (Pike, 2023).

1.3. Lösungsansatz

Im Betrieb kam die Idee auf, die OpenVPN-Client-Library in einer speichersicheren Programmiersprache zu reimplementieren.

Um dieses Unterfangen strukturiert anzugehen, muss:

1. das OpenVPN-Protokoll analysiert werden
2. die bestehende Software-Lösung analysiert und dokumentiert werden
3. die Schnittstelle der aktuell genutzten OpenVPN-Library dokumentiert werden
4. eine geeignete speichersichere Programmiersprache identifiziert werden
5. der Teil des Protokolls, der für einen Client-Prototypen relevant ist, identifiziert werden
6. der relevante Teil für einen Prototypen in der gewählten Programmiersprache plattformübergreifend (portabel) implementiert werden
7. die bestehende Software-Lösung mit dem erarbeiteten Prototypen verglichen werden, um die Eignung des Prototypen als Ersatz für die bestehende Lösung zu ermitteln.

Die komplette Implementierung ist im Rahmen einer Bachelorarbeit unrealistisch, aber die Dokumentation des Protokolls sowie die Implementierung eines teilweise funktionalen Prototyps scheint in drei Monaten möglich zu sein.

1.4. Umsetzung

Nach einem kurzen Überblick über Kryptografie und VPNs wird das Protokoll gründlich, aber nicht vollständig beschrieben und Problematiken mit dem Protokoll in einem Fazit erörtert. Danach wird die bereits implementierte Client-Bibliothek analysiert und relevante Komponenten und Schnittstellen identifiziert. Es werden die Programmiersprachen Go und Rust auf ihre Eignung zur Implementierung eines VPN-Clients verglichen und anhand der Vergleichskriterien wird Go ausgewählt. Anhand der Analysen des Protokolls und der bestehenden Software-Lösung werden Anforderungen an den Prototypen sowie seine Schnittstellen definiert. Die Implementierung des Prototyps und die Herausforderungen dabei werden erörtert. Der Prototyp wird aufgrund technischer Probleme und einem Mangel an Zeit nicht fertiggestellt, implementiert aber schon viele wichtige Anforderungen. Abschließend wird die bestehende Implementierung mit dem Prototypen verglichen, wobei neben den funktionalen Anforderungen auch qualitative Aspekte der Software-Entwicklung berücksichtigt werden.

2. Überblick

Bevor wir uns VPNs genauer ansehen, ergibt es Sinn, den Begriff kurz zu erörtern. Ferguson & Huston (1998) haben diesen Begriff wie folgt definiert:

> „A VPN is private network constructed within a public network infrastructure […]."
>
> — Ferguson & Huston (1998)

Was in dieser Definition etwas zu kurz kommt, ist das „Virtual" in Virtual Private Network. Raymond (1993) definiert „virtual" als „logical" oder auch als vom Betriebssystem simuliert. Man könnte also vielleicht zur Definition von Ferguson & Huston (1998) ergänzen:

> Bei VPNs handelt es sich um logische Netze, die vom Betriebssystem simuliert werden.

Es gibt verschiedene Anwendungsfälle, bei denen VPNs zum Einsatz kommen können. Ein häufiges Szenario für die Verbindung von verschiedenen Netzen ist die Site-to-Site-Verbindung. Eine Site-to-Site-Verbindung verbindet zwei Netzwerke miteinander. Das ist vor allem bei Unternehmen ein verbreiteter Einsatz-Modus, um z. B. verschiedene Firmen-Standorte miteinander zu vernetzen. In so einem Szenario könnte zum Beispiel ein Client im Netz des einen Netzwerks auf einen Server im Netz eines anderen Standorts zugreifen.

In Abbildung 1 ist eine vereinfachte schematische Darstellung eines Site-to-Site-VPNs dargestellt:

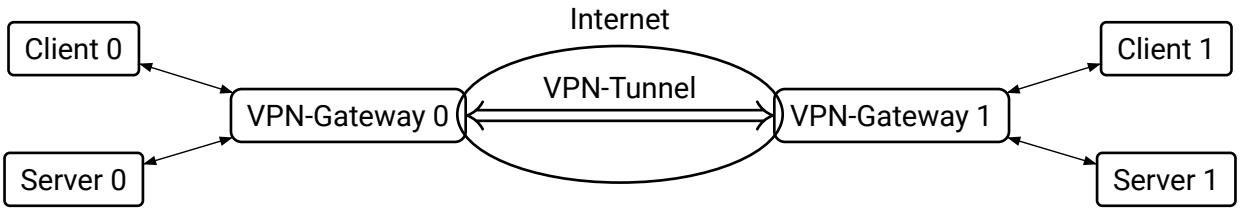

Abbildung 1: Site-to-Site-VPN

Wenn man nun einige Mitarbeiter im Homeoffice oder Außendienst hat, sind diese nicht in einem Netzwerk der Firma. Für solche Fälle gibt es End-to-Site-Verbindungen. Eine End-to-Site-Verbindung verbindet einen einzelnen Client mit einem Netzwerk (siehe Abbildung 2). Dieses Szenario wird bei der bestehenden Bibliothek und dem zu entwickelnden Prototypen angenommen.

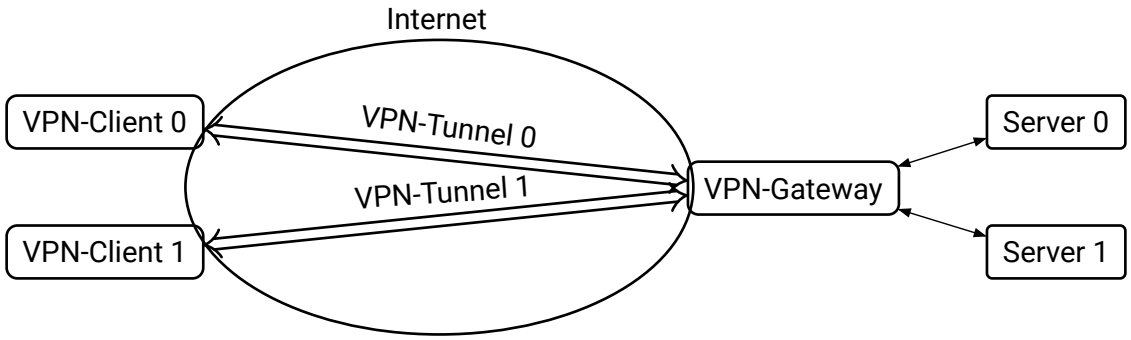

Abbildung 2: End-to-Site-VPN

2.1. Informationssicherheit in VPNs

Der Einsatz von Kryptografie dient meist einem der folgenden Ziele der Informationssicherheit, die auf Englisch als „CIA-Triad“ bekannt sind (Stobitzer, 2017):

- Confidentiality (Vertraulichkeit)
- Integrity (Integrität)
- Availability (Verfügbarkeit)

Manchmal wird statt „Availability“ „Authenticity“ (dt. Authentizität) als alternatives Ziel genannt (Bundesamt für Sicherheit in der Informationstechnik, 2023).

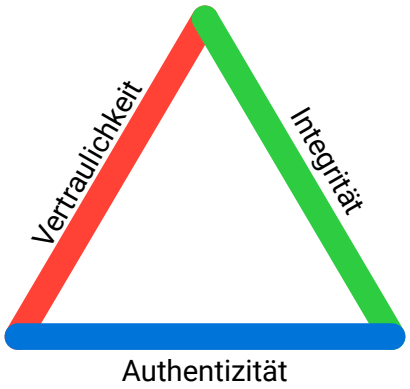

Abbildung 3: Die Schutzziele der Informationssicherheit nach Bundesamt für Sicherheit in der Informationstechnik (2023)

2.1.1. Vertraulichkeit

Definition 1: Vertraulichkeit

Vertraulichkeit bedeutet, dass Unbefugten kein Informationsgewinn möglich ist.

Vertraulichkeit kann über Verschlüsselung der Informationen hergestellt werden. Verschlüsselung kann mit symmetrischer Verschlüsselung umgesetzt werden, wie dem Advanced Encryption Standard (AES) (National Institute of Standards and Technology, 2001), dem ChaCha-Verschlüsselungsverfahren (Nir & Langley, 2018) oder mit Public-Key-Verschlüsselung wie der Rivest-Shamir-Adleman (RSA) Verschlüsselung (Rivest et al., 1978) oder dem Elgamal-Verschlüsselungsverfahren (Elgamal, 1985).

Public-Key-Kryptografie, auch asymmetrische Kryptografie genannt, unterscheidet sich von der symmetrischen Kryptografie in der Anzahl der Schlüssel und wie diese Schlüssel eingesetzt werden:

Bei der symmetrischen Kryptografie kommt ein Schlüssel zum Einsatz, den sowohl Sender als auch Empfänger nutzen, um Informationen zu verschlüsseln und auch wieder zu entschlüsseln. Die Operation der Kryptografie ist also mit dem selben Schlüssel reversibel.

Bei der asymmetrischen Kryptografie kommt ein Schlüsselpaar zum Einsatz, das aus privatem und öffentlichem Schlüssel besteht. Das Schlüsselpaar wird mit kryptografischen Verfahren generiert, sodass der private Schlüssel Nachrichten entschlüsseln kann, die mit dem öffentlichen Schlüssel verschlüsselt wurden.

Der private Schlüssel muss geheim bleiben, aber der öffentliche Schlüssel kann frei geteilt werden.

Die Eigenschaft, die asymmetrische Kryptografie ausnutzt, ist die Tatsache, dass die Operationen mit privatem und öffentlichem Schlüssel leicht sind, ohne privaten Schlüssel aber praktisch irreversibel

sind. Man spricht von einer Falltürfunktion (Rivest et al., 1978), da selbst Supercomputer ausreichend große Schlüssel niemals ermitteln können (Andrew, 2021).

2.1.2. Integrität

Definition 2: Integrität

Integrität bedeutet, dass Informationsmanipulation von Unbefugten zumindest erkannt werden kann.

Um die Integrität einer Nachricht zu überprüfen, werden Prüfsummen eingesetzt. Prüfsummen werden vom Sender erzeugt und dem Empfänger mitgeteilt. Der Empfänger kann aus der Nachricht eine eigene Prüfsumme erzeugen. Wenn die eigene Prüfsumme mit der des Senders übereinstimmt, ist die Nachricht mit sehr hoher Wahrscheinlichkeit unverfälscht (Bundesamt für Sicherheit in der Informationstechnik, o. J.).

Überträgt man nun allerdings Daten über einen öffentlichen Kanal, könnte eine einfach Prüfsumme von einem Angreifer nach der Manipulation der Nachricht neu erzeugt werden. Daher werden in der Praxis Hash-based Message Authentication Codes (HMACs) verwendet. HMACs werden nicht nur mit einer Nachricht, sondern auch einem geheimen Schlüssel erzeugt, den nur Sender und Empfänger haben. Ein HMAC kann daher von potenziellen Angreifern praktisch nicht neu erzeugt werden (Krawczyk et al., 1997).

2.1.3. Verfügbarkeit

Definition 3: Verfügbarkeit

Verfügbarkeit bedeutet, dass die korrekte Funktionsweise eines Systems gewährleistet ist. Andersrum heißt es, dass es nicht möglich ist, das System ohne Befugnis von der korrekten Funktionsweise abzuhalten.

Verfügbarkeit ist im Kontext von VPNs vor allem in Hinsicht auf Server relevant. Ein Beispiel für die Verletzung der Verfügbarkeit ist eine erfolgreiche Denial of Service (DoS)-Attacke (Stobitzer, 2017). Eine solche DoS-Attacke haben Quarkslab SAS (2017) erfolgreich für bestimmte Konfigurationen des OpenVPN-Protokolls konstruiert.

2.1.4. Authentizität

Definition 4: Authentizität

Authentizität bedeutet, dass die Identität eines Kommunikationspartners überprüfbar und echt ist.

Authentizität kann kryptografisch über Signaturen gewährleistet werden. Typische Signaturen bauen auf Public-Key-Kryptografie auf. Damit können in einigen Fällen auch Nachrichten mit dem eigenen privaten Schlüssel so „signiert“ werden, dass man unter Verwendung des öffentlichen Schlüssels die Authentizität einer Nachricht beweisen kann.

OpenVPN kann optional HMACs nutzen (bekannt als „TLS-Auth“ oder „HMAC authenticated control channel packets“), um die Authentizität von Clients schon beim Aufbau der Verbindung zu

prüfen (Schwabe, 2016). Da der OpenVPN-Server somit alle Nachrichten ohne valide HMAC verwerfen kann, ist der DoS-Angriff von Quarkslab SAS (2017) so nicht mehr möglich.

2.1.5. Schlüsselaustausch

VPNs stehen wie alle verschlüsselten Verbindungen vor dem Problem des sicheren Schlüsselaustauschs. Das Problem besteht darin, dass die Kommunikationspartner die Kommunikation über ein öffentliches, abhörbares Medium wie das Internet aufbauen müssen.

Gängige VPN-Protokolle wie IPSec (Kaufman et al., 2014), WireGuard (Donenfeld, 2015) und OpenVPN nutzen dabei alle den Diffie-Hellman-Schlüsselaustausch (Schwabe, 2016), (Rescorla & Dierks, 2008). Der Diffie-Hellman-Schlüsselaustausch erlaubt es, einen Schlüssel über einen unsicheren Kanal mit einem Kommunikationspartner auszutauschen (Rescorla, 1999).

Der grobe Ablauf ist dabei wie folgt:
1. Es werden öffentliche Parameter mit dem Kommunikationspartner festgelegt.
2. Die öffentlichen Parameter werden mit dem eigenen privaten Schlüssel kryptografisch kombiniert (was nicht rückgängig zu machen ist).
3. Diese kryptografisch kombinierten Schlüsseldaten werden über den öffentlichen Kanal ausgetauscht.
4. Die empfangenen Schlüsseldaten des Kommunikationspartners werden mit dem eigenen privaten Schlüssel kryptografisch kombiniert.
5. Das Ergebnis ist ein geteilter Schlüssel, der dann für symmetrische Verschlüsselung verwendet werden kann.

2.2. VPN-Protokolle

Es gibt mehrere verbreitete VPN-Protokolle, die heute noch zum Einsatz kommen. Zu den am häufigsten verwendeten Protokollen zählen IPSec, OpenVPN und WireGuard (Lyons, 2025).

IPSec unterscheidet sich dabei von den anderen beiden insofern, als dass es auf Internet-Protokoll-Ebene agiert (Frankel & Krishnan, 2011; Kaufman et al., 2014). Mit Hardwarebeschleunigung ist IPSec eine schnelle und relativ leicht einsetzbare Lösung (Frikin, 2022). Außerdem erfreut sich IPSec breiter Integration in Betriebssystemen: So ist es in den Linux-Kernel integriert und unter Android, iOS, macOS und Windows verfügbar (The strongSwan Team, 2025). IPSec hat den Ruf, schwierig zu konfigurieren zu sein und verschiedene Software-Versionen zu haben, die untereinander nicht vollständig kompatibel sind (Frikin, 2022).

WireGuard ist ein relativ neues Protokoll, das eine hohe Flexibilität aufweist, obwohl es weniger Konfigurationsoptionen bietet. Eine Implementierung ist außerdem seit 2020 mit Version 5.6 im Linux-Kernel (Donenfeld, 2020). Der Datendurchsatz von WireGuard kann in vielen Szenarien mit IPSec mithalten oder ist sogar schneller (Donenfeld, 2022).

OpenVPN zeichnet sich vor allem durch eine hohe Konfigurabilität aus (Lyons, 2025). Es kann aufgrund seiner Flexibilität vielseitig eingesetzt werden, zum Beispiel in Unternehmen zur Vernetzung von Filialen oder für den Zugriff auf interne Netzressourcen durch Mitarbeiter im Homeoffice. Es ist allerdings nicht besonders schnell (Donenfeld, 2022).

Um hier einen Eindruck der Performance-Charakteristiken zu vermitteln, folgen die Ergebnisse eines Firmen-internen Benchmarks, bei dem der VPN-Durchsatz von Securepoint Firewalls der G5-Serie getestet wurde. Dazu wurde ein Site-to-Site-VPN zwischen der jeweiligen G5-Firewall und einem deutlich leistungsstärkeren Prototypen mit Default-Einstellungen konfiguriert. Im internen Netz jeder Firewall befanden sich jeweils ein Rechner. Diese Rechner haben mit dem Messungstool `iperf3` den Datendurchsatz über das jeweilige VPN ermittelt ([Firmen-interne Quelle], Mario Rhein, 2025):

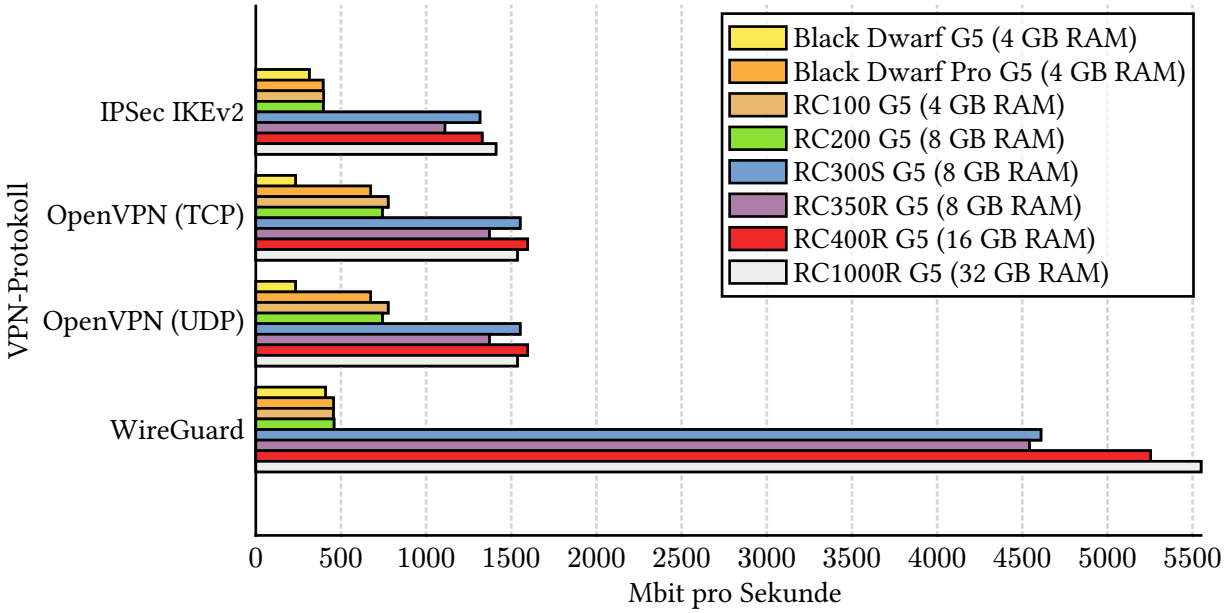

Abbildung 4: Ergebnisse eines Firmen-internen VPN-Benchmarks der G5-Modelle ([Firmen-interne Quelle], Mario Rhein, 2025)

Abbildung 4 zeigt, dass OpenVPN bei den mittelstarken Modellen Black Dwarf Pro, RC100 und RC200 sehr gut performt. Bei den stärkeren Modellen zeigt hingegen WireGuard die beste Performance; sowohl IPSec als auch OpenVPN laufen auf RC300S, RC350R, RC400R und RC1000R nicht einmal halb so gut: Sie bieten ca. 1500 Mbit pro Sekunde Daten-Durchsatz, während WireGuard auf der RC1000R über 5500 Mbit/s erreicht.
Da die Verschlüsselungsalgorithmen der verschiedenen VPN-Protokolle auf ihren jeweiligen Standard-Einstellungen waren, könnte eine Angleichung die Vergleichbarkeit verbessern. WireGuard unterstützt ausschließlich ChaCha20-Poly1305, deshalb würde sich dieser Algorithmus für einen Benchmark anbieten (Donenfeld, 2015).

2.3. OpenVPN im OSI-Schichten-Modell

OpenVPN nutzt vom Betriebssystem bereitgestellte, virtuelle Netzwerkgeräte. Über diese Netzwerkgeräte werden die Nutzdaten an das VPN gesendet oder davon empfangen. Je nach Konfiguration arbeitet OpenVPN dabei entweder auf der Sicherungsschicht mit TAP-Devices oder der Netzwerkschicht mit TUN-Devices (Yonan, 2018). TAP-Devices verarbeiten Ethernet-Frames, während TUN-Devices IP-Packets verarbeiten (Krasnyansky et al., 2002).

Die Nutzdaten werden vom Betriebssystem an dieses virtuelle Netzwerkgerät geroutet und von dort durch OpenVPN entgegengenommen, verschlüsselt und verpackt. Das Ergebnis wird in der Regel über das Internet übertragen und auf der Gegenseite wieder entpackt, entschlüsselt und über das dortige virtuelle Netzwerkgerät in das Zielnetzwerk geroutet.

Die Kommunikation zwischen den Endpunkten erfolgt über das Transmission Control Protocol (TCP) oder das User Datagram Protocol (UDP) (Transport-Schicht). Um die Nutzdaten übertragen zu können, muss erst eine Verbindung aufgebaut werden. Dazu wird eine Sitzung durch den Client initiiert, die Verbindungspartner authentifiziert und kryptografisch abgesichert (Sitzungsschicht). Nach der erfolgreichen Aushandlung der Verbindungsparameter wird diese verschlüsselte Verbindung für den sicheren Transport der Nutzdaten verwendet (Schwabe, 2016). Da OpenVPN an sich schon eine Anwendung ist, wird das OpenVPN-Protokoll der Anwendungsschicht zugeordnet. OpenVPN-Anwendungen transportieren also als Nutzdaten Ethernet-Frames oder IP-Packets, die

dann selbst als Nutzdaten die Daten der höheren Schichten enthalten, wie etwa das Hypertext Transfer Protocol Secure (HTTPS).

Die Abbildung 5 visualisiert beispielhaft den Weg eines IP-Packets von einem Client zu einem Server.

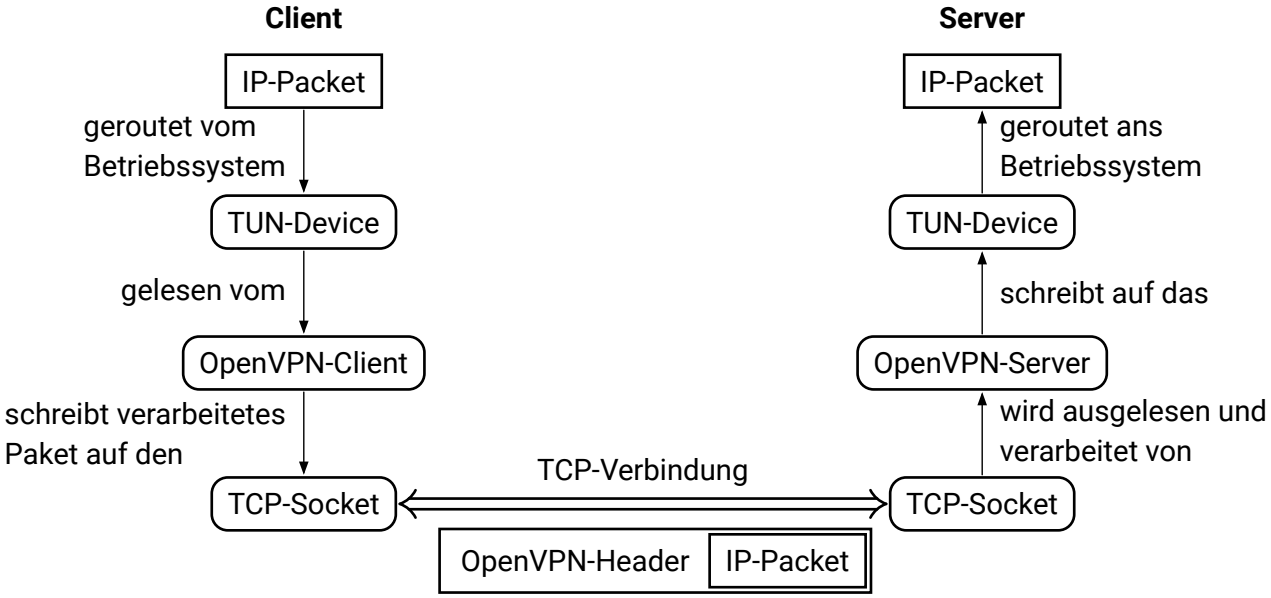

Abbildung 5: Beispielhafter Weg eines IP-Packets über einen OpenVPN-Tunnel

3. Analyse des OpenVPN-Protokolls

Damit das OpenVPN-Protokoll implementiert werden kann, muss es zunächst verstanden werden. Das OpenVPN-Protokoll hat sich seit der Einführung stetig weiterentwickelt. Dadurch gibt es viele Einstellungsmöglichkeiten, die heute nicht mehr empfohlen werden. Dazu gehört der Static-Key-Mode, bei dem ein Pre-shared Key zur Verschlüsselung genutzt wird. Ohne Session Keys und andere moderne kryptografische Praktiken ist dieser Modus nicht nur unsicher, sondern auch `deprecated` (Schwabe, 2016).

Der empfohlene OpenVPN-Modus ist der Transport Layer Security (TLS)-Mode. OpenVPN im TLS-Mode unterscheidet grundsätzlich zwischen zwei Channels: Der Control-Channel dient der Authentifizierung und Aushandlung der Verbindungsparameter, während der Data-Channel die ausgehandelten Parameter nutzt, um Daten symmetrisch verschlüsselt zu übertragen. Control-Channel und Data-Channel teilen sich dabei eine Verbindung (über UDP oder TCP) und werden über Opcodes unterschieden.

3.1. Struktur von OpenVPN-Paketen

OpenVPN-Pakete unterscheiden sich für TCP und UDP in einem Punkt: TCP-Pakete beginnen mit zwei Bytes, die die Länge des restlichen Pakets beinhalten. Dies ist erforderlich, da bei TCP die Daten ab einer gewissen Größe segmentiert werden. Überschreiten die zu sendenden Daten die *Maximum Segment Size*, werden die Daten in Segmente unterteilt übertragen und müssen beim Empfänger wieder zusammengesetzt werden (Eddy, 2022). Opcodes befinden sich im ersten Byte (bei TCP im dritten) eines jeden Pakets in den höherwertigen 5 Bits. Darauf folgen 3 Bits Key-ID (Schwabe, 2016). Die darauf folgenden Bytes beinhalten die Daten für Control-Channel oder Data-Channel.

Tabelle 1 zeigt diesen Aufbau schematisch. In der ersten Zeile ist eine Beschreibung des Paket-Abschnitts, in der zweiten Zeile die Länge des Abschnitts und in der dritten Zeile sind weitere Hinweise.

Tabelle 1: Aufbau eines OpenVPN-Pakets

Paket-Länge	Opcode	Key-ID	Daten
2 Byte	5 Bit	3 Bit	0 bis n Byte
nur TCP			von Control-Channel oder Data-Channel

Neben `CONTROL`-Packets und `DATA`-Packets gibt es außerdem noch `ACK`-Packets, die zu den `CONTROL`-Packets zählen. Sie dienen dazu, die für TLS erforderliche Zuverlässigkeit des Control-Channels sicherzustellen, was bei UDP sonst nicht gegeben wäre. Allerdings können auch andere `CONTROL`-Packets bis zu vier Acknowledgements beinhalten.

3.2. Opcodes

„Opcode“ steht für „operation code“. Die Opcodes dienen zur Unterscheidung verschiedener OpenVPN-Pakete. Der Begriff wird öfter für Prozessor-Instruktionen verwendet (Intel Corp., 1973) und ist in diesem Kontext am ehesten mit TCP Control-Bits vergleichbar (Eddy, 2022). Ähnlich wie bei TCP gibt es bei OpenVPN zum Beispiel Opcodes, um eine Verbindung zu initiieren und einen Opcode um Pakete als empfangen zu kommunizieren (Schwabe, 2016).

Die vollständige Liste aller OpenVPN-Opcodes mit ihrem jeweiligen Wert, Namen und Verwendungszweck sieht wie folgt aus:

Tabelle 2: Übersicht über OpenVPN-Opcodes (Schwabe, 2016)

Wert	Opcode	Verwendungszweck
1	CONTROL_HARD_RESET_CLIENT_V1	obsolet
2	CONTROL_HARD_RESET_SERVER_V1	obsolet
3	CONTROL_SOFT_RESET_V1	Initiiert erneuten TLS-Handshake bei einer stehenden Verbindung
4	CONTROL_V1	Kontrollpaket, verkapselt meist TLS-Handshake
5	ACK_V1	Bestätigt Empfang für bis zu 8 Kontrollpakete
6	DATA_V1	Datenpaket für Transport durch das VPN
7	CONTROL_HARD_RESET_CLIENT_V2	Initiierung der Verbindung
8	CONTROL_HARD_RESET_SERVER_V2	Bestätigt Initiierung der Verbindung
9	DATA_V2	Datenpaket mit Peer-ID
10	CONTROL_HARD_RESET_CLIENT_V3	Initiierung der Verbindung mit TLS-Crypt
11	CONTROL_WKC_V1	CONTROL_V1 für TLS-Crypt

Obsolete Opcodes

`CONTROL_HARD_RESET_CLIENT_V1` und `CONTROL_HARD_RESET_SERVER_V1` waren die Opcodes zum Verbindungsaufbau mit „TLS Key method 1“. Diese Art des Verbindungsaufbaus wird seit 2020 nicht mehr unterstützt, da sie nur zur Abwärtskompatibilität für OpenVPN-Clients vor Version 2.0 erforderlich war (Schwabe, 2020). Das Release-Datum von OpenVPN 2.0 war nicht auffindbar. GitHubs erster Versions-Tag war `v2.1_rc1` von November 2006 (OpenVPN Inc. and contributors, 2006). OpenVPN schreibt in der Software-Dokumentation, dass die Version `2.0-beta17` im November 2004 erschien (OpenVPN Inc., 2025b).

Da diese Opcodes also seit ungefähr 20 Jahren nur noch dem Zweck der Abwärtskompatibilität dienen, wird hier auf genauere Ausführungen zur „TLS Key method 1“ verzichtet. Schwabe (2016) schreibt außerdem, dass diese Opcodes in Zukunft für andere Zwecke wiederverwendet werden könnten, wenn alle anderen möglichen Opcodes belegt sein sollten.

Opcode `CONTROL_SOFT_RESET_V1`
Dieser Opcode leitet einen erneuten TLS-Handshake ein. Dies geschieht, sofern nicht anders konfiguriert, nach einer Stunde.

Opcode `CONTROL_V1`
Dieser Opcode markiert ein Kontrollpaket. Pakete beinhalten beispielsweise den TLS-Handshake oder Nutzernamen, Passwort und Schlüsselmaterial, nachdem der TLS-Handshake die Verbindung erfolgreich abgesichert hat. Ein `CONTROL_V1`-Packet kann bis zu 4 Acknowledgements beinhalten.

Opcode `ACK_V1`
Dieser Opcode dient dazu, Kontrollpakete als empfangen an die andere Seite zu kommunizieren. Ein `ACK_V1`-Packet kann bis zu 8 Acknowledgements beinhalten.

Opcode `DATA_V1`
Dieser Opcode markiert Data-Channel-Packets.

Opcode `CONTROL_HARD_RESET_CLIENT_V2`
Dieser Opcode initiiert den Verbindungsaufbau mit dem Server.

Opcode `CONTROL_HARD_RESET_SERVER_V2`
Dieser Opcode beantwortet die Verbindungsaufbau-Nachricht des Clients.

Opcode `DATA_V2`
Dieser Opcode markiert Data-Channel-Packets, die eine zusätzliche Peer-ID von 24 Bit beinhalten.

Opcode `CONTROL_HARD_RESET_CLIENT_V3`
Dieser Opcode initiiert den Verbindungsaufbau mit dem Server, wenn `tls-crypt-v2` konfiguriert ist.

Opcode `CONTROL_WKC_V1`
Äquivalent zu `CONTROL_V1`, allerdings nur für `tls-crypt-v2`. „WKC“, oder WK_c steht für „Wrapped Client Key“.

3.3. Control-Channel-Pakete

Die Initiierung einer OpenVPN-Session (vgl. Abbildung 6) nutzt Control-Channel-Pakete. Ihre Struktur ist im OpenVPN-RFC-Entwurf von Schwabe (2016) wie folgt definiert:

Tabelle 3: Aufbau eines Control-Channel-Pakets

Eigene Session-ID	Anzahl ACKs	ACKs	Peer Session-ID	Paket-ID	Daten
8 Byte	1 Byte	4 Byte × Anzahl ACKs	8 Byte	4 Byte	0 bis n Byte
		nur wenn Anzahl ACKs > 0	nur wenn Anzahl ACKs > 0	nicht in ACK_V1	je nach Opcode

Tabelle 3 zeigt in der ersten Zeile den Inhalt des Paket-Abschnitts, in der zweiten Zeile die Länge und in der dritten Zeile stehen Kommentare, wie zum Beispiel die Ausnahme, dass `ACK_V1`-Packets keine Paket-ID enthalten. Der Grund dafür ist, dass `ACK_V1`-Packets nicht acknowledged werden dürfen (Schwabe, 2016). Würde man jedes `ACK_V1` mit einem `ACK_V1` beantworten, wäre man in einer Endlosschleife aus Acknowledgements gefangen.

3.4. OpenVPN-Session-Aufbau

Laut Quarkslab SAS (2017) geht der Verbindungsaufbau wie folgt vonstatten:

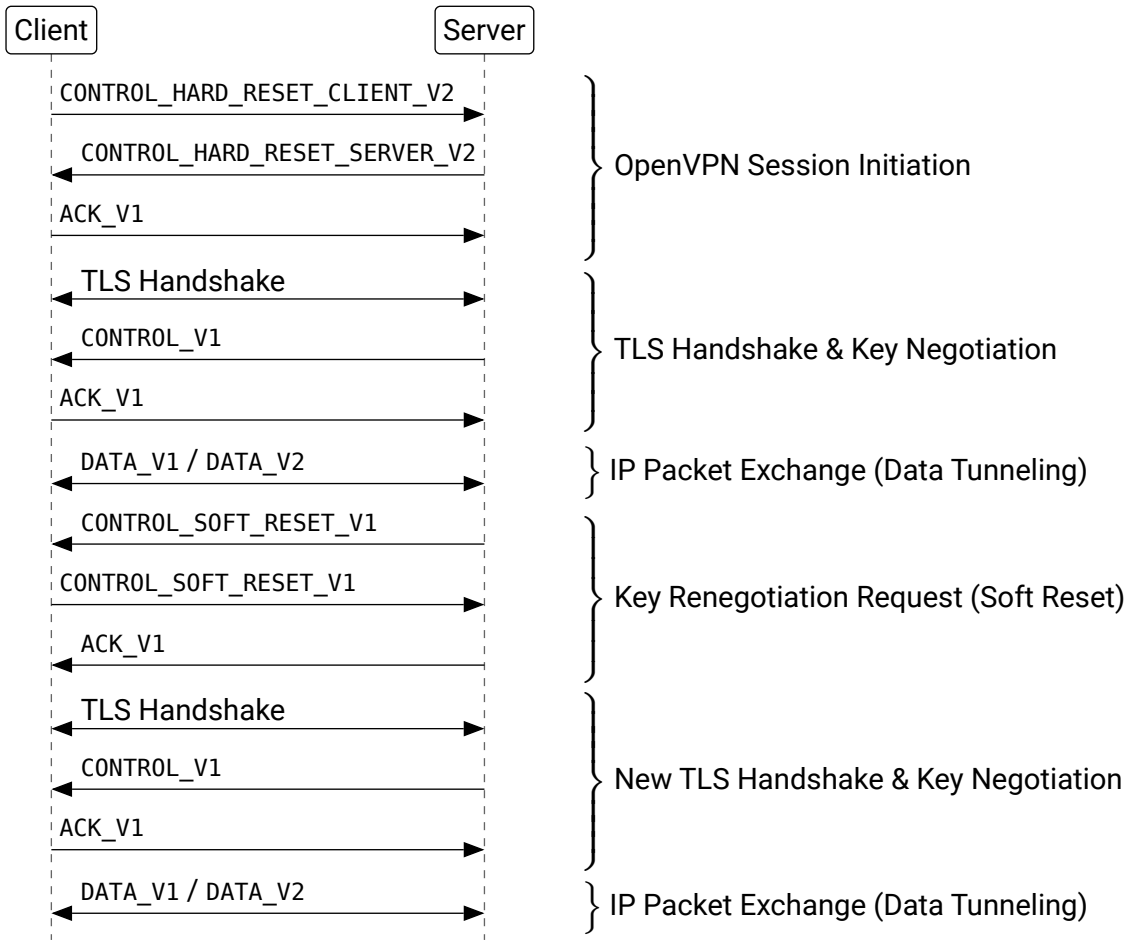

Abbildung 6: Ablauf einer OpenVPN-Session (Quarkslab SAS, 2017)

3.5. Initiierung der OpenVPN-Session

Die Initiierung der OpenVPN-Session besteht aus einem Drei-Wege-Handschlag, vergleichbar mit dem von TCP (Quarkslab SAS, 2017), vgl. Abbildung 6.

1. Der Client verschickt ein `CONTROL_HARD_RESET_CLIENT_V2`.
   - 2 Bytes für die Länge des restlichen Pakets (nur bei TCP)
   - 5 Bit `CONTROL_HARD_RESET_CLIENT_V2` (`0x07`), gefolgt von 3 Bit Key-ID 0
   - 8 Bytes Client-Session-ID (randomisiert)
   - 1 Byte für die Anzahl an ACKs, hier immer 0
   - 4 Bytes Packet-ID, hier immer 0
2. Der Server antwortet mit `CONTROL_HARD_RESET_SERVER_V2`.
   - 2 Bytes für die Länge des restlichen Pakets (nur bei TCP)
   - 5 Bit `CONTROL_HARD_RESET_SERVER_V2` (`0x08`), gefolgt von 3 Bit Key-ID 0
   - 8 Bytes Server-Session-ID (randomisiert)
   - 1 Byte für die Anzahl an ACKs, hier immer 1 für das `CONTROL_HARD_RESET_CLIENT_V2`-Packet
   - 1 × 4 Bytes ACK für Packet-ID vom `CONTROL_HARD_RESET_CLIENT_V2`-Packet, hier immer 0
   - 8 Bytes Client-Session-ID vom `CONTROL_HARD_RESET_CLIENT_V2`-Packet
   - 4 Bytes Packet-ID, hier immer 0
3. Der Client antwortet mit einem `ACK_V1`.
   - 2 Bytes für die Länge des restlichen Pakets (nur bei TCP)

- 5 Bit `ACK_V1` (`0x05`), gefolgt von 3 Bit Key-ID 0
- 8 Bytes Client-Session-ID (randomisiert)
- 1 Byte für die Anzahl an ACKs, hier 1 für das Paket vom Server
- 4 Bytes für das Acknowledgement der Packet-ID des `CONTROL_HARD_RESET_CLIENT_V2`-Packets
- 8 Bytes für die Session-ID des Peers, bekannt aus dem `CONTROL_HARD_RESET_CLIENT_V2`-Packet

3.6. TLS-Handshake in OpenVPN

Der TLS-Handshake und was darin passiert war in Schwabe (2016) zum Zeitpunkt meiner Recherche noch nicht beschrieben. Daher habe ich versucht, mit Wireshark nachzuvollziehen, was für Nachrichten in welchem Format ausgetauscht werden müssen. Ich habe die Ausgabe etwas editiert, um sie übersichtlicher zu gestalten:

```
No. Source          Destination    Protocol Length Info
 1  192.168.178.22  195.4.128.60   OpenVPN  82     MessageType: P_CONTROL_HARD_RESET_CLIENT_V2
 2  195.4.128.60    192.168.178.22 OpenVPN  94     MessageType: P_CONTROL_HARD_RESET_SERVER_V2
 3  192.168.178.22  195.4.128.60   OpenVPN  90     MessageType: P_ACK_V1
 4  192.168.178.22  195.4.128.60   TLSv1.3  391    Client Hello
 5  195.4.128.60    192.168.178.22 TLSv1.3  1264   Server Hello, Change Cipher Spec,
                                                    Application Data (reassembled)
 6  195.4.128.60    192.168.178.22 OpenVPN  1506   MessageType: P_CONTROL_V1
 7  192.168.178.22  195.4.128.60   OpenVPN  90     MessageType: P_ACK_V1
 8  195.4.128.60    192.168.178.22 TLSv1.3  435    Application Data
 9  192.168.178.22  195.4.128.60   OpenVPN  90     MessageType: P_ACK_V1
10  192.168.178.22  195.4.128.60   TLSv1.3  1506   Change Cipher Spec (reassembled)
11  192.168.178.22  195.4.128.60   OpenVPN  1043   MessageType: P_CONTROL_V1, Application Data
12  195.4.128.60    192.168.178.22 OpenVPN  90     MessageType: P_ACK_V1
13  192.168.178.22  195.4.128.60   TLSv1.3  532    Application Data
14  195.4.128.60    192.168.178.22 TLSv1.3  276    Application Data
15  192.168.178.22  195.4.128.60   OpenVPN  90     MessageType: P_ACK_V1
16  195.4.128.60    192.168.178.22 TLSv1.3  303    Application Data
17  192.168.178.22  195.4.128.60   OpenVPN  90     MessageType: P_ACK_V1
18  192.168.178.22  195.4.128.60   TLSv1.3  117    Application Data
19  195.4.128.60    192.168.178.22 OpenVPN  90     MessageType: P_ACK_V1
20  195.4.128.60    192.168.178.22 TLSv1.3  355    Application Data
21  192.168.178.22  195.4.128.60   OpenVPN  90     MessageType: P_ACK_V1
22  192.168.178.22  195.4.128.60   OpenVPN  168    MessageType: P_DATA_V2
```

Listing 2: Verkürzter Auszug eines Wireshark-Capture beim Aufbau einer OpenVPN-Verbindung (TCP) via „Passepartout VPN“ unter macOS (gefiltert nach „openvpn“)

Leider wurden die TCP-Pakete aufgrund ihrer Länge segmentiert. Deswegen gehören in Listing 2 einige Pakete, die separat aufgeführt wurden, logisch zusammen:

- Pakete 6 und 8 bilden ein OpenVPN-`CONTROL_V1`-Paket und
- Pakete 10 und 11 bilden ein OpenVPN-`CONTROL_V1`-Paket mit TLS-Change-Cipher-Spec.

Die ersten drei Pakete sind der Handshake, der in Abschnitt 3.5 beschrieben ist. Darauf folgt ein TLS-Handshake, in dem auf ein „Client Hello“ (Paket 4) vom Client der Server mit einem „Server Hello“ (Paket 5) antwortet. Wireshark hat einige Pakete, wie den TLS-Handshake, TLS („TLSv1.3“) statt OpenVPN zugeordnet. Tatsächlich ist der TLS-Handshake aber in OpenVPN-Paketen enthalten, seine Daten sind also in Control-Channel-Paketen gekapselt (vgl. Tabelle 3). Laut dem intern entwickelten Client ([Firmen-interne Quelle], Bastian Kummer, 2018) werden für den TLS-Handshake die Zertifikate aus der OpenVPN-Konfigurationsdatei verwendet. Theoretisch hätte auch das „Client Hello“ (Paket 4) das Acknowledgement des `P_CONTROL_HARD_RESET_SERVER_V2` übernehmen können, da `CONTROL_V1`-Pakete bis zu vier Acknowledgements enthalten dürfen (Schwabe, 2016). Was der Server in dem `CONTROL_V1`-Paket 6 (und Paket 8) tut, ist mir nicht ganz klar. Eventuell gehört das Paket noch zum „Server Hello“ von TLS: Denn je nach Ansicht behauptet Wireshark, dass das Paket 5 zu den anderen beiden gehört, oder eben nicht.

Laut Schwabe (2016) wird erwartet, dass Control-Channel-Nachrichten kleiner sind als 1250 Byte. Größere Nachrichten müssen deshalb auf OpenVPN-Protokoll-Ebene aufgeteilt werden. Control-Channel-Nachrichten sollen außerdem mit einem „NUL“-/0-Byte enden. Jeder TLS-Record soll einer Nachricht zuzuordnen sein.

3.7. Key-Exchange

Für die Beobachtung des weiteren Protokollablaufs war der Wireshark-Capture leider nicht so hilfreich, wie ich es mir erhofft hatte, denn die folgenden Pakete mit „Application Data“ und `ACK_V1` sind verschlüsselt und daher nicht aufschlussreich. Glücklicherweise habe ich weitere sehr knappe Beschreibungen des OpenVPN-Protokolls gefunden, die erwähnen, was genau über TLS übertragen wird (OpenVPN Inc., 2025c; Stipakov & Lichtenheld, 2022). Die Quelle ist aber anscheinend aus altem Quelltext kopiert, denn es wird dort noch „Key method 1“ erklärt, die im RFC-Entwurf schon als obsolet markiert ist (Schwabe, 2016).

Tabelle 4 zeigt eine Nachricht zum Austausch der Sitzungsparameter im Klartext.

Tabelle 4: OpenVPN-Nachricht zum Austausch der Sitzungsparameter (OpenVPN Inc., 2025c; Stipakov & Lichtenheld, 2022)

Null-Bytes	Key-Method	Key-Random	Options-String	Username	Password	Peer Info
4 Byte	1 Byte	48 Byte für Pre-Master, 32 Byte für Random 1, 32 Byte für Random 2	String-Länge + 2 Byte	String-Länge + 2 Byte	String-Länge + 2 Byte	String-Länge + 2 Byte

Die extra zwei Bytes für die Strings in der Nachricht enthalten die Länge des Strings und stehen vor dem String. Hier ist zu beachten, dass die String-Länge inklusive eines abschließenden Null-Bytes zu berechnen ist. Ist ein String leer, ist die Länge 0 (`0x0000`) und es werden nur die Längen-Bytes geschrieben.

Der Options-String (manchmal auch OCC-String) hat nur noch begrenzten Nutzen in moderneren Versionen von OpenVPN. Er wurde zur Warnung vor Inkompatibilitäten zwischen Client und Server konzipiert, die mit modernen Implementierungen umgangen werden. Dort werden Verbindungsparameter dynamisch ausgehandelt (Schwabe, 2016).
Der intern entwickelte Client setzt für den Options-String „`V0 UNDEF`“.

Peer Info enthält Informationen, die nach dem Schema „`IV_<KEY>=<VALUE>\n`“ aufgebaut sind (die spitzen Klammern markieren Variablen). Einige Werte müssen unterstützt werden, während andere optional sind (Schwabe, 2016).

Die folgenden Werte müssen laut RFC-Entwurf unterstützt sein:
- `IV_CIPHERS`: Verschlüsselungsalgorithmen für den Data-Channel, separiert mit „`:`“
- `IV_PROTO`: natürliche Zahl als String, die Zahl wird als Bit-Liste interpretiert
- `IV_SSO`: Multi-Faktor-Authentifizierung, z. B. via TOTP

Die folgenden anderen Werte werden von der internen Implementierung übermittelt ([Firmen-interne Quelle], Bastian Kummer, 2018):
- `IV_PLAT`: Betriebssystem des Clients
- `IV_VER`: OpenVPN-Version des Clients
- `IV_NCP`: veraltet, ersetzt durch `IV_CIPHERS`; „2“ bedeutet, dass AES-128-GCM and AES-256-GCM unterstützt werden.

Die `IV_PROTO`-Bits haben die folgenden Bedeutungen (aus Schwabe (2016) direkt übernommen), die zu großem Teil Bezug nehmen auf die Control-Channel-Nachrichten, die in Abschnitt 3.8 noch erklärt werden:

- Bit 0: Reserviert, soll 0 sein
- Bit 1: Der Client unterstützt den Peer-IDs
- Bit 2: Der Client erwartet eine `PUSH_REPLY` und der Server kann diese Antwort senden, ohne eine `PUSH_REQUEST` zu empfangen
- Bit 3: Der Client unterstützt den neuen Key-Exchange (Rescorla, 2010)
- Bit 4: Der Client ist in der Lage, zusätzliche Argumente in einer `AUTH_PENDING`-Nachricht zu verarbeiten
- Bit 5: Der Client unterstützt die Feature-Aushandlung im P2P-Modus
- Bit 6: Der Client ist in der Lage, die Option „`dns`“ zu verarbeiten (veraltet)
- Bit 7: Der Client ist in der Lage, `EXIT`-Nachrichten zu senden und akzeptiert die Option „`protocol-flags pushed`“
- Bit 8: Der Client kann `AUTH_FAILED,TEMP`-Nachrichten verarbeiten
- Bit 9: Der Client kann dynamisches TLS-Crypt (V2) verwenden
- Bit 10: Der Client kann das Epoch-Datenformat verwenden. Dieses Format verwendet das AEAD-Tag am Ende und verfügt über einen 48-Bit-Paketzähler zusammen mit einer 16-Bit-Schlüssel-ID, die eine 64-Bit-Paket-ID bilden
- Bit 11: Der Client ist in der Lage, die `DNS`-Nachrichten der Push-Option zu verarbeiten
- Bit 12: Der Client ist in der Lage, `PUSH_UPDATE`-Nachrichten zu verarbeiten

Der Server antwortet mit einer Nachricht in ähnlichem Format, lässt aber die 48 Byte für den Pre-Master-Key in seiner Antwort weg. Daraus können nur die Schlüssel für den Data-Channel generiert werden (Stipakov & Lichtenheld, 2022), wobei Schwabe (2016) zusätzlich einen neueren Prozess beschreibt. Da der alte Prozess in der intern entwickelten Implementierung noch genutzt wird, wird dieser hier einmal beschrieben:

Aus dem
- Pre-Master-Key,
- „Random 1“ vom Client und
- „Random 1“ vom Server

wird ein Master-Secret erzeugt (Key-Derivation). Dazu wird eine eine pseudo-zufällige Funktion genutzt, die für TLS 1.0 definiert wurde. Als weiterer Input dient der String „`OpenVPN master secret`“ ([Firmen-interne Quelle], Bastian Kummer, 2018).

Aus diesem
- Master-Secret,
- „Random 2“ vom Client,
- „Random 2“ vom Server,
- Client-Session-ID und
- Server-Session-ID

wird ein Session-Key erzeugt. Dafür wird dieselbe pseudo-zufällige Funktion wie für das Master-Secret verwendet. Als weiterer Input dient der String „`OpenVPN key expansion`“ ([Firmen-interne Quelle], Bastian Kummer, 2018).

Dieser „Data-Channel-Key“ wird zur weiteren Verwendung in vier gleich große, je 64 Byte lange Keys, aufgeteilt ([Firmen-interne Quelle], Bastian Kummer, 2018; Stipakov & Lichtenheld, 2022):
- „Cipher encrypt key“
- „HMAC encrypt key“
- „Cipher decrypt key“
- „HMAC decrypt key“

wobei „encrypt“ und „decrypt“ von Client-Seite aus beschrieben sind.

Der neue Prozess zur Key-Derivation, den Schwabe (2016) beschreibt, nutzt den in RFC 5705 (Rescorla, 2010) beschriebenen Prozess, um die Data-Channel-Keys zu erzeugen. Wird dieser neue Prozess verwendet, sollen Pre-Master, Random 1 und Random 2 zufällige Bytes enthalten, die aber nicht genutzt werden.

3.8. Control-Channel-Nachrichten

Nachdem die Schlüssel für den Data-Channel ausgehandelt sind, folgt die weitere Konfiguration, wenn dies unterstützt wird. Diese Nachrichten sind nicht mehr bit- und byteweise definiert, sondern nutzen ein Text-basiertes Protokoll (Schwabe, 2016):

Dabei beginnt der Client mit einer `PUSH_REQUEST`-Nachricht. Auf diese antwortet der Server mit einer `PUSH_REPLY`-Nachricht, die kommaseparierte Konfigurationsoptionen enthält (nach dem Schema „`PUSH_REPLY,option 1,option 2`“). Die Konfigurationsoptionen sind dabei die, die auch in einer Konfigurations-Datei angegeben werden können, wie z. B. DNS-Server, IP-Adressen oder Gateways, die für das VPN auf dem Client konfiguriert werden sollen.

Es gibt noch weitere Nachrichten wie

- `PUSH_UPDATE`, um Änderungen an der Konfiguration vorzunehmen,
- `AUTH_PENDING`, um erforderliche, noch nicht erfolgte Multi-Faktor-Authentifizierung zu kommunizieren,
- `RESTART`, um eine Beendigung der Verbindung mit Neuaufbau zu erbitten,
- `HALT`, um eine Beendigung der Verbindung ohne Neuaufbau zu erbitten,
- `AUTH_FAILED`, um Probleme mit der Authentifizierung zu kommunizieren; diese Probleme können als temporär markiert sein und sogar eine Dauer vor dem nächsten Versuch enthalten,
- `EXIT`, um die Beendigung der Verbindung zu kommunizieren,
- `CR_RESPONSE`, um Challenge-Response-Authentication Base-64-encodiert zu beantworten,
- `INFO` und `INFO_PRE`, um fehlende Authentifizierungsparameter zu übermitteln,
- `ACC`, um Nachrichten für den App Control Channel auszutauschen.

Diese weiteren Nachrichten werden von dem intern entwickelten Client nicht unterstützt ([Firmen-interne Quelle], Bastian Kummer, 2018). Es ist davon auszugehen, dass ihre Relevanz in der Praxis gering ist; abgesehen von `AUTH_FAILED` und `AUTH_PENDING`, die einen recht offensichtlichen Nutzen haben.

3.9. Data-Channel

OpenVPN unterstützt für den Data-Channel mehrere Verschlüsselungsalgorithmen. Neben AES in den Modi „GCM“ (Galois/Counter Mode) und „CBC“ (Cipher Block Chaining) mit den gängigen Schlüssellängen von 128, 192 und 256 Bit wird auch der neuere ChaCha20-Poly1305 unterstützt (OpenVPN Inc., 2025d). Dieser zeichnet sich dadurch aus, dass es anders als AES-CBC eine „Authenticated Encryption with Associated Data“ (AEAD) ist, wie auch AES-GCM. ChaCha20-Poly1305 ist aber auf Hardware ohne AES-Hardwarebeschleunigung deutlich schneller als AES (Schirrmacher, 2016). Des Weiteren wurden früher Algorithmen unterstützt, die heute als nicht mehr sicher gelten.

Tabelle 5 zeigt die Einordnung der Verschlüsselungsalgorithmen von OpenVPN Inc.

Tabelle 5: Einordnung der unterstützten Algorithmen für den Data-Channel (OpenVPN Inc., 2025d)

Empfohlene Algorithmen	Optionale Algorithmen	Veraltete Algorithmen
AES-256-GCM	AES-256-CBC	BF-CBC
AES-128-GCM	AES-192-CBC	DES-CBC
CHACHA20-POLY1305	AES-128-CBC	DES-EDE3-CBC
	AES-192-GCM	DESX-CBC
		none (keine Verschlüsselung)

Data-Channel-Packets nutzen die Opcodes `DATA_V1` oder `DATA_V2`, wobei `DATA_V1`-Packets veraltet sind. Ihr Aufbau unterscheidet sich, je nach dem, ob ein AEAD-Algorithmus verwendet wird oder nicht.

Tabelle 6 zeigt die Struktur der AEAD-Data-Channel-Packets.

Tabelle 6: Aufbau eines `DATA_V2`-Pakets mit AEAD-Cipher

Peer-ID	Packet-ID	Payload	Authentication-Tag
3 Byte	8 Byte	variabel	16 Byte
nur in `DATA_V2`	Schutz vor Replay-Attacken	verschlüsselt	

Bei Data-Channel-Packets wird die Key-ID genutzt, die Teil jedes OpenVPN-Packets ist (vgl. Tabelle 1), um die Session-Keys zu identifizieren.

Die Peer-ID, die in `DATA_2`-Packets enthalten ist, identifiziert Clients und erlaubt damit den Wechsel von IP-Adressen und Ports. Durch sie muss eine Verbindung nicht neu aufgebaut werden, sondern die bestehende Session und damit verbundene Sitzungsparameter können trotz neuer IP-Adresse oder Port weiter genutzt werden.

Die Packet-ID schützt vor Replay-Angriffen und besteht aus zwei Byte Epoch und sechs Byte Epoch Counter. Sie wird außerdem für den Initialisierungsvektor (auch IV oder Nonce; Kurzwort für „Number used once“) zur Entschlüsselung genutzt.

Der Authentication-Tag kann mit Session-Key und Initialisierungsvektor erzeugt werden, um die empfangenen Daten auf Integrität (vgl. Definition 2) und Authentizität (vgl. Definition 4) zu prüfen. Bei `DATA_V2`-Packets sind Opcode, Key-ID, Peer-ID, Packet-ID und Payload signiert, aber bei `DATA_V1`-Packets sind nur Packet-ID und Payload signiert (Schwabe, 2016).

Nicht-AEAD-Data-Channel-Packets sind wie folgt aufgebaut:

Tabelle 7: Aufbau eines `DATA_V2`-Pakets mit Nicht-AEAD-Cipher

Peer-ID	HMAC	IV	Packet-ID	Payload
3 Byte	ca. 20-32 Byte, je nach Algorithmus	16, 24 oder 32 Byte, je nach Algorithmus	4 Byte	variabel
nur in `DATA_V2`			verschlüsselt, nur bei AES-CBC	verschlüsselt

OpenVPN unterstützt hier, entgegen dem was OpenVPN Inc. (2025d) behauptet, AES in den Modi CBC, OFB und CTR (Schwabe, 2016). Davon wird an einigen Stellen aber nur CBC überhaupt erwähnt (OpenVPN Inc., 2025d). Deswegen wird hier vor allem auf den AES-CBC-Modus eingegangen.

Der HMAC hat eine Länge, die vom gewählten Hashing-Algorithmus abhängt. SHA-256 beispielsweise erzeugt einen Hash mit 32 Byte Länge. Andere Hash-Verfahren können aber längere Hashes erzeugen, wie z. B. SHA-3 mit bis zu 64 Byte, oder MD5 mit 16 Byte, das aber als unsicher gilt. Der HMAC wird über dem verschlüsselten Rest des Pakets erzeugt (IV, ggf. Packet-ID, Payload). Ist der HMAC inkorrekt, muss das Paket ohne Entschlüsselung verworfen werden.

Der IV wird bei CBC pseudo-zufällig gewählt. Seine Länge hängt vom gewählten Verschlüsselungsalgorithmus ab:
- AES-128: 16 Byte
- AES-192: 24 Byte
- AES-256: 32 Byte

Der CBC-Modus enthält eine Packet-ID. OFB und CTR nutzen die ersten acht Byte als Packet-ID.

Cipher Negotiation oder Negotiable Crypto Parameters (NCP) erlauben es, den symmetrischen Verschlüsselungsalgorithmus während des Verbindungsaufbaus zwischen Server und Client auszuhandeln.

3.10. TLS-Auth

TLS-Auth erlaubt es, alle Kontroll-Pakete zu signieren. Dadurch kann der Server sämtliche Pakete ohne eine gültige Signatur ignorieren. Dieses Feature erlaubt dem Server einen besseren Schutz vor Denial-of-Service-Angriffen, da Angreifer keine solche Signatur erzeugen können. Ein solcher Angriff wird in Quarkslab SAS (2017), Kapitel 5.1 demonstriert.

Die HMAC-Signatur wird mithilfe eines „OpenVPN Static Key V1“ erzeugt. Wird der Hashing-Algorithmus nicht spezifiziert, wird SHA-1 verwendet (Schwabe, 2016).

Tabelle 8 zeigt die Struktur von TLS-Auth-Paketen. TLS-Auth-Pakete enthalten weitere Felder, die in den Plaintext-Control-Channel-Paketen nicht enthalten sind (vgl. Tabelle 3):

Tabelle 8: Aufbau eines Control-Channel-Pakets mit TLS-Auth (Schwabe, 2016)

Eigene Session-ID	HMAC	Replay-Packet-ID	Anzahl ACKs	ACKs	Peer Session-ID	Paket-ID	Daten
8 Byte	16-64 Byte	8 Byte	1 Byte	4 Byte × Anzahl ACKs	8 Byte	4 Byte	0 bis n Byte

Die Replay-Packet-ID ist zusammengesetzt aus 4 Byte Paket-ID und 4 Byte Timestamp. Sie werden allerdings in der umgekehrten Reihenfolge als 64-Bit-Counter genutzt (Schwabe, 2016).

Zur Erzeugung des HMAC wird der statische Key und das Pseudo-Paket in Tabelle 9 genutzt.

Tabelle 9: Pseudo-Paket, mit dem der TLS-Auth-HMAC konstruiert wird (Schwabe, 2016)

Replay-Packet-ID	Opcode & Key-ID	Eigene Session-ID	Anzahl ACKs	ACKs	Peer Session-ID	Paket-ID	Daten
8 Byte	1 Byte	8 Byte	1 Byte	4 Byte × Anzahl ACKs	8 Byte	4 Byte	0 bis n Byte

Sowohl Opcode als auch Key-ID werden im HMAC berücksichtigt. Außerdem wechselt die Replay-Packet-ID ihre Position an den Anfang des Pakets. Der HMAC selbst ist logischerweise nicht in dem Pseudo-Paket.

3.11. TLS-Crypt

TLS-Crypt verschlüsselt die OpenVPN-Pakete des Control-Channels mit einem statischen Pre-shared Key.

TLS-Crypt-V2 geht noch einen Schritt weiter: Hier wird der Schlüssel aus dem Client-Zertifikat genutzt, und so die Nutzung unterschiedlicher Schlüssel für jeden Client ermöglicht.

Beide TLS-Crypt-Modi sind recht aufwendig in ihrer Konstruktion (Schwabe, 2016) und werden von der internen Implementierung nicht unterstützt ([Firmen-interne Quelle], Bastian Kummer, 2018). Aus diesen Gründen wird auf eine detaillierte Erklärung dieses Teils des Protokolls an dieser Stelle verzichtet.

3.12. Fazit zum OpenVPN-Protokoll

Im Folgenden möchte ich meine Analyse mit einer kurzen Kritik des Protokolls abschließen, um auf die Komplexität des Protokolls und den unverschlüsselten Handshake einzugehen.

3.12.1. Mehr als TLS

Ich halte das OpenVPN-Protokoll für komplizierter, als es sein müsste:
Statt die bestehende TLS-Verbindung zu nutzen, um darüber die Nutzdaten verschlüsselt zu transportieren, wird diese nur für den Control-Channel genutzt, um darüber Nutzer-Authentifikation abzuwickeln und Sitzungsparameter auszuhandeln. Das ist wahrscheinlich historisch gewachsen, um UDP über den eingebauten Mechanismus für Zuverlässigkeit unterstützen zu können, aber den Data-Channel nicht zuverlässig machen zu müssen (Schwabe, 2016).
Bei UDP gibt es mit Datagram Transport Layer Security (DTLS) (Rescorla et al., 2022) und Quick UDP Internet Connections (QUIC) (Iyengar & Thomson, 2021) inzwischen aber offene Standards, die die Zuverlässigkeit beim Handshake gewährleisten und Verbindungen mit TLS absichern können. „wstunnel“ bietet beispielsweise ein anderes TLS-basiertes VPN auf TCP-Websockets (Gerard, 2025), Ferrumgate hat ein auf QUIC aufbauendes VPN-Protokoll entwickelt (Ferrumgate, 2023) und Fortinet bietet in den eigenen VPN-Produkten einen DTLS-Modus (candawi, 2023).

3.12.2. Fingerprinting

OpenVPN muss wegen des eigenen Mechanismus für Zuverlässigkeit die OpenVPN-Verbindung unverschlüsselt initiieren:
Damit weist die Initiierung mit Opcodes, Key-ID, Session-IDs, Packet-IDs und Acknowledgements Metadaten im Klartext auf. Diese Metadaten unterscheiden sich nicht stark von Handshake zu Handshake und erzeugen somit ein leicht erkennbares Byte-Pattern, was das Protokoll identifizierbar macht.
Es eignet sich damit nicht zur Umgehung von Zensur, da Regierungen mit Zugriff auf die Netzwerkinfrastruktur eines Landes dieses Byte-Pattern nutzen können, um OpenVPN-Verbindungen zu unterbinden (Xue et al., 2024).
Um abwärtskompatibel zu sein, wird das Problem selbst bei TLS-Crypt nicht komplett behoben und enthält weiter einen unverschlüsselten Opcode, Key-ID und Session-ID. Der Opcode am Beginn eines jeden Pakets wird auch bleiben müssen, um die Pakete differenziert behandeln zu können. Ohne Opcode wäre es unmöglich, TLS für UDP abzusichern, da der Opcode im Endeffekt für die Initiierung der zuverlässigen OpenVPN-Verbindung erforderlich ist.

TLS, DTLS und QUIC weisen eventuell auch gewisse Byte-Patterns auf; durch ihre ständige oder wachsende Verwendung wäre Fingerprinting allerdings vermutlich erschwert. Damit könnte ein VPN-Protokoll das TLS, DTLS oder QUIC nutzt, unter dem anderen Traffic wie HTTPS im Internet weitgehend unerkannt bleiben.

3.12.3. Plaintext-Handshake ohne Integritätsnachweis

Neben dem DoS-Angriff auf einen OpenVPN-Server durch einen fehlenden Authentizitätsnachweis (Quarkslab SAS, 2017) gibt es noch ein weniger schwerwiegendes Problem im Handshake zur OpenVPN-Sitzungsinitiierung:
Er verfügt über keine kryptografischen Integritätsmechanismen. Somit ist theoretisch ein Machine-in-the-Middle-Angriff denkbar, bei dem die OpenVPN-Pakete manipuliert werden. Es ist dabei nicht möglich, die potenzielle Manipulation dieser Pakete zu erkennen. Dadurch könnte ein erfolgreicher Verbindungsaufbau verhindert werden (Denial of Service). Wenn TLS-Auth oder TLS-Crypt konfiguriert wurden, gibt es dieses Problem nicht, da diese Modi Integritätsmechanismen auch in die ersten OpenVPN-Pakete integrieren.

3.12.4. Viele Konfigurationsmöglichkeiten

Die vielen Konfigurationsmöglichkeiten erschweren eine sichere und erfolgreiche Konfiguration und erhöhen die Komplexität der Software. WireGuard geht die entgegengesetzte Richtung, hat sehr wenige Konfigurationsoptionen und kann in ca. 4000 Zeilen Code implementiert werden (Donenfeld, 2015).
Ein Beispiel für die unsichere Konfiguration von OpenVPN ist die Konfigurationsmöglichkeit von unsicheren Verschlüsselungsalgorithmen. Die werden zwar nicht empfohlen, sind aber auch nicht verboten (OpenVPN Inc., 2025d). WireGuard hingegen erlaubt gar keine Konfiguration an der Stelle, sondern erzwingt ChaCha20-Poly1305.

Die Konfigurabilität von OpenVPN ist aber auch ein Vorteil, da es beispielsweise die Zuweisung von IP-Adressen dynamisch über DHCP unterstützt (Yonan, 2018), was WireGuard nicht erlaubt (Donenfeld, 2015).
Ein weiterer Vorteil der Konfigurabilität: Sollte ein konfigurierter Verschlüsselungsalgorithmus in Zukunft gebrochen werden, erlaubt OpenVPN einen Wechsel des Algorithmus, während man bei WireGuard auf ein Software-Update warten müsste.

4. Analyse der bestehenden Software

Securepoints OpenVPN-Client-Library kann sowohl als Standalone-Binary kompiliert werden als auch als Shared-Library. Die Software unterstützt verschiedene Betriebssysteme:

- Linux
- Android (wie Linux, aber TUN-Device wird vom Betriebssystem verwaltet)
- iOS
- macOS (als iOS-App, über Apples „Designed for iPad“)
- FreeBSD
- Windows

Die Software hat eine Abhängigkeit auf OpenSSL für TLS im Control-Channel und die Verschlüsselung im Data-Channel.

4.1. Struktur des Quellcodes

Abgesehen von der `main.c` gibt es zu jeder `.c`-Datei eine `.h`-Header-Datei.

`libovpn-client` und `ovpn-client` bieten eine Application Programming Interface (API) für die Nutzung. `ovpn-client` enthält außerdem den Programm-Loop, der die Verbindung initiiert und Pakete verarbeitet.

Das Modul `options` implementiert das Parsen von Konfigurationsdateien. Es definiert das `struct options_s`, in dem die geparsten Konfigurationsoptionen gespeichert werden. Für TLS-Verbindungen stellt `bio_s_packet_buffer` gebufferte I/O-Funktionen bereit und kapselt dabei die Funktionalität aus `openssl/bio.h`. Das Modul `buffer` enthält eine eigene Buffer-Implementierung mit einer festen Größe von 2048 Byte. Mit `dbg` werden Logging-Funktionen bereitgestellt, während `packet_buffer` Funktionen zum gebufferten Schreiben und Lesen von Paketen implementiert. Das Modul `random` stellt plattformabhängige, kryptografisch sichere Pseudozufallszahlengeneratoren bereit. Das Modul `ssl` enthält Funktionen zum Parsen von OpenSSL-Chiffren, Zertifikaten und Private Keys sowie zur Ver- und Entschlüsselung von Paketen. Über `tun` wird die Konfiguration eines TUN-Devices ermöglicht, einschließlich DNS-, MTU- und Gateway-Einstellungen. Die Datei `main.c` bildet den Einstiegspunkt der Anwendung: Sie parst eine Konfigurationsdatei und startet anschließend den VPN-Tunnel.

4.2. Schnittstellen der Bibliothek

Die Bibliothek stellt eine Schnittstelle zum Verbindungsaufbau in C zur Verfügung, die allerdings über zwei Header-Dateien verteilt ist. Der Sinn hinter dieser Separation erschließt sich mir nicht.

`ovpn-client.h` (Listing 3) verfügt über Funktionen, die die bisher empfangenen und gesendeten Bytes zurückgeben (Zeilen 1 und 2), sowie über eine Funktion, die mit einer geparsten OpenVPN-Konfiguration ein VPN startet (Zeile 3).
Außerdem gibt es Funktionen, die das VPN wieder beenden (Zeilen 4 und 5), oder pausieren (Zeile 6). Es gibt allerdings keine Funktion, um die pausierte Sitzung wieder zu starten.

C

```c
1 long long ovpn_in_bytes(void);
2 long long ovpn_out_bytes(void);
3 int ovpn_client_run(struct options_s *opt);
4 void ovpn_client_stop(void);
5 void ovpn_client_logout(void);
6 void ovpn_client_hibernate(void);
```

Listing 3: Auszug der Schnittstellendefinition aus `ovpn-client.h`

`libovpn-client.h` (Listing 4) nutzt selbst `ovpn-client.h`, ergänzt sie aber, da es eine Funktion enthält, die das Verbinden über einen String statt über ein `struct options_s` erlaubt (Zeile 1). Die Funktion in Zeile 2 hingegen ruft nur `ovpn_client_stop()` auf.

```C
1 int ovpn_client_connect_with_string(char *config_string);
2 int ovpn_client_disconnect(void);
```

Listing 4: Auszug der Schnittstellendefinition aus `libovpn-client.h`

Eine weitere in den Clients genutzte Schnittstelle ist die Funktionsdefinition für Log-Aufrufe (Zeile 1). Diese Funktion kann von Clients implementiert werden, um mit der Funktion in Zeile 2 den voreingestellten Log-Nachrichten-Handler zu ersetzen. Somit können zum Beispiel Nachrichten der Library in der Datenbank einer Client-Applikation gespeichert werden.

```C
1 typedef void (*msg_handler_t)(int, char *);
2 void msg_set_handler(msg_handler_t new_msg_handler);
```

Listing 5: Auszug der Schnittstellendefinition aus `dbg.h`

4.3. Konfiguration

Ein zentraler Bestandteil der bestehenden Software ist der Konfigurations-Parser in `options.h` und `options.c`. Dieser Parser parst die Konfigurations-Datei zu einer Konfiguration vom Typ `struct options_s`. Um den Parser zu verstehen, muss man zuerst den Aufbau einer OpenVPN-Konfigurationsdatei verstehen. Die meisten Konfigurationsoptionen lassen sich dabei als einzeiliges n-Tupel aus Option und Wert (Key-Value-Pair) verstehen.

Beispiele:

- `dev tun` konfiguriert, welche Art von virtuellem Netzwerkinterface angelegt werden soll.
  - `tun` für `tun`-Devices, die auf OSI-Layer 3 (IP) operieren,
  - `tap` für `tap`-Devices, die auf OSI-Layer 2 (Ethernet) operieren.
- `cipher AES-256-GCM` konfiguriert die Verschlüsselung des Data-Channels.
  - die validen Optionen sind in Tabelle 5 aufgeführt.
- `auth SHA256` konfiguriert den Hashing-Algorithmus für die Authentifizierung im Data-Channel.
- `proto tcp` kann global für alle Server das Protokoll auf Vermittlungs- und Transport-Schicht-Ebene festlegen.
  - Die validen Optionen sind `TCP`, `UDP`, `TCPv6` und `UDPv6`, wobei die Groß-/Kleinschreibung ignoriert wird und das `'v'` in den IPv6-basierten Protokollen optional ist.
- `remote janhopp.de 1194 udp` konfiguriert einen Server, und auf welchem Port er erreichbar ist.
  - Als erster Wert kann sowohl eine IP-Adresse, als auch eine Domain aufgeführt werden.
  - Der zweite Wert ist ein optionaler Port, der Standard-Port für OpenVPN ist 1194.
  - Der dritte Wert ist optional das Transport-Protokoll, falls es sich von dem über `proto` gesetzten Wert unterscheidet, oder `proto` nicht gesetzt ist.
- `cert`, `ca`, `key` und `tls-auth` haben einen Datei-Pfad als Wert, wenn sie einzeilig sind.
  - `tls-auth` hat dabei die Besonderheit, kein gängiges Key-Format zu verwenden.

Neben den einzeiligen Optionen gibt es auch mehrzeilige Optionen. Mehrzeilige Optionen starten dabei mit `<option>` und enden mit `</option>`. Dies ist nützlich, um nur eine Konfigurations-Datei zu haben, denn so lassen sich Zertifikate und Schlüssel in die Konfigurations-Datei integrieren. Zu den hier unterstützten mehrzeiligen Optionen zählen `cert`, `ca`, `key` und `tls-auth`.

Zwingend erforderlich laut der `validate`-Funktion für `struct options_s` sind `cert`, `ca`, `key`, `remote`, und `link-mtu` oder `tun-mtu`, die sich gegenseitig ausschließen.

4.4. Main-Loop

Der Main-Loop des Programms organisiert den Verbindungsaufbau und verhält sich dabei vergleichbar mit einem Mealy-Automaten. Die Darstellung dieses Automaten wäre allerdings unübersichtlich. Der Main-Loop wechselt seinen Zustand, gespeichert als `enum ctrl_channel_state_e ctrl_state` in `struct ovpn_ctx_s` je nach seinem aktuellen Zustand und dem Opcode des aktuellen Pakets (vgl. Tabelle 2).

Listing 6 zeigt die möglichen Zustände des Control-Channels.

```c
#define STATE_NO_INPUT 0x10

enum ctrl_channel_state_e
{
  STATE_INIT = 0x0,
  STATE_WAIT_FOR_HARD_RESET = 0x1,
  STATE_SSL_CONNECT = 0x2 | STATE_NO_INPUT,
  STATE_SSL_CONNECT_WANT_READ = 0x3,
  STATE_SSL_GEN_KEY_DATA = 0x4 | STATE_NO_INPUT,
  STATE_SSL_WRITE_KEY_DATA = 0x5 | STATE_NO_INPUT,
  STATE_SSL_WRITE_KEY_DATA_WANT_READ = 0x6,
  STATE_SSL_READ_KEY_DATA = 0x7,
  STATE_SSL_ESTABLISHED = 0x8,
  STATE_SSL_HIBERNATE = 0x9,

  STATE_ANY = 0xa,
};
```

Listing 6: `ctrl_channel_state_e`*-Enum aus* `ovpn-client.c`

Die Zustandsübergänge sind über die Funktionen in der Funktionsmatrix in Listing 7 definiert, das verwendete Makro und die Funktionsdefinition befinden sich darüber.

```c
#define _H(ctrl_state, opcode) ((ctrl_state << 4) + opcode)

typedef int (*ovpn_handler_fn)(struct ovpn_ctx_s *, struct ovpn_parsed_frame_s *);

ovpn_handler_fn ctrl_handler[512] = {
  [_H(STATE_WAIT_FOR_HARD_RESET, P_CONTROL_HARD_RESET_SERVER_V2)] =
ovpn_handle_hard_reset_server,
  [_H(STATE_SSL_CONNECT, NOOP)] = ovpn_handle_ssl_connect,
  [_H(STATE_ANY, P_CONTROL_HARD_RESET_SERVER_V2)] = ovpn_handle_hard_reset_server,
  [_H(STATE_SSL_ESTABLISHED, P_CONTROL_SOFT_RESET_V1)] = ovpn_handle_soft_reset,
  [_H(STATE_SSL_CONNECT_WANT_READ, P_CONTROL_V1)] = ovpn_handle_ssl_connect,
  [_H(STATE_SSL_GEN_KEY_DATA, NOOP)] = ovpn_handle_gen_ssl_key_data,
  [_H(STATE_SSL_WRITE_KEY_DATA, NOOP)] = ovpn_handle_ssl_key_data,
  [_H(STATE_SSL_WRITE_KEY_DATA_WANT_READ, P_CONTROL_V1)] = ovpn_handle_ssl_key_data,
  [_H(STATE_SSL_READ_KEY_DATA, P_CONTROL_V1)] = ovpn_handle_ssl_key_data,
  [_H(STATE_SSL_ESTABLISHED, P_CONTROL_V1)] = ovpn_handle_ctrl,
};
```

Listing 7: Control-Channel-Funktionsmatrix aus `ovpn-client.c` *(gekürzt)*

Nachdem über diese Zustände eine Verbindung aufgebaut und konfiguriert ist, beginnt der Datentransfer. Die ans TUN-Device geleiteten, zu sendenden Daten werden verschlüsselt, in ein OpenVPN-Paket gekapselt und über den Socket an den Server geschickt. Die vom Socket

empfangenen OpenVPN-Pakete werden geparst und ihre Nutzdaten entschlüsselt und an das TUN-Device weitergeleitet.

Abbildung 7 stellt die Komponenten und ihr Zusammenspiel in der Software vom Quellcode abstrahiert dar. Sie zeigt den OpenVPN-Client als System, das IP-Packets an das TUN-Device empfängt, verschlüsselt, mit OpenVPN-Metadaten einpackt und über den Socket an den Server verschickt. In entgegengesetzter Richtung werden OpenVPN-Packets vom Socket empfangen, entpackt, entschlüsselt und die Daten als IP-Packets an das TUN-Device weitergeleitet.
Der Data-Channel erhält sein Schlüsselmaterial vom Control-Channel. Dieser besteht aus einem Reliability Layer, einer TLS-Session und der Session Negotiation. Der Reliability Layer stellt über Acknowledgements die Zuverlässigkeit sicher, damit eine TLS-Session aufgebaut werden kann. Über die TLS-Session werden dann die Sitzungsparameter ausgehandelt und Schlüsselmaterial ausgetauscht, das im Data-Channel zur Ver- und Entschlüsselung genutzt wird.

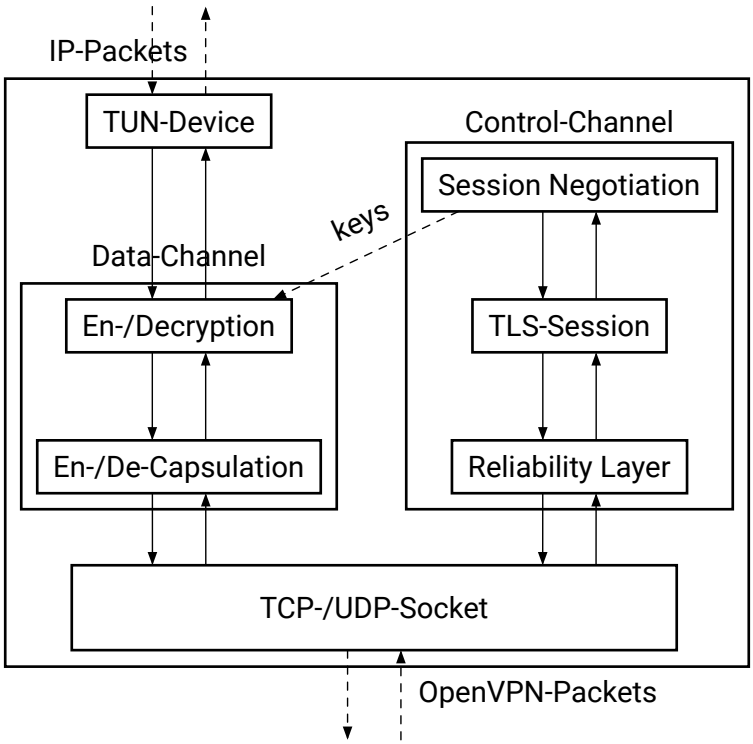

Abbildung 7: Überblick der notwendigen Komponenten für einen OpenVPN-Client

4.5. Data-Channel

Die statischen Funktionen `ovpn_link_write`, `ovpn_link_read` und `ovpn_parse_frame` in `ovpn-client.c` serialisieren und deserialisieren die Pakete, die über die Verbindung zum Server gesendet und empfangen werden. Dabei ist `ovpn_link_write` für die Serialisierung aber nicht für die Verschlüsselung zuständig. Die Parameter sind aber nicht bloß die zu übermittelnden Bytes, sondern auch das `struct ovpn_ctx_s`, das alle für die Sitzung relevanten Daten enthält. Dabei wird `ovpn_link_write` für die Serialisierung aller verschiedenen Paket-Formate genutzt, die sich im Aufbau teils stark unterscheiden. Daher ist die Funktion recht verschachtelt. `ovpn_link_read` ist dafür relativ simpel gehalten, da die Serialisierung nach dem Lesen in `ovpn_parse_frame` erfolgt.

Die kryptografischen Funktionen des Data-Channels unterscheiden sich, je nachdem, ob eine AEAD-Cipher eingesetzt wird. Die Funktionssignaturen in Listing 8 sind recht lang, da sie für viele Pointer die Länge als weiteren Parameter benötigen. Das ist bei C notwendig, da Pointer an sich

keine Länge besitzen. Neben `cipher`, `key`, `iv` und den Cipher- und Klartextparametern (`in` und `out`) wird ein Funktionspointer übergeben, der Fehler loggen kann. Bei den AEAD-Funktionen kommen noch `aad` (Additional Authenticated Data) und der Authentication-Tag `tag` hinzu.

```c
int encrypt_data(const EVP_CIPHER *cipher, const void *key, size_t key_len,
                 const void *iv, size_t iv_len, void *in, size_t in_size,
                 void *out, int *out_size, void (*err_cb)(char *, ...));
int decrypt_data(const EVP_CIPHER *cipher, const void *key, size_t key_len,
                 const void *iv, size_t iv_len, void *in, size_t in_size,
                 void *out, int *out_size, void (*err_cb)(char *, ...));

int encrypt_data_aead(const EVP_CIPHER *cipher, const void *key, size_t key_len,
                      const void *iv, size_t iv_len, void *in, size_t in_len,
                      void *out, int *out_len, const void *aad, size_t aad_len,
                      void *tag, size_t *tag_len, void (*err_cb)(char *, ...));
int decrypt_data_aead(const EVP_CIPHER *cipher, const void *key, size_t key_len,
                      const void *iv, size_t iv_len, void *in, size_t in_len,
                      void *out, int *out_len, const void *aad, size_t aad_len,
                      void *tag, size_t tag_len, void (*err_cb)(char *, ...));
```

Listing 8: Funktionssignaturen zur Verschlüsselung des Data-Channels aus `ssl.h`

4.6. Weitere Komponenten

Es gibt noch weitere Komponenten der Software, die nicht zentraler Bestandteil der Implementierung des OpenVPN-Protokolls, aber dennoch zu erwähnen sind.

4.6.1. `tun`-Geräteverwaltung

Um ein VPN erfolgreich auf dem Gerät zu starten, ist ein TUN-Device erforderlich. Dies wird so konfiguriert, dass die Routen, die über das VPN erreichbar sein sollen, auf das TUN-Device zeigen. Es gibt über Makros definierte „Setter“ für DNS, Maximum Transmission Unit (MTU) und Gateways für IPv4 und IPv6 sowie Funktionen zum Hinzufügen von Routen und IP-Adressen zur TUN-Device-Konfiguration. Mit `tun_open()` kann ein neues TUN-Device erstellt und die Konfiguration darauf angewendet werden (Listing 9). `tunConfig_reset()` wird zum Speichermanagement verwendet, da dies in C mit `free()` und `memset()` manuell erfolgen muss.

```c
#define tunConfig_set_dns(cfg, dns, cnt) (cfg)->dnsServer = dns, (cfg)->dnsCnt = cnt
#define tunConfig_set_mtu(cfg, mtu_value) (cfg)->mtu = mtu_value
#define tunConfig_set_remote_gateway(cfg, arg) (cfg)->remote_gateway = arg
#define tunConfig_set_remote_gateway_ip6(cfg, arg) (cfg)->remote_gateway_ip6 = arg

void tunConfig_add_route(tunConfig *cfg, char *dst, char *msk, char *router);
void tunConfig_add_address(tunConfig *cfg, char *addr, char *mask);
void tunConfig_reset(tunConfig *cfg);
int tun_open(tunConfig *config, tun_ctx_s *tun_ctx);
int tun_close(tun_ctx_s *tun_ctx);
```

Listing 9: Angepasster Auszug aus `tun.h`

4.6.2. `buffer`-Verwaltung

`buffer` besteht aus drei Feldern, die über Funktionen manipuliert werden:

- `data`, eine Array von 2048 Byte,
- `pos`, die Position in `data`, von der gelesen werden soll,
- `len`, die Länge von lesbaren Daten in `data`.

Dabei sollen die Funktionen einen sicheren Zugriff erlauben. Leider kam es in der Vergangenheit aber selbst mit diesen Funktionen zu den eingangs beschriebenen Abstürzen (vgl. Listing 1).

`ovpn-client.c` sowie `packet_buffer.c` und darüber `bio_s_packet_buffer.c` verwenden dieses `struct buffer_s`, um darin Daten für die Weiterverarbeitung (z. B. Ver- und Entschlüsselung) zwischenzuspeichern.

4.6.3. Makefile

Die `Makefile` (Listing 10) konfiguriert den Build der Library. Leider funktioniert sie in diesem Zustand vor allem auf FreeBSD und Linux, nicht aber unter macOS und Windows. Das liegt an dem in Zeile 2 konfigurierten Pfad `/usr/lib`, da dieser auf macOS und Windows nicht existiert oder nicht die notwendigen Dateien enthält. Der Build-Prozess für diese Plattformen wird über andere Dateien in den jeweiligen Projekten konfiguriert. Abgesehen davon ist die Konfiguration der `Makefile` ein aufwendiger Prozess, so muss z. B. jede neue Header-Datei im Projekt explizit in `HEADERS` ergänzt werden. Außerdem muss selbst der `clean`-Befehl (Zeilen 20 bis 22), also das Löschen der Build-Artefakte, explizit definiert werden. Das alles erlaubt zwar eine sehr exakte Konfiguration des Builds, erfordert jedoch auch viel Fachwissen.

Makefile

```
1  OPENSSL_CFLAGS =
2  OPENSSL_LIBS = -L/usr/lib -lcrypto -lssl
3  CFLAGS = -ggdb -Wall -Wextra -Wsign-compare -Werror -std=c99  ${OPENSSL_CFLAGS} -fPIC
4  LDFLAGS = -ggdb -Wall -Wextra -Wsign-compare -Werror -std=c99 ${OPENSSL_LIBS} -pthread
5
6  OBJECTS = ovpn-client.o bio_s_packet_buffer.o packet_buffer.o buffer.o ssl.o dbg.o
   tun.o random.o options.o contrib/strsep.o
7  EXEC_OBJECTS = ${OBJECTS} main.o
8  LIB_OBJECTS = ${OBJECTS} libovpn-client.o
9
10 HEADERS = bio_s_packet_buffer.h buffer.h dbg.h options.h packet_buffer.h random.h
   ssl.h tun.h contrib/strsep.h
11
12 ovpn-client: ${EXEC_OBJECTS} ${HEADERS}
13   ${CC} ${EXEC_OBJECTS} -o ovpn-client ${LDFLAGS}
14
15 libovpn-client.so: ${LIB_OBJECTS}
16   ${LD} -shared -soname $@.1 -o $@.1.0 ${LIB_OBJECTS} ${OPENSSL_LIBS}
17   ln -fs $@.1.0 $@.1
18   ln -fs $@.1 $@
19
20 clean:
21   rm -f ${OBJECTS} main.o libovpn-client.o ovpn-client libovpn-client.so*
22   rm -rf doc
```

Listing 10: Auszug aus der Makefile-Datei der internen OpenVPN-Client-Bibliothek

5. Planung der Reimplementierung

Vor dem Beginn der Implementierung mussten einige Entscheidungen getroffen werden:

- Welche Teile des Protokolls sollen initial unterstützt werden?
- Wie soll die Schnittstelle für Clients definiert werden?
- Welche Programmiersprache sollte verwendet werden?

5.1. Anforderungen an die Software

FA 1: Funktionale Anforderung

OpenVPN-Konfiguration einlesen

Die Bibliothek muss das OpenVPN-Konfigurationsformat implementieren und unterstützen, um darauf aufbauend die Verbindung herstellen zu können.

FA 2: Funktionale Anforderung

OpenVPN-Handshake unterstützen

Die Bibliothek muss den vollständigen OpenVPN-Handshake-Prozess implementieren und unterstützen, um damit die Datenübertragung auszuhandeln.

FA 3: Funktionale Anforderung

OpenVPN-Daten-Tunneling unterstützen

Die Bibliothek muss in der Lage sein, VPN-Nutzdaten über einen gesicherten Tunnel zu übertragen.

FA 4: Funktionale Anforderung

Plattformübergreifende Nutzung ermöglichen

Die Bibliothek kann den Einsatz auf verschiedenen Plattformen (z. B. macOS, Linux, Android, iOS) erlauben, sollte aber zumindest die Erweiterbarkeit um weitere Plattform-Unterstützung erlauben.

FA 5: Funktionale Anforderung

Routing konfigurieren (plattformabhängig)

- Auf macOS und Linux soll die Bibliothek Routing selbst übernehmen können.
- Auf Android und iOS soll Routing durch die Plattform-APIs gesteuert werden.

Es gibt noch eine paar weitere Anforderungen, die aber optional sind. Sie gehen teilweise weit über den Umfang eines Prototypen hinaus und werden hier nur zum Zweck der Vollständigkeit erwähnt.

FA 6: Funktionale Anforderung

***(Optional)* Soft-Resets unterstützen**

Die Bibliothek kann Unterstützung für OpenVPN-Soft-Resets bereitstellen.

FA 7: Funktionale Anforderung

***(Optional)* TLS-Auth (Authenticated Control Channel Packets)**

Die Bibliothek kann signierte (TLS-Auth) Control-Channel-Pakete unterstützen.

FA 8: Funktionale Anforderung

***(Optional)* TLS-Crypt und TLS-Crypt-V2 (Encrypted Control Channel Packets)**

Die Bibliothek kann verschlüsselte (TLS-Crypt und TLS-Crypt-V2) Control-Channel-Pakete unterstützen.

Abwesend sind hier Anforderungen für `ovpn_client_hibernate()` (vgl. Listing 6), um die Verbindung zu pausieren. Dieses Feature ist dazu geeignet, in nativen Clients implementiert zu werden, statt in einer plattformunabhängigen Bibliothek, da gerade mobile Plattformen dafür eigene Mechanismen besitzen. Außerdem übersteigt das Feature den Umfang eines Prototypen.

5.2. Anforderungen an die Schnittstelle der Software-Bibliothek

Wegen ihrer Übersichtlichkeit sind die Funktionssignaturen hier schon in Go angegeben.

SA 1: Schnittstellenanforderung

Connect: Verbindungsaufbau initiieren und Tunnel starten

Parameter:
- Konfiguration als `string`

Rückgabewert:
- ob die Operation erfolgreich war als `bool`

Funktionssignatur in Go:

```
func Connect(config string) bool
```

SA 2: Schnittstellenanforderung

Disconnect: Verbindungsabbau initiieren und Tunnel stoppen

Parameter: keine erforderlich

Rückgabewert:
- ob die Operation erfolgreich war als `bool`

Funktionssignatur in Go:

```
func Disconnect() bool
```

SA 3: Schnittstellenanforderung

InBytes: Gibt Information über empfangene Bytes seit Start der Software

Rückgabewert:
- empfangene Bytes als natürliche Zahl

Funktionssignatur in Go:

```
func InBytes() uint64
```

SA 4: Schnittstellenanforderung

OutBytes: Gibt Information über gesendete Bytes seit Start der Software

Rückgabewert:
- gesendete Bytes als natürliche Zahl

Funktionssignatur in Go:

```
func OutBytes() uint64
```

SA 5: Schnittstellenanforderung

SetLogHandler: Funktion zum Setzen einer Logging-Funktion innerhalb von Apps statt in Go nativ

Parameter:
- Funktion, die Log-Level und Log-Nachricht entgegennimmt, und diese loggt

Funktionssignatur in Go:

```
func SetLogHandler(logger func(logLevel int, message string))
```

5.3. Eingrenzung

Zur prototypischen Entwicklung gehören nur die Features, die zwingend erforderlich sind, um das OpenVPN-Protokoll in simpelster Form abzubilden.

TLS-Auth, TLS-Crypt oder TLS-Crypt-V2 zähle ich nicht dazu, da sie zusätzliche Features bieten, die über ein „Minimum Viable Product“ hinausgehen.

Zur Portabilität der Software-Bibliothek trägt neben der Wahl der Programmiersprache auch die Architektur der Software bei. Dazu gilt es, darauf zu achten, Plattform-Abhängigkeiten bei der Entwicklung zu erkennen und mit geeigneten Mitteln zu abstrahieren. Das ermöglicht die spätere Erweiterung um andere Plattformen, die über die Entwicklung eines Prototypen hinausgeht.

Da auch die interne Originalimplementierung kein Ethernet-Bridging erlaubt, wird auch hier im Folgenden nur IP-Packet-Tunneling betrachtet.

5.4. Vergleich von Programmiersprachen

Für die Reimplementierung der OpenVPN-Library gab es im Betrieb zwei Vorschläge für die Wahl der Programmiersprache: Go und Rust. Um die Entscheidung verständlich darzulegen, werden beide Sprachen kurz vorgestellt.

5.4.1. Go

Go, auch bekannt als Golang, wurde 2007 bei Google entwickelt und 2009 unter einer Open-Source-Lizenz veröffentlicht (Donovan & Kernighan, 2016). Ziel der Sprache war es, die Entwicklung moderner, skalierbarer Software durch eine Kombination aus Einfachheit, Effizienz und starken Tooling-Mechanismen zu erleichtern (Pike, 2023). Die Motivation hinter der Entwicklung war, dass Google komplexe Software-Projekte in C++ und Java hatte. Diese hatten lange Compile-Zeiten und boten für die oft eingesetzte Nebenläufigkeit nur unkomfortable Lösungen.

Go ist statisch typisiert, kompiliert und bietet integrierte Unterstützung für Nebenläufigkeit über sogenannte Goroutines, was sie besonders für netzwerk- und servernahe Anwendungen prädestiniert. Die Syntax ist bewusst minimalistisch gehalten und verzichtet auf komplexe Sprachfeatures wie klassische Vererbung.

Ein herausragendes Merkmal von Go ist das standardisierte Tooling: Das Kommandozeilenwerkzeug `go` übernimmt nahezu den gesamten Build- und Dependency-Management-Prozess. Darüber hinaus unterstützt das Tool auch Dokumentation, Testing, Formatting und Linting (Donovan & Kernighan, 2016), was in Programmiersprachen wie C, C++ oder Java über separates Tooling erledigt wird. Abhängigkeiten werden über eine zentrale Datei (go.mod) verwaltet.

Das Paket-Ökosystem ist gut etabliert, mit einer Vielzahl an qualitativ hochwertigen Open-Source-Bibliotheken für Bereiche wie Netzwerkkommunikation, Kryptographie und Webentwicklung (Avelino & contributors, 2025). Gleichzeitig legt das Go-Design Wert auf eine starke Standardbibliothek, sodass viele Anwendungsfälle ohne externe Abhängigkeiten realisierbar sind (Donovan & Kernighan, 2016).

5.4.2. Rust

Rust ist eine systemnahe, kompilierte Programmiersprache, die seit 2009 von Mozilla Research gesponsert und 2015 in Version 1.0 veröffentlicht wurde. Sie wurde mit dem Ziel entwickelt, Speichersicherheit, Nebenläufigkeit und Performance miteinander zu vereinen, ohne auf Garbage Collection angewiesen zu sein (Klabnik & Nichols, 2018).

Rust positioniert sich somit als moderne Alternative zu Sprachen wie C und C++, die in sicherheitskritischen Bereichen (Betriebssysteme, Embedded, Netzwerkinfrastruktur) weit verbreitet sind, aber bekannte Schwächen im Umgang mit Speicher und Nebenläufigkeit aufweisen.

Das zentrale Sprachmerkmal von Rust ist das innovative Speichermanagement ohne Garbage Collector (Klabnik & Nichols, 2018). Statt automatischer Speicherbereinigung basiert Rust auf einem Ownership-Modell mit Borrowing und Lifetimes. Der sogenannte Borrow-Checker analysiert zur Compile-Zeit, ob Speicher korrekt genutzt wird und verhindert damit viele typische Fehler wie:

- Use-after-free

- Null-Pointer-Dereferences
- Data Races

Dies führt zu hoher Sicherheit und sehr guter Laufzeit-Performanz, allerdings auch zu einer steileren Lernkurve: Man muss die Konzepte von Ownership und Lifetimes beherrschen, um effizient mit Rust zu arbeiten.

Rust verfügt mit `cargo` über ein modernes, integriertes Tooling-System für Paketverwaltung, Builds, Tests und Dokumentation. Außerdem gibt es ein aktives Ökosystem, in dem zahlreiche qualitativ hochwertige Bibliotheken verfügbar sind; von kryptografischen Werkzeugen über Webserver bis hin zu Low-Level-Systembibliotheken.

5.4.3. Vergleich der Softwareverwaltung

Beide Sprachen bieten gutes Tooling für moderne Softwareentwicklung (Donovan & Kernighan, 2016; Klabnik & Nichols, 2018). Sowohl Go als auch Rust verfügen über eine Konfigurations-Datei (`go.mod` bzw. `cargo.toml`), die deutlich leichtere Konfiguration als `CMakeLists.txt` oder `Makefile` für C oder C++ ermöglichen.

Beide erlauben die einfache Einbettung von Libraries aus dem Internet.

5.4.4. Vergleich des Speichermanagements

Da die entstehende Library in Zukunft auch auf Smartphones und Laptops mit begrenzten Speicherressourcen zum Einsatz kommen soll, ist eine Betrachtung des Speichermanagements und damit der zu erwartenden Speichereffizienz und Performance sinnvoll.

Rust bietet den Vorteil, dass die Binaries keine Laufzeitumgebung für Garbage-Collection mitliefern müssen, da das Speichermanagement manuell erfolgt (Klabnik & Nichols, 2018). Das manuelle Speichermanagement geht allerdings mit einem höheren initialen Entwicklungsaufwand einher.

Go hingegen setzt auf einen Garbage-Collector, was zwar für die Laufzeit-Performanz nachteilig ist, aber auch leichtere Software-Entwicklung erlaubt (Donovan & Kernighan, 2016).

5.4.5. Vergleich der Compile-Zeit

Um möglichst schnell Feedback zu Validität des Codes und Verhalten der zu entwickelnden Software bekommen zu können, ist ein schneller Compiler wünschenswert.

Rust ist bekannt dafür, einen langsamen Compiler zu haben, da der Borrow-Checker komplexe Analysen durchführen muss, um die Speichersicherheit der Programme zu verifizieren (The Rust Project Contributors, 2025b).

Go hingegen wurde mit dem Ziel entwickelt, einen möglichst schnellen Compiler zu ermöglichen (Pike, 2023).

5.4.6. Vergleich der Einbettung in mobile Apps

Da Securepoint mobile Apps entwickelt, ist das Tooling der Programmiersprachen zur Einbettung von nativen Libraries in die Apps für Android und iOS relevant.

Go verfolgt mit `gomobile` und `gobind` einen Ansatz, bei dem die API einer Library automatisch aus Go-Code generiert werden kann. Dieser ist dann im Code für Android- oder iOS-Apps aufrufbar (Google LLC, 2025a). Securepoint hat in einer App bereits Go mit diesem Tooling eingesetzt ([Firmen-interne Quelle], Securepoint GmbH, 2025).

Rust verfolgt zwar einen ähnlichen Ansatz, ist aber manueller in der Umsetzung: Neben C-Headern müssen relativ komplexe, in Rust geschriebene FFI-Bindings programmiert werden.

5.4.7. Entscheidung

Aufgrund der begrenzten Zeit für eine Bachelorarbeit, der Erfahrung im Team mit Go sowie der zu erwartenden Lernkurve habe ich entschieden, Go zu verwenden.

Im Bereich der Verwaltung von Software-Abhängigkeiten sind sich die beiden Sprachen recht ähnlich.

Das Speichermanagement von Rust ist zwar für High-Performance-Anforderungen wünschenswert, für die Nutzung in mobilen Apps hingegen würde der zusätzliche Aufwand im Speichermanagement höchstwahrscheinlich die Entwicklung verzögern. Die schnelleren Compile-Zeiten von Go sind zwar nicht strikt erforderlich, könnten aber den Entwicklungsworkflow positiv beeinflussen.

Nicht nur das Tooling von Go zur Einbettung in mobilen Apps ist für den potenziellen Einsatz hilfreich. Auch die Erfahrung im Betrieb mit Go sollte die Integration zusätzlich vereinfachen.

6. Implementierung der Software

In diesem Kapitel wird die technische Umsetzung des OpenVPN-Clients in Go beschrieben. Ziel der Implementierung ist es, die wesentlichen Funktionen der bestehenden Library schrittweise nachzubilden und dabei eine saubere und nachvollziehbare Struktur zu schaffen.

Den Anfang macht die Test-Strategie, die zeigt, wie bereits während der Entwicklung durch Unit Tests sichergestellt wird, dass zentrale Komponenten korrekt funktionieren. Darauf folgt die Beschreibung der CI-Pipeline, die den automatisierten Test- und Build-Prozess unterstützt.

Im weiteren Verlauf wird der Konfigurations-Parser vorgestellt, der für das Einlesen und Verarbeiten der OpenVPN-Konfigurationsdateien zuständig ist. Die Strukturierung der OpenVPN-Pakete bildet die Grundlage für die Kommunikation mit dem Server. Anschließend werden die einzelnen Schritte zur Verbindungsherstellung erläutert: der OpenVPN-Handshake, der TLS-Handshake sowie die Aushandlung der Sitzungsparameter. Nach erfolgreichem Verbindungsaufbau übernimmt der Data-Channel die Übertragung der Nutzdaten.

Außerdem werden die Probleme geschildert, die während der Entwicklung der einzelnen Komponenten aufgetreten sind.

6.1. Test-Strategie

Um eine zügige und fehlerarme Entwicklung des Prototypen zu erleichtern, habe ich Tests als zentralen Bestandteil der Entwicklung eingesetzt. Tests erlauben es, große Änderungen so vorzunehmen, dass dabei auftretende Probleme früh erkannt und behoben werden können (Myers et al., 2012).

Der aktuelle Fokus liegt auf Unit-Tests, die einzelne Komponenten und Funktionen isoliert testen. Diese Tests decken zentrale Logik wie Konfigurationsverarbeitung und Paket-Verarbeitung ab. Durch die frühe Integration von Unit-Tests kann die Korrektheit der Implementierung fortlaufend überprüft werden. Dabei ist es gerade in der Prototyping-Phase leicht, die Architektur gut testbar zu gestalten (Myers et al., 2012).

Langfristig ist geplant, die Testabdeckung durch End-to-End-Tests zu erweitern. Diese sollen sicherstellen, dass die gesamte Bibliothek korrekt mit einem OpenVPN-Server interagiert und alle Komponenten zuverlässig zusammenspielen. Während Unit Tests schnelle Rückmeldung bei Änderungen am Code liefern, bieten End-to-End-Tests zusätzlich eine ganzheitliche Absicherung gegen Integrationsfehler.

Dieses Vorgehen unterscheidet sich von der bisherigen Software-Lösung darin, dass bisher keine in Software definierten Tests eingesetzt wurden ([Firmen-interne Quelle], Bastian Kummer, 2018).

6.2. CI-Pipeline

Damit die Software-Tests möglichst oft, aber nicht nur manuell ausgeführt werden, habe ich eine CI-Pipeline definiert. Die CI-Pipeline erlaubt eine regelmäßige Kontrolle, da sie nach jedem „Git-Push“ (The Git Project Contributors, 2025) ausgeführt wird. In ihr wird zunächst eine Go-Build-Umgebung konfiguriert. Danach wird die interne Anforderung geprüft, ob alle Quelltext-Dateien mit einem Copyright-Hinweis beginnen. Dazu wird Googles `addlicense`-Tool genutzt. Im Anschluss wird mit dem Befehl `go build` geprüft, ob aus dem Code erfolgreich eine Binary kompiliert werden kann. Nach der Ausführung der in Software definierten Tests, deren Ergebnisse gespeichert werden, wird der Go-Formatter (`go fmt`) ausgeführt, der den Code auf einheitliche Einrückung und andere Formatierung prüft. Zum Schluss wird noch mit `golint` auf häufige Programmierfehler und die Einhaltung von Programmier-Konventionen geprüft. Listing 11 zeigt einen Auszug aus der `ci.yml`-Datei, die das beschriebene Verhalten konfiguriert:

yaml

```yaml
on: [ push ]

jobs:
  check_license-build-test-format-lint:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - name: Setup Go
        uses: actions/setup-go@v5
        with:
          go-version: "1.24.x"
      - name: Install addlicense
        run: go install github.com/google/addlicense@latest
      - name: Check license
        run: addlicense -check -l mit -s -c "Securepoint GmbH" .
      - name: Build project
        run: go build
      - name: Run project tests
        run: |
                set -o pipefail
                go test ./... -v -json | tee TestResults.json
      - name: Upload Go test results
        uses: actions/upload-artifact@v4
        with:
          name: Go-Test-Results
          path: TestResults.json
      - name: Format Go code
        run: gofmt -l .
      - name: Install golint
        run: go install golang.org/x/lint/golint@latest
      - name: Run golint
        run: golint -set_exit_status ./...
```

Listing 11: GitHub-Actions-Skript ci.yml *zur Qualitätsüberprüfung des Quellcodes*

6.3. Schnittstellenimplementierung

Die Implementierung der Schnittstelle war ein anfängliches Unterfangen, da es den Einstiegspunkt in die Software-Bibliothek definiert. Dieser Einstiegspunkt sowie die anderen Schnittstellen-Funktionen werden von der `main.go`-Datei genutzt. Die `main.go` definiert den Einstiegspunkt in das Programm und ist der in diesem Projekt genutzte Client für die definierte API und die dahinter

stehende Implementierung. Anders als bei der bisherigen Implementierung ist hier die gesamte Schnittstelle in einer einzigen Datei definiert (Listing 12).

`Connect()` nimmt eine OpenVPN-Konfiguration als String entgegen und parst daraus ein `struct Options`, das die unterstützten Konfigurationsoptionen enthält. Es wird geprüft, ob die geparsten Optionen valide sind und dann der Verbindungsaufbau begonnen. Schlägt ein Funktionsaufruf fehl, wird `false` zurückgeben, anderenfalls `true`. Das kann z. B. beim Verbindungsaufbau passieren.
`Disconnect()` stoppt den Client. Dabei werden Socket und TUN-Device geschlossen. Schlägt ein Funktionsaufruf fehl, wird `false` zurückgeben, anderenfalls `true`. Das kann z. B. bei dem Schließen eines Sockets auftreten.
`InBytes()` gibt die vom UDP- oder TCP-Socket empfangenen Bytes zurück. `OutBytes()` gibt die über den UDP- oder TCP-Socket gesendeten Bytes zurück.
`SetLogHandler()` setzt eine Funktion als Logger. Die Funktion bekommt ein `LogLevel` (`Debug`, `Info`, `Warn` oder `Error`), sowie eine Log-Nachricht als String übergeben. Sie muss nicht verwendet werden: Es gibt einen Logger, der auf `stdout` schreibt. Über den `LogLevel` kann man z. B. `Debug`-Nachrichten verwerfen. Um aber zu verhindern, dass es kein Logging gibt, kann hier kein „`nil`“-Logger gesetzt werden.

go

```go
func Connect(config string) bool {
  opt, err := options.ParseOptionsFromString(config)
  if err != nil {
    dbg.Errorf("Failed to parse options: %s", err)
    return false
  } else if !opt.IsValid() {
    dbg.Errorf("Invalid options")
    return false
  }
  return ovpnclient.Run(*opt)
}

func Disconnect() bool {
  return ovpnclient.Stop()
}

func InBytes() uint64 {
  return ovpnclient.InBytes()
}

func OutBytes() uint64 {
  return ovpnclient.OutBytes()
}

func SetLogHandler(handler func(level dbg.LogLevel, message string)) {
  if handler != nil {
    dbg.SetMsgHandler(handler)
  }
}
```

Listing 12: Verkürzter Auszug aus `libovpn-client.go`

6.4. Konfigurations-Parser

Der Konfigurations-Parser orientiert sich stark an der internen Originalimplementierung. Die Konfigurationsdatei wird zeilenweise gelesen; das erste Wort bestimmt, welche Funktion zum Parsen der restlichen verwendet wird. Eine Ausnahme bilden hier die Zertifikate und Keys, die mehrzeilig sein können, wenn z. B. statt `cert <cert>` das erste Wort ist. In dem Fall wird der Modus des Parsers

gewechselt und alle Zeilen von z. B. `<cert>` bis `</cert>` in einem String der geparsten Konfiguration gespeichert. Worauf ich hier verzichtet habe, ist die Validierung der Zertifikate oder das direkte Parsen in ein Go `x509.Certificate`. Zertifikatsdateien sind wahrscheinlich fast immer von Maschinen generiert, weshalb die Validierung nicht so dringlich ist. Die Validierung ließe sich aber ergänzen, wenn es denn gewünscht wird.

Ein wesentlicher Unterschied ist, dass Domains für `remote` (Server) aufgelöst und alle IP-Adressen sofort validiert werden. Dazu wird auf die Go-Standard-Library zurückgegriffen, die Funktionen für diesen Zweck bereitstellt. Das hat den Vorteil dass im restlichen Programm nur valide IP-Adressen behandelt werden müssen: Eine Fehlerquelle fällt weg.

Ein kleinerer Unterschied ist die Aufteilung in kleinere Komponenten, was sich in modernen Programmiersprachen wie Go anbietet: `options.go` enthält das `struct` und seine Methoden wie `IsValid`, während der Parser in `parse.go` definiert wird. Damit werden Funktionalitäten in kleinere, besser überschaubare Bereiche strukturiert.

Es gibt außerdem Software-Tests, die den Parser mit verschiedenen Konfigurationen auf korrekte Funktion überprüfen können. Komplexere Subkomponenten werden separat mit eigenen Unit-Tests geprüft, ansonsten wird der gesamte Parser als Einheit getestet. Dieses Vorgehen limitiert den Implementierungs- und Maintenance-Aufwand für die Tests, ohne die komplexeren und somit fehleranfälligen Funktionen in den Tests zu vernachlässigen.

6.5. Strukturierung der OpenVPN-Pakete

OpenVPN-Pakete haben je nach Verbindungszustand und Inhalt unterschiedliche Strukturen (vgl. Abschnitt 3). Um Pakete zu verarbeiten, werden sie anhand ihres Opcodes erkannt und zu einem `struct` deserialisiert. Zu verschickende Pakete halten ihre Daten in einem `struct`, auf das eine `Bytes`-Methode definiert ist. Diese Methode nutzt die Daten des `struct`, um daraus ein OpenVPN-Paket als Byte-Array zu serialisieren. Diese Bytes können dann über die Verbindung zum Server verschickt werden.

Listing 13 illustriert ein vom Client zu verschickendes Paket anhand des initialen `CONTROL_HARD_RESET_CLIENT_V2`-Pakets. Das `struct` (Zeilen 1 bis 3) hält die Daten, in diesem Fall nur die eigene Session-ID. Die `Bytes`-Methode serialisiert für das `HardResetClientV2Packet p` ein Byte-Array. Als erstes werden der Opcode und die Key-ID (hier 0, da es noch keine Keys gibt) in den Byte-Buffer geschrieben (Zeile 7). Danach werden die eigene Session-ID, die Anzahl von ACKs im Paket (hier immer 0), sowie die Packet-ID (hier immer 0) in den Buffer geschrieben (Zeilen 8 bis 10).

Es bietet sich auch hier wieder an, auf die Go-Standard-Library zurückzugreifen. Sie bietet eine Buffer-Implementierung, die der in der internen Originalimplementierung in C recht ähnlich ist. So lässt sie sich anstatt einer eigenen Implementierung verwenden, womit der Implementierungs- und Maintenance-Aufwand reduziert wird. Außerdem kann man davon ausgehen, dass die offizielle Implementierung von hoher Qualität und gut getestet ist, wodurch ihre Verwendung das Fehlerpotenzial senkt.

```go
type HardResetClientV2Packet struct {
  OwnSessionID SessionID
}

func (p HardResetClientV2Packet) Bytes() []byte {
  buf := bytes.Buffer{}
  buf.WriteByte(OpcodeKeyIDByte(byte(PControlHardResetClientV2), 0))
  buf.Write(p.OwnSessionID.Bytes())
  buf.WriteByte(0)              // 1 byte for acked packet IDs length
  buf.Write([]byte{0, 0, 0, 0}) // 4 bytes for packet ID
  return buf.Bytes()
}
```

Listing 13: Definition eines OpenVPN-Hard-Reset-Paket vom Client in Go

Listing 14 zeigt die Implementierung der Antwort auf einen Client-Hard-Reset. Das vom Client empfangene `CONTROL_HARD_RESET_SERVER_V2`-Paket muss zunächst geparst werden. Das wird in `ParseHardResetServerV2Packet` implementiert. Die Funktion deserialisiert einen `Buffer`, indem es Bytes nach dem erwarteten Aufbau liest und interpretiert. Tritt ein Fehler auf, wird statt eines `HardResetServerV2Packet` nur ein `error` zurückgegeben. Die Rückgabe in dem Format eines Tupels aus erwarteter Rückgabe und Error ist dabei typisch für Go. Darüber wird explizites Error-Handling erzwungen, wie man in Zeile 14 sehen kann. Dort führt ein erkannter Fehler sofort zur Beendigung der Funktion und die aufrufende Funktion muss den aufgetretenen Fehler behandeln. Tritt kein Fehler auf, wird am Ende der Funktion ein Pointer auf das in Zeilen 1 bis 7 definierte `struct` zurückgegeben. Dieses `struct` kann dann genutzt werden. In diesem Fall wird es als Teil des OpenVPN-Handshakes acknowledged (vgl. Abbildung 6).

```go
type HardResetServerV2Packet struct {
  KeyID           uint8
  RemoteSessionID *SessionID
  AckedPacketIDs  []PacketID
  OwnSessionID    *SessionID
  RemotePacketID  PacketID
}

func ParseHardResetServerV2Packet(packet []byte) (*HardResetServerV2Packet,
  error) {
  buffer := bytes.NewBuffer(packet)
  // keyID und remoteSessionID dem Buffer entnehmen...
  ackedPacketIDsLength, err := buffer.ReadByte()
  if err != nil {
    return nil, err
  }
  // ackedPacketIDs, ownSessionID und remotePacketID dem Buffer entnehmen
  parsed := &HardResetServerV2Packet{
    // struct wird mit den geparsten Werten befüllt
  }
  return parsed, nil
}
```

Listing 14: Gekürzte Definition eines OpenVPN-Hard-Reset-Paket vom Server in Go

6.6. OpenVPN-Handshake

Der OpenVPN-Handshake wurde in der internen Originalimplementierung mit einer State-Machine abgebildet. Das hat die Nachvollziehbarkeit im Code leider eher verschlechtert, da Funktionsaufrufe über Zustände und Handler-Matrix (vgl. Listing 6, Listing 7) und nicht explizit erfolgen. Deshalb hat die Go-Implementierung einen anderen Ansatz verfolgt: Statt über Zustände wurde der Kontrollfluss explizit definiert. Jeder Funktionsaufruf folgt auf den vorangegangen, statt in einem Main-Loop über das weitere Vorgehen zu entscheiden. Dieser Ansatz ist meines Erachtens nach besser lesbar. Es besteht die Gefahr, dass er nicht so gut skaliert wie der Ansatz mit Funktionsmatrix. Für einen Prototypen ist diese Architektur aber ausreichend.

Listing 15 zeigt den OpenVPN-Handshake in dieser expliziten Implementierung. Der Handshake wird vom Client initiiert (Zeile 2), wonach auf eine Antwort vom Server gewartet wird (Zeilen 3 und 4). Bei Empfang einer Antwort wird das Paket geparst (Zeile 8), seine Session-ID gespeichert und die Packet-ID in einer Liste ergänzt (Zeilen 12 und 13). Danach wird ein Acknowledgement-Paket für die Server-Antwort verschickt (Zeile 14).

```go
1  func (s *Session) Handshake() error {
2    s.SendHardReset()
3    packet := make([]byte, 2048)
4    _, err := s.ReaderWriter.Read(packet)
5    if err != nil {
6      return err
7    }
8    parsed, err := packets.ParseHardResetServerV2Packet(packet)
9    if err != nil {
10     return err
11   }
12   s.RemoteSessionID = *parsed.RemoteSessionID
13   s.AddUnacked(parsed.RemotePacketID)
14   err = s.SendAck()
15   if err != nil {
16     return err
17   }
18   return nil
19 }
```

Listing 15: Vereinfachte Darstellung der OpenVPN-Handshake-Implementierung in Go

Die `Session`, die den Handshake durchführt, enthält dabei eine `conn.OVPNConn` (Listing 16), die `net.Conn` aus der Go-Standard-Library erweitert. `net.Conn` stellt Lese- und Schreibmethoden für unter anderem UDP- und TCP-Sockets in Go bereit. `conn.OVPNConn` kann auf einer UDP- oder TCP-Verbindung aufbauen. Es abstrahiert das OpenVPN-Framing für TCP-Verbindungen, indem es Pakete mit einem Längen-Präfix versieht (vgl. Tabelle 1). Außerdem erhebt eine `conn.OVPNConn` die Menge der Bytes, die empfangen und gesendet werden.

```go
1 type OVPNConn interface {
2   net.Conn
3   GetInBytes() uint64
4   GetOutBytes() uint64
5 }
```

Listing 16: Definition der `conn.OVPNConn`*, die für UDP und TCP implementiert wird*

6.7. TLS-Handshake

Bei dem Aufbau der TLS-Session, über die der Control-Channel die Verbindungsparameter austauschen sollte, kam es leider zu Problemen bei der Implementierung.

Zunächst war die Konstruktion der Verbindung eine Herausforderung, die sich aber bewältigen ließ. Das Problem bestand darin, dass der TLS-Handshake im OpenVPN-Protokoll in OpenVPN-Pakete mit einem entsprechenden Header für Opcodes und andere Informationen gekapselt werden muss. Dazu habe ich eine eigene Go `net.Conn` implementiert, die dieses Verhalten unterstützt: `CtrlChannelConn` in Listing 17 dient zur Übertragung von `CONTROL_V1`-Paketen, und unterstützt das Empfangen und Senden von Acknowledgements in den Paketen. Das `struct CtrlChannelConn` implementiert eine `conn.OVPNConn`, die eine Go `net.Conn` implementiert (vgl. Listing 16). Außerdem wird der Traffic über die `CtrlChannelConn` in Control-Channel-Pakete eingekapselt. Dazu wurden die Methoden `Read` und `Write` überschrieben.

go

```go
type CtrlChannelConn struct {
  conn.OVPNConn
  session *Session
}

func (c *CtrlChannelConn) Read(b []byte) (int, error) {
  n, err := c.session.ReaderWriter.Read(b)
  if packets.OpcodeFromByte(b[0]).IsAck() {
    return n, nil
  }
  packet, err := packets.ParseUnauthenticatedControlPacket(b[:n])
  if err != nil {
    return 0, err
  }
  if packets.OpcodeFromByte(b[0]).IsControl() { // ACKs MUST NOT be acknowledged
    c.session.AddUnacked(packet.OwnPacketID)
    err = c.session.SendAck()
  }
  copied := copy(b, packet.Payload)
  return copied, err
}

func (c *CtrlChannelConn) Write(b []byte) (int, error) {
  packet := packets.NewUnauthenticatedControlPacket(
    0, // keyID
    c.session.OwnSessionID,
    c.session.PopUnacked(),
    c.session.RemoteSessionID,
    c.session.LastSentPacketID+1,
    b,
  )
  c.session.LastSentPacketID++
  return c.session.OVPNConn.Write(packet.Bytes())
}
```

Listing 17: Auszug der Implementierung `CtrlChannelConn` *in Go*

Dieser Datentyp sollte dann als `net.Conn` von der TLS-Implementierung der Go-Standard-Library zum Aufbau einer TLS-Verbindung genutzt werden können:
`tls.Client(ctrlChannelConn, tlsConfig)`.

Leider hat der TLS-Handshake aber weder über TCP noch über UDP erfolgreich funktioniert. Es gibt stattdessen die Fehler „`error: failed to authenticate over TLS: local error: tls: bad record MAC`“ und „`failed to authenticate over TLS: unexpected EOF`“, anderenfalls hängt sich die Verbindung beim Lesen auf.

Ich habe die Vermutung, dass nicht alles Relevante zum TLS-Handshake in Schwabe (2016) steht. Der Versuch, die Paket-Größe zu limitieren, hat das Problem nicht behoben. Dieser Workaround wurde in der internen Originalimplementierung verwendet ([Firmen-interne Quelle], Bastian Kummer, 2018). Meine Vermutung ist, dass der TLS-Handshake nicht einfach über den `tls.Client` der Go-Standard-Library ausgeführt werden kann, auch wenn der Handshake in Control-Channel-Paketen gekapselt ist. Eventuell könnte uTLS Abhilfe schaffen. uTLS ist ein Fork des `crypto/tls`-Pakets der Go-Standard-Library. Es bietet tieferen Zugriff auf die Komponenten, sodass man damit vielleicht die TLS-Records in den Paketen besser kontrollieren kann (Refraction Networking, 2025). uTLS findet auch bei dem in Go geschriebenen OpenVPN-Client des Open Observatory of Network Interference (2025) Verwendung. Dann könnte man das Verhalten im Kapitel „Control message framing“ von Schwabe (2016) exakt abbilden.

Aufgrund dieser Probleme habe ich die darauf folgenden Komponenten entwickelt, auch wenn sie sich ohne Control-Channel nicht testen ließen.

6.8. Aushandlung der Sitzungsparameter

Die Modellierung der Authentisierung (Listing 18) mit Nutzername und Passwort sowie die Aushandlung der Sitzungsparameter entspricht der Definition in Tabelle 4. Der Options-String ist dabei aus der Originalimplementierung übernommen. Hier gibt es wieder die Aufteilung in Datenhaltung in einem `struct` und eine `Bytes`-Methode, mit der die Nachricht serialisiert werden kann. Aus den Strings werden in `EncodeStringBytes` Byte-Arrays generiert, die in den ersten zwei Byte ihre Länge enthalten und mit einem Null-Byte enden.

go

```go
type AuthMessage struct {
  KeyRandom KeyRandom
  Username  string
  Password  string
  PeerInfo  string
}

func (authMsg *AuthMessage) Bytes() []byte {
  buf := bytes.Buffer{}
  buf.Write(make([]byte, 4))
  buf.WriteByte(0x02) // "key_method2_write"
  buf.Write(authMsg.KeyRandom.Bytes())
  buf.Write(packets.EncodeStringBytes("V0 UNDEF")) // options string
  buf.Write(packets.EncodeStringBytes(authMsg.Username))
  buf.Write(packets.EncodeStringBytes(authMsg.Password))
  buf.Write(packets.EncodeStringBytes(authMsg.PeerInfo))
  return buf.Bytes()
}
```

Listing 18: Auszug aus `authmessage.go`

Die Peer Info wird dabei analog zur Originalimplementierung in Go erzeugt (Listing 19). Der einzige Unterschied ist in `platform.GetPlatform()`. Während die Originalimplementierung den C-Präprozessor direkt in der Funktion nutzt, wird die Abfrage der Plattform bei Go in plattformspezifischen Dateien hinterlegt. Damit lässt sich die Plattform-Unterstützung nach und

nach ausbauen, die unterschiedlichen Implementierungen sind aber in einzelne Dateien getrennt. Diese werden vom Go-Compiler nach dem Namensschema „`name_plattform.go`“ erwartet, in diesem Beispiel also `platform_android.go`, `platform_darwin.go` (iOS und macOS), `platform_freebsd.go`, `platform_linux.go` und `platform_windows.go`.

```go
func PeerInfo(ciphers string) string {
  return fmt.Sprintf(
    "IV_PLAT=%s\nIV_VER=0.1\nIV_PROD=sp-ovpn\nIV_NCP=2\nIV_CIPHERS=%s\n",
    platform.GetPlatform(),
    ciphers,
  )
}
```

Listing 19: Auszug aus `peerinfo.go`

Das Parsen der Antwort auf eine `AuthMessage` erfolgt nach dem Schema wie es in Listing 14 präsentiert ist, nur eben für den Inhalt von Tabelle 4. Es wird der darin vom Server genannte Verschlüsselungsalgorithmus für den Data-Channel genutzt.

6.9. Data-Channel

Der Data-Channel modelliert das Senden und den Empfang der Pakete. Er ver- und entpackt sie und ver- und entschlüsselt sie. Da die Kapselung mit `Bytes`-Methoden und `Parse`-Funktionen bereits erörtert wurden (vgl. Listing 13 und Listing 14), liegt der Fokus im Folgenden auf der Verschlüsselung und der Interaktion des Data-Channels mit dem Betriebssystem.

Um eine möglichst einheitliche Nutzung der verschiedenen Verschlüsselungsalgorithmen zu ermöglichen, habe ich alle unterstützten Algorithmen mit dem folgenden `struct` dargestellt (Listing 20). Das `struct` besteht dabei aus dem Namen, unter dem der Verschlüsselungsalgorithmus im OpenVPN-Protokoll bekannt ist (vgl. Tabelle 5), der `KeySize` und `NonceSize`, die zur Ver- und Entschlüsselung mit entweder `NewAEAD` oder `NewBlock` dienen. Das ist abhängig von der Art des Algorithmus, der in `IsAEAD` definiert wird.

```go
type CipherSpec struct {
  Name      string
  KeySize   int
  NonceSize int
  IsAEAD    bool
  NewAEAD   func(key []byte) (cipher.AEAD, error)  // only for AEAD modes
  NewBlock  func(key []byte) (cipher.Block, error) // only for non-AEAD modes
}
```

Listing 20: Auszug aus `ciphers.go`

Damit ließen sich die empfohlenen und optionalen Algorithmen aus Tabelle 5 in einer Map abbilden. Um die Algorithmen einzusetzen, sind Funktionen mit den folgenden Funktionssignaturen definiert (Listing 21). Die ersten beiden Zeilen zeigen dabei die Funktionssignaturen der Chiffren, die separate Authentifizierung benötigen (AES-CBC), während die unteren beiden Funktionen für die AEAD-Chiffren verwendet werden. Diese Funktionen erlauben zusätzlich „Additional Authenticated Data“ (hier `aad`) und benötigen den „Authentication Tag“ (hier `tag`), der für zum Nachweis der Authentizität und der Integrität benötigt wird. Die Signaturen der Funktionen sind gut vergleichbar mit denen aus Listing 8:

```go
func EncryptBlock(block cipher.Block, iv, input []byte) ([]byte, error)
func DecryptBlock(block cipher.Block, iv, input []byte) ([]byte, error)

func EncryptAEAD(aead cipher.AEAD, iv, plain, aad []byte) (cipher, tag []byte, error)
func DecryptAEAD(aead cipher.AEAD, iv, cipher, aad, tag []byte) ([]byte, error)
```

Listing 21: Funktionssignaturen der Verschlüsselungsfunktionen in Go

Bisher wurden bei der Implementierung Control-Channel-Pakete behandelt, oder Data-Channel-Pakete in ihrer Verarbeitung erklärt. Sockets und `net.Conn` aus der Go-Standard-Library wurden erwähnt, nicht aber die TUN-Devices, die Herkunft oder Ziel der IP-Pakete auf Client-Seite sind (vgl. Abbildung 5, Abbildung 7). Sie bilden die Systemgrenze zwischen Betriebssystem und OpenVPN-Client-Software. Um TUN-Devices in Go zu verwenden, kann man die Library `water` nutzen. `water` verwaltet TUN- und TAP-Devices für Linux, macOS und Windows. Der Vorteil dieser Library ist, dass sie plattformabhängigen Code über eine einheitliche Programmierschnittstelle bereitstellt (Gao, 2020). Die interne Library in C hat dies hingegen selbst implementieren müssen. In Listing 22 ist eine verkürzte Version der TUN-Device-Konfiguration und die dazugehörige `Open`-Methode dargestellt, die damit ein TUN-Device öffnet. Die Funktionen `setMtu`, `setAddresses` und `setRoutes` setzen die entsprechenden Werte für das TUN-Device mithilfe von Shell-Commands wie `route` und `ifconfig`. Für das in `Open` ungenutzte Feld `DNSServers` fehlt einfach noch die entsprechende Konfigurationsfunktion. Da „`interface`“ in Go ein Keyword der Programmiersprache ist wird das `water.Interface` „`ifce`“ genannt.

```go
type Config struct {
  Mtu           uint16
  Addresses     []netip.Prefix
  Routes        []InetRoute
  DNSServers    []netip.Addr
}

func (cfg *Config) Open() (*water.Interface, error) {
  ifce, err := water.New(water.Config{
    DeviceType:             water.TUN,
    PlatformSpecificParams: water.PlatformSpecificParams{},
  })
  if err != nil { return nil, err }
  err = setMtu(ifce.Name(), cfg.Mtu)
  if err != nil { return nil, err }
  err = setAddresses(ifce.Name(), cfg.Addresses)
  if err != nil { return nil, err }
  err = setRoutes(cfg.Routes)
  if err != nil { return nil, err }
  return ifce, nil
}
```

Listing 22: Auszug aus `tun.go`

Das `water.Interface` bietet `Read`- und `Write`-Methoden, wie eine `net.Conn`, um Pakete zu empfangen und zu senden. Die weitere Konfiguration von TUN-Devices unterstützt die `water`-Bibliothek nicht. Die Konfiguration erfolgt daher über Befehle wie „`ifconfig`“ und „`route`“. Diese Befehle sind nicht plattformübergreifend einheitlich. Der `route`-Befehl auf Linux funktioniert zum Beispiel anders als auf macOS: Während Linux IPv4 und IPv6 dort mit „`-4`“ und „`-6`“ differenziert, wird dies unter macOS mit „`-inet`“ und „`-inet6`“ erreicht. Diese Unterschiede wurden daher

plattformspezifisch z. B. in `tun_linux.go` und `tun_darwin.go` implementiert. Dadurch bleibt die Software plattformübergreifend einsetzbar.

Anders als die Desktop-Betriebssysteme erzeugen und verwalten Android und iOS ihre TUN-Devices selbst. Anwendungen haben über Programmierschnittstellen indirekten Zugriff auf sie. Einerseits ist das sicherer, da es keinen Root-Zugriff oder Rechte zur Erzeugung eines TUN-Devices für die Anwendung erfordert, andererseits müssen Clients für diese Plattformen besondere, plattformspezifische Lösungen implementieren. Zur Vollendung der Implementierung dieser OpenVPN-Library wäre dieses Feature für Securepoint wichtig, es übersteigt aber den Rahmen dieser Bachelorarbeit.

7. Vergleich der Implementierungen

Zur Bewertung des im Rahmen dieser Arbeit entwickelten Go-Prototyps muss ein strukturierter Vergleich mit der C-basierten OpenVPN-Client-Bibliothek vorgenommen werden. Ziel ist es, Stärken und Schwächen der Implementierungen, insbesondere im Hinblick auf Wartbarkeit und Leistung herauszuarbeiten. Da der Go-Prototyp zum Zeitpunkt der Analyse noch nicht vollständig implementiert ist, liegt der Schwerpunkt nicht auf einem Funktionsvergleich oder auf Performance-Messungen, sondern auf qualitativen Aspekten der Softwareentwicklung.

7.1. Erfüllung der Anforderungen

Die bestehende Implementierung erfüllt alle primären funktionalen Anforderungen. Auch die optionalen Anforderungen des „Soft-Reset“ (Funktionale Anforderung 6) und „TLS-Auth“ Funktionale Anforderung 7 sind erfüllt. Allein „TLS-Crypt“ (Funktionale Anforderung 8) ist nicht implementiert.

Der Prototyp hingegen ist unvollständig. Es können zwar Konfigurationen und Konfigurations-Dateien eingelesen werden (Funktionale Anforderung 1), aber schon der Handshake ist nur unvollständig implementiert (Funktionale Anforderung 2). Das hat auch die Datenübertragung über den Client beeinflusst (Funktionale Anforderung 3): Teile der Implementierung sind fertig, sie sind aber nicht vollständig, da dort die Integration ohne Funktionale Anforderung 2 nicht möglich war. Die Umsetzung der plattformspezifischen Features ist zwar nicht vollständig, aber die Infrastruktur ist implementiert (Funktionale Anforderung 4). Die Konfiguration der TUN-Devices ist zumindest für Linux und macOS möglich (Funktionale Anforderung 5). Die Schnittstellenimplementierung wurde wie in Abschnitt 5.2 beschrieben umgesetzt (vgl. Listing 12).

7.2. Qualitative Analyse

Ein sinnvoller Vergleich von Software erfordert mehr als einen reinen Funktionsabgleich. Da der Prototyp nicht fertiggestellt werden konnte, sollen zunächst Kriterien wie Testbarkeit sowie die Komplexität und Struktur des Codes berücksichtigt werden. Diese Bewertung erfolgt hauptsächlich anhand qualitativer Einschätzungen.

7.2.1. Build-Tooling

Sowohl C als auch Go verfügen über schnelle Compiler, es gibt aber dennoch Qualitätsunterschiede in den Standard-Toolchains der beiden Sprachen (Pike, 2023; The Rust Project Contributors, 2025b). Diese Unterschiede möchte ich im Folgenden erörtern.

Go bietet simples und modernes Build-Tooling, was die Verwendung von externen Abhängigkeiten für beispielsweise Kryptografie oder die Verwendung von TUN-Devices sehr einfach ermöglicht. Es sind außerdem der Formatter mit `go fmt`, ein Linter mit `go vet` und eine Testumgebung mit `go test` direkt in dem Compiler integriert, wodurch ihre händische Installation und Konfiguration entfällt.

Auch ist die `go.mod` (Listing 23) als Projektverwaltungsdatei übersichtlich, explizit und funktioniert unabhängig vom Betriebssystem, anders als die `Makefile` der C-Bibliothek (vgl. Listing 10):

```go
module golang-vpn

go 1.24

require (
  github.com/songgao/water v0.0.0-20200317203138-2b4b6d7c09d8
  golang.org/x/crypto v0.40.0
)

require golang.org/x/sys v0.34.0 // indirect
```

Listing 23: Komplette `go-mod`*-Datei des Prototyps*

Die Abhängigkeiten lassen sich über die Kommandozeile mit z. B. `go get github.com/songgao/water` hinzufügen und mit `go get -u all` auf die neueste Version aktualisieren. Hingegen muss OpenSSL für die C-Bibliothek entweder auf Desktop-Betriebssystemen vorinstalliert sein oder auf mobilen Plattformen separat kompiliert und in die Apps integriert werden. Dadurch ist jede Aktualisierung der OpenSSL-Abhängigkeit in den Apps ein manueller Prozess, der Minuten und bis zu einer Stunde in Anspruch nimmt, je nach dem, wie schnell die OpenSSL-Library kompiliert wird. Während man aber bei der C-Bibliothek noch OpenSSL-Shared-Libraries zum Betrieb benötigt, konstruiert der Go-Compiler eine einzelne, alles enthaltende Binary.

Bei der Einbettung der neuen Go-Implementierung in mobile Apps wäre zu erwarten, dass dies leichter ist als mit der bestehenden Lösung: Die Anbindung der Bibliothek an Apps für Android und iOS wird von `gomobile` fast komplett automatisiert (Google LLC, 2025a) und wurde bei Securepoint bereits erfolgreich eingesetzt ([Firmen-interne Quelle], Securepoint GmbH, 2025).

7.2.2. Lines of Code

Die Anzahl an Code-Zeilen eines Projekts sind zwar nicht unbedingt eine verlässliche Metrik, aber dafür sehr leicht zu erheben. Gerade bei unterschiedlichen Programmiersprachen ist die Vergleichbarkeit aber nicht unbedingt gegeben. Trotzdem kann die Metrik Aufschluss über die Komplexität und damit die Wartbarkeit der Software geben, wenn der Unterschied besonders groß ist. Dabei ist eine kleine Anzahl Zeilen erstrebenswert, weil tendenziell weniger Fehler auftreten können. Um die in Abbildung 8 gezeigten Werte zu ermitteln, wurde das Tool `cloc` verwendet, das mit der Zählung von Code-Zeilen, Leerzeilen und Kommentarzeilen eine um diese Werte bereinigte Analyse der Lines of Code ermöglicht (Danial, 2025).

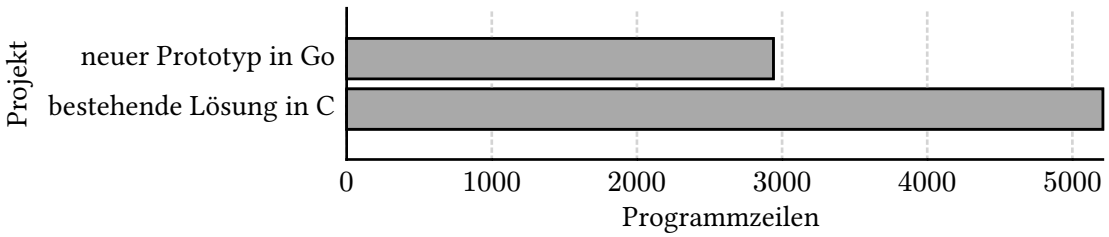

Abbildung 8: Lines of Code in den Projekten, bereinigt um Leer- und Kommentarzeilen

Der Prototyp ist noch nicht fertig implementiert, weshalb er bei dieser Messung einen Vorteil hat. Deswegen hat er mit 2940 Zeilen eine geringere Anzahl an Zeilen als die bestehende Lösung mit 5210 Zeilen. Erst wenn beide Implementierungen dieselben Features bieten, ist der Vergleich aussagekräftiger. Der Trend zum jetzigen Zustand ist aber vielversprechend und spricht eher für den Prototypen. Ich würde davon ausgehen, dass noch maximal 2000 Zeilen für die Unterstützung aller

Features der Originalimplementierung erforderlich sein werden. Der Umfang der bestehenden Software-Lösung ist im Vergleich also auch nicht schlecht.

7.2.3. Vorteile der Go-Standard-Library

Ein Punkt, der für den Prototypen spricht, ist die Tatsache, dass deutlich mehr Komponenten von Software-Abhängigkeiten wie der Go-Standard-Library implementiert sind. So wurden `buffer.c` und `buffer.h` nicht zu Go portiert, sondern `bytes.Buffer` aus der Standard-Bibliothek verwendet. Die interne Implementierung in der C-Bibliothek hingegen ist durch Instabilität negativ aufgefallen (vgl. Listing 1 und [Firmen-interne Quelle] (2024)). Außerdem verfügt die Go-Standard-Library über Funktionen, die Daten wie z. B. IP-Adressen und X.509-Zertifikate validieren können, sowie über leicht nutzbare APIs für Netzwerkverbindungen, Textverarbeitung und Kryptografie (Donovan & Kernighan, 2016).

7.2.4. Abstraktionen

Die Methoden in Listing 13 und Listing 14 strukturieren den Code im Prototypen für die verschiedenen OpenVPN-Paket-Typen, wohingegen die bestehende Implementierung mit `ovpn_link_write`, `ovpn_link_read` und `ovpn_parse_frame` alle unterschiedlichen Paket-Typen über die selben Funktionen serialisieren und deserialisieren. Diese Entscheidung resultiert in langen, unleserlichen Funktionen, die die Wartbarkeit verschlechtern (Khan et al., 2006).

7.2.5. Software-Tests

Ein klarer Vorteil für den Prototypen ist die Existenz von Software-Tests. Sie erlauben es, automatisiert Fehler in Programmen zu finden (Myers et al., 2012). Aktuell sind 105 verschiedene Test-Cases definiert, der Großteil für den Konfigurations-Parser und die Serialisierung und Deserialisierung der OpenVPN-Pakete. Die Code-Coverage, also der Anteil am Programm-Code, der bei Tests ausgeführt wird, liegt nach `go test ./... -cover` bei 74.6% für De-/Serialisierung der Pakete, bei 43.8% für den Konfigurations-Parser und für Kryptografie bei 56.1%. Manche Komponenten haben gar keine Tests. Allerdings ist Code-Coverage unter Kritik, da die Zahlen oft nicht aussagekräftig sind und sich manipulieren lassen (Borenkraout, 2024).
Das Testen von kritischen Komponenten, wie der De-/Serialisierung von Paketen, hilft bei der Erkennung von z. B. Regressionen (Myers et al., 2012). Die Originalimplementierung verfügt nicht über Software-Tests, wodurch Fehler leichter unentdeckt bleiben können und dann erst bei Ausführung auftreten, beispielsweise auf Kundengeräten.

7.3. Größe der Binary

Da die Libraries unter anderem auf Smartphones eingesetzt werden sollen, sollten sie keine zu großen Binaries produzieren.

Die Binary-Dateien der beiden Clients sind nicht direkt vergleichbar, da Go eine einzelne Datei erzeugt, während die C-Bibliothek externe Abhängigkeiten nutzt. `libopenvpn-lib.so` selbst ist 287.040 Byte groß, aber zusammen mit C-Standard-Library `libc++_shared.so` (1.794.776 Byte), sowie `libcrypto.so` (4.260.232 Byte) und `libssl.so` (687.520 Byte) von OpenSSL kommt man auf ca. 7 Megabyte. Die Library des Prototypen, `libgolang-vpn.so`, ist ca. 5 Megabyte groß (4.795.490 Byte). Damit ist die Library des Prototypen tatsächlich kleiner als die der Originalimplementierung, wenn man ihre Abhängigkeiten in der Berechnung berücksichtigt.
Ich finde dieses Ergebnis recht überraschend, da die Shared Library des Prototypen eine Runtime für Garbage-Collection enthalten muss. Andererseits kann Go eventuell ungenutzte Funktionen der Abhängigkeiten entfernen, während die die Implementierung in C die vollständigen Shared-Libraries der Abhängigkeiten benötigt. Beide Libraries haben vertretbare Größen, die z. B. das Limit des Google Play Store von 200 Megabyte nicht übersteigen (Google LLC, 2025c).

7.4. Benchmark

Auch wenn mein eigener Prototyp noch nicht einsatzbereit ist, soll in Vorbereitung auf seine Fertigstellung der Durchsatz anderer Clients gemessen werden. Es wurden der offizielle OpenVPN-Client v2.6.14 (OpenVPN contributors, 2025) und der intern entwickelte Client ([Firmen-interne Quelle], Bastian Kummer, 2018) getestet.

Die Testumgebung bestand aus einer Ubuntu-Linux Virtual Machine (VM), die sowohl als OpenVPN- als auch als `iperf3`-Server diente. Der jeweilige OpenVPN-Client lief auf dem Host-Betriebssystem und verband sich zum OpenVPN-Server auf der VM. Die OpenVPN-Verbindung war dabei über UDP und mit `AES-256-GCM` konfiguriert. Über diese Verbindung wurden dann jeweils drei `iperf3`-Tests auf dem Host-Betriebssystem gestartet. Zusätzlich wurden zwei Baselines erfasst: Eine direkte Verbindung zu `localhost` und eine direkte Verbindung zur VM ohne VPN.

Abbildung 9 zeigt den `iperf3`-Durchsatz über alle vier Verbindungen. Die direkte Verbindung zum `localhost` (Baseline) erreicht 106 Gbit/s im Durchschnitt von drei Tests. Das ist das praktisch erreichbare Maximum auf dem System. Die direkte Verbindung zur VM ohne VPN hat einen deutlich geringeren durchschnittlichen Durchsatz von 3,99 Gbit/s. Allein das virtualisierte VM-Netzwerk verringert den für die VPN-Clients erreichbaren Durchsatz also enorm. Beide Clients erreichen im Durchschnitt über drei `iperf3`-Tests ca. 375 Mbit/s.

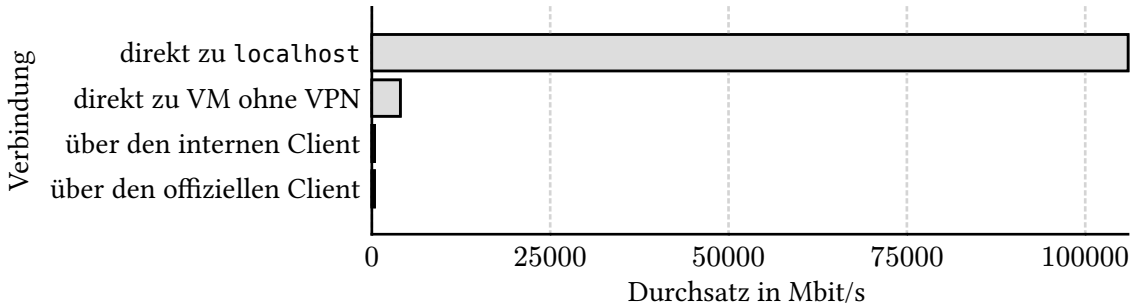

Abbildung 9: `iperf3`*-Benchmark der OpenVPN-Clients im Vergleich mit Baseline-Messungen*

Da in Abbildung 9 die Benchmark-Ergebnisse der OpenVPN-Clients nicht so gut erkennbar sind, folgt noch eine Darstellung ohne die Baselines. Man kann in Abbildung 10 sehen, dass beide OpenVPN-Clients fast gleich abschneiden, auch wenn die interne Implementierung einen minimalen Vorsprung (379 Mbit/s) vor dem offiziellen Client hat (373 Mbit/s). Das sind zwar im Vergleich mit der Baseline zur VM ohne VPN weniger als 10 %, aber immer noch fast das vierfache der Median-Downloadgeschwindigkeit deutscher Festnetzanschlüsse von 101 Mbit/s (Statista GmbH, 2025).

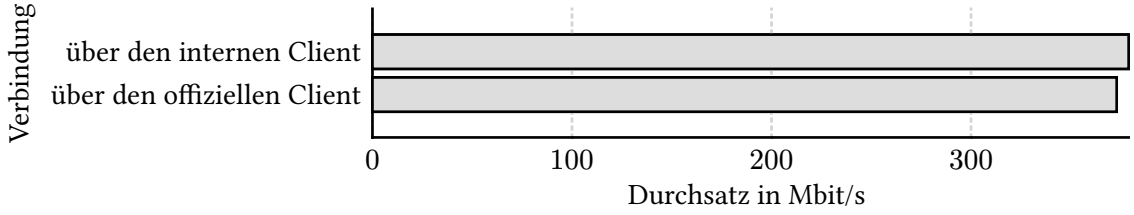

Abbildung 10: `iperf3`*-Benchmark der OpenVPN-Clients ohne Baseline-Messungen*

Bei einem Benchmark des Prototypen in Go wäre zu erwarten, dass er etwas schlechter performen würde als die beiden in C geschriebenen Clients, da Go-Binaries eine Runtime für Garbage-Collection enthalten, die einen kleinen Performance-Overhead erzeugt.

8. Fazit

Ausgangspunkt der Arbeit war die Analyse der bestehenden Situation bei Securepoint, bei der Abstürze der bestehenden OpenVPN-Bibliothek durch Speicherzugriffsverletzungen, suboptimale Performance auf bestimmten Plattformen (z. B. iOS), sowie ein hoher Integrationsaufwand und eine unkomfortable Build-Konfiguration zu wiederkehrenden Problemen führten. Diese Herausforderungen motivierten die Suche nach einer Alternative zur bestehenden C-basierten Lösung.

Im Rahmen dieser Arbeit wurde das OpenVPN-Protokoll detailliert technisch analysiert. Dabei wurden sowohl die Struktur einzelner OpenVPN-Pakete als auch der Aufbau von Sessions umfassend dokumentiert. Im Anschluss wurde die bestehende C-Implementierung untersucht. Dabei wurde deutlich, dass einige technische Altlasten (z. B. eigene Buffer-Logik und fehlende Testabdeckung) die Wartbarkeit und Erweiterbarkeit erschweren können. Insbesondere der in C notwendige manuelle Umgang mit Speicher und die unzureichende Isolation von Plattform-spezifischem Code stellen erhebliche Risiken dar.

Basierend auf einem fundierten Vergleich moderner Programmiersprachen wurde Go für die Reimplementierung ausgewählt. Die Entscheidung begründet sich in der Kombination aus Speichersicherheit, einfacher Cross-Plattform-Kompatibilität, guter Tooling-Unterstützung und einer insgesamt reduzierten Komplexität gegenüber Alternativen wie Rust.

Die Implementierung des Prototyps fokussierte sich auf zentrale Teile des Protokolls, etwa die Paketstruktur, TLS-Handshake, den Aufbau von Sessions und die Verarbeitung der Data-Channel-Nachrichten. Die Software wurde mit besonderem Fokus auf Testbarkeit und Trennung von plattformspezifischen Abhängigkeiten entworfen. Der Einsatz automatisierter Tests wurde dabei von Anfang an mitgedacht, was die Produktreife in Zukunft verbessern kann.

Auch wenn der vollständige Funktionsumfang des ursprünglichen Clients aus Zeitgründen nicht komplett nachgebildet werden konnte, zeigt der entwickelte Prototyp exemplarisch, dass die gewählte Sprache und Architektur prinzipiell geeignet sind, Funktionsparität mit der bestehenden Lösung zu erreichen.

Die Ergebnisse dieser Arbeit legen nahe, dass eine vollständige Migration des OpenVPN-Clients zu Go technisch machbar und langfristig potenziell vorteilhaft ist. Zukünftige Arbeiten sollten sich auf die Vervollständigung und Erweiterung des Funktionsumfangs konzentrieren. Auch eine Performance-Evaluierung der Go-Implementierung wäre wünschenswert.

Sollte die neue Library weiterentwickelt werden, empfiehlt sich ein Security-Audit der Software. Damit könnten gegebenenfalls Sicherheitslücken gefunden und geschlossen werden, bevor die Library in Produkten von Securepoint eingesetzt wird.

9. Glossar

AES. Advanced Encryption Standard. symmetrisches, Block-basiertes Verschlüsselungsverfahren 7

API. Application Programming Interface. Programmierschnittstelle, um mit einem Programm in Software zu interagieren 4, 23, 29, 33, 35, 46

CI-Pipeline. automatisch ausgeführtes Programm zum Testen von Software-Änderungen in der zentralen Code-Verwaltung; CI ist kurz für "Continuous Integration" 34, 35

ChaCha-Verschlüsselungsverfahren. symmetrisches, Strom-basiertes Verschlüsselungsverfahren 7

DTLS. Datagram Transport Layer Security. TLS über unzuverlässige Verbindungen 21

DoS. Denial of Service 8, 9, 22

Elgamal-Verschlüsselungsverfahren. asymmetrisches Verschlüsselungsverfahren 7

GNU GPLv2. GNU General Public License Version 2. Copyleft-Lizenz; fordert die Copyleft-Lizenzierung bei Nutzung einer mit ihr lizenzierter Software 3

HMAC. Hash-based Message Authentication Code. Integritätsbeweismechanismus für Nachrichten über unsichere Kanäle 8, 9, 19, 20

HTTP. Hypertext Transfer Protocol

HTTPS. Hypertext Transfer Protocol Secure. HTTP über eine TLS-Verbindung 11, 21

QUIC. Quick UDP Internet Connections. UDP-basiertes Transportprotokoll, das bessere Performanz als TCP zum Ziel hat 21

RSA. Rivest-Shamir-Adleman. asymmetrisches Kryptosystem 7

Serialisierung. Übersetzung von Daten in ein speicher- oder transportfähiges Format 26, 46

Socket. Kommunikationsschnittstelle des Betriebssystems, unter anderem für Protokolle auf Transportschicht 4, 11, 25, 26, 36, 39, 43

Software-Test. in Programmcode definierte Prüfung, die Fehler eines Programms erkennen kann 35, 37, 46

TCP. Transmission Control Protocol. zuverlässiges, verbindungsorientiertes Protokoll auf Transportschicht 10, 11, 12, 14, 15, 21, 36, 39, 41, 50, 51

TLS. Transport Layer Security. Verschlüsselungsprotokoll zur sicheren Datenübertragung über unsichere, zuverlässige Netze 11, 12, 13, 15, 16, 20, 21

TUN-Device. Virtuelle Netzwerkadapter des Betriebssystems 10, 11, 23, 25, 26, 27, 36, 43, 44

UDP. User Datagram Protocol. verbindungsloses Protokoll auf Transportschicht 10, 11, 12, 21, 36, 39, 41, 47, 51

VM. Virtual Machine. Software-basiertes Computersystem, das auf einem physischen Host-Rechner läuft und sich wie ein eigenständiger Computer mit Betriebssystem verhält 47

VPN. Virtual Private Network. Privates Netzwerk, das vom Betriebssystem simuliert wird, meist um auf entfernte private Netzwerkressourcen zuzugreifen 3, 5, 6, 18, 23, 47

10. Abbildungsverzeichnis

11. Tabellenverzeichnis

12. Verzeichnis weiterer Darstellungen

Bibliographie

[Firmen-interne Quelle], Bastian Kummer (2018) *Multiplatform OpenVPN-Client-Library in C*. Verfügbar unter: https://development.intern.securepoint.de/bastiank/bsd-ovpnclient.

[Firmen-interne Quelle], J.H. (2022a) *iOS/OpenVPN: Refactored tun-write operations by moving from callback-based to faster fd-based implementation*. Verfügbar unter: https://development.intern.securepoint.de/sms/sms-vpn-common/-/commit/455df625d5584ee9319fe1d422645f84205604d9.

[Firmen-interne Quelle], J.H. (2022b) *iOS: Simplified tun-write operations by moving from callback-based to faster fd-based implementation*. Verfügbar unter: https://development.intern.securepoint.de/bastiank/bsd-ovpnclient/-/commit/c20d1456dea24c2567d693cc455527fa8d6e3056.

[Firmen-interne Quelle], J.H. (2024) *Fehler #36283: OpenVPN-Library Crash in buffer_write*. Verfügbar unter: https://redmine.intern.securepoint.de/issues/36283.

[Firmen-interne Quelle], Mario Rhein (2025) *Benchmark: Durchsatz IPERF durch VPN-Tunnel*. Verfügbar unter: https://doc.intern.securepoint.de/books/benchmarks-und-belastungstests-2025/page/durchsatz-iperf-durch-vpn-tunnel.

[Firmen-interne Quelle], Securepoint GmbH (2025) *GitHub-Repository: Securepoint Cloud Shield für Android*. Verfügbar unter: https://github.com/Securepoint/quantum-query-app-android.

Andrew (2021) *Why are quantum computers faster at breaking RSA-2048 encryption than classical supercomputers are?*. Verfügbar unter: https://physics.stackexchange.com/q/651972.

Apple Inc. (2025a) *Data: init(bytes:count:)*. Verfügbar unter: https://developer.apple.com/documentation/foundation/data/init(bytes:count:).

Apple Inc. (2025b) *What is XNU?*. Verfügbar unter: https://github.com/apple-oss-distributions/xnu.

Avelino, T. und contributors (2025) *Awesome Go*. Verfügbar unter: https://awesome-go.com/.

Borenkraout, M. (2024) *The Harmful Misuse of Code Coverage*. Verfügbar unter: https://matanbobi.dev/posts/why-i-dont-like-code-coverage.

Bundesamt für Sicherheit in der Informationstechnik (2023) „CON.1 Kryptokonzept“.

Bundesamt für Sicherheit in der Informationstechnik (ohne Datum) *Was ist der Prüfsummencheck?*. Verfügbar unter: https://www.bsi.bund.de/dok/504478.

candawi (2023) *Technical Tip: Using DTLS to improve SSL VPN performance*. Verfügbar unter: https://community.fortinet.com/t5/FortiGate/Technical-Tip-Using-DTLS-to-improve-SSL-VPN-performance/ta-p/193881.

Danial, A. (2025) *cloc Source Code*. Verfügbar unter: https://github.com/AlDanial/cloc.

Devietti, J. *u. a.* (2008) „Hardbound: architectural support for spatial safety of the C programming language“, *SIGOPS Oper. Syst. Rev.*, 42(2), S. 103–114. Verfügbar unter: https://doi.org/10.1145/1353535.1346295.

Donenfeld, J.A. (2015) „WireGuard: Next Generation Kernel Network Tunnel“. Verfügbar unter: https://www.wireguard.com/papers/wireguard.pdf.

Donenfeld, J.A. (2020) *WireGuard 1.0.0 for Linux 5.6 Released*. Verfügbar unter: https://lore.kernel.org/wireguard/CAHmME9qOpDeraWo5rM31EWQW574KEduRBTL-+0A2ZyqBNDeYkg@mail.gmail.com/T/#u.

Donenfeld, J.A. (2022) *WireGuard: Benchmarking*. Verfügbar unter: https://www.wireguard.com/performance/#benchmarking.

Donovan, A.A.A. und Kernighan, B.W. (2016) *The Go Programming Language*.

Eddy, W. (2022) *Transmission Control Protocol (TCP). Request for Comments*, RFC Editor. Verfügbar unter: https://doi.org/10.17487/RFC9293.

Elgamal, T. (1985) „A public key cryptosystem and a signature scheme based on discrete logarithms“, *IEEE Transactions on Information Theory* [Preprint]. Verfügbar unter: https://doi.org/10.1109/TIT.1985.1057074.

Ferguson, P. und Huston, G. (1998) „What is a VPN?“.

Ferrumgate (2023) *A New VPN over QUIC protocol with Rust*. Verfügbar unter: https://ferrumgate.com/blog/new-vpn-over-quic-protocol/.

Frankel, S. und Krishnan, S. (2011) *IP Security (IPsec) and Internet Key Exchange (IKE) Document Roadmap. Request for Comments*, RFC Editor. Verfügbar unter: https://doi.org/10.17487/RFC6071.

Frikin, E. (2022) *How to troubleshoot IPsec VPN misconfigurations*. Verfügbar unter: https://www.redhat.com/de/blog/troubleshoot-ipsec-vpn.

Gao, S. (2020) *water, A simple TUN/TAP library written in native Go*. Verfügbar unter: https://github.com/songgao/water.

Gerard, E.-R. (2025) *wstunnel: Tunnel all your traffic over Websocket or HTTP2*. Verfügbar unter: https://github.com/erebe/wstunnel.

The Git Project Contributors (2025) „Git Source Code Management Reference“. Verfügbar unter: https://git-scm.com/docs.

Google LLC (2025c) *App-Größe reduzieren, Android Developer Docs*. Verfügbar unter: https://developer.android.com/topic/performance/reduce-apk-size.

Google LLC (2025b) *go.mod file reference*. Verfügbar unter: https://go.dev/doc/modules/gomod-ref.

Google LLC (2025a) *gomobile Documentation: Build a library for Android and iOS*. Verfügbar unter: https://pkg.go.dev/golang.org/x/mobile/cmd/gomobile#hdr-Build_a_library_for_Android_and_iOS.

Intel Corp. (1973) *MCS-4 Assembly Language Programming Manual - The INTELLEC 4 Microcomputer System Programming Manual (Preliminary ed.).*

Iyengar, J. und Thomson, M. (2021) *QUIC: A UDP-Based Multiplexed and Secure Transport. Request for Comments*, RFC Editor. Verfügbar unter: https://doi.org/10.17487/RFC9000.

Kaufman, C. *u. a.* (2014) *Internet Key Exchange Protocol Version 2 (IKEv2). Request for Comments*, RFC Editor. Verfügbar unter: https://doi.org/10.17487/RFC7296.

Khan, R.A., Mustafa, K. und Ahson, S.I. (2006) *Software Quality: Concepts and Practices*.

Klabnik, S. und Nichols, C. (2018) *The Rust Programming Language*.

Krasnyansky, M., Yevmenkin, M. und Thiel, F. (2002) *Universal TUN/TAP device driver*. Verfügbar unter: https://www.kernel.org/doc/html/latest/networking/tuntap.html.

Krawczyk, H., Bellare, M. und Canetti, R. (1997) *HMAC: Keyed-Hashing for Message Authentication. Request for Comments*, RFC Editor. Verfügbar unter: https://doi.org/10.17487/RFC2104.

Lyons, K. (2025) *VPN Types and Their Protocols Explained: When to Use Them*. Verfügbar unter: https://blog.openvpn.net/what-is-a-vpn-protocols-and-types.

Myers, G., Badgett, T. und Sandler, C. (2012) *The Art of Software Testing, Third Edition*. Verfügbar unter: https://community.fortinet.com/t5/FortiGate/Technical-Tip-Using-DTLS-to-improve-SSL-VPN-performance/ta-p/193881.

National Institute of Standards and Technology (2001) *Advanced Encryption Standard (AES)*. Washington, D.C.. Verfügbar unter: https://doi.org/doi:10.6028/NIST.FIPS.197-upd1.

Nir, Y. und Langley, A. (2018) *ChaCha20 and Poly1305 for IETF Protocols. Request for Comments*, RFC Editor. Verfügbar unter: https://doi.org/10.17487/RFC8439.

Open Observatory of Network Interference (2025) *minivpn Source Code*. Verfügbar unter: https://github.com/ooni/minivpn.

OpenVPN contributors (2025) *OpenVPN 2 Source Code*. Verfügbar unter: https://github.com/OpenVPN/openvpn.

OpenVPN Inc. (2025a) *Hardening OpenVPN Security*. Verfügbar unter: https://openvpn.net/community-resources/hardening-openvpn-security/.

OpenVPN Inc. (2025c) *OpenVPN Protocol*. Verfügbar unter: https://openvpn.net/community-docs/openvpn-protocol.html.

OpenVPN Inc. (2025b) *Protocol Compatibility*. Verfügbar unter: https://openvpn.net/community-resources/protocol-compatibility/.

OpenVPN Inc. (2025d) *Tutorial: Change the Data-Channel Encryption Cipher*. Verfügbar unter: https://openvpn.net/as-docs/tutorials/tutorial--change-encryption-cipher.html#tutorial--change-the-data-channel-encryption-cipher.

OpenVPN Inc. and contributors (2006) *Git Tag v2.1_rc1*. Verfügbar unter: https://github.com/OpenVPN/openvpn/tags?after=v2.1_rc2.

OpenVPN Inc. and contributors (2025) *OpenVPN-Quellcode*. Verfügbar unter: https://github.com/OpenVPN/openvpn.

Pike, R. (2023) *What We Got Right, What We Got Wrong, GopherConAU*. Verfügbar unter: https://www.youtube.com/watch?v=yE5Tpp2BSGw.

Quarkslab SAS, O.S.T.I.F. (2017) *OpenVPN 2.4.0 Security Assessment*. Verfügbar unter: https://ostif.org/wp-content/uploads/2017/05/OpenVPN1.2final.pdf.

Raymond, E.S. (1993) *The New Hacker's Dictionary, Third Edition*. MIT Press.

Refraction Networking (2025) *uTLS Source Code*. Verfügbar unter: https://github.com/refraction-networking/utls.

Rescorla, E. (1999) *Diffie-Hellman Key Agreement Method. Request for Comments*, RFC Editor. Verfügbar unter: https://doi.org/10.17487/RFC2631.

Rescorla, E. (2010) *Keying Material Exporters for Transport Layer Security (TLS). Request for Comments*, RFC Editor. Verfügbar unter: https://doi.org/10.17487/RFC5705.

Rescorla, E. und Dierks, T. (2008) *The Transport Layer Security (TLS) Protocol Version 1.2. Request for Comments*, RFC Editor. Verfügbar unter: https://doi.org/10.17487/RFC5246.

Rescorla, E., Tschofenig, H. und Modadugu, N. (2022) *The Datagram Transport Layer Security (DTLS) Protocol Version 1.3. Request for Comments*, RFC Editor. Verfügbar unter: https://doi.org/10.17487/RFC9147.

Rivest, R.L., Shamir, A. und Adleman, L. (1978) „A Method for Obtaining Digital Signatures and Public-Key Cryptosystems“. Verfügbar unter: https://doi.org/10.1145/359340.359342.

Ronin, V. (2016) *How to properly manage dependencies for C/C++ project?*. Verfügbar unter: https://softwareengineering.stackexchange.com/q/339046.

Rust contributors (2025) *The Cargo Book: The Manifest Format*. Verfügbar unter: https://doc.rust-lang.org/cargo/reference/manifest.html.

The Rust Project Contributors (2025a) *Memory allocation and lifetime - The Rust Reference*. Verfügbar unter: https://doc.rust-lang.org/reference/memory-allocation-and-lifetime.html.

The Rust Project Contributors (2025b) *Rust: Frequently Asked Questions*. Verfügbar unter: https://prev.rust-lang.org/en-US/faq.html#why-is-rustc-slow.

Schirrmacher, D. (2016) *Standardisierung von Verschlüsselungs-Algorithmus ChaCha20 für TLS abgeschlossen*. Heise Medien. Verfügbar unter: https://www.heise.de/news/Standardisierung-von-Verschluesselungs-Algorithmus-ChaCha20-fuer-TLS-abgeschlossen-3249531.html.

Schwabe, A. (2016) *OpenVPN Wire Protocol (work in progress)*. Verfügbar unter: https://openvpn.github.io/openvpn-rfc/openvpn-wire-protocol.html.

Schwabe, A. (2020) *[Openvpn-devel] [PATCH v3 5/9] Remove key-method 1*. Verfügbar unter: https://sourceforge.net/p/openvpn/mailman/message/37066487/.

Securepoint GmbH (2025) *Securepoint Wiki: VPN-Clients*. Verfügbar unter: https://wiki.securepoint.de/VPN.

Statista GmbH (2025) *Durchschnittliche Verbindungsgeschwindigkeit der Internetanschlüsse (Festnetz) in Deutschland von Oktober 2020 bis Juni 2025*. Verfügbar unter: https://de.statista.com/statistik/daten/studie/416534/umfrage/durchschnittliche-internetgeschwindigkeit-in-deutschland.

Stipakov, L. und Lichtenheld, F. (2022) *OpenVPN Control packet wire format*. Verfügbar unter: https://github.com/OpenVPN/openvpn-rfc/blob/master/wip/control_channel.txt#L65.

Stobitzer, C. (2017) *IT-Schutzziele: Vertraulichkeit, Integrität, Verfügbarkeit*. Verfügbar unter: https://www.kryptowissen.de/schutzziele.php.

The strongSwan Team (2025) *strongSwan Portability*. Verfügbar unter: https://strongswan.org/.

Turner, S. (2014) *Security vulnerabilities of the top ten programming languages: C, Java, C++, Objective-C, C#, PHP, Visual Basic, Python, Perl, and Ruby*. Verfügbar unter: https://aabri.com/manuscripts/131731.pdf.

Xue, D. *u. a.* (2024) „OpenVPN is Open to VPN Fingerprinting“, *Commun. ACM*, 68(1), S. 79–87. Verfügbar unter: https://doi.org/10.1145/3618117.

Yonan, J. (2018) „The OpenVPN 2.4 Reference Manual“. Verfügbar unter: https://openvpn.net/community-docs/community-articles/openvpn-2-4-manual.html.