Technická univerzita v Košiciach
Fakulta elektrotechniky a informatiky

Ľahká kryptografia vo VPN sieťach

Diplomová práca

2023 **Bc. Marek Rohač**

Technická univerzita v Košiciach
Fakulta elektrotechniky a informatiky

Ľahká kryptografia vo VPN sieťach

Diplomová práca

Študijný program:	Počítačové siete
Študijný odbor:	Informatika
Školiace pracovisko:	Katedra elektroniky a multimediálnych telekomunikácií (KEMT)
Školiteľ:	prof. Ing. Miloš Drutarovský, CSc.
Konzultant:	

Košice 2023 **Bc. Marek Rohač**

Abstrakt v SJ

Cieľom práce je oboznámiť čitateľa s problematikou VPN sieti. Tento úkon realizujeme pomocou opisu princípov fungovania VPN, klasifikáciou na základe viacerých aspektov a následnou charakteristikou viacerých známych VPN protokolov. Následne sa zameriame na charakteristiku pojmu ľahkej kryptografie. Z tohto odvetvia opíšeme kryptografickú permutáciu XOODOO a možnosti jej použitia. V praktickej časti analyzujeme, demonštrujeme a experimentálne odmeriame implementáciu XOODOO permutáciu v jednoduchej VPN sieti. VPN sieť vytvára voľne dostupný program DSVPN v jazyku C s otvoreným zdrojovým kódom. V rámci praktickej časti sa zameriame aj na rozšírenie kompatibility v programe DSVPN pre OS Windows. V závere práce zhrnieme dosiahnuté výsledky a ponúkneme čitateľovi možnosti ďalšieho rozšírenia práce.

Kľúčové slová v SJ

DSVPN, Ľahká kryptografia, Linux, VPN, Windows, XOODOO

Abstrakt v AJ

The aim of the work is to familiarize the reader with the issue of VPN networks. We perform this task by describing VPN principles, classification by several aspects, and subsequent characterization of several known VPN protocols. Subsequently, we will focus on the characteristics of the concept of lightweight cryptography. From this branch, we will describe the XOODOO cryptographic permutation and the possibilities of its use. In the practical part, we analyze, demonstrate and experimentally measure the implementation of XOODOO permutation in a simple VPN network. The VPN network is created by the freely available open-source C program DSVPN. As part of the practical part, we will also focus on the practical extension of compatibility in the DSVPN program for OS Windows. At the end of the work, we will summarize the achieved results and offer the reader possibilities for further expansion of the work.

Kľúčové slová v AJ

DSVPN, Lightweight Cryptography, Linux, VPN, Windows, XOODOO

Bibliografická citácia

ROHAČ, Bc. Marek. *Ľahká kryptografia vo VPN sieťach*. Košice: Technická univerzita v Košiciach, Fakulta elektrotechniky a informatiky, 2023. 99s. Vedúci práce: prof. Ing. Miloš Drutarovský, CSc.

67865

TECHNICKÁ UNIVERZITA V KOŠICIACH

FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Katedra elektroniky a multimediálnych telekomunikácií

ZADANIE DIPLOMOVEJ PRÁCE

Študijný odbor: **Informatika**

Študijný program: **Počítačové siete**

Názov práce:

Ľahká kryptografia vo VPN sieťach

Lightweight Cryptography in VPN Networks

Študent: **Bc. Marek Rohač**

Školiteľ: **prof. Ing. Miloš Drutarovský, CSc.**

Školiace pracovisko: **Katedra elektroniky a multimediálnych telekomunikácií**

Konzultant práce:

Pracovisko konzultanta:

Pokyny na vypracovanie diplomovej práce:

Na základe dostupných informácií naštudujte a opíšte kryptografickú permutáciu XOODOO využívanú v ľahkej kryptografii. Analyzujte a opíšte dostupné implementácie, ktoré využívajú XOODOO permutáciu v zabezpečení VPN siete. V jazyku C vytvorte demonštračné aplikácie jednoduchej VPN siete využívajúce XOODOO permutáciu na platformách Linux alebo Windows a experimentálne overte ich funkčnosť. Analyzujte aké výpočtové nároky má vytvorená implementácia. Dokumentáciu k diplomovej práci vytvorte tak, aby ju bolo možné využiť aj v špecializovaných predmetoch zameraných na bezpečnosť informačných a komunikačných systémov.

Jazyk, v ktorom sa práca vypracuje: slovenský

Termín pre odovzdanie práce: 21.04.2023

Dátum zadania diplomovej práce: 31.10.2022

TECHNICKÁ UNIVERZITA V KOŠICIACH FAKULTA ELEKTROTECHNIKY A INFORMATIKY -12-

.....

prof. Ing. Liberios Vokorokos, PhD.

dekan fakulty

Čestné vyhlásenie

Vyhlasujem, že som záverečnú prácu vypracoval samostatne s použitím uvedenej odbornej literatúry.

Košice, 28. 5. 2023 .....

Vlastnoručný podpis

Poďakovanie

Na tomto mieste by som rád poďakoval svojmu vedúcemu práce *prof. Ing. Milošovi Drutarovskému, CSc.* za jeho čas a odborné vedenie počas celého riešenia záverečnej práce.

Obsah

Zoznam obrázkov

Zoznam tabuliek

Zoznam zdrojových kódov

Zoznam skratiek

AES Advanced Encryption Standard.

AH Authentication Header.

CHAP Challenge Handshake Authentication Protocol.

ECC Eliptic Curve Cryptography.

ECDH Elliptic-Curve Diffie–Hellman.

ECDSA Elliptic Curve Digital Signature Algorithm.

ESP Encapsulating Security Payloads.

FSKD Full-State Keyed Duplex.

FTP File Transfer Protocol.

FW FireWall.

GRE Generic Routing Encapsulation.

GW GateWay.

HTTP HyperText Transfer Protocol.

HTTPS HyperText Transfer Protocol Secure.

IOT Internet Of Things.

IP Internet Protocol.

IPSec Internet Protocol Security.

ISAKMP Internet Security Association and Key Management Protocol.

KP Cryptographical Primitive.

L Layer.

L2F Layer 2 Forwarding protocol.

L2TP Layer 2 Tunneling Protocol.

LAC L2TP Access Concentrator.

LNS L2TP Network Server.

LTS Long Term Support.

LWC LightWeight Cryptography.

LWCA LightWeight Cryptography Algorithm.

MS-CHAP MicroSoft Challenge Handshake Authentication Protocol.

MTU Maximum transmission unit.

NAS Network Access Server.

NAT Network Address Translation.

NIST National Institute of Standards and Technology.

OS Operating System.

OSI Open Systems Interconnection reference model.

PAP Password Authentication Protocol.

PDV Packet Delay Variation.

PPP Point-to-Point Protocol.

PPPoE PPP over Ethernet.

PPTP Point-to-Point Tunneling Protocol.

RFC Request For Comments.

RSA Rivest–Shamir–Adleman.

SA Security Associations.

SMTP Simple Mail Transfer Protocol.

SSL Secure Socket Layers.

SSTP Secure Socket Tunneling Protocol.

TCP Transmission Control Protocol.

TCP/IP Transmission Control Protocol/Internet Protocol reference model.

TLS Transport Layer Security.

UDP User Datagram Protocol.

VB Virtual Box.

VM Virtual Machine.

VPN Virtual Private Network.

XOF eXtendable-Output Function.

Úvod

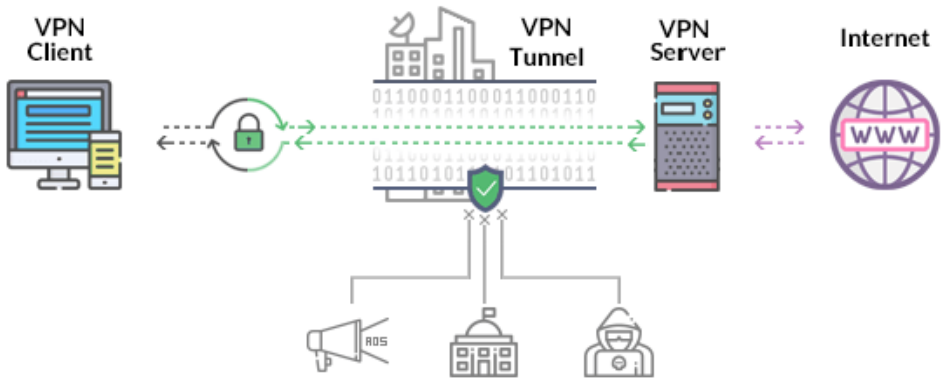
Virtuálna privátna sieť, tiež známa ako VPN, sa stala bežnou a veľmi využívanou technológiou na zabezpečenie sieťovej premávky. S VPN sa stretávame takmer v každej sfére. Domácnosti ju zvyknú používať na získanie prístupu k im nedostupným zdrojom. Vo sfére biznisu zasa s cieľom najlepšieho zabezpečenia dát v pomere s rýchlosťou, ktorú VPN implementácia poskytuje.

Cieľom práce je priblížiť čitateľovi informácie o VPN sieti. Postupnou charakteristikou, klasifikáciou na základe viacerých aspektov a opisom už existujúcich VPN protokolov. Následne špecifikujeme dôležitý prvok kvalitnej VPN siete, zabezpečenie pomocou kryptografie. Konkrétne sa zameriame na pomerne novú podkategóriu ľahká kryptografia. Tento pojem v práci charakterizuje. Z ľahkej kryptografie sme si za účelom opisu vybrali kryptografickú perrmutáciu XOODOO. XOODOO tvorí základ kryptografického balíka Xoodyak, jedného z finalistov štandardizačného procesu Národného inštitútu pre štandardy a technológie (NIST) v kategórií ľahkej kryptografie. V práci ju opíšeme a vysvetlíme možnosti jej použitia. Následne sa presunieme k jej praktickému použitiu za účelom zabezpečenia jednoduchej siete. VPN sieť vytvoríme pomocou voľne dostupného programu DSVPN napísaného v jazyku C. Program realizuje autentizované šifrovanie s využitím XOODOO permutácie. Experimentálne overíme funkčnosť. Krok po kroku implementáciu VPN v DSVPN zanalyzujeme. Následne doplníme zdrojový kód o kompatibilitu s operačným systémom Windows. Vykonané zmeny opíšeme. Súčasťou práce je aj experimentálne meranie počtu potrebných cyklov a rýchlosti šifrovacieho algoritmu počas bežnej prevádzky.

Dosiahnuté výsledky práce vyhodnotíme v samotnej kapitole tejto práce. Spomenuté činnosti budú vykonané v prostredí virtuálnych strojov na dvojici obrazov operačných systémov Microsoft Windows a Linux s distribúciou Ubuntu. Pri tvorbe, úprave a preklade zdrojového kódu použijeme jazyk C s GCC prekladačom.

1 Virtual Private Network – VPN

Virtuálna privátna sieť (ďalej **VPN**) je jeden zo spôsobov prepojenia zariadení, tak že internetová komunikácia medzi nimi je privátna, resp. zabezpečená aj v prípade používania nezabezpečenej, verejnej siete. Bezpečnosť spojenia je docielená pomocou kryptografických protokolov v tuneli, ktorý VPN vytvára. Pod pojmom tunel sa v skutočnosti myslí virtuálna zašifrovaná linka, ktorou je dátový paket prenášaný po sieti medzi koncovými zariadeniami. V skutočnosti tunel vzniká pomocou procesu zapuzdrenia dát. V závislosti od toho na akej úrovni OSI referenčného modelu sa pohybujeme. Táto technológia patrí aktuálne k najpoužívanejším spôsobom pripojenia sa medzi 2 rôznymi internetovými doménami. Najčastejší výskyt je možné sledovať v korporačnom prostredí, pričom cieľom je rozšírenie možností bezpečného pripojenia sa k firemnej sieti. Vzhľadom na firemné tajomstvá je nutné aby bolo takéto spojenie bezpečné a zamestnanci sa mohli pripojiť z rôznych miest. Vďaka uvedeným vlastnostiam je následne možná aj práca z domu (z ang. *Home office*), ktorá môže byť benefitom pre obe strany. Ukážka použitia VPN je znázornená pomocou 1.1. Naprieč VPN tunelom sa pohybujú zašifrované dáta.



Obr. 1.1: Ukážka typického VPN spojenia

1.1 Výhody a nevýhody VPN sieti

Medzi výhody VPN sietí patrí najmä zvýšenie bezpečnosti pripojenia. Všetka sieťová premávka je zašifrovaná pomocou kryptografického algoritmu. V závislosti od kvality implementácie sú informácie prenášane po takejto sieti nečitatelné. Ďalším typickým znakom je poskytnutie anonymity. VPN dokáže zamaskovať našu IP adresu a zároveň aj geografickú lokáciu. Tým, že sa pripojíme na zariadenie v inej sieti, tak získavame prístup k všetkým povoleným lokáciam tohto vzdialeného zariadenia. Týmto spôsobom dokážeme napríklad pristupovať k interným dátam. VPN je šikovný nastroj na ochranu vo verejnej nezabezpečenej sieti, ktorá poskytuje útočníkom rôzne možnosti útoku alebo zneužitia pripojeného zariadenia.

Medzi hlavné nevýhody patrí spomalenie internetového spojenia. Tým že, sa dodatočne spracúvavajú sieťové dáta, tak nastáva nechcené spomalenie. Táto dlhšia odozva siete sa predlžuje priamo úmerne od vzdialenosti servera, na ktorý sa pripájame. Ďalšou nepísanou nevýhodou je cena. Väčšina VPN poskytovateľov poskytuje služby za pomerne dráhe poplatky. Pri voľbe poskytovateľa je taktiež nutné prečítať si podmienky pripojenia. Niektorý poskytovatelia uchovávajú dáta, ktoré naprieč spojením prechádzajú. Poslednou nevýhodou je zložitá konfigurácia pri niektorých VPN protokoloch. Samozrejme uvedený problém nastáva len pri prvotnej konfigurácií VPN siete.

1.2 Charakteristika a definovanie pojmov

Obsahom tejto podkapitoly je zavedenie a následne stručná charakteristika pojmov, potrebných na pochopenie problematiky VPN sieti.

1.2.1 Tunel a tunelovacie rozhrania

Tunel v počítačových sieťach predstavuje virtuálne spojenie medzi jedným alebo viacerými zariadeniami. Jeho obsahom sa prenášajú enkapsulované, resp. zapuzdrené dáta. Proces pridávania dát k pôvodným sa v tejto súvislosti zvykne taktiež označovať ako **tunelovanie**. V počítačových sieťach je týmto spôsobom možné zmeniť, resp. zameniť použitý protokol za iný. Príkladom by mohlo byť tunelovanie z IPv4 do IPv6. V uvedenom prípade sa z pôvodného paketu použijú dáta a k ním sa pridajú údaje potrebné na smerovanie v IPv6 sieti. Výhodou je, že týmto spôsobom vieme sprostredkovať kompatibilitu naprieč viacerými sieťami s

rôznymi protokolmi. Obdobne vieme do procesu tunelovania začleniť aj bezpečnostné prvky v podobe šifrovania a autentizácie pôvodných dát. V súvislosti s tunelmi sa používateľ môže stretnúť s pojmami tunelovací protokol a tunelovacie rozhranie.

Tunelovací protokol predstavuje súbor činností, ktoré sá v procese tunelovania vykonávajú. V súčasnosti existuje veľa protokolov, ktoré sú štandardne špecifikované a bežne používané v sieťovej komunikácií. Najčastejšie sa používateľ stretne s tunelovaním pri VPN sieťach. Niektoré VPN v sebe nesú názov použitého tunelovacieho protokolu. Za spomenutie stojí protokol *Generic Routing Encapsulation* (ďalej GRE). GRE je tunelovací protokol vyvinutý spoločnosťou Cisco v roku 1994. Ponúka širokú paletu možností tunelovania viacerých sieťových protokolov vo virtuálnych spojeniach medzi zariadeniami, tzv. z ang. *point-to-(multi)point*. Niektoré variácie sa používajú aj pri vytváraní VPN. Viac informácií o GRE je dostupné na [1] a [2].

Tunelovacie rozhranie, resp. adaptér je virtuálne zariadenie, ktoré je možné vytvárať v jadre daného operačného systému. Jedná sa o kompletne softvérové riešenie. Dôležité je, že nie každý OS má natívnu podporu pre vytváranie virtuálnych tunelovacích rozhraní. Je preto potrebné používať ovládače tretích strán, ktoré danú funkcionalitu implementujú. Od roku 2000 podporujú tieto rozhrania Solaris, Linux a BSD operačné systémy. Poznáme dva typy tunelovacích rozhraní. Konkrétne TAP a TUN rozhranie. Nedajú sa použiť spolu, pretože pracujú na rôznych vrstvách. TUN simuluje správanie zariadenia na sieťovej vrstve (L3) a teda spracúva pakety a používa sa pri smerovaní. Na druhej strane TAP rozhranie pracuje o úroveň nižšie s ethernetovými rámcami (L2). Používa sa na premostenie sieťovej premávky. TUN/TAP rozhrania je možné použiť na odosielanie a príjmanie dát. Dáta odoslané OS do TUN/TAP sú dodávané do programu, ktorý rozhrania vytvoril. Na druhej strane používateľ môže v programe taktiež odoslať dáta do rozhraní. V tomto prípade TUN/TAP vloží dáta do sieťového zásobníka OS. Tým emuluje ich príjem z externého zdroja.

Technológia TUN/TAP je pomerne známa a zaužívaná. Svoje uplatnenie zohráva pri softvérovej implementácií VPN. Okrem toho sa bežne používa aj vo sfére virtuálnych strojov. Napríklad aj vo voľne dostupný nástroji na virtualizáciu – VirtualBox. Pri vytvorení a následnom používaní obrazu, dochádza k vytváraní tunelovacích rozhraní. Prostredníctvom zvolenej konfigurácie stroja sa rozhrania nakonfigurujú. Týmto spôsobom sa pre používateľa sprístupní možnosť komunikácie virtuálneho stroja s inými zariadeniami. Viac informácií o problematike je možné nájsť na [3], [4] a [5], odkiaľ bol obsah tejto kapitoly čerpaný.

1.2.2 Charakteristika referenčných modelov

Pred podkapitolou 1.3.2 je potrebné pred samotnou klasifikáciou vysvetliť pojmy Referenčný model prepojenia otvorených systémov[1] (ďalej OSI) a Model opisujúci balíky internetových protokolov, známy ako, TCP/IP referenčný model.

Úlohou uvedených referenčných modelov je vizualizácia postupu spracovania dát od používateľa až k ich odoslaniu zo zariadenia[2]. Pojem spracovanie dát znamená opísanie toho ako dochádza v jednotlivých abstraktných vrstvách k pretransformovaniu používateľských dát na súbor jednotiek a núl, ktoré sú následne odoslané do iného zariadenia.

OSI Model vznikol v skorších fázach evolúcie počítačových sietí. Vychádza skôr z teoretického než praktického prístupu. Pozostáva zo 7 abstraktných vrstiev[3]:

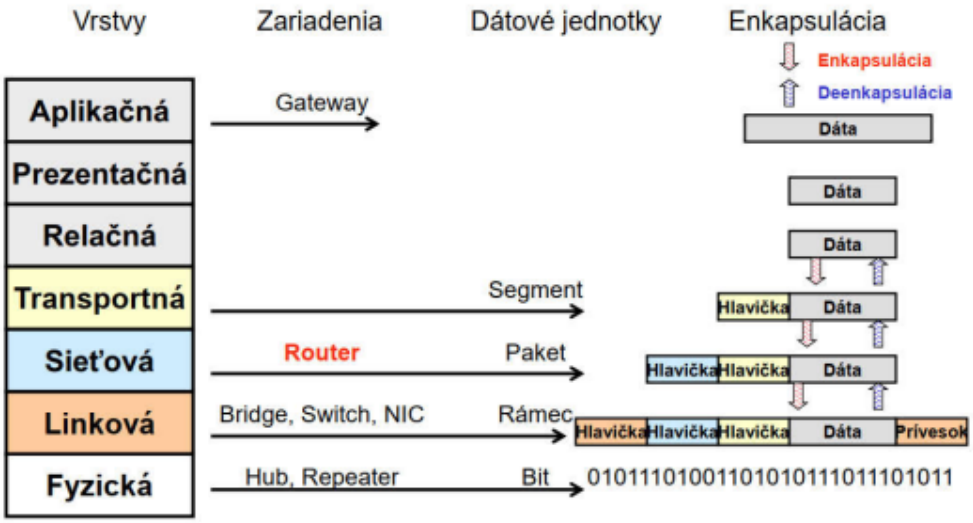
1. **fyzická vrstva** – z ang. *Physical Layer* (ďalej L1),
2. **spojová vrstva** – z ang. *Data Link Layer* (ďalej L2),
3. **sieťová vrstva** – z ang. *Network Layer* (ďalej L3),
4. **transportná vrstva** – z ang. *Transport Layer* (ďalej L4),
5. **relačná vrstva** – z ang. *Session Layer* (ďalej L5),
6. **prezenčná vrstva** – z ang. *Presentation Layer* (ďalej L6),
7. **aplikačná vrstva** – z ang. *Application Layer* (ďalej L7),

V schéme 1.2 sa nachádzajú jednotlivé vrstvy spoločne so zariadenia, ktoré pracujú s jednotlivými dátami. Zároveň je v schéme znázornený proces (de)enkapsulácie. Schéma bola prebratá z [6].

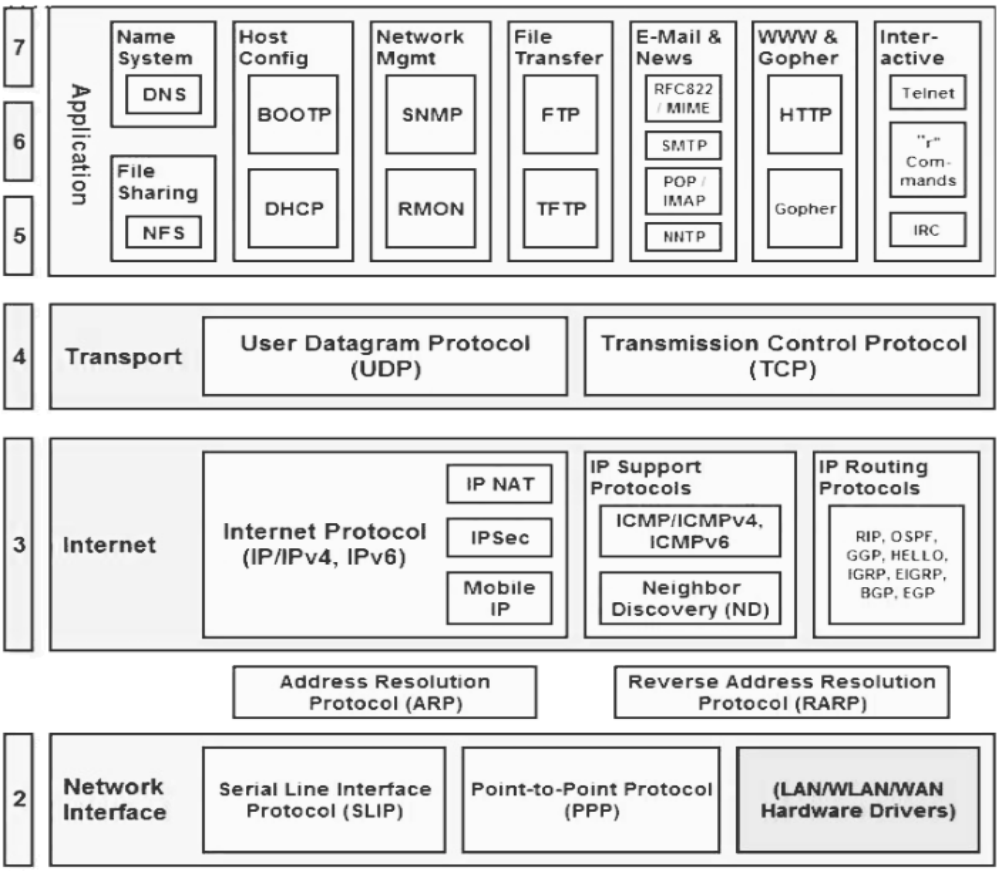
Čím je väčšie číslo vrstvy tým bližšie sa dáta nachádzajú pri používateľovi. Vďaka uvedeným vlastnostiam je tento model vhodnejší pri začiatku štúdia spracovania sieťových dát v počítači. Z rovnakého dôvodu sa taktiež viac stretávame s jeho použitím pri opise funkcionality riešenia ako s TCP/IP novším modelom. Viac informácií o problematike nájde čitateľ v [**osi**].

Na druhej strane TCP/IP model vznikol z praktického prístupu. Hlavný rozdiel je v počte abstraktných vrstiev, ktorý je v prípade TCP/IP zmenšený na 4 vrstvy [7]. TCP/IP model je zobrazený pomocou schémy 1.3. V uvedenej schéme sú znázornené niektoré z protokolov, ktoré sa na jednotlivých vrstvách používajú.

[1] z ang. *Open Systems Interconnection reference model*
[2] z ang. *end-to-end data communication*
[3] z ang. *layer*



Obr. 1.2: Schéma postupného spracovania dát jednotlivými vrstvami OSI modelu [6]



Obr. 1.3: Schéma TCP/IP modelu s niektorými protokolmi

Aplikačná vrstva (L5-7) zahŕňa protokoly používané väčšinou aplikácií na poskytovanie užívateľských služieb alebo výmenu aplikačných dát cez sieťové

pripojenia, ktoré je vytvorené protokolmi na nižšej úrovni. Spája vrstvy L5 až L7 OSI modelu. Príklady známych protokolov aplikačnej vrstvy sú HyperText Transfer Protocol (HTTP), File Transfer Protocol (FTP), Simple Mail Transfer Protocol (SMTP) a iné. Údaje, resp. dáta sú pri spracovaní kódované podľa L4 protokolov. Sú zapuzdrené do protokolových jednotiek transportnej vrstvy, tzv. **segmentov**.

Transportná vrstva (L4) vytvára základné dátové kanály, ktoré aplikácie používajú na výmenu dát. Vrstva vytvára konektivitu medzi hostiteľmi, ktorá je nezávislá od siete, štruktúry užívateľských dát a smerovacích informácií. Konektivita na transportnej vrstve môže byť kategorizovaná ako orientovaná na spojenie, implementovaná v protokole TCP, alebo UDP bez orientácie na spojenie. Uvedené protokoly sú stručne charakterizované nižšie, v tejto podkapitole.

Protokoly v tejto vrstve zabezpečujú:

- riadenie chýb – z ang. *error control* [8],
- segmentáciu dát – z ang. *segmentation* [9],
- riadenie toku dát – z ang. *flow control* [10],
- riadenie preťaženia – z ang. *congestion control* [11],
- adresovanie aplikácií pomocou portov – z ang. *application addressing* [12].

Výstupom transportnej vrstvy sú segmenty, ktoré sú spracované v ďalšej vrstve referenčného modelu.

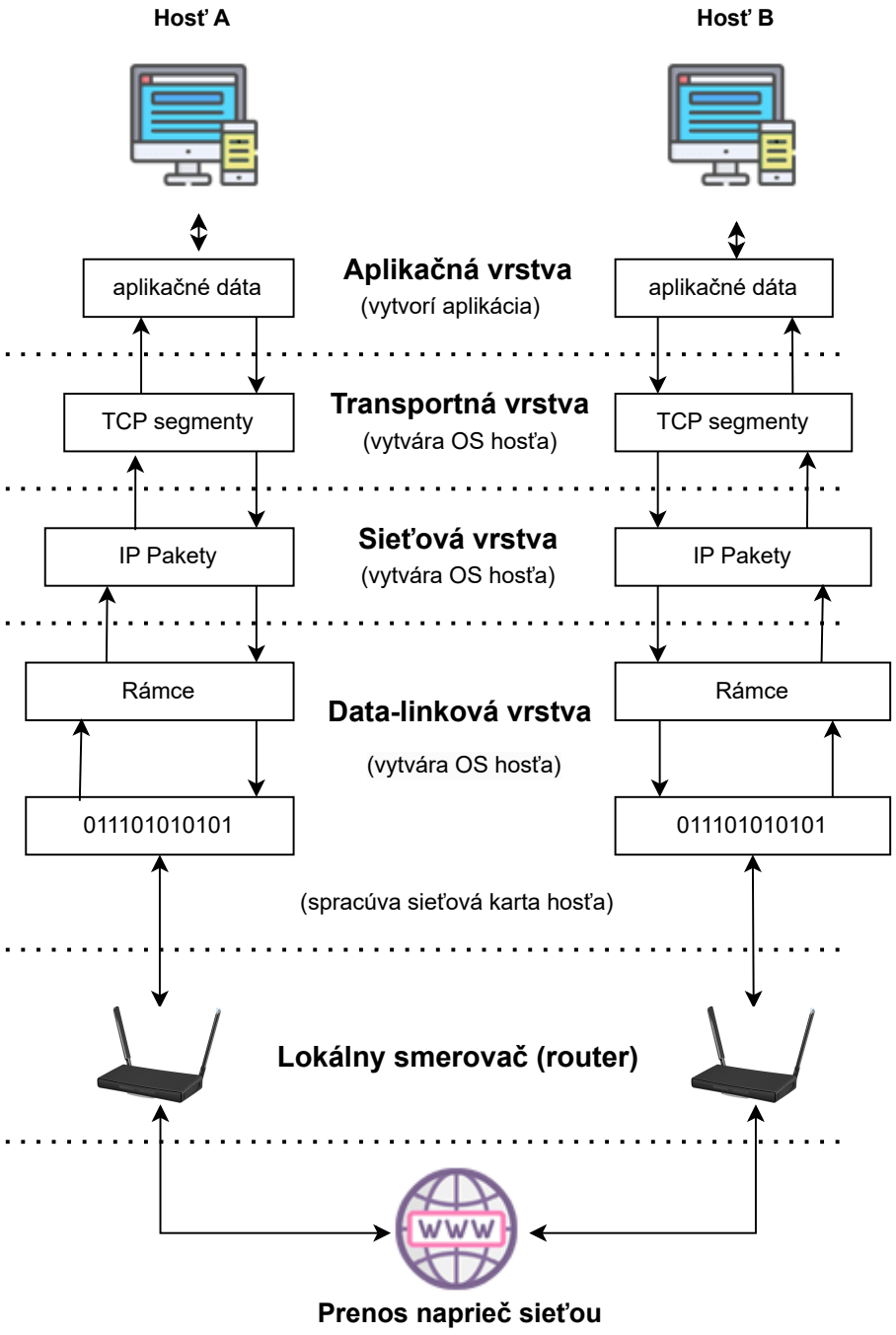
Internetová, resp. Sieťová vrstva (L3) je zodpovedná za odosielanie paketov cez jednu alebo viac sietí. S touto funkcionalitou internetová vrstva umožňuje sieťovanie, prepojenie rôznych IP sietí a v podstate vytvára internet. Z L4 segmentov tvorí pakety, tak že pridá informácie potrebné na ďalšie smerovanie.

Linková vrstva (L1-2) sa používa na presun paketov medzi rozhraniami internetovej vrstvy dvoch rôznych hostiteľov na rovnakom linku. Procesy vysielania a prijímania paketov na linke je možné konfigurovať. Zariadenia nazývané prepínače [4], vykonávajú rámcovania[5]. Pripravujú pakety z L3 vrstvy na prenos pridaním ďalších informácií. Týmto úkonom vzniknú rámce. Tie sa prenášajú do fyzickej vrstvy a cez prenosové médium až k hostiteľovi. Fyzická vrstva predstavuje prvok, cez ktoré sú dáta prenášane. Napríklad optický a ethernetový kábel.

Vyššie uvedené procesy prípravy a spracovania dát sú znázornené pomocou schémy 1.4. V tejto schéme je znázornený proces vytváranie sieťových dát, ich

[4] z ang. *switch*
[5] z ang. *frame*

následne smerovanie naprieč sieťou a spracovanie na cieľovou zariadení. V bežnej prevádzke sa jedná o obojsmernú komunikáciu. Z uvedeného dôvodu sú šipky obojstranné.



Obr. 1.4: Proces formovania a spracovania sieťových dát naprieč dvoma zariadeniami

Protokol riadenia prenosu (z ang. *Transmission Control Protocol*, ďalej TCP) je komunikačný protokol orientovaný na nadviazanie a udržanie sieťového spojenia medzi zariadeniami. Môže byť použitý pri úlohe príjímateľa aj odosielateľa (z ang. *full-duplex*). Úlohou je spoľahlivý prenos dát medzi komunikantmi. Odoslanie a príjem dát je v rovnakom poradí. Protokol zároveň obsahuje mechanizmy na kontrolu výskytu chýb. Svoj názov má podľa dvoch najdôležitejších protokolov:

- Protokol riadenia prenosu – z ang. *Transmission Control Protocol*,
- Internet protokol – z ang. *Internet Protocol*.

Na začiatku 21. storočia je 95% paketov používaných na internete TCP. Bežné aplikácie používajúce TCP sú webové (HTTP/HTTPS protokoly), slúžiace na e-mailovú komunikáciu (SMTP/POP3/IMAP) a prenos súborov (z ang. *File Transfer Protocol – FTP*). Minimálna dĺžka hlavičky TCP je 20 bajtov a maximálna dĺžka 60 bajtov. Po pridaní údajov TCP hlavičky k prenášaným dátam, vzniká tzv. *segment*.

V súčasnosti je možne TCP protokol implementovať softvérovo aj hardvérovo. Pri prvom z uvedených je problémom závislosť na OS a následne aj vysoká vyťaženosť procesora pri príprave a spracovaní dát. Pri hardvérovom riešení je výhodou optimalizácia a implementácia bez potreby dodatočnej úpravy OS. Hardvérové implementácie sa realizujú pomocou koprocesorov umiestnených vo vnútri procesora. Následkom toho môžeme dnes bežne pozorovať umiestnenie spomenutých zariadení na našich zariadeniach.

Podrobnejšie informácie o TCP protokole je možné nájsť na [13]. V uvedenej publikácií sa nachádza opis TCP hlavičky, metódy nadviazania a ukončenia spojenia. Obdobne je spomenuté ako dochádza k prenosu dát pomocou sekvenčných čísel. Ak má používateľ nejasnosti v fungovaní TCP protokolu, odporúča sa danú publikáciu prečítať.

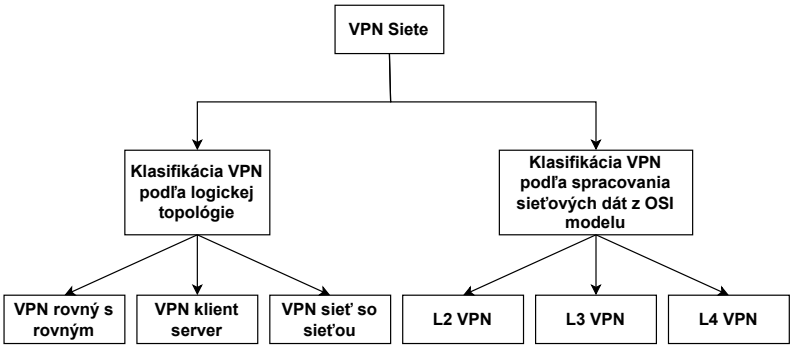
Používateľský datagramový protokol (z ang. *User Datagram Protocol*, ďalej UDP) je jedným zo základných komunikačných IP protokolov. Používa sa na odosielanie správ iným hostiteľom v sieti. Správy sú prenášané ako datagramy v paketoch. UDP nevyžaduje predchádzajúcu komunikáciu na nastavenie komunikačných kanálov alebo dátových ciest. Používa jednoduchý komunikačný model bez spojenia s minimom protokolových mechanizmov. Poskytuje kontrolné súčty pre integritu údajov a čísla portov na adresovanie rôznych funkcií v zdroji a cieli datagramu.

Narozdiel od TCP) neposkytuje žiadnu záruka doručenia správy alebo duplicitnej ochrany. UDP) je vhodný na účely, kde kontrola a oprava chýb buď nie sú

potrebné, alebo sa vykonávajú v aplikácii. Aplikácie citlivé na čas často používajú UDP, pretože zahadzovanie paketov je vhodnejšie ako čakanie na pakety oneskorené v dôsledku opätovného prenosu. Príklad použitia môžu byť streamovacie služby. Podrobnejšie informácie o UDP) protokole nájde čitateľ v [**udp**].

1.3 Klasifikácia VPN sietí

V súčasnosti má čitateľ k dispozícii veľa rôznych internetových zdrojov o problematike VPN. Uvedené sú rôzne možnosti klasifikácie VPN siete. V rámci tejto práce klasifikujeme VPN siete podľa logickej topológie, použitých protokolov a vrstiev, na ktorých sú postupy aplikované. Obsahom tejto podkapitoly je rozdelenie a opis jednotlivých typov VPN sieti. Rozdelenie zavedené v tejto práci je možné vidieť v schéme 1.5. V závere kapitoly čitateľ nájde sumárne zaradenie opísaných VPN protokolov do uvedenej klasifikácie.



Obr. 1.5: Klasifikácia VPN sieti

1.3.1 Rozdelenie VPN sieti podľa logickej topológie

Podľa topológie, v ktorej VPN spojenie prebieha rozdeľujeme VPN do 3 kategórií:

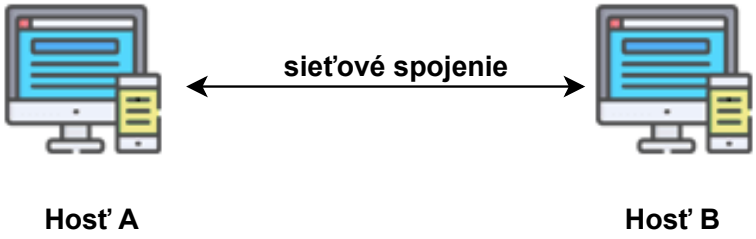
- **VPN rovný s rovným** – *z ang. Peer to Peer VPN*,
- **VPN klient a server** – *z ang. Client to Server VPN*,
- **VPN sieť so sieťou** – *z ang. Site to Site VPN*.

VPN rovný s rovným

Uvedený spôsob vytvára zabezpečený tunel medzi dvoma rovnocennými uzlami, resp. zariadeniami[6], ktorý spoločne komunikujú cez verejnú sieť. Medzi zariade-

[6] z ang. *peers*

niami je vytvorený tunel. Každý koniec má priradenú svoju IP adresu. Z uvedeného modelu vyplýva aj následná limitácia. VPN tunel vznikne iba medzi dvoma komunikujúcimi zariadeniami. Z toho dôvodu nie je toto použitie časté. Na obrázku 1.6 je znázornený uvedený typ VPN spojenia.



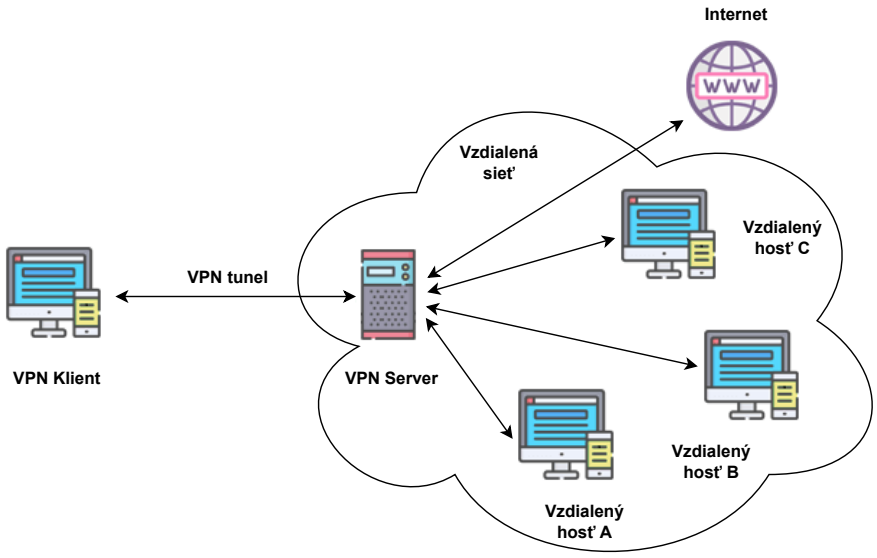
Obr. 1.6: Ukážka spojenia hosťov typu rovný s rovným

VPN Klient a Server

Tento typ spojenia pozostáva z pripojenia medzi nerovnocennými dvoma alebo viacerými zariadeniami. Najjednoduchší model musí pozostávať z jedného VPN servera a VPN klienta. Princíp spočíva vo vytvorení zabezpečeného tunela, ktorý je použitý na prenos dát medzi uvedenými zariadeniami. Zároveň je však možné vytvoriť $1 : N$ takýchto spojení. N reprezentuje počet VPN klientov, resp. prepojení, ktorý VPN Server dokáže nadviazať. Tento parameter je závislý najmä od hardvérových prostriedkov daného servera.

Úloha Klienta spočíva v presmerovaní všetkej svojej sieťovej komunikácie cez zabezpečený tunel, ktorý vznikol medzi ním a Serverom. Tento úkon je najčastejšie realizovaný presmerovaním trafiky cez sieťovú bránu[7] (ďalej GW). V danom OS, na ktorom VPN klient beží, je teda potrebné zmeniť IP adresu GW na adresu VPN servera. Vďaka tomu nastane presmerovania komunikácie. Tento úkon je väčšinou realizovaný programovo pomocou aplikácií. Typicky nastolia spojenie medzi Klientom a Serverom. Následne upravia sieťové nastavenia systému. Spomenuté úkony sú vysoko závislé od OS a daného programovacieho jazyka, prostredníctvom ktorého sú úpravy realizované. Úloha Servera na druhej strane spočíva vo vytvorení možnosti pripojenia pre jedného alebo viacerých klientov. Následne Server zastupuje klientovu prítomnosť v danej sieti. Teda spracúva požiadavky Klienta a komunikuje s ostatnými zariadeniami. Komunikácia ďalej však už nie je zabezpečená pomocou šifrovania alebo tunelu. Tento fakt je znázornený na obrázku 1.7.

[7] z ang. *GateWay*



Obr. 1.7: Ukážka pripojenia VPN klienta na VPN server

V súčasnosti je tento spôsob považovaný za najviac používaný. VPN server slúži ako vstupná brána do internej siete. Vďaka tomu je možné sprístupniť zdroje pre používateľov z rôznych oblastí sveta. Používateľ sa taktiež môže stretnúť s pomenovaním model **uzol k sieti**, resp. z ang *point-to-site*. Obidva pojmy sú správne a predstavujú rovnakú myšlienku zapojenia VPN.

Sieť so Sieťou VPN

VPN model Sieť so Sieťou vytvára zabezpečený tunel medzi dvoma rôznymi sieťami naprieč verejnou sieťou. Model pozostáva z 2 zariadení – VPN servera a VPN Koncentrátora[8].

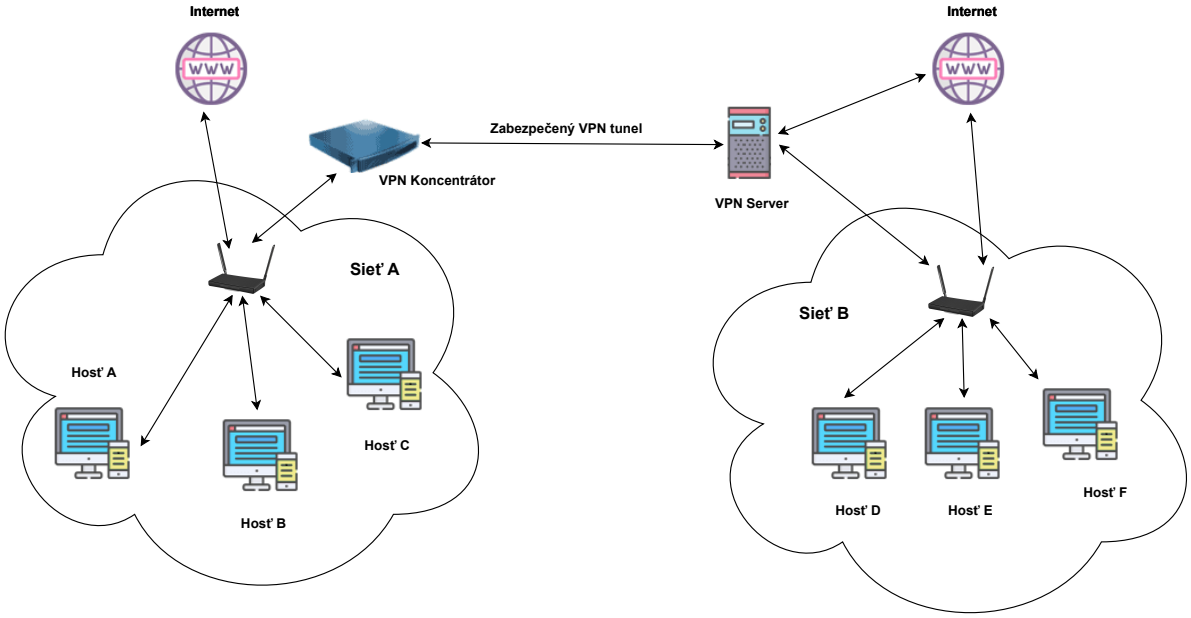
VPN Koncentrátor je typ sieťového zariadenia, ktoré poskytuje zabezpečené VPN spojenie a doručenie dát. Zvyčajne je to špecializovaný smerovač[9]. Dokáže vytvárať veľké množstvo VPN tunelov. Používa sa na nastolenie VPN modelu sieť so sieťou. Funkcionalita koncentrátora pozostáva z:

- nastolenie a konfiguráciu VPN tunela – z ang. *Establish and Configure tunnels*,
- autentizáciu používateľa– z ang. *Authenticate users*,
- priradenie IP adries používateľov k tunelom – z ang. *Assign tunnel/IP addresses to users*,
- šifrovanie a dešifrovanie dát – z ang. *Encrypt and Decrypt data*,

[8] z ang. *VPN Concentrator*
[9] z ang. *router*

- zabezpečiť integritu doručenia – z ang. *Ensure end-to-end delivery of data*.

Model Sieť so sieťou je používaný najmä v biznis svete pri spojení vedľajšej pobočky s hlavnou, ktoré sa nachádzajú na rozdielnych geografických lokalitách. Pomocou schémy 1.8 je znázornený tento model.



Obr. 1.8: Ukážka spojenia VPN siete typu sieť so sieťou

Používateľ však málo kedy pozná o aký presne druh VPN sa jedná. Najčastejšie sa stretne s označením, z ang *Remote Access VPN*. Jedná sa o vyššie uvedené klient-server VPN spojenie.

Informácia z tejto podkapitoly boli čerpané z [14].

1.3.2 Rozdelenie VPN sieti podľa vrstiev referenčného modelu

VPN siete môžeme klasifikovať podľa vrstvy (ďalej L) referenčného modelu. Rozdelenie vytvoríme na základe vrstvy pôvodných sieťových dát, ktoré VPN aplikácia spracúva. Stručná charakteristika referenčných modelov je obsahom podkapitoly 1.2.2. Každá dobrá VPN by v svojej implementácií mala realizovať šifrovanie. Preto použijeme túto činnosť ako príklad bloku, pri ktorom dochádza k už spomenutému spracovaniu pôvodných dát.

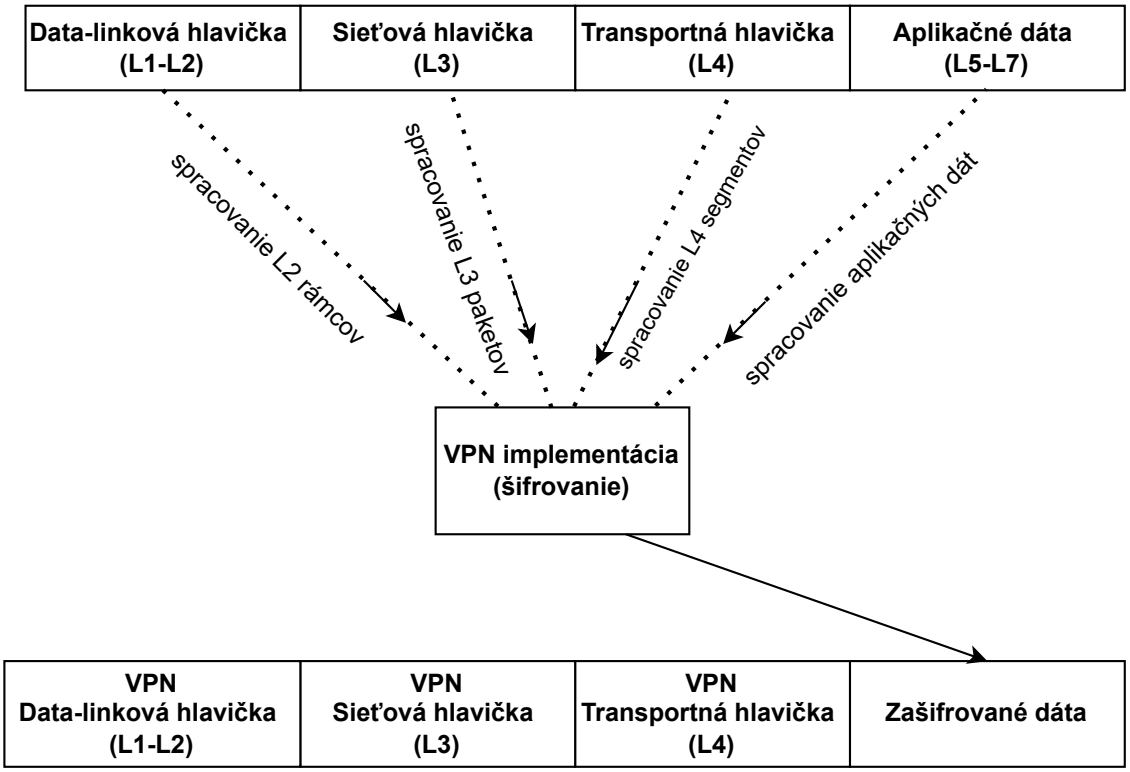
Klasifikácia VPN sietí podľa OSI modelu:

- **L2 VPN** – VPN na spojovej vrstve,
- **L3 VPN** – VPN na sieťovej vrstve,

- **L4 VPN** – VPN na transportnej vrstve.

Pri implementácií VPN komunikácie medzi zariadeniami je pre pochopenie dôležité určiť aké dáta vstupujú do VPN algoritmu. Pomocou tejto informácie a klasifikácie vyššie následne dokážeme zaradiť do určitej kategórie.

Princíp použitia spracovania dát za pomoci bloku na šifrovanie je znázornený pomocou schémy 1.9. Na schéme sa nachádzajú prerušované čiary. Tie reprezentujú scenár aký typ dát môže VPN spracúvať.



Obr. 1.9: Ukážka formovania sieťových dát po spracovaní pôvodných dát

Zo schémy je viditeľné, že spracovaním nižšej vrstvy dochádza k zväčšeniu bloku dát, ktoré je potrebné spracovať. To môže negatívne ovplyvniť výslednú rýchlosť celého systému. Okrem uvedenej vlastnosti je ďalším problémom prenos vrámci rôznych sieti. Z uvedených dôvodov je preto najrozšírenejší spôsob VPN spracovania dát na pomedzi L3 až L4 vrstve.

Typický sa s L2 a L3 VPN sieťami môže používateľ stretnúť na špecializovaných sieťových zariadeniach. Konkrétne na routroch a switchoch. Prvé z uvedených je využívané najmä pri smerovaní, respektíve určení cesty smerom von z lokálnej siete až k cieľovej destinácií. Tento úkon je vykonaný za pomoci aplikácie smerovacích protokolov. Router je možné použiť aj na smerovanie v rámci lokálnej siete. V porovnaní so switchom však nedosahuje porovnateľný výkon. Na druhej strane klaasický switch je možné použiť len vrámci lokálnej siete. V súčasnosti

sa používajú aj tzv. L3 switche. V porovnaní s routrom je ich prednosťou vyššia rýchlosť spracovania prichádzajúcich paketov pri väčšom počte pripojených zariadení. Všetky uvedené typy sa však dajú realizovať aj na konečnom zariadení v podobe softvérovej implementácie.

1.4 Protokoly vo VPN sieťach

V predchádzajúcej podkapitole sme klasifikovali VPN siete na základe zapojenia kryptografického bloku do OSI referenčného modelu. V tejto podkapitole pomocou uvedeného rozdelenia, zaradíme a charakterizujeme zopár protokolov, ktoré sa typicky používajú vo VPN sieťach. Pred začiatkom by som rád čitateľa upozornil, že aktuálne neexistuje svetový štandard na vytváranie VPN spojení. Dôsledkom toho existuje veľké množstvo rôznych protokolov. Vrámci tejto podkapaitoly si predstavíme niektoré z nich.

1.4.1 Point-to-Point Tunneling Protocol (PPTP)

PPTP umožnuje vytvoriť zabezpečené VPN spojenie k inej internetovej sieti. TCP/IP sieťový protokol PPTP zabezpečuje bezpečný prenos dát z klienta do privátneho servera vo VPN sieti. Jedná sa o starší Microsoft L2 protokol, ktorý bol definovaný v roku 1996. Je rozšírením L2 PPP protokolu. PPTP zapuzdruje PPP pakety do IP datagramov. Následne ich prenáša cez sieť.

Zabezpečenie prenosu pomocou PPTP pozostáva typicky z 3 po sebe nasledujúcich procesov:

1. PPP pripojenie a komunikácia – z ang. *PPP Connection and Communication*,
2. riadenie spojenia pomocou PPTP protokolu – z ang. *PPTP Control Connection*,
3. prenos dát PPTP tunelom – z ang. *PPTP Data Tunneling*.

Prechod medzi procesmi je možný len ak došlo k úspešnému dokončeniu predchádzajúceho kroku. V prvom procese sa PPTP klient pripája k serveru s prístupom na internet[10] (ďalej NAS). Používa sa pri tom PPP protokol, ktorý nastolí spojenie a zašifruje pakety. V druhom kroku následne vznikne TCP pripojenie medzi NAT a PPTP serverom. Použitý je port 1723. Uvedeným postupom nám

[10] z ang. *Network Access Server*

Obr. 1.10: Schéma zložených sieťových dát po PPTP spracovaní

vznikne medzi zariadeniami PPTP Tunel. Po úspešnom vytvorení konektivity dochádza k zapuzdrovaniu prichádzajúcich zašifrovaných PPP dát do PPTP protokolu a ich prenosu cez tunel. PPTP zapuzdruje dáta do tzv. IP datagramov, ktoré už obsahujú zašifrovaný PPP paket. Datagramy su vytvorené pomocou GRE protokolu, ktorý bol opísaný v **??**. Na schéme 1.10 je znázornená schém PPTP Paketu.

PPTP Server po prijatí dáta rozbalí, dešifruje PPP paket a následne ho smeruje v rámci lokálnej siete.

Je dôležité poznamenať, že PPTP klient môže mať priamy prístup na internet. V uvedenom prípade sa nevytvára prvotné PPP spojenie až k internetovému poskytovateľovi. Viac informácií o protokole PPTP nájde čitateľ na [15] a [16], ktoré boli zdrojom pri tvorbe tejto podkapitoly.

Bezpečnosť a použitie protokolu

Protokol vznikol v júny 1996. Implementácia poskytuje používateľovi:

- **Autentizáciu** – z ang. *Authentication*, overenie totožnosti používateľa pomocou mena a hesla. Na výber boli autentizačné protokoly *Challenge Handshake Authentication Protocol* (CHAP) [17], *MicroSoft Challenge Handshake Authentication Protocol* (MS-CHAP) [18] a *Password Authentication Protocol* (PAP) [19].

- **Kontrolu prístupu** – z ang. *Access Control*, po úspešnej autentizácií je následne prístup používateľa riadený na základe pravidiel a politiky prístupu daného OS.

- **Šifrovanie dát** – z ang. *Data Encryption*, je vykonané pomocou vopred zdieľaného kľúča, ktorý sa získa odvodením z hašovanej hodnoty uloženého hesla používateľa. Haš je vstupom do prúdovej šifry RC4 [20] a výstupom je 40-bitový kľúč relácie.

- **Filtrovanie PPTP paketov** – z ang. *PPTP Packet Filtering*, možnosť zapnutia filtrovania paketov len autentizovaných PPTP klientov.

- **Preddefinované Firewall pravidla pre PPTP** – z ang. *PPTP with Firewalls and Routers*, PPTP má štandardne definovaný TCP port 1723 a ID 47 v IP protokole. Vďaka tomu je možné jednoducho presmerovať tok dát.

Protokol je aktuálne štandardne zahrnutý v každej distribúcií OS Windows aj Linux. Pri vytváraní VPN siete teda používateľ môže zvoliť uvedený protokol na zabezpečený prenos dát v lokálnej, ale aj naprieč verejnou sieťou. Výhodou protokolu je vysoká kompatibilita naprieč rôznymi platformami. Nevýhodou je veľké množstvo zraniteľných miest, možností zneužitia a použitie starších kryptografických algoritmov. Aktuálne existujú protokoly poskytujúce lepšiu bezpečnosť ako uvedená implementácia. Z uvedených dôvodov sa tento protokol neodporúča používať.

1.4.2 Layer 2 Tunneling Protocol (L2TP)

Ďalším z VPN tunelovacích protokolov je L2TP. Špecifikovaný bol v roku 2000 v dokumente RFC[11] 2661 [21]. Inšpirovaný dvoma staršími protokolmi L2F[12] [22] a PPTP.

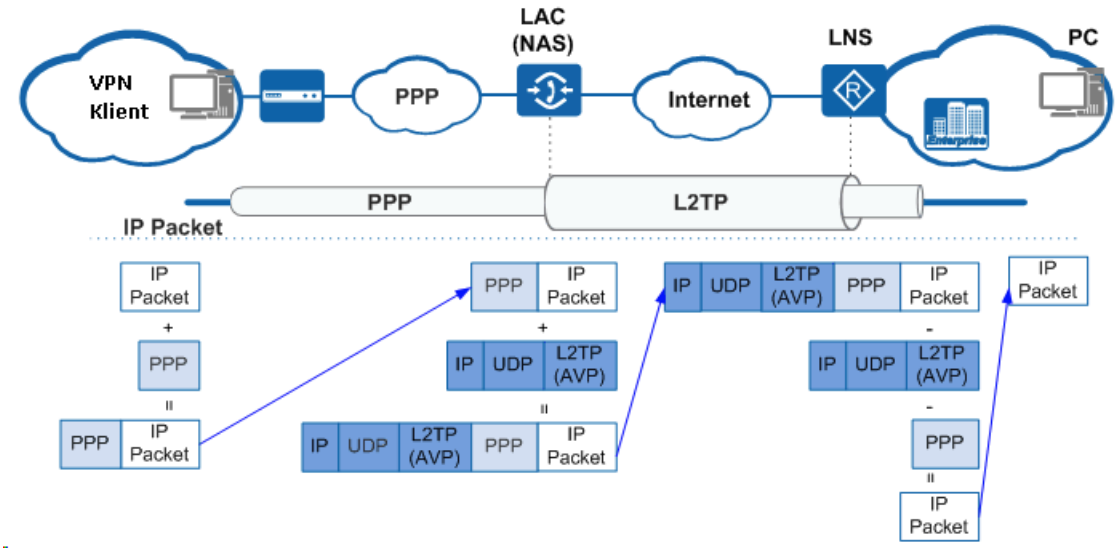
L2TP sieť pozostáva primárne z 3 zariadení:

1. **PPP terminál** – ľubovoľné zariadenie na vykonanie PPP enkapsulácie na dáta a pripojenie k LAC[13]. Môže to byť aj samotné zariadenie, ktoré sa pripája.

2. **L2TP prístupového servera** – LNS[14], jeden z koncov tunela, ktorý deenkapsuluje dáta a poskytuje prístup do lokálnej sieti. Na zariadení prebieha autentizácia používateľa, nastolenie PPP relácie[15] a L2TP tunela s LAC. Nasadzuje sa na hranici medzi privátnou a verejnou sieťou, zvyčajne ako sieťová brána na opustenie danej súkromnej siete[16]. Obdobne poskytuje funkcionalitu prekladu adries z privátnych do verejných a opačne. Uvedená funkcionalitá sa nazýva Network Address Translation (NAT). Viac o tomto protokole je dostupné na [23].

3. **L2TP koncentrátora** – LAC, zariadenie umiestnené medzi LNS a clientom. Slúži na preposielanie paketov oboma smermi. V smere k LNS vytvára L2TP tunel. LAC server môže byť nasadený aj na PPP termináli a pracovať ako PPPoE[17] server. Zvyčajne je koncentrátor nasadený na NAS, ale je možné ho nasadiť na ľubovoľné sieťové zariadenie.

[11] z ang. *Request For Comments*
[12] z ang. *Cisco Layer 2 Forwarding Protocol*
[13] z ang. *L2TP access concentrator*
[14] z ang. *L2TP Network Server*
[15] z ang. *session*
[16] z ang. *gateway*
[17] z ang. *PPP Over Ethernet*



Obr. 1.11: Proces tunelovanie naprieč L2TP VPN protokolom [24]

V schéme 1.11 je znázornené ako sú spomenuté zariadenia zapojené v logickej topológií. Na schéme je znázornený aj proces enkapsulácie PPP paketov, prenosu dát naprieč tunelom a následnej deenkapsulácie. Schéma bola mierne poupravená a prebraná z [24].

L2TP používa namiesto TCP protokolu UDP, ktorého výhodou je rýchlejší prenos bez kontroly prijatia na druhej strane spojenia. Pri vzniku tunela používa UDP port 1701. Iniciátor tohto procesu následne vyberá náhodne z nečinných portov a smeruje nim pakety s portom 1701. Prijímač po prijatí paketu taktiež náhodne určí nečinný port a preposiela ním pakety prijaté iniciátorom. Takto zvolené portové čísla sa používajú až kým nie je komunikácia cez tunel ukončená.

L2TP vytvára 2 druhy spojenia počas vytvárania konektivity medzi LAC a LNS.

- **tunelové spojenie** – z ang. *tunnel connection*, napomáha k nastoleniu viacero tunelov medzi zariadeniami. Pozostáva z jedného alebo viacerých relačných spojení. Žiadosť o vytvorenie vytvára LAC server po prijatí PPP žiadosť od vzdialeného používateľa. LAC a LNS si vymenia informácie potrebné na vznik spojenia ako sú napríklad autentizačné informácie tunela a ID. Po úspešnom vyjednávaní[18] vznikne tunel, ktorý je identifikovateľný pomocou dohodnutého ID.

- **relačné spojenie** – z ang. *session connection*, reprezentuje PPP spojenie naprieč tunelom. Môže vzniknúť až keď je tunel úspešne vytvorený.

[18] z ang. *negotiation*

Po vytvorení oboch spojení následne odchádza k prenosu zapuzdrených PPP paketov naprieč týmto tunelom.

L2TP ponúka kompatibilitu pre mnohé platformy a jednoduchú konfiguráciu. OS Windows, Linux a Mac majú tento protokol zabudovaný v sebe. Výhodou je použitie UDP, vďaka tomu je možné protokol používať aj v nestabilnom sieťovom prostredí. Nevýhodou je zníženie prenosovej rýchlosti. L2TP používa taktiež vopred zdieľané kľúče a v prípade ak sa nezhodujú, tak dochádza k zastaveniu chodu. L2TP je podporuje iba limitovaný počtet portových čísel. Samotná implementácia L2TP neposkytuje, respektíve nezabezpečuje žiadne šifrovanie alebo autentizáciu paketa. Na zabezpečenie sa používa v kombinácií s iným protokolom. Veľmi známa je implementácia L2TP/IPSec.

V roku 2005 vznikla 3. verzia protokolu, ktorá priniesla zväčšenie dĺžky ID v L2TP hlavičke, zo 16 na 32 bitov. Rozšírenie tunelového autentizačného mechanizmu a oddelenie L2TP dát súvisiacich s PPP protokolom. Viac o tomto upravenom protokole je možné nájsť na [25].

Viac informácií o L2TP problematike je možné nájsť v [26], [21], [24], odkiaľ boli informácie z tejto podkapitole čerpané.

1.4.3 Internet Protocol Security (IPSec)

IPSec je otvorený štandard. Vďaka tomu vďačí za veľkú popularitu a pravidelné aktualizácie kryptografických algoritmov. Protokol sa využíva na zabezpečenie bezpečnosti v sieti. IPSec môžeme klasifikujeme ako L3 protokol nakoľko pracuje s dátami zo sieťovej vrstvy. Má dva režimy:

- **transportný režim** – po prijatí paketu z vyššej vrstvy sú smerovacie dáta zachované a na základe nich sú dáta odosielané ďalej. K zvyšným dátam je pridaná hlavička použitého IPSec protokolu. Princíp pridania sieťových dát k pôvodným je znázornený na 1.12
- **tunelovací režim** – zapuzdruje paket. Pridáva novú IP hlavičku a hlavičku IPSec protokolu (ESP/AH) k pôvodnému nezapuzdrenému paketu. Uvedená skutočnosť je zobrazená na obrázku 1.13

Obr. 1.12: IPSec transportný režim

Obr. 1.13: IPSec tunelovací režim

Za účelom poskytnutie zabezpečeného spojenia, protokol vykonáva autentizáciu, šifrovanie a vyjednávanie, resp. výmenu potrebných kľúčov. Jednotlivé činnosti sú realizované pomocou týchto IPSec protokolov:

- **Autentizačná hlavička** – z ang. *Authentication Header*, pridáva k prepravovanému paketu dáta na zabezpečenie dátovej integrity a pôvodu. Chráni proti z ang. *Replay attack* [27]. Dáta v tomto režime nie sú šifrované. AH hlavička je vygenerovaná v závislosti od toho v akom IPSec režime je protokol použítý.

- **Bezpečnostné zapuzdrenie nákladu** – z ang. *Encapsulating Security Payloads*, narozdiel od AH, ESP aj šifruje dáta z vyššej vrstvy. Vďaka tomu dochádza k zabezpečeniu dôvernosti, datovej integrity a autentizácie pôvodu. Výsledná hlavička závisí od dát zo sieťovej vrstvy a konfigurácie IPSec.

- **Internet Security Association and Key Management Protocol** – ďalej ISAKMP, protokol na autentizáciu a výmenu kľúčov. Slúži taktiež na vytvorenie parametra SA, ktorý sa používa v hlavičke AH/ESP.

IPSec je možné nakonfigurovať, tak aby používal AH a ESP selektívne alebo aj súčasne. V závislosti od konfigurácie následne dochádza k zapuzdrovaniu prichádzajúceho paketu. Používateľ má na výber viacero štandardných kryptografických algoritmov. Príkladmi sú AES [28], RSA [29], Diffie-Hellman [30] a eliptické krivky (ECDSA [31] aj ECDH [32]).

Nabudúce sa opytat ci biks predmet moze byt zdroj alebo ine. [6]

1.4.4 Secure Socket Tunneling Protocol (SSTP)

SSTP je bežný L2 VPN protokol, ktorý zapuzdruje PPP ramce cez HTTPS protokol [33]. Spolieha sa na SSL, resp. TLS, ktoré sú opísané v úvode práce. Vďaka tomu umožňuje ľahší prechod cez väčšinu firewallov a proxy brán. Teda blokovanie takto vytvoreného VPN spojenia je pre poskytovateľov internetu a správcov siete zložitejšie.

SSTP bol vytvorený v roku 2007 spoločnosťou Microsoft. Primárne pre platformu Windows. Cieľom bolo poskytnúť bezpečnejšiu náhradu za PPTP a L2TP.

V súčasnosti sa považuje za jeden zo štandartných protokolov. Je dostupný vo viacerých operačných systémoch vrátane Linuxu a BSD. Pravidelne udržiavaný o čom svedčia aj priebežná aktualizácia dokumentácie na Microsoft dokumentačných stránkach. Informácie k tejto podkapitole boli čerpané z [34]. V uvedenom zdroji je možné nájsť podrobnejšie informácie o protokole.

Proces enkapsulácie s využitím protokolov je znázornený pomocou schémy 1.14. Pri vytváraní SSTP segmentu dochádza k zapuzdreniu PPP rámcov pomo-

Obr. 1.14: Proces enkapsulácie pôvodných PPP rámcov naprieč SSTP protokolom

cou HTTPS s využitím TCP protokolu s portovým číslom 443. Po úspešnom nadviazaní TCP a overení SSL/TLS spojenia dochádza k spracovaniu SSTP hlavičky. Po úspešnom odstránení sa následne získa prístup k pôvodnému PPP rámcu.

Podobne ako tomu bolo v prípade L2TP protokolu, opísaného vyššie, tak z hľadiska premávky putujú naprieč tunelom dva druhy paketov. Vytvára a odosiela ich klient aj server. Majú špecifický formát a musia byť prenášané po bajtoch a v sieťovom poradí bajtov[19], z ľava do prava, teda od najvýznamnejšieho bitu po najmenej významný[20]. Prvé 4 parametre v oboch hlavičkách sú rovnaké:

1. **Verzia** – z ang. *Version*, má veľkosť 8-bitov, používa sa pri komunikácií a vyjednávaní SSTP verzie, ktorá sa má použiť. Prvé 4 bity signalizujú majoritnú verziu a zvyšné minoritnú verziu.

2. **Reservované** – z ang. *Reserved*, 7 bitov nastavených na 0, rezervované pre budúce použitie, pri spracovaní sa ignorujú.

3. **C** – 1 bit, indikátor pre dátový (0) a kontrolný SSTP paket (1).

4. **Dĺžka paketu** – z ang. *LengthPacket*, 16-bitový parameter, pozostáva z:

   - **R** – 4 bity, pripravené na budúce použitie, nastavené na 0 a ignorované,
   - **Dĺžka** – z ang. *Length*, 12 bitov, špecifikuje bajtovú veľkosť celého SSTP paketu

Rozdielnosť medzi hlavičkami vzniká v:

- **Dátové SSTP pakety** – z ang. *SSTP Data Packets* [35], obsahujú 5. parameter

[19] z ang. *network byte order*

[20] sieťové spracovanie dát známe aj ako z ang. *Big Endian*

  - **5. Dáta** – pole variabilnej dĺžky. Obsahuje zapuzdrený náklad z vyššej vrstvy.

- **Kontrolné SSTP pakety** – z ang. *SSTP Control Packets* [36], obsahuje 3 dodatočné parametre:

  - **5. Typ správy** – z ang. *Message Type*, 16-bitové pole s SSTP správou o stave spojenia. Celkovo 9 možných správ [36].
  - **6. Číselné atribúty** – z ang. *Num Attributes*, 16-bitové pole, ktoré špecifikuje počet parametrov v správe
  - **7. Atribúty** – z ang. *Atributes*, zoznam parametrov s variabilnou veľkosťou.

Protokol vo všeobecnosti nepodporuje spojenie typu sieť k sieti. Primárne je orientovaný na pripojenie klienta k sieti, za účelom získania vzdialeného prístupu. To isté platí pri autentizácií. Je možné autentizovať iba používateľa, iné možnosti nie sú podporované (zariadenie, smart card, počítať,...). V prípade nestabilného spojenia, je možný vysoký výskyt straty paketov. Dôvodom je použitie TCP protokolu.

Bezpečnosť SSTP je sprostredkovaná za pomoci HTTPS protokolu, ktorý používa SSL/TLS. Aplikované kryptografické algoritmy závisia od verzie SSL/TLS, ktorá je použitá v implementácií. TLS bol predmetom opisu v úvode práce. SSTP sa považuje za veľmi bezpečný protokol. Na druhej strane použitie robustných šifrovacích a autentizačných algoritmov, dosť spomaľuje výsledné SSTP VPN pripojenie.

Viac informácií o protokole môže čitateľ nájsť v [34], odkiaľ boli aj informácie pre túto podkapitolu čerpané.

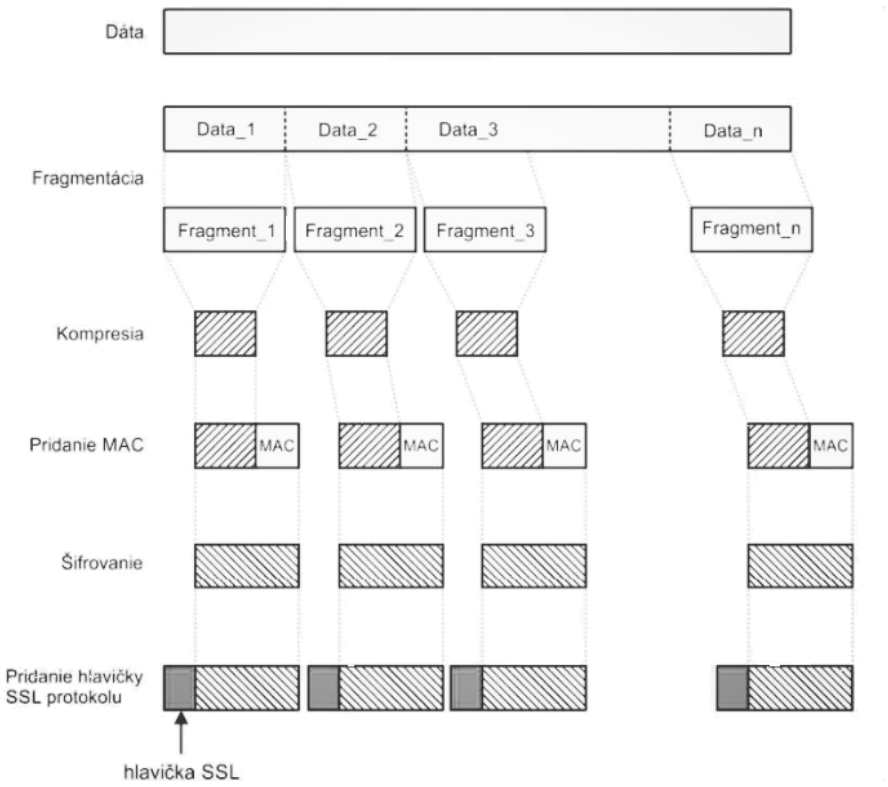
1.4.5 Transport Layer Security – TLS

TCP ani UDP protokol sam o sebe nezabezpečí dáta, ku ktorým sa pridáva hlavička. Dôsledkom toho vznikli viaceré protokoly slúžiace na autentizované šifrovanie aplikačných dát. Najznámejší je protokol zabezpečenia prenosu – TLS.

Zabezpečenie dát bolo prvotne vykonávané pomocou protokolu *Secure Sockets Layer* – SSL. Tento spôsob používa certifikáty na odšifrovanie dát. SSL malo od svojho vytvorenia dlhý vývoj, ktorý smeroval až k doteraz najpoužívanejšiemu TLS vo verzii 1.3. Inými slovami, TLS protokol je nástupca SSL pričom obsahuje rôzne úpravy a vylepšenia najmä z hľadiska rýchlosti. Zároveň sa v dnešnej dobe

neodporúča používanie SSL protokolu. Dôsledkom optimalizácií je, že klienta komunikujúci so serverom cez HTTPS protokol s TLS 1.3 je rýchlejší ako v prípade použitia nešifrovaného HTTP variantu.

TLS pracuje na pomedzi aplikačnej a transportnej vrstvy. Spôsob spracovania dát je zobrazený pomocou schémy 1.15 s SSL operáciami, prebratej z [6]. Postup v SSL a TLS ostáva zachovaný.



Obr. 1.15: Prehľad operácií v SSL protokole [37]

Aplikačné dáta sa rozdelia na menšie fragmenty. Následne sa vykoná kompresia pomocou kompresného algoritmu. Z týchto dát sa vypočíta a pridá autentizačný kód správy (ďalej MAC [**mac**]), taktiež označovaný ako tag. Na uvedených dáta sa vykoná šifrovanie a nakoniec sa pridá TLS hlavička.

TLS sa v niektorej literatúre zvykne označovať aj ako VPN protokol. Z hladiska funkcionality VPN však s týmto tvrdením nemôžeme úplne súhlasiť. TLS spracúva dáta konkrétnej aplikácie. Jeho úlohou je zabezpečenie aplikačných dát pri bežnom prenose. Z hľadiska VPN by sme však mohli tunelovať naprieč sieťou len jednú konkrétnu aplikáciu. Ako vidíme jedná sa o veľmi špeciálny prípad. Situácia sa však mení v prípade implementácie TLS protokolu do aplikácie ako je napríklad webový prehliadač. V takomto prípade by sme za pomoci manipulácie dát vychádzajúcich z prehliadača dokázali sprostredkovať činnosť VPN. Ako príklad realizácie by mohli byť rôzne typy rozšírení, ktoré si môžeme do prehlia-

dačov doinštalovať v podobe pluginov. Z tohto dôvodu zaradíme tento protokol vrámci rozdelenia VPN sieti do L4 kategórie aj napriek faktu, že sa nejedná o tunelovací protokol.

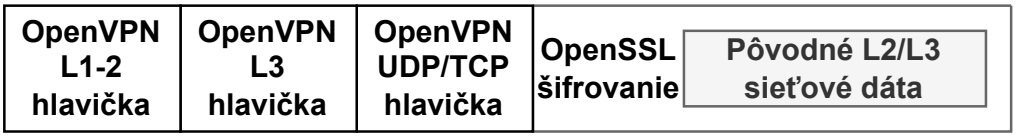
Viac informácií o TLS protokole, jednotlivých verziách a optimalizáciách je možné nájsť na [37].

1.4.6 Ostatné populárne VPN protokoly

Vrámci tejto práce si predstavíme ešte dvojicu veľmi populárnych VPN riešení. Koncepčne sú riešenia pri vytváraní tunelu a prenose správ podobné vyššie opísaným protokolom. Z uvedeného dôvodu protokoly len stručne charakterizujeme.

OpenVPN

OpenVPN je jeden z najznámnejších voľne dostupných VPN tunelovacích protokolov. Vznikol v roku 2001. Dostupné ako firmvérové riešenie pre výrobcov sieťových zariadení. Pre používateľa je dostupná vo forme softvéru, ktorý je potrebné nainštalovať na cieľovú platformu. Ponúka aj grafické rozhranie. Zabezpečené spojenie naprieč internetom je sprostredkovane prostredníctvom SSL/TLS protokolu. Dokáže pracovať s dátami na L2 aj L3 vrstve v závislosti od konfigurácie. Vďaka voľnému prístupu do zdrojového kódu poskytuje jednoduché možnosti na skúmanie výsledných riešení, validáciu a prípadnú úpravu podľa potrieb konkrétneho používateľa. Protokol je napísaný v jazyku C. Bezpečnostné prvky su implementované pomocou OpenSSL knižnice. Knižnica v sebe zahŕňa funkcie potrebné na šifrovanie, autentizáciu, výmenu kľúčov a mnoho ďalšieho. Na obrázku 1.16 je znázornená výstupna štruktúra dát po zapuzdrení.



Obr. 1.16: Zloženie sieťových dát po spracovaní OpenVPN

Aktuálne používa na šifrovanie AES s 256-bitovou veľkosťou kľúča. V súčasnosti sa OpenVPN považuje za najbezpečnejšiu implementáciu VPN dostupnú pre viacero platforiem. Používateľ ma možnosti vo voľbe protokolu TCP alebo UDP. Podporuje aj prácu s IPv6. OpenVPN nie je kompatabilná s ostatnými protokolmi. Je preto potrebné mať implementáciu na serveri aj klientovi. Nevýhodou je pomerne veľké, energeticky a výpočtovo náročné prevedenie.

Viac informácií o tomto komplexnom programe je možné nájsť v [14], [38] a priamo na stránke OpenVPN[21]. Z uvedených zdrojov boli aj informácie čerpané. Zdrojový kód je dostupný napríklad na Githube[22].

WireGuard

WireGuard je najnovší protokol z vyššie uvedených. Pracuje iba na L3 vrstve. Jedná sa o implementáciu s voľne dostupným zdrojovým kódom. Vznikol v roku 2015. Cieľom projektu bolo vytvoriť jednoduchý protokol, ktorý sa ľahko používa, dosahuje vysoké prenosové rýchlosti a poskytuje kvalitnú bezpečnosť pre používateľa. Vo výsledku sa to vývojárom podarilo. Od roku 2020 sa stal súčasťou Linuxového jadra, konkrétne verzie Linux 5.6 kernel. Aktuálne podporuje veľké množstvo OS vrátane Androidu, Windowsu, MacOS, OS založené na BSD [39] a ďalšie. Za úspešne prevedenie vďačí WireGuard implemetovaniu svojich funkcionalít do samotných jadier OS, čo značne zrýchľuje spracovanie dát. Samozrejmostou ostáva aj využitie moderných, extrémne rýchlych kryptografických algoritmov a podpora pre IPv4 a IPv6.

Protokoly použité vo WireGuarde sú:

- **X25519** [40] – zabezpečuje výmenu kľúčov vďaka kryptografii s eliptickým krivkami [41] (ďalej ECC), ponúka 128-bitovú bezpečnosť s veľkosťou kľúča 256-bitov. Považuje sa za jednu z najrýchlejších kriviek v ECC.
- **ChaCha20** [42] – zabezpečuje symetrické šífrovanie.
- **Poly1305** [43] – vytvára autentifikačný kód správy[23]. Veľmi častá je kombinácia ChaCha20-Poly1305 za účelom autentizovaného šifrovania s pridruženými dátami[24].
- **SipHash** [44] – algoritmus sa používa na vytvorenie a mapovanie kľúčov k hodnotám v hashovacej tabuľke. Hashovacie tabuľky sú známy pojem v oblasti dátových štruktúr. Ich hlavnou výhodou je vysoká rýchlosť v porovnaní s ostatnými možnosťami.
- **BLAKE2s** [45] – hashovacia funkcia, rýchlejšia než aktuálne štandardy z rodiny SHA.
- **UDP** – protokol na prepravu zapuzdrených dát.

[21] https://openvpn.net/faq/what-is-openvpn/
[22] https://github.com/OpenVPN/openvpn/
[23] z ang. *Message Authentication codes*
[24] z ang. *Authenticated encryption with associated data*

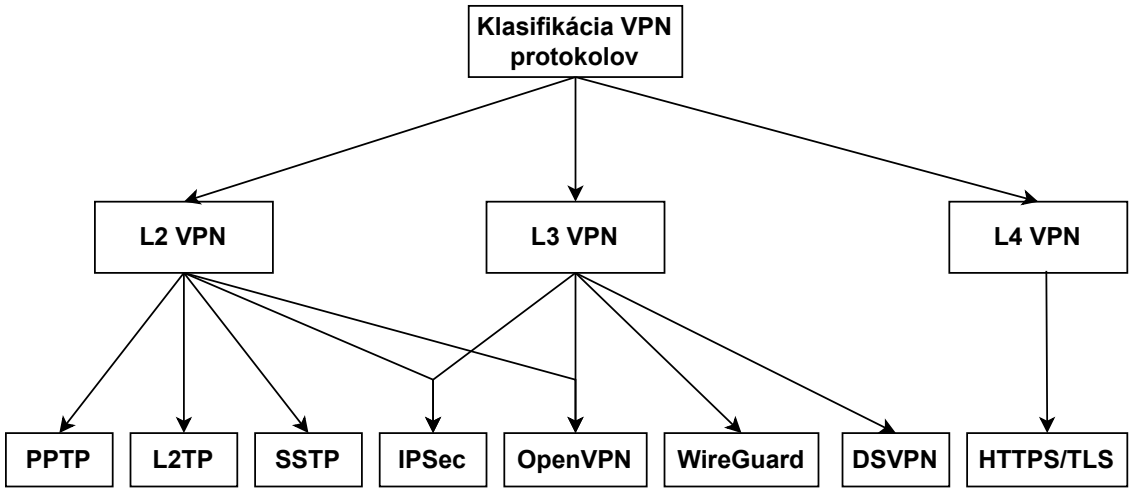
WireGuard podporuje aj použitie vopred zdieľaného kľúča za účelom symetrického šifrovania. Dôvodom je pokrok v oblasti kvantových počítačov, ktoré predstavujú riziko pre algoritmy založené na asymetrickom šifrovaní.

Viac informácií o tomto protokole je k dispozícií na [46] a [47], odkiaľ boli informácie aj čerpané. Zdrojový kód je rozdelený do viacerých repozitárov. Zoznam je dostupný na stránkach Wireguardu[25].

1.5 Zhrnutie VPN sieti

V súčasnosti je trend, využívanie VPN na súkromné účely. Dôvodov môže byť viacero. Napríklad prístup k lokálne blokovaným doménam, anonymita v internetovom prostredí, zabezpečenie pripojenia a iné. V tomto prípade vstupujú do popredia tzv. VPN poskytovatelia (z ang. *VPN Providers*). Ktorý za poplatok poskytujú výhody pripojenia cez VPN k verejnej sieti. Pre čitateľa dávame do popredia bezplatnú implementáciu VPN, VPNHood s voľne prístupným kódom[26].

V schéme 1.17 sme si pre čitateľa pripravili zaradenie charakterizovaných protokolov do klasifikácie VPN. Do schémy sme zaradili aj DSVPN, ktorá bude predmetom analýzy v ďalších častiach práce.



Obr. 1.17: Klasifikácia spomenutých VPN protokolov

[25]https://www.wireguard.com/repositories/

[26]dostupné na `https://github.com/vpnhood/VpnHood`

2 Ľahká kryptografia

V počítačovej sfére sa do popredia dostávajú zariadenia, ktorých hardvérové prostriedky a ponúkaný výkon sú podstatne nižšie ako v prípade bežne dostupných zariadení na domáce, resp. komerčné použitie. Príkladom môžu byť IOT zariadenia, senzorové uzly, mikro-ovládače a podobné. Dôležitou vlastnosťou je aj komunikácia medzi sebou alebo inými zariadeniami. V dôsledku toho je nutné riešiť aj otázku zabezpečenia, resp. bezpečnosti takého prenosu dát.

Kryptografia je oblasť počítačovej vedy, ktorá sa zaoberá práve spomenutou problematikou. Hlavným cieľom je utajiť správu pri jej ceste z bodu A do B. Teda od odosielateľa (tvorcu) správy, až k jej prijímateľovi. Dôsledkom tohto úkonu dochádza k zabezpečeniu 3 hlavných úloh kryptografie:

- **ochrana osobných údajov** (dôvernosť) – z ang. *Data Privacy*,
- **autenticita údajov** (prišla z miesta, kde sa uvádza) – z ang. *Data Authenticity*,
- **integrita údajov** (nebolia upravená počas prenosu) – z ang. *Data Integrity*.

Pojem kryptografia vznikol už pomerne dávno a bol už viackrát charakterizovaný. Z toho dôvodu sa tejto problematike ďalej nebudeme venovať. Ak by však bol čitateľ v tejto oblasti nový, odporúčame najprv načerpať viac informácií. Kvalitné spracovanie problematiky je možné nájsť v [50].

V súčasnosti má používateľ možnosť vybrať si zo širokej ponuky kryptografických algoritmov. Výber je volený na základe potrebnej funkcionality, ktorú sa snažíme implementovať. Napríklad za účelom symetrického šifrovania by sme mohli použiť AES [28], ktorý je aktuálne používaný ako štandardný kryptografický algoritmus. Detailný opis jednotlivých blokov a postupov použitých v AES-e, je obsahom rôznych vydaní. Viac informácií o problematike nájde čitateľ v [50].

V prípade zariadení spomenutých v úvode kapitoly, však môže nastať problém s výpočtovým výkonom pri realizácií algoritmov. Vo výsledku trvá vykonávanie funkcionality omnoho dlhšie ako v prípade normálnych zariadení. V roku

2005 bol prvýkrát definovaný pojem, z ang. *Lightweight Cryptography*. Konkrétne v práci [51]. Vrámci tejto práce voľne preložíme tento pojem ako **Ľahká kryptografia** (ďalej LWC).

LWC algoritmy sú mnoho násobne efektívnejšie ako súčasne používané konvenčné kryptografické štandardy. Dôvodom je ich vysoká rýchlosť a nízky počet potrebných procesorových inštrukcií na vykonanie cielenej funkcionality. V súčasnosti je častým javom použitie označenia *lightweight* pre ľubovoľný kryptografický algoritmus. Jedná sa o implementácie, ktoré svojimi vlastnosťami spĺňajú základné požiadavky LWC. Tie boli definované samotnými autormi už v diele [51]. Sú nimi:

- **minimalizácia spotreby zdrojov zariadenia** – veľkosť kódu, používaných dát a spotreby energie,
- **poskytnutie vysokého stupňa bezpečnosti**,
- **odolnosť voči útoku cez postranné kanály** – z ang. *side-channel attack*, napríklad útokom na analýzu výkonu a na časovanie,
- **jednoduchá implementácia a efektívnosť,**
- **nízka pamäťová stopa** – z ang. *low memory footprint*.

V práci budeme tieto algoritmy označovať ako *Lightweight Cryptographic Algorithm* (ďalej LWCA). LWCA tvoria kryptografické algoritmy, ktoré splňajú vyššie stanovené vlastnosti a teda je možné ich aj nasadiť do tzv. *low resource* zariadení. V závislosti od výslednej implementácie sa sledujú požiadavky na daný algoritmus. V prípade hardvérovej implementácie najväčšiu rolu pri tvorbe optimálneho algoritmu zohráva energetická spotreba a potrebná veľkosť čipu (chip size iba v zdroji – In hardware implementations, chip size and/or energy consumption are the important measures to evaluate the lightweight properties.). Na druhej strane, pri softvérovej implementácií je dôležitým aspektom veľkosť algoritmu a/alebo jeho využitie RAM pamäte. Čím sú uvedené parametre nižšie, tým výhodnejší je nasadenie daného algoritmu pre cieľové zariadenie. V prípade slabých zariadení sa pri výslednej implementácií LWCA môžeme stretnúť aj s kompromisom medzi ponúknutou bezpečnosťou a efektivitou.

V rámci tejto kapitoly si predstavíme jeden z novších LWCA algoritmov. Konkrétne balíček Xoodyak s kryptografickou permutáciou XOODOO [52] a jeho použitie. Xoodyak sa stal jedným z 10 finalistov v LWC štandardizačnom procese Národného inštitútu pre štandardy a technológie (ďalej NIST) [53]. Aktuálne ešte stále prebieha výber štandardu.

Viac podrobností nájde čitateľ v [51], [54] a [53], odkiaľ boli aj informácie z tejto kapitoly čerpané.

2.1 Kryptografická permutácia XOODOO a jej variácie

Permutácia je operácia, ktorá mení pozíciu vstupných prvkov, tak aby vo výsledku vzniklo nová usporiadaná množina. Kryptografické permutácie sú špeciálne navrhnuté matematické algoritmy, tak aby bolo možné využiť ich za účelom šifrovania a dešifrovania. Operácie vykonané v takýchto permutáciách musia byť invertibilné. Kryptografické permutácie tvoria základný stavebný blok pri následnej tvorbe ďalších kryptografických blokov. Množina základných stavebných blokov sa v kryptografii spoločne označuje ako kryptografické primitíva (ďalej KP). Príkladom môžu byť hašovacie funkcie, generátory náhodných čísel a podobne. Viac informácií nájde čitateľ v [55].

XOODOO je sada 384-bitových kryptografických permutácií parametrizovaných počtom kôl. Funkcia kola/rundy[1] funguje na 12 slovách[2] po 32 bitoch. Vďaka tomu je efektívna aj na menej výkonných procesoroch nižšej triedy. Vytvoril ju tím Keccak, ktorý stojí za viacerými úspešnými kryptografickými algoritmami. Napríklad hashovacie funkcie z rodiny SHA-3 a iné – [56]. XOODOO algoritmus vznikol po vytvorení tzv. Kravatte autentizačno-šifrovacieho algoritmu [57], založené na Keccak-p permutácií [58]. Ten sa ukázal ako dostatočne rýchly na širokom spektre platforiem. Avšak nezapadá do kategórie LWCA.

Tím Keccak vypracoval nové riešenie. Ním bol port medzi ich prvotným Keccak-p dizajnom a Gimli-ho [59] permutačným algoritmom. Vo výsledku autori zlúčili lepšie realizované prvky z oboch algoritmov do jedného celku. Primárny problém samotnej Gimli permutácie bol v slabom prejave zmeny výstupu po malých zmenách vo vstupnej správe. Táto vlastnosť sa v kryptografii označuje pomocou anglického pojmu, tzv. *propagation properties*[3]. Novo-vzniknuté riešenie autori pomenovali XOODOO. Na základe rôznych variácií tohto kryptografického primitíva sa im následne podarilo vytvoriť sadu vysoko efektívnych kryptografických funkcií. Medzi sady, ktorých jadro tvorí XOODOO, patrí Xoodyak a Xoofff. Xoofff pozostáva zo zlúčenia Farfalle konštrukcie [60] so XOODOO permutáciou.

[1] z ang. *round*

[2] z ang. *words*

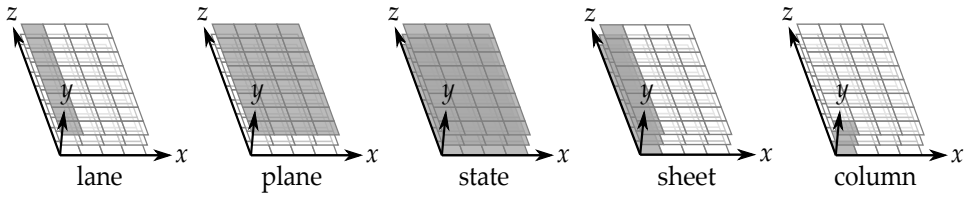
[3] Cieľom je aby aj zmena jedného bitu na vstupe, ovplyvnila čo najviac bitov vo výstupe – tzv. *Lavínový efekt*

Xoodyak ma narozdiel od Xoofff duplexovú konštrukciu [61]. Vo výsledku máme ľahko prenosnú, všestrannú, kryptografickú knižnicu. Je vhodná do výkonovo obmedzených prostredí. Môže sa použiť pre väčšinu kryptografických funkcií, ktoré používajú symetrický kľúč. Napríklad hashovanie, šifrovanie, výpočet MAC alebo autentizované šifrovanie. O kvalite riešenia napovedá aj fakt, že sada Xoodyak je jedným z 10 finalistov v oblasti ľahkej kryptografie NIST štandardizačného procesu. V rámci tejto kapitoly opíšeme kryptografické primitívum XOODOO a následne balík Xoodyak. Informácie o téme boli čerpané z týchto zdrojov: [52], [62], [63], [64], [65],[66].

2.1.1 XOODOO permutácia

XOODOO je permutácia, definovaná počtom rúnd. Má klasickú iteračnú štruktúru. Teda opakovane sa vola rundová funkcia s aktuálnym stavom. Pre pochopenie operácií je nutné pochopiť určité označenie použité v algoritme.

Stav – **state**, pozostáva z 3 rovnako veľkých horizontálnych rovín – **planes**[4]. Každá z týchto rovín obsahuje štyri paralelne 32-bitové pruhy – **lanes**)[5]. Okrem tejto charakteristiky je možné opísať stav ako množinu zloženú zo stĺpcov – **columns**[6], pričom jeden stĺpec obsahuje 4 bity v jednej rovine. Stav je teda tvorený zo stĺpcov usporiadaných v poli o rozmere $4 \times 3 \times 32 = 384$ bitov. Posledná položka na opis stavu sú tzv. listy – **sheets**[7]. List sa skladá z 3 na sebe uložených pruhov. Uvedené pojmy sú znázornené pomocou schémy 2.1, ktorá bola prebraná z [63].



Obr. 2.1: Grafické znázornenie terminológie využitej v kryptografickej permutácii XOODOO [63]

Roviny majú index y. Index $y = 0$ zodpovedá spodnej rovine a vrchná rovina má index $y = 2$. Bit je označený s indexom z vrámci množiny pruhov. List ozna-

[4]V jednej rovine je 128 bitov
[5]V jednom pruhu je 32 bitov
[6]V jednom stĺpci je 12 bitov
[7]V jednom liste je 96 bitov

čujeme pomocou indexu x. Pozícia pruhu v stave je definovaná pomocou dvoch súradníc (x, y). Konkrétny bit je možné reprezentovať v stave pomocou trojice súradníc (x, y, z). Pri určení stĺpca sú potrebné 2 súradnice (x, z). Pred spustením samotného algoritmu musí používateľ vykonať mapovanie 384-bitovej správy voči horizontálnym rovinám. Tento úkon sa realizuje pomocou vzorca 2.1.

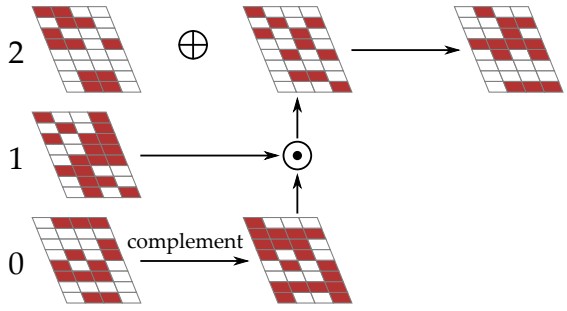
$$i = z + 32(x + 4y) \tag{2.1}$$

Výhodou XOODOO je, že celý stav, 384 bitov, dokáže byť uložený v 12 registroch po 32 bitov. Vďaka tomu ideálne vyhovuje nízko výkonným 32-bitovým zariadeniam.

Rundová funkcia pozostáva z 5 krokov:

1. miešanie vrstvy – z ang. *a mixing layer* θ,
2. posun rovín – z ang. *a plane shifting* ρ_{west},
3. pridanie rundových konštánt – z ang. *the addition of round constants* ι,
4. nelineárna vrstva – z ang. *a non-linear layer* χ,
5. posun rovín – z ang. *an another plane shifting* ρ_{east}.

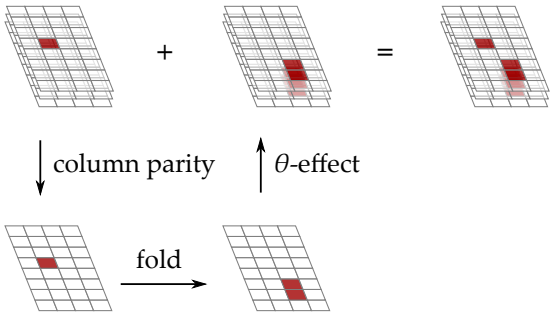
Opis jednotlivých krokov je znázornený pomocou schémy 2.2, 2.3, 2.4 ktoré sú prebraté z [63].



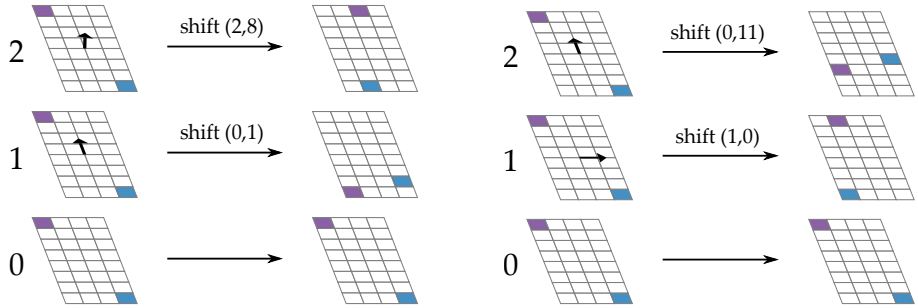
Obr. 2.2: Grafické znázornenie operácie χ [63]

K spomenutej schéme 2.4, by sme rád doplnili, že posun je realizovaný na každom bite. Na schéme sú ilustrované len posuny 2 bitov.

Tabuľka 2.5 vysvetľuje jednotlivé operácie, ktoré sa v algoritme používajú. Algoritmus permutácie je následne znázornený pomocoou 2.6. Súčasťou implementácie sú aj tak zvané rundové konštanty. Tie je možné vidieť v 2.1. Uvedené ilustrácie boli prebrané z dokumentu [63].



Obr. 2.3: Grafické znázornenie operácie ρ [63]



Obr. 2.4: Ilustrácia miešania vrstiev ρ_{west} (vľavo) a ρ_{east} (vpravo)[63]

A_y	Plane y of state A
$A_y \lll (t, v)$	Cyclic shift of A_y moving bit in (x, z) to position $(x + t, z + v)$
$\overline{A_y}$	Bitwise complement of plane A_y
$A_y + A_{y'}$	Bitwise sum (XOR) of planes A_y and $A_{y'}$
$A_y \cdot A_{y'}$	Bitwise product (AND) of planes A_y and $A_{y'}$

Obr. 2.5: Charakteristika operácií v algoritmickom zápise kryptografickej permutácie XOODOO [63]

2.1.2 Kryptograficý balíček Xoodyak

Xoodyak možno považovať za všestranný kryptografický nástroj. Je vhodný pre väčšinu operácií využívajúcich symetrický kľúč. Napríklad generovanie pseudonáhodných bitov, autentizáciu, šifrovanie a iné. Tím Keccak použil pri návrhu duplexnú konštrukciu. Konkrétne variant s plným stavom a využitím kľúča. Tento dizajn označujeme ako Full-State Keyed Duplex (FSKD). Viac o tejto konštrukcii si čitateľ môže prečítať v [61]. Operačný režim, v ktorom Xoodyak pracuje sa nazýva Cyklista – z ang. *Cyclist*. Tento názov získal ako opozitum k pomenovaniu režimu Motorista, ktorý je možné nájsť v Keyak schéme [67]. Narozdiel od

Definícia XOODOO permutácie s počtom rúnd n_r

Parametre: Počet rúnd n_r

for rundový index i from $1 - n_r$ to 0 **do**

$A = \mathrm{R_i}(A)$

$\mathrm{R_i}$ **je špecifikovaná nasledujúcou sekvenciou krokov:**

θ :

$$P \leftarrow A_0 + A_1 + A_2$$
$$E \leftarrow P \lll (1,5) + P \lll (1,14)$$
$$A_y \leftarrow A_y + E \text{ for } y \in \{0,1,2\}$$

ρ_{west} :

$$A_1 \leftarrow A_1 \lll (1,0)$$
$$A_2 \leftarrow A_2 \lll (0,11)$$

ι :

$$A_0 \leftarrow A_0 + \mathrm{C}_i$$

χ :

$$B_0 \leftarrow \overline{A_1} \cdot A_2$$
$$B_1 \leftarrow \overline{A_2} \cdot A_0$$
$$B_2 \leftarrow \overline{A_0} \cdot A_1$$
$$A_y \leftarrow A_y + B_y \text{ for } y \in \{0,1,2\}$$

ρ_{east} :

$$A_1 \leftarrow A_1 \lll (0,1)$$
$$A_2 \leftarrow A_2 \lll (2,8)$$

Obr. 2.6: Algoritmický zápis kryptografickej permutácie XOODOO [63]

i	c_i
0	0x00000012
-1	0x000001A0
-2	0x000000F0
-3	0x00000380
-4	0x0000002C
-5	0x00000060
-6	0x00000014
-7	0x00000120
-8	0x000000D0
-9	0x000003C0
-10	0x00000038
-11	0x00000058

Tabuľka 2.1: Súbor rundových konštánt kryptografického algoritmu XOODOO [63]

uvedeného balíka Keyak, nie je Xoodyak limitovaný len na autentizované šifro-

vanie. Je jednoduchší hlavne kvôli tomu že neobsahuje paralelné varianty.

Režim Cyklista

Režim Cyklista funguje na princípe kryptografických permutácií f, teda zmeny usporiadania bitov za pomoci tajného kľúča a matematických operácií. Parametrami sú veľkosti blokov R_{hash}, R_{kin}, R_{kout} a veľkosť račety, resp. západky[8] [68] $\ell_{ratchet}$. Uvedený pojem sa v kryptografii používa vo forme obrazného pomenovania. Cieľom je poukázať na jednoduchý pohyb vpred, ale s ťažkým, resp. zložitejším pohybom naspäť. Dôležité je, že uvedený scenár je vyvolaný zámerným dizajnom. Šírka permutácie b' je definovaná pomocou vzorca 2.2. Všetky uvedené parametre sú v bajtoch. Pre označenie prázdneho slova budeme používať E.

$$max(R_{hash}, R_{kin}, R_{kout}) + 2 \leq b' \tag{2.2}$$

Cyklista operuje v dvoch režimoch – **hašovací a kľúčový**[9]. Inicializácia prebieha pomocou príkazu `CYCLIST(K,id,counter)`. Ak sa parameter K rovná prázdnemu slovu E, tak potom nastane spustenie v hašovacom režime. Aktuálne nie je do implementácie zakomponovaná možnosť zmeny režimu po inicializácií. Vývojári však túto vlastnosť nevylúčili pre prípadné aktualizácie balíka.

Dostupné funkcie závisia od režimu, v ktorom sa Cyklista spúšťa. Medzi ne patria `ABSORB()` a `SQUEEZE()`. Možno ich volať v oboch režimoch, zatiaľ čo funkcie `ENCRYPT()`, `DECRYPT()`, `SQUEEZEKEY()` a `RATCHET()` sú dostupné len pre kľúčový režim. Účel každej funkcie je nasledujúci:

- `ABSORB(X)` absorbuje vstupný reťazec X,
- $C \leftarrow$ `ENCRYPT(P)` zašifruje P do C a absorbuje P,
- $P \leftarrow$ `DECRYPT(C)` dešifruje C do P a absorbuje P,
- $Y \leftarrow$ `SQUEEZE(L)` vytvára L-bajtový výstup, ktorý závisí od doteraz absorbovaných dát,
- $Y \leftarrow$ `SQUEEZEKEY(L)` funguje ako `SQUEEZE(L)`, ale používa sa za účelom generovania odvodeného kľúča,
- `RATCHET()` transformuje stav na nevratný tak, aby sa zabezpečila dopredná bezpečnosť[10] [69].

[8] z ang. *the ratchet size*
[9] z ang. *hash and keyed mode*.
[10] z ang. *Forward secrecy*

Stav bude závisieť od postupnosti volaní funkcií a od jeho vstupných reťazcov. Presnejšie povedané, zámerom je, že akýkoľvek výstup závisí od postupnosti všetkých vstupných reťazcov a volaní, tak že akékoľvek dva nasledujúce výstupné reťazce budú výstupom rôznych domén. Napríklad volanie `ABSORB(X)` znamená, že výstup bude závisieť od reťazca X. Na druhej strane `ABSORB()` vo funkcii `ENCRYPT(P)` vytvorí výstup závislý aj od P z funkcie šifrovania. Okrem uvedených závislostí ovplyvňujú výstup aj iné dizajnové riešenia. Príkladom je minimalizácia pamäťovej stopy. Vo výsledku teda výstup závisí od počtu predchádzajúcich volaní funkcie `SQUEEZE()` a predtým spracovaných textov pomocou funkcií `ENCRYPT()` a `DECRYPT()`. Viac informácií o režime je dostupných v kapitole 7.2, publikácie [63]. Algoritmický zápis jednotlivých funkcií a doplňujúce informácie o režime Cyklista je možné nájsť v [70], konkrétne v kapitole 2.2.

Definícia a bezpečnosť

Xoodyak je definovaný pomocou operatívneho režimu Cyklista nasledovne:

$$CYCLIST[f, R_{hash}, R_{kin}, R_{kout}, L_{ratchet}] \tag{2.3}$$

Kde jednotlivé parametre majú veľkosti:

1. f – permutácia XOODOO so šírkou 48 bajtov $(384$ bitov$)$,
2. R_{hash} – 16 bajtov,
3. R_{kin} – 44 bajtov,
4. R_{kout} – 24 bajtov,
5. $L_{ratchet}$ – 16 bajtov.

Takto definované parametre algoritmu dokážu poskytnúť 128-bitovú bezpečnosť v oboch režimoch Cyklistu. Samozrejmosťou je, že v prípade kľúčového režimu, musí byť veľkosť kľúča rovná alebo väčšia ako 128 bitov. Viac informácií o kryptografickej bezpečnosti algoritmov je možné nájsť v [71].

Viac informácií o bezpečnosti Xoodyak-a je možné nájsť v [**7.3**, 63] a [70], odkiaľ boli informácie čerpané.

2.1.3 Možnosti použitia Xoodyak algoritmu

Obsahom tejto podkapitoly sú uvedené postupy ako a za akých okolností je daný balík možné použiť.

Použitie hašovacieho režimu

Xoodyak sa dá aplikovať ako hašovacia funkcia. Konkrétne je možné ju použiť ako funkciu na rozšírenie výstupu[11] (ďalej XOF). Nominálne, resp. nie bežné použitie by v tomto prípade bolo nasledujúce:

```
CYCLIST(E,E,E) //spustenie v hašovacom režime,
ABSORB(X)      //absorpcia vstupného reťazca X,
SQUEEZE(L)     //vytvára L-bajtový výstup,
               //závislý od doteraz absorbovaných dát.
```

V tomto prípade by sa kryptografická bezpečnosť algoritmu pohybovala v závislosti od veľkosti výstupu L. Konkrétne v intervaloch:

- **odolnosť voči kolízií** – z ang. *collision resistance* [72], min(8L/2, 128) bitov,
- **odolnosť voči** – z ang. *preimage and second preimage resistance* [73], min(8L, 128) bitov,
- **odolnosť voči** – z ang. *m-target preimage resistance* [73], min(8L - log m, 128) bitov.

Bežná je však absorpcia sekvencie viacerých reťazcov.

Použitie kľúčového režimu

Inicializácia režimu začína použitím príkazu `CYCLIST(K,id,counter)`. Autori uviedli celkovo 6 spôsobov použitia. V nich sa opisuje význam parametrov id, $counter$ $nounce$ a podobne aj možnosti použitia funkcií, spomenutých na začiatku podkapitole.

- **Použitie na zabránenie viac-cieľového útoku** – z ang. *Two ways to counteract multi-target attacks*,
- **Tri spôsoby spracovania jednorázového kľúča** – z ang. *Three ways to handle the nonce*,
- **Autentizované šifrovanie** – z ang. *Authenticated encryption*,
- **Autentizované relačné šifrovanie** – z ang. *Session authenticated encryption*,
- **Použitie funkcie `RATCHET()`** – z ang. *Ratchet*,
- **Pohyblivé subkľúče** – z ang. *Rolling subkeys*,

[11] z ang. *eXtendable-Output Function*

Použitie na zabránenie viac-cieľového útoku – id parameter je voliteľný identifikátor kľúča K. Pre každý tajný kľúč by mal byť jedinečný. Ponúka možnosť zabránenia viac-cieľovým útokom. Jedná sa o útok na viacero používateľov daného kryptografického systému, resp. viacero kľúčov používateľa. Viac o tomto útoku je možné nájsť v [74]. V prípade použitia id parametra algoritmus rozšíri veľkosť kľúča. Následne pri vyhľadávaní kľúčov nedochádza k degradácií bezpečnosti a veľkosť tajného kľúča môže ostať 128 bitov. Týmto spôsobom bude zachovaná aj rovnaká bezpečnosť systému pred útokom. Príklad za účelom šifrovania správy P za pomoci tajného kľúča K s id, je na nasledujúci:

```
CYCLIST(K,id,E)
ABSORB(nounce)
C ← ENCRYPT(P).
```

Tri spôsoby spracovania jednorázového kľúča – 3. parameter pri inicializácií režimu cyklista je *counter*, resp. počítadlo. Jedná sa o dátový prvok vo forme bajtového reťazca, ktorý môže byť inkrementovaný. Spracúva sa po jednej číslici. Vďaka tomu sa obmedzuje počet informácií, ktoré vie útočník využiť pre rôzne vstupy. Pri inicializácií používateľ zvolí veľkosť počítadla v intervale $2 \leq B \leq 256$. Predpokladá sa, že počítadlo je reťazec z množiny $\mathbb{Z}_{\mathbb{B}}^{*}$. Potom ak je počítadlo inicializované ako prázdny reťazec, tak množina všetkých možných hodnôt po inkrementácií je $\mathbb{Z}_{\mathbb{B}}^{\not\perp}$. Pri každom ďalšom navýšení sa zvýši hodnota za $*$. Spracovanie prebieha od najvýznamnejšieho bitu. Čím menšia je hodnota B, tým menší je počet možných vstupov pri každej iterácií permutácie. Vďaka tomu je zabezpečená lepšia ochrana pred tzv. z ang *Power analysis* [75] útokmi a jeho variantami. Za účelom zamedzenia týchto útokov sa používa `ABSORB(nounce)` v prípade ak si počítadlo pri inicializácií zvolíme prázdny reťazec E.

Autentizované šifrovanie – za účelom autentizovaného šifrovanie je Xoodyak možné použiť nasledovne:

```
CYCLIST(K,nonce,E)
ABSORB(A)
C ← ENCRYPT(P)
T ← SQUEEZE(t)
return (C,T)
```

Pričom K je tajný kľúč s pridruženým jednorázovým heslo a dátami A k heslu. Z týchto údajov sa získa autentizačný tag t. Jeho veľkosť je rovná 16-bajtom. V prípade dešifrovania by postupnosť volaní vyzerala nasledovne:

```
CYCLIST(K,nonce,E)
```

```
ABSORB(A)
C ← ENCRYPT(P)
T ← SQUEEZE(t)
if T = T' then
  return P,
else
  return ⊥  //vyjadrenia hodnoty false
```

Kryptografická bezpečnosť takto implementovaného šifrovania je minimom z uvedených hodnôt $min(184, K, 8t)$.

Autentizované relačné šifrovanie – pracuje so sekvenciou správ a autentizačných tagov. V princípe sa opakuje viacero blokov z predchádzajúcej možnosti použitia. Prvé volanie je rovnaké ako v predchádzajúcom príklade. Implementácia však zahŕňa aj možnosť vytvoriť tag bez toho aby bolo nutné použiť funkcie `ABSORB()`, `ENCRYPT()` alebo obe súčasne pred volaním `SQUEEZE()`. Aj napriek tomu bude možné vytvoriť nový tag na základe predchádzajúcich dát, uložených v pamäti.

Použitie funkcie `RATCHET()` – používateľ môže v režime kľúča kedykoľvek volať funkciu `RATCHET()`. Volanie spôsobí prepísanie časti stavu s nulami. Vďaka tomu je nemožné vypočítať stav pred volaním funkcie `RATCHET()`. Týmto spôsobom je možné zťažiť snahu pri pokuse obnoviť vnútorný stav, napríklad pri útoku postrannými kanálmi. `RATCHET()` je možné použiť obdobne aj pri autentizovanom šifrovaní. Konkrétne pred alebo za funkciou vytvárania tagu `SQUEEZE()`.

```
CYCLIST(K,nonce,E)
ABSORB(A)
C ← ENCRYPT(P)
RATCHET()   //príklad volania pred
T ← SQUEEZE(t)
RATCHET()   //príklad volania za
```

Obidva spôsoby majú svoje výhody. Volanie pred vytváraním tagu je najfektívnejšie. Dôvodom je, že vďaka tomu je potrebné len jedno extra volanie permutácie. V prípad volania funkcie `RATCHET()` za `SQUEEZE()` sa šifrovaný text prenáša, funkcionalita `RATCHET()` je vykonaná asynchrónne a algoritmus môže spracovať ďalšiu správu, určenú na šifrovanie.

Pohyblivé subkľúče – je alternatíva k použitiu dlhodobého tajného kľúča K s inkrementovanými pridruženým jednorázovým kľúčom $nounce$ alebo identifikátorom kľúča id. Subkľúč/e sa vytvára/jú pomocou funkcie `SQUEZEEKEY()`. Pri

šifrovaní je teda možné nahradiť procesy inkrementácie a ukladania jednorázových hesiel pri každom použití K, pomocou použitia pohyblivých subkľučov.

```
K_1 ← K and i ← 1 //inicializacia tajneho kluca
while condition do
CYCLIST(K,E,E) //inicializácia novej Xoodyak inštancie
K_{i+1} ← SQUEEZEKEY(ℓ_sub)
RATCHET()       //volitelne
ABSORB(A_i)
C_i ← ENCRYPT(P_i) //sifrovanie
T_i ← SQUEEZE(t)  //vypocet tagu
⟹ output(C_i,T_i) //caka na dalsiu správu
i ← i + 1
```

Parameter ℓ_{sub} je vo veľkosti 32 bajtov (256 bitov). Táto veľkosť by mala byť dostatočná aby sa zabránilo kolíziám za predpokladu, že v subkľúčoch nevznikla kolízia. Použitie týmto spôsobom ponúka odolnosť voči útokom cez postranné kanály. Tajný kľúč nie je po dodávke prvého subľúča používaný. Vďaka tomu nie je ani vystavený možnému útoku. Každým ďalším šifrovaním sa mení aj použitý kľúč na šifrovanie, čo veľmi sťažuje možnosti ďalšej analýzy.

Použitie za účelom autentizovaného šifrovania s bežným heslom

Protokoly na výmenu kľúčov, ako napríklad Diffie-Hellman, poskytujú vo výsledku bežný tajný kľuč. Pred symetrickým šifrovaním je však potrebná jeho úprava. Za týmto účelom je možné použiť Cyklistu v hašovacom režime. Následne po spracovaní bežného kľúča použijeme odvodený kľuč na spustenie režimu kľúča. Funkcionalita je jednotlivých volaní je znázornená nižšie.

```
CYCLIST(E, E, E)      //hasovaci rezim
ABSORB(ID)            //ID pouziteho protokolu
ABSORB(K_A)           //verejny kluc A
ABSORB(K_B)           //verejny kluc B
ABSORB(K_AB)          //tajny kluc
K_D ← SQUEEZE(ℓ)

CYCLIST(K_D, nonce, E) //rezim kluca
ABSORB(A)
C ← ENCRYPT(P)
T ← SQUEEZE(t)
return (C, T)
```

Ak platí $\ell \leq R_{hash}$, tak implementácia dokáže efektívne zreťaziť tajný kľúč K_D

a reinicializovať cyklistu `CYCLIST($K_D$,$E$,$E$)`. Dôvodom je, že tajný kľúč je už súčasťou stavu permutácie. Je len potrebné nastaviť zvyšné parametre. Note also that if at least one of the public key pairs is ephemeral, the common secret KAB is used only once and no nonce is needed. ??[**3.3**, 70]

3 Charakteristika zariadenia, nástrojov a konfigurácia prostredí

Pri praktickej časti tejto práce budeme realizovať naše experimenty na jednom fyzickom zariadení. Jedná sa o osobný prenosný počítač značky Asus. V tabuľke 3.1, je uvedená špecifikácia tohto zariadenia. Nad týmto zariadením budeme následne spúšťať virtuálne obrazy OS pomocou vizualizačných nástrojov. Týmto spôsobom izolujeme činnosť programov od nášho natívneho OS. Počas celej doby experimentovania bude zariadenie pripojené k elektrickej sieti s cieľom dosiahnuť čo najvyšší výkon.

Komponenty	**Zariadenie** **Asus TUF A15**
Model	F506IU-AL006T
Verzia OS	Win 11 Home; 64-bit.; v.22H2
Zostava OS	22621.1555
CPU	AMD Ryzen 7 Mobile 4800H
RAM	16 GB DDR4 2x1600 MHz
Úložisko	SSD OM8PCP3512F-AB

Tabuľka 3.1: Technická špecifikácia použitého fyzického zariadenia

3.1 Nástroje použité v obrazoch a v práci

Pri práci vo virtuálnych obrazoch bolo potrebné pracovať s viacerými programami. V tejto podkapitole si ich stručne predstavíme.

Visual Studio Code

Visual Studio Code, taktiež známe ako VSCode, je odľahčená verzia textového editora. Program je dostupný pre Windows, macOS a Linux. Obsahuje zabudovanú podporu pre JavaScript, TypeScript, Node.js a má bohatý ekosystém rozšírení pre ďalšie jazyky ako napríklad C++, C#, C, Java, Python, PHP, Go, .NET a ďalšie. Jedná sa o bezplatný program, dostupný na webe[1]. VSCode v práci používame ako primárny editor na úpravu, ladenie a preklad kódu.

GCC prekladač a balík Make

Programy, ktoré v práci používané sú napísané v jazyku C. Kvôli potrebe prekladu kódu sme do virtuálnych OS nainštalovali aj prekladače a balík Make na jednoduchý bezstarostný preklad. Na OS Linux je inštalácia pomerne jednoduchá. Stačí zadať do terminálu tieto príkazy.

```
sudo apt install gcc
sudo apt install make
```

V prípade OS Windows používame balíček Winlibs.

Winlibs balík

Ako naznačuje názov jedná sa o balík s knižnicami jazyka C a C++ určený pre OS Windows. Jedná sa o voľne dostupný balík[2]. V prípade použitia je postup inštalácie zložitejší ako na Linuxe. Používateľ musí manuálne balíček stiahnuť. Po stiahnutí musí importovať uvedený balíček, resp. cestu k nemu, do premenných prostredia OS Windows. Jeden zo spôsobov je uvedený aj na Winlibs stránke.

Počas práce sme používali GCC s verziou 12.2.0, MinGW-w64 10.0.0 (UCRT) - release 2. Uvedený balík obsahuje aj program make.

Tunelovacie rozhranie Wintun

Wintun je veľmi jednoduchý a minimalistický ovládač pre vytvorenie tunelovacieho rozhrania v systéme Windows. Wintun komunikuje s jadrom OS a poskytuje používateľovi prístup k sieťovému rozhraniu na zapisovanie a čítanie paketov. Rozhranie sa správa rovnako ako v prípade natívneho Linuxového a BSD ovládača. Wintun bol pôvodné navrhnutý pre implementáciu VPN protokolu WireGuard. Autori sa však rozhodli zverejniť túto časť samostatne čím otvorili cestu

[1] `https://code.visualstudio.com/download`

[2] dostupne na https://winlibs.com/

experimentovaniu v L3 sieťovej vrstve na OS Windows. Jediné obmedzenie vzniká na strane OS Windowsu. Aby sme dokázali spustiť vo Windowse ľubovoľný ovládač, tak musí byť digitálne podpísaný. Kvôli tomu autori poskytujú na svojej stránke[3] už podpísaný ovládač. Používateľ potrebuje stiahnuť dynamickú knižnicu `wintun.dll` a hlavičkový súbor `wintun.h` . Pribaliť ju do projektu a následne použiť. Obdobne je na stránke vytvorené demo ako príklad použitia.

V rámci tejto práce sme sa rozhodli vyskladať ovládač aj vlastnými silami. Používateľ na to potrebuje vykonať 3 kroky:

- **inštalácia Windows Driver Kit** – konkrétne v našom prípade verziu pre Windows 10, version 2004, pretože sme inštaláciu skúšali na OS Windows 10 s verziou 21H2. Okrem toho táto verzia je posledná kompatibilná možnosť pre Visual Studio 2019. Dostupné na Microsoft webe[4],
- **inštalácia Visual Studio IDE 2019** – popri inštalácii by sa mal automatický nainštalovať aj Windows Software Development Kit. Ak by tak nenastalo odporúčame doinštalovať. Visual Studio postačuje vo verzii Community.
- **bcdedit /set testsigning on** – zadať tento príkaz do príkazového riadka spusteného ako administrátor a následne reštartovať.

Po uvedených krokoch sa nám OS spustí do testovacieho režimu. V ňom dokážeme následne modifikovať pôvodné riešenie a vyskladať si vlastný ovládač. Ako uviedli autori, pre jeho použitie bez spusteného testovacieho režimu OS je nutné ovládač podpísať.

Pre čitateľa sme pripravili aj vlastnú demo ukážku použitia ovládača v jazyku C. Zdrojový kód je obsahom príloh. Konkrétne v priečinku `wintun_demo`. Priečinok obsahuje zdrojový kód programu, podpísaný `wintun.dll`, balíček make a jednoduché `readme.txt` s pokynmi. Viac informácií nájde používateľ v dokumentácií Wintun, dostupnej na webe[5].

Zdrojový kód na meranie počtu cyklov

Pri experimentálnych meraniach kryptografickej permutácie XOODOO sme sa rozhodli použiť funkcie na meranie počtu vykonaných inštrukcií. Ukážku zdrojového kódu je možné si pozrieť v 3.1.

[3] `https://www.wintun.net/`

[4] `https://learn.microsoft.com/en-us/windows-hardware/drivers/other-wdk-downloads`

[5] `https://git.zx2c4.com/wintun/about/`

Zdrojový kód 3.1: Zdrojový kód funkcií na zmeranie počtu vykonaných cyklov

```
1  //start of measurement
2  static __inline__ uint64_t cpucyclesS(){
3    unsigned cycles_low, cycles_high;
4    __asm__ volatile ("CPUID\n\t"
5    "RDTSC\n\t"
6    "mov␣%%edx,␣%0\n\t"
7    "mov␣%%eax,␣%1\n\t": "=r" (cycles_high), "=r" (cycles_low)::
8    "%rax", "%rbx", "%rcx", "%rdx");
9    return  (((uint64_t)cycles_high << 32) | cycles_low );
10 }
11 // end of measurement
12 static __inline__ uint64_t  cpucyclesE(){
13   unsigned cycles_low, cycles_high;
14   __asm__ volatile ("RDTSCP\n\t"
15   "mov␣%%edx,␣%0\n\t"
16   "mov␣%%eax,␣%1\n\t"
17   "CPUID\n\t": "=r" (cycles_high), "=r" (cycles_low)::
18   "%rax", "%rbx", "%rcx", "%rdx");
19   return (((uint64_t)cycles_high << 32) | cycles_low );
20 }
```

Uvedený kód bol prevzatý z bakalárskej práce [76]. Viac informácií o funkcionalite nájde čitateľ v uvedenom diele.

Umelá inteligencia založená na OpenAI

V čase vytvárania práce sa pre širokú verejnosť vypustila umelá inteligencia ChatGPT založená na OpenAI. Jedná sa o pomerne dobrý nástroj pri vyhľadávaní čiastkových riešení. V našom prípade nám bola nápomocná najviac pri problémoch súvisiacich s riešením kompatibility v kóde. Odpoveď na otázku sme týmto spôsobom dokázali nájsť podstatne skôr, než tradičným prehľadávaním internetu. Rád by som však upozornil čitateľa, že nástroj nie je vhodné používať za účelom vytvorenia komplexného riešenia. Nakoľko tento nástroj ešte nie je dostatočne vyspelý.

Aktuálne dostupná verzia pre verejnosť má isté limitácie. Konkrétne sa jedná o jej dátový model na základe, ktorého bola učená. Informácie obsiahnuté v modeli su z roku 2021. V niektorých momentoch to preto poskytovalo pre používateľa neaktuálne alebo neplatné informácie.

Čitateľ si môže ChatGPT vyskúšať na stránke[6]. Odhadujeme že, vývoj tohto nástroja bude v blízkej dobe veľmi napredovať vzhľadom k získanej popularite.

Program Wireshark na analýzu sieťovej premávky

Wireshark je veľmi populárny a známy program s otvoreným zdrojovým kódom. Je vhodný na použitie za účelom analýzy sieťových protokolov, vyhľadávaní chýb pre smerovaní a vytváraní nových protokolov. Poskytuje používateľovi možnosti zachytávať sieťovú premávku a následne na základe analýzy týchto dát dokážeme vyhodnotiť správanie, výkon alebo prípadné bezpečnostné problémy riešenia.

Práca s programom je relatívne jednoduchá a intuitívna. Používateľ si dokáže zvoliť ľubovoľné sieťové rozhranie a následne monitorovať čo sa deje. Zo získaných dát je možne určiť všetky podstatné sieťové informácie o prenášaných dátach.

Viac o programe je možné nájsť na stránke[7] programu, odkiaľ boli aj informácie z tejto podkapitoly čerpané.

3.2 Prostredie virtuálnych strojov pomocou VirtualBox

Pri testovaní funkcionality VPN sme použili virtualizačný nástroj (ďalej VM) Virtual Box (ďalej VB), spoločnosti Oracle, vo verzii 6.1.38. VB je voľne dostupný. Inštalácia je jednoduchá a rýchla. Viac informácii o nástroji je možné dohľadať v [77].

Pre použitie je potrebné aby mal používateľ k dispozícií obraz operačného systému (ďalej OS). Tie nie je problém získať ani pre OS Windows a podobne, avšak pri našej práci sme zvolili využitie voľno dostupného OS – **Linux Ubuntu** vo verziách 22.04.3(LTS[8])– OSS. Následne sme vykonali inštaláciu spomenutých nástrojov a vytvorili klon – OSC. Pri opise práce použijeme označenia OSS pre VPN server a OSC pre klienta.

Pri jednoduchej inštalácií VM sme použili konfiguráciu s 2048 MB RAM a 2 jadrami. (Minimálna inštalácia). V prípade potreby dávame do popredia návod na prípravu Windows OS v VM – [78], avšak postup je triviálny. Po inštalácii sme OS aktualizovali pomocou príkazov:

[6] `https://chat.openai.com/`
[7] `https://www.wireshark.org/`
[8] z ang. Long Term Support

```
sudo apt-get update
sudo apt-get upgrade
```

Následne sme doinštalovali potrebné súčasti k VM OS vo verzii ako je VB, teda 6.1.30. Dôvodom bolo zväčšenie rozlíšenia a využívanie možnosti zdieľaného priečinka s OS, na ktorom daný VB beží. Za účelom správneho fungovania priečinka bolo nutné v termináli použiť príkaz:

```
sudo usermod -aG vboxsf $(whoami)
```

a následne reštartovať OS.

Pri príprave prostredia S OS Windows sme postupovali obdobne. Zo stránok Microsoftu sme stiahli obraz s OS Windows 10. Následne sme konfigurovali obraz podľa prednastavených nastavení. Vo virtuálnom stroji sme priradili obrazu po 4 gigabajty pamäte RAM a 4 procesory. Po spustení sa systém začne automaticky aktualizovať. Nechali sme mu preto priestor 10 minút. Následne sme OS reštartovali a po štarte sme inštalovali potrebné nástroje. V prípade potreby ďalšieho zariadenia s OS Windows sme si len vytvárali klon tohto už predinštalovaného systému.

3.2.1 Zmena sieťových adaptérov

VB ponúka rôzne možnosti nastavenia sieťových adaptérov. Aktuálne su dostupné tieto:

- Not attached – zariadenie nemá žiaden sieťový adaptér a teda ani prístup do žiadnej siete,
- Network Address Translation (ďalej NAT) – VM host používa lokálnu adresu a s vonkajším svetom komunikuje vďaka portovým číslam a prekladu adries na adresu natívneho zariadenia,
- NAT Network – typ internej NAT siete, vďaka ktorej je možná taktiež komunikácia smerom von
- Bridge adapter – vytvorenie virtuálneho rozhrania s vlastnou IP adresou,
- Internal – interná sieť medzi VM bez prístupu do natívneho zariadenia,
- Host-Only – slúži na vytvorenie podsiete medz natívnym zariadením a skupinou priradených VM bez prístupu na internet,
- Generic driver – všeobecné rozhranie s možnosťou výberu iného ovládača na vytvorenie sieťového rozhrania,

- Cloud-Based – používa sa na pripojenie lokálneho VM do vzdialenéj cloud služby.

Konektivita jednotlivých možností je znázornená pomocou tabuľky 3.2. Host reprezentuje natívne zariadenie. NET prístup na internet. Viac informácií o režimoch je dostupných na [79].

Režim	Konektivita				
	VM –> Host	**VM <– Host**	**VM1 <–> VM2**	**VM –> NET**	**VM <– NET**
Host-only	+	+	+	-	-
Internal	-	-	+	-	-
Bridged	+	+	+	+	+
NAT	+	konfigurácia portov	-	+	konfigurácia portov
NAT network	+	konfigurácia portov	+	+	konfigurácia portov

Tabuľka 3.2: Konektivita jednotlivých sieťových adaptérov

Vzhľadom k našim potrebám, teda obojsmerná komunikácia medzi 2 VM, sú pre nás relevantné režimy bridge a NAT network. Ako si môžeme všimnúť, NAT vyžaduje dodatočnú konfiguráciu portov v prípadoch kedy chceme aby nastala komunikácia medzi dvoma VM. Z tohto dôvodu je pre čo najjednoduchší prístup zvoliť práve režim bridge. Ten priradí VM vlastnú IP adresu, pomocou, ktorej stroj komunikuje.

Viac o jednotlivých režimov je taktiež možné nájsť v [80]. Autor sa venuje postupu konfigurácie jednotlivých režimov spoločne s ich opisom.

4 Implementacia jednoduchej VPN siete

Za účelom demonštrácie jednoduchej VPN siete sme zvolili voľne dostupnú implementáciu v jazyku C. V tejto kapitole postupne rozoberieme funkcionalitu a správanie vzniknutej VPN siete.

4.1 Dead Simple VPN

Dead Simple VPN je voľne dostupný[1] program, napísaný v jazyku C. Určený je pre operačný systém Linux. Autorom je Frank Denis. DSVPN rieši najbežnejší prípad použitia VPN, teda pripojenie klienta k VPN serveru cez nezabezpečenú sieť. Následne sa klient dostane na internet prostredníctvom servera. Tak ako bolo uvedené napríklad v obrázku 1.1.

DSVPN používa protokol riadenia prenosu – TCP [7]. Medzi ďalšie pozitíva patrí:

- Používa iba modernú kryptografiu s formálne overenými implementáciami.
- Malá a konštantná pamäťová stopa. Nevykonáva žiadne dynamické alokovanie pamäte (z ang. *heap memory*).
- Malý (25 KB) a čitateľný kód. Žiadne vonkajšie závislosti (z ang. *Dependencies*).
- Funguje po preklade GCC prekladačom. Bez dlhej dokumentácia, žiaden konfiguračný súbor, dodatočná konfigurácia. DSVPN je spustiteľná jednoriadkovým príkazom na serveri, obdobne na klientovi. Bez potreby konfigurácie brány firewall a pravidiel smerovania.
- Funguje na Linuxe (kernel $\geq$ 3.17), macOS a OpenBSD, DragonFly BSD, FreeBSD a NetBSD v klientskych a point-to-point režimoch. Pridanie podpory pre iné operačné systémy je triviálne.

[1] https://github.com/jedisct1/dsvpn

- Nedochádza k úniku IP medzi pripojeniami, ak sa sieť nezmení. Blokuje IPv6 na klientovi, aby sa zabránilo úniku IPv6 adries.

V uvedenej VPN autor zakomponoval aj možnosť pokročilejších nastavení. Celkový súhrn vstupných parametrov pri štarte programu je takýto:

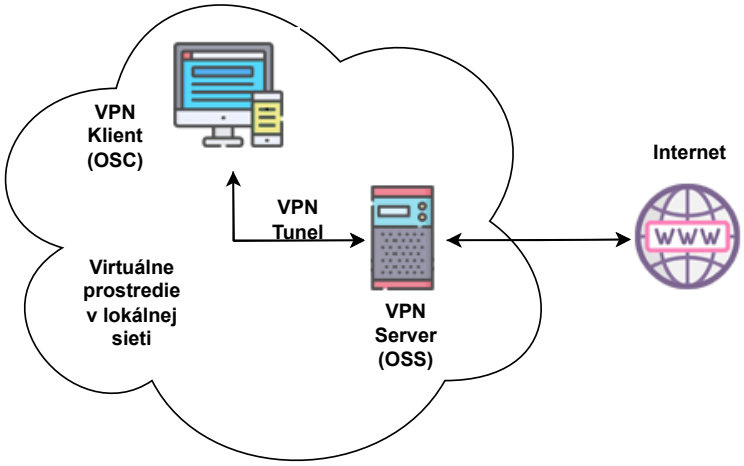
```
./dsvpn   server
<key file>
<vpn server ip or name>|"auto"
<vpn server port>|"auto"
<tun interface>|"auto"
<local tunnel ip>|"auto"
<remote tunnel ip>"auto"
<external ip>|"auto"

./dsvpn   client
<key file>
<vpn server ip or name>
<vpn server port>|"auto"
<tun interface>|"auto"
<local tunnel ip>|"auto"
<remote tunnel ip>|"auto"
<gateway ip>|"auto"
```

Väčšina parametrov je v zdrojovom kóde prednastavených na automatické hodnoty. Príkladom je port 443, vytvorenie rozhrania tun0, prevzatie externej IP adresy zo siete a ďalšie. Používateľ teda môže spúšťať DSVPN pomocou príkazu s najmenej 2 parametrami v prípade servera a tromi pre prípad klienta. Dôvodom je, že klient potrebuje mať určenú IP adresu servera, na ktorý sa má pripojiť (3.parameter). Druhý parameter v poradí je už spomenutá cesta k zdieľanému 256-bitovému kľúču.

4.2 Kryptografia použitá v DSVPN

DSVPN používa v svojej implementácií malú sebestačnú kryptografickú knižnicu – *Charm*[2]. Jej autorom je tvorca DSVPN. Implementácia umožňuje autentizované šifrovanie (z ang.*authenticated encryption*) a hašovanie kľúčov (z ang. *keyed*

[2] `https://github.com/jedisct1/charm`



Obr. 4.1: Schéma architektúry jednoduchej VPN siete počas experimentu

hashing). Správnosť implementácie algoritmu v knižnici programátor overil pomocou nástroja **Cryptol**[3]. Uvedený nástroj slúži na zápis algoritmu do matematickej špecifikácií. Tým poskytne možnosť jednoduchšej a hlavne korektnej implementácie zvoleného kryptografického algoritmu. Zároveň je možné program využiť aj na verifikáciu vytvoreného riešenia. Obdobne sú v repozitári knižnice ponechané overovacie skripty pre jednoduché spustenie.

Kryptografický algoritmus použitý v DSVPN je Xoodoo permutácia v duplex móde, pričom môže byť jednoducho nahradená napríklad Gimli-im[4] [81] alebo Simpira384[5] [82]. Pri zmene musí používateľ zasiahnuť do zdrojového kódu v súbore **charm.c**, ktorého obsahom sú kryptografické primitíva.

4.3 Experimentálne overenie VPN

DSVPN sme prakticky overili pomocou dvojice virtuálnych strojov OSS a OSC, ktorých opis je obsahom 3.2. Na zariadení OSS sme pomocou balička make a GCC prekladača vykonali inštaláciu DSVPN. Obdobný postup je aplikovaný aj vo VM OSC. Na OSS spúšťame VPN Server, ktorý nám poskytne IP adresu, prostredníctvom ktorej budeme komunikovať s vonkajším svetom. V schéme 4.1 je znázornená architektúra sieťa, v ktorej bude vykonaný experiment.

Na spustenie a vytvorenie spojenie vykonáme nasledujúce úkony:

1. Vygenerovanie zdieľaného kľúča:

[3] `https://cryptol.net/index.html`
[4] `https://github.com/jedisct1/gimli`
[5] `https://github.com/jedisct1/simpira384`

```
dd if=/dev/urandom of=vpn.key count=1 bs=32
```

– zdieľaný kľúč, ktorý sme vygenerovali, sa nám uložil do súboru *vpn.key*. Jeho veľkosť je 32 bajtov, teda 256 bitov. Kľúč je potrebné vložiť do priečinka s programom *dsvpn* v oboch zariadeniach – OSS aj OSC alebo zadať cestu ako parameter, kde sa kľúč nachádza.

2. OSS zariadenie:

```
sudo ./dsvpn server vpn.key auto
2340 auto 10.8.0.254 10.8.0.2
```

– tento príkaz zabezpečí spustenie VPN servera na prostredí OSS s IP adresou. Príkaz `sudo` nám spustí program s administrátorskými právami. Piaty parameter nastavuje IP adresu servera. V našom prípade **`auto`**, použije aktuálne používanú IP na komunikáciu s vonkajším prostredím. Príkazom ďalej definujeme portové číslo 2340, ktoré sa použije pri nastolení TCP spojenie medzi klientom a serverom. Poslednou konfiguráciou je priradenie mena a IP adresy tunelov, ktoré bude využívať naše zariaadenie – 10.8.0.254 a druhý koniec tunela – 10.8.0.2. Používateľ má ešte možnosť nastaviť tzv. External IP. Tú by sme využili ak by sme spúšťali DSVPN na routri poskytovateľa internetu. Obdobne na klientovi vieme nastaviť gateway IP, ktorá slúži na presmerovanie komunikácie k serveru. Po spustení príkazu si vieme overiť našu konfiguráciu[6].

3. OSC zariadenie:

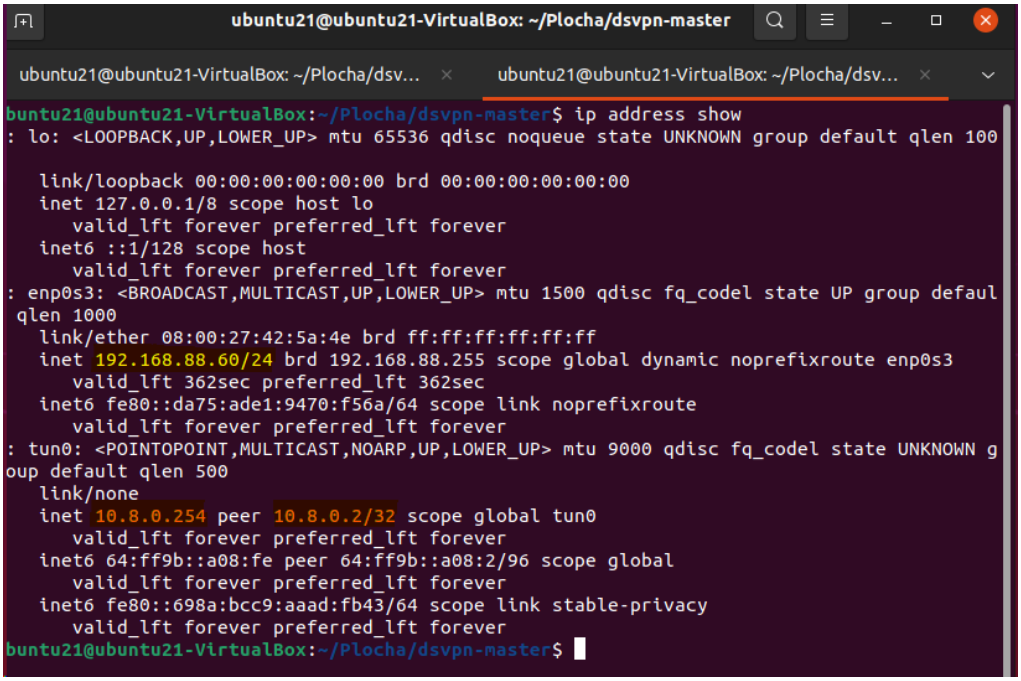
```
sudo ./dsvpn client vpn.key 192.168.88.62
2340 auto 10.8.0.2 10.8.0.254
```

– uvedený príkaz zabezpečí, že sa pripojíme na VPN Server, ktorý ma ip adresu *192.168.88.62* s portom 2340. Následne vzniká TCP spojenie. Dôležité je si všimnúť poradie adries tunelov. Je opačné ako v prípade servera.

4. V prípade úspešnej konektivity sa operácia podarila a pre okolitý svet sme viditelný pomocou IP adresy, ktorú sme zvolili.

Na overenie správnosti funkcionality nám postačí jednoduchý sieťový príkaz `traceroute`. Napríklad `traceroute google.sk`[7]. Prvá z uvedených adries je práve tá, ktorú dané zariadenie používa.

[6] Pomocou `ip address show tun0` overíme IPv4 adresu VPN tunela.
[7] vo Windows CMD prostredí: `tracert google.sk`



Obr. 4.2: Zistenie IP adries VPN Servera na VM OSS

V našom prípade bolo nutné použiť lokálne adresy vzhľadom na to, že obe VM bežia na jednom hosťovskom počítači. Obidva zariadenia sú tým pádom pripojené k jednému internetovému poskytovateľovi, čo má za následok takmer rovnaké smerovanie k vzdialenej doméne.

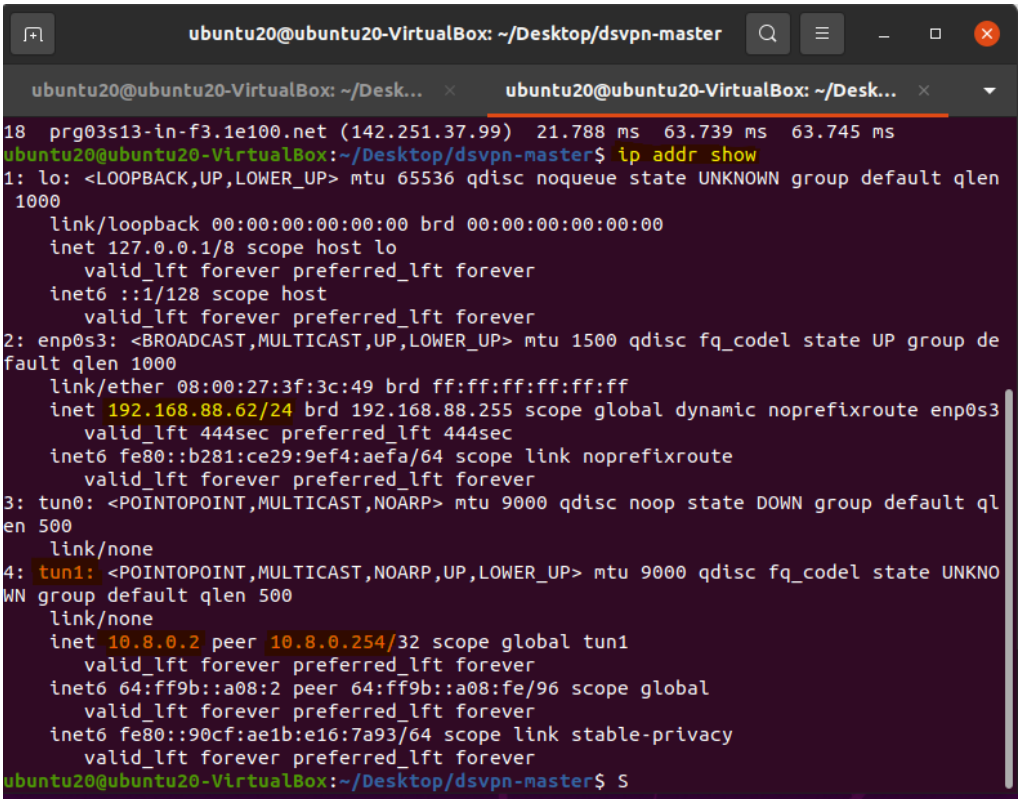
Proces zistenia IP adresy VPN servera, po spustení, a overenie funkčnosti je následne znázornené pomocou 4.2,4.3, 4.4. V 4.2 sme žltou farbou znázornili IP adresu, na ktorej je VPN server dostupný. Oranžová farba znázorňuje IP adresy tunelu medzi serverom a klientom v tomto poradí. Následne v 4.3 môžeme vidieť to isté pre klienta.

Nakoniec, v 4.4 môžeme vidieť ako klient pri internetovej komunikácií používa namiesto svojej vlastnej, adresu poskytnutú VPN Serverom na zariadení OSS – žltou zvýraznená IP. Červenou je zaškrnutá farba poskytovateľa internetu.

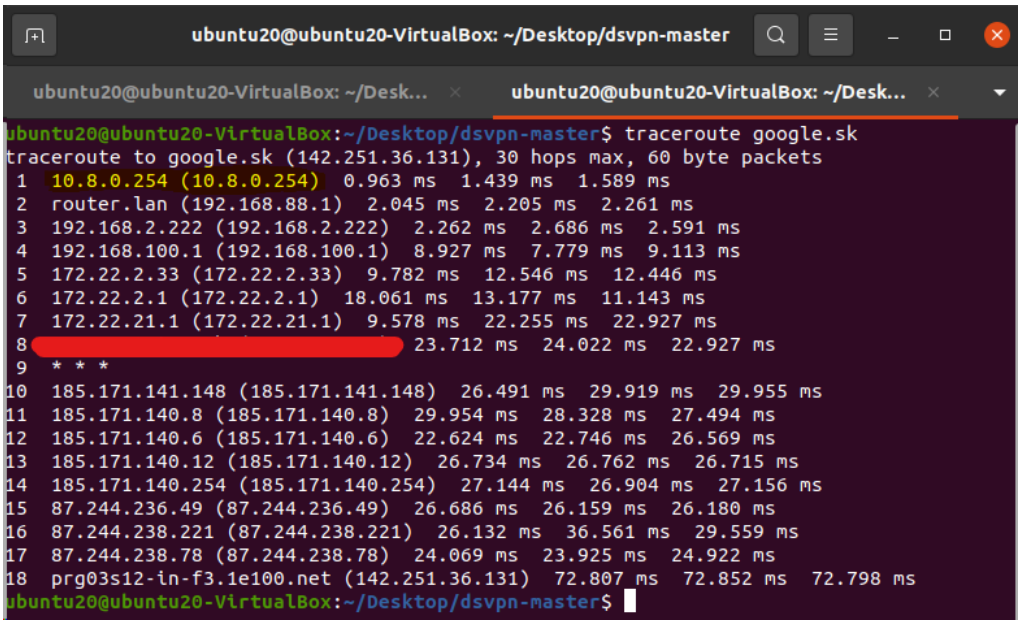
4.4 Princíp fungovania programu

DSVPN vytvára VPN sieť medzi VPN klientom a VPN serverom. Na obidvoch zariadeniach je potrebné spustiť program pomocou korešpondujúcich príkazov.

Princíp vysvetlíme na praktickom príklade. Klienta umiestnime na lokálnu sieť bez prístupu na internet. Následne vymažeme všetky smerovacie pravidlá v zariadeni. Klient po tomto kroku nedokáže komunikovať so žiadným zariadením nakoľko nemá žiadne prednastavené pravidlá kam by smeroval internetovú ko-



Obr. 4.3: Zistenie IP adries VPN Klienta na VM OSC



Obr. 4.4: Overenie funkcionality DSVPN pomocou traceroute

munikáciu. V uvedenej lokálnej sieti bude prítomný taktiež VPN server, ktorý ako jediný dokáže komunikovať s okolitým svetom. Po spustení DSVPN na obidvoch zariadeniach, dochádza k spojeniu. Jedná sa o TCP spojenie vytvorené soketmi. Úlohou soketového TCP spojenia je preposielanie už zašifrovaných dát v novom L3 pakete. Následne DSVPN overuje stav soketov a tunelovacieho rozhrania. Ak je tun rozhranie pripravené na čítanie (obsahuje pakety), tak dochádza k ich spracovaniu (šifrovaniu) a preposlaní cez soket. Po prijatí sa dáta dešifrujú a smerujú na základe dát z L3 IP hlavičky.

Toto je celý princíp zabezpečenej komunikácie medzi Klientom a Serverom vo VPN sieti na L3 vrstve. Aby sme docielili plnohodnotnú funkcionalitu VPN, tak potrebujeme do tohto riešenia zakomponovať smerovanie. Tým, že dáta smerujeme von z tejto siete, tak je ešte potrebné upraviť niektoré systémové smerovacie pravidla.

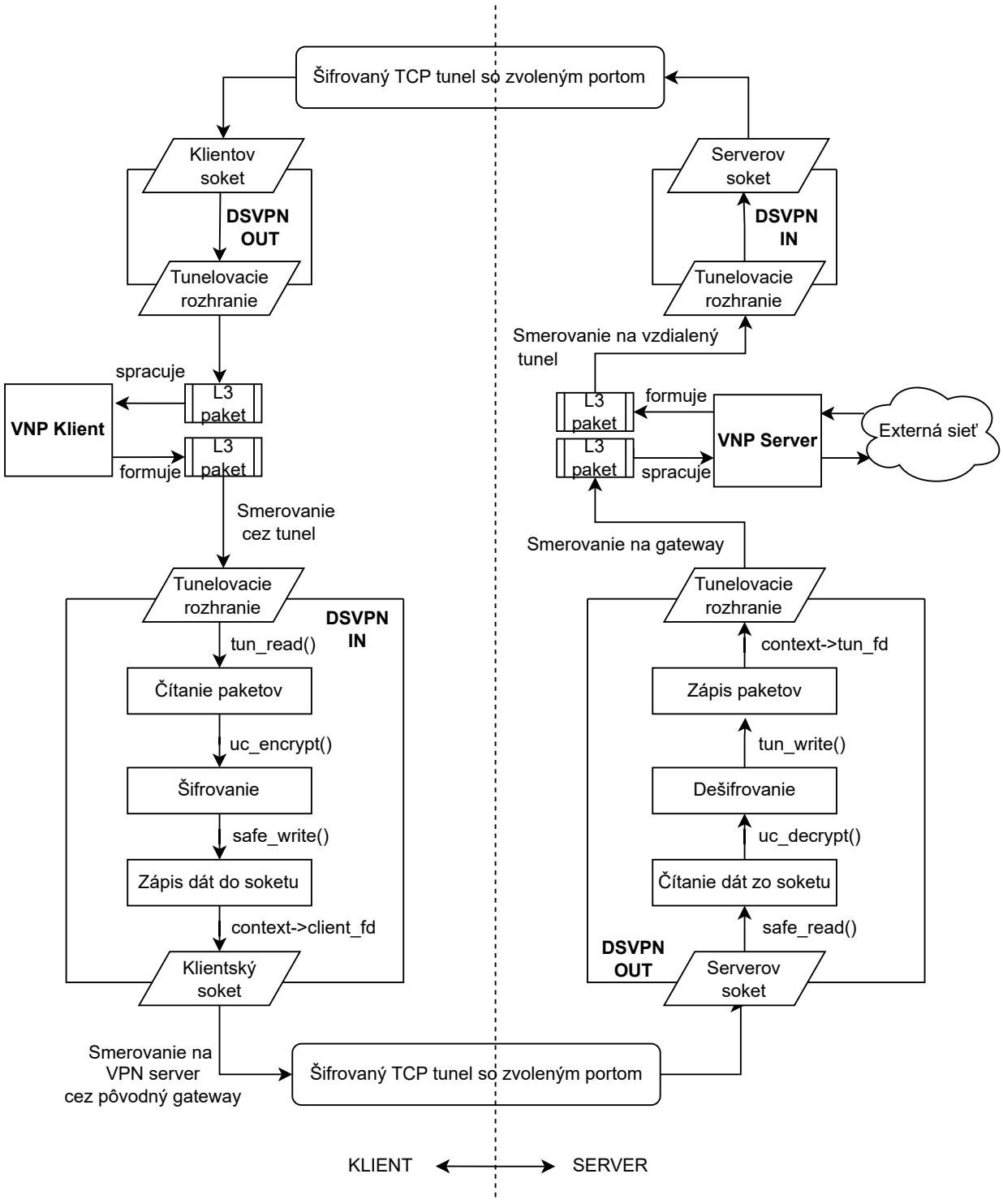
Okrem načrtnutého príkladu s prístupom na internet, dokáže klient takto získať konektivitu k jemu nedostupným segmentom siete. Samozrejme za predpokladu, že sú dostupné pre server. Ako príklad môže byť diskové úložisko. Prípadne prístup na internetové domény. V angličtine sa takýto termín označuje ako *remote access*, teda vzdialený prístup. Je to tiež veľmi obľúbený a používaný typ VPN siete.

Funkcionalita DSVPN je znázornená pomocou schémy 4.5. V nej môžeme na ľavej strane vidieť čo sa deje s paketom pri požiadavke smerom od VPN klienta k VPN serveru. OS na základe požiadavky formoju L3 pakety. Po presmerovaní paketu na tunel DSVPN zašifruje tieto dáta. Následne ich zapíše na soket, ktorý tvorí so serverom TCP spojenie. Server tieto dáta spracuje a vykoná potrebné náležitosti. Po získaní odpovede zasa Server odpovedá klientovi rovnakým mechanizmom. Čitateľovi by som rád doplnil, že v schéme sme pre DSVPN vytvorili 2 bloky. **DSVPN IN** a **DSVPN OUT**. Funkcionalita je znázornená iba v spodnej časti schémy. Jedná sa však o identické bloky.

4.5 Analýza zdrojového kódu DSVPN

Pri analýze sa zameriame výhradne na dôležité časti kódu DSVPN pre programovací jazyk C. Repozitár pozostáva z jedného make-file balíčka[83]. 3 hlavičkových (.h) a k ním korešpondujúcimi zdrojovými kódmi (.c), s pomenovaním:

1. **charm.h** – kryptografická knižnica so 6 funkciami,
2. **os.h** – funkcie čítania a zápisu paketov, vytvorenia, používania a zrušenia



Obr. 4.5: Ukážka prenosu paketu naprieč DSVPN

tunelovacieho rozhrania,

3. **vpn.h** – deklarácia konštánt, endianity[84] a niektorých závislostí OS.

V spomenutých hlavičkových súboroch sa nachádzajú deklarácie funkcií, ktoré VPN používa. Definície sú obsahom .c súborov, ako je v jazyku C zaužívaným

zvykom. Obsahom tejto podkapitoly je analýza týchto kódov.

4.5.1 Súbor charm.h a charm.c

Obsahom sú prevažne funkcie slúžiace pri behu kryptografického algoritmu XOODOO. Charm.h pozostáva z 6 funkcií. Ich implementácia nie je až tak rozsiahla. Zaberá celkovo 337 riadkov. Princípy použité v algoritme bolí opísané v kapitole 2.

Zdrojový kód 4.1: Obsah charm.h

```
1  void uc_state_init(uint32_t st[12], const unsigned char key[32],
2                     const unsigned char iv[16]);
3  void uc_encrypt(uint32_t st[12], unsigned char *msg,
4                  size_t msg_len, unsigned char tag[16]);
5  int uc_decrypt(uint32_t st[12], unsigned char *msg,
6                  size_t msg_len,
7                  const unsigned char *expected_tag,
8                  size_t expected_tag_len);
9  void uc_hash(uint32_t st[12], unsigned char h[32],
10              const unsigned char *msg, size_t len);
11 void uc_memzero(void *buf, size_t len);
12 void uc_randombytes_buf(void *buf, size_t len);
```

Z názvov funkcií je pomerne jasné čo sa v jednotlivých volaniach deje.

4.5.2 Súbor os.h a os.c

Obsahom su funkcie, ktorých úlohami sú čítanie alebo pridanie GW, vytvorenie a nastavenie tunelu v danom OS. Následne je aplikovaná úprava firewall pravidiel, tak aby všetka komunikácia bola presmerovaná na VPN server. Úprava firewall pravidiel je závislá od role, pod ktorou je VPN spustená. Teda či sa jedná o server alebo klienta. Celkovo je obsahom 12 funkcií, ktoré riešia uvedené úlohy. Detail je možné vidiet v 4.2.

Zdrojový kód 4.2: Obsah ZK os.h

```
1  ssize_t safe_read(const int fd, void *const buf_, size_t count,
2                    const int timeout);
3  ssize_t safe_write(const int fd, const void *const buf_,
4                     size_t count,
5                     const int timeout);
6  ssize_t safe_read_partial(const int fd, void *const buf_,
7                            const size_t max_count);
8  ssize_t safe_write_partial(const int fd, void *const buf_,
9                             const size_t max_count);
10
11 typedef struct Cmds {
12  const char *const *set;
13  const char *const *unset;
14 } Cmds;
15
16 Cmds firewall_rules_cmds(int is_server);
17 int shell_cmd(const char *substs[][2], const char *args_str,
18               int silent);
19 const char *get_default_gw_ip(void);
20 const char *get_default_ext_if_name(void);
21 int tcp_opts(int fd);
22 int tun_create(char if_name[IFNAMSIZ], const char *wanted_name);
23 int tun_set_mtu(const char *if_name, int mtu);
24 ssize_t tun_read(int fd, void *data, size_t size);
25 ssize_t tun_write(int fd, const void *data, size_t size);
```

Samozrejmosťou je implementácie čítania a zápisu dát z TUN tunelu, soketu a príkazového riadku.

4.5.3 Súbor vpn.h a vpn.c

Hlavičkový súbor obsahuje prevažne definovanie niektorých parametrov potrebných na správnu funkcionalitu VPN, spoločne s korekciou pre niektoré OS. Viď. 4.3. Ostatné parametre, ktoré bolí opísané pre spustenie, su definované práve v tomto súbore (porty, IP adresy, MTU atď.).

Zdrojový kód 4.3: Obsah ZK vpn.h

```
1  /*UNIX-like OS Dependent Libraries*/
2  #include <sys/ioctl.h>
3  #include <sys/socket.h>
4  #include <sys/types.h>
5  #include <sys/uio.h>
6  #include <sys/wait.h>
7  #include <net/if.h>
8  #include <netinet/in.h>
9  #include <netinet/tcp.h>
10 /*End UNIX-like OS Dependent Libraries*/
11 /*OS setup dependencies*/
12 #ifdef __linux__
13 #include <linux/if_tun.h>
14 #endif
15
16 #ifdef __APPLE__
17 #include <net/if_utun.h>
18 #include <sys/kern_control.h>
19 #include <sys/sys_domain.h>
20 #endif
21
22 #ifdef __NetBSD__
23 #define DEFAULT_MTU 1500
24 #else
25 #define DEFAULT_MTU 9000
26 #endif
27 /*End of OS setup dependencies*/
```

Main() – Beh programu

Pred samotným opisom by som rád upozornil na jeden fakt. Autor DSVPN používa vo veľkej miere zápis pomocou tzv. ternárnych operátorov. Viac informácií o tejto problematike je možné nájsť v [85]. V skratke, používa sa podmienený zápis hodnoty do premennej.

Vpn.c je hlavným zdrojovým kódom DSVPN. V jeho vnútri nájdeme hlavnú, štartovaciu, funkciu main. Zároveň má prilinkované aj vyššie uvedené knižnice. Ako prvé dochádza k inicializácií premennej štruktúry `Context`4.4. Štruktúra v

sebe nesie všetky premenné, potrebné na správne fungovanie programu.

Zdrojový kód 4.4: Štruktúra Context

```
1  typedef struct Context_ {
2    const char *  wanted_if_name;
3    const char *  local_tun_ip;
4    const char *  remote_tun_ip;
5    const char *  local_tun_ip6;
6    const char *  remote_tun_ip6;
7    const char *  server_ip_or_name;
8    const char *  server_port;
9    const char *  ext_if_name;
10   const char *  wanted_ext_gw_ip;
11   char          client_ip[NI_MAXHOST];
12   char          ext_gw_ip[64];
13   char          server_ip[64];
14   char          if_name[IFNAMSIZ];
15   int           is_server;
16   int           tun_fd;
17   int           client_fd;
18   int           listen_fd;
19   int           congestion;
20   int           firewall_rules_set;
21   Buf           client_buf;
22   struct pollfd fds[3];
23   uint32_t      uc_kx_st[12];
24   uint32_t      uc_st[2][12];
25 } Context;
```

Následne dochádza k načítaniu vopred zdieľaného kľúča pomocou pomocnej funkcie
`load_key_file()`4.5. Úlohou je prečítanie kľúča, pričom sa používa funkcia z os.c – `safe_read()`. Realizuje sa znakové spočítanie, v ktorom je zakomponovaná funkcia `poll()` [86]. Dôvodom je, že `safe_read()` sa používa aj pri čítaní zo soketu a tunelu. V linuxe je čítanie z týchto typov rovnaké. Tato časť kódu je však podmienená špecifickou chybou, ktorú je možné dostať len pri čítaní soketu, respektíve tunelu. V prípade, ak sa prečítalo 32 bajtov, `safe_read()` vracia 0. Následne sa inicializuje stav v XOODOO.

Zdrojový kód 4.5: Načítanie zdieľaného kľúča

```
1  static int load_key_file(Context *context, const char *file)
2  {
3   unsigned char key[32];
4   int           fd;
5
6   if ((fd = open(file, O_RDONLY)) == -1) {
7     return -1;
8   }
9   if (safe_read(fd, key, sizeof key, -1) != sizeof key) {
10    (void) close(fd);
11    return -1;
12  }
13  uc_state_init(context->uc_kx_st, key,
14                (const unsigned char *) "VPN Key Exchange");
15  uc_memzero(key, sizeof key);
16
17  return close(fd);
18 }
```

Ako môžeme vidieť v implementácii sú bežne použité smerníky. Vo výsledku, tak dokážeme preniesť zmenu hodnôt na viaceré premenné.

V prípade, ak používateľ pri štarte nezmenil parameter pre IP adresu sieťovej brány[8] (ďalej GW), tak sa používa pôvodná. Tá sa získa pomocou funkcie `get_default_gw_ip()`, ktorá je deklarovaná v 4.2. Prostredníctvom shell príkazu `ip route` a `read_from_shell_command()` funkcie, dochádza k extrakcii informácií priamo z príkazového riadka. Tento krok je teda závislý od OS, v ktorom používateľ pracuje. Nasleduje overenie návratových hodnôt z funkcií iba v prípade ak je DSVPN spustené ako Klient.

Program v prípade servera pokračuje s `get_default_ext_if_name()`. Obdobne ako v predchádzajúcom odstavci je realizácia funkcionality, vykonaná pomocou terminálu. Podstata spočíva v zistení mena tunelovacieho rozhrania.

Po nastavení parametrov sa dostávame k vytvoreniu tunelovacieho rozhrania. V tomto kroku autor používa funckiu `tun_create`. Úloha je vysoko závislá od OS. Dôsledkom toho je možné vidieť vetvenie funkcionality vzhľadom k bežiacemu OS. DSVPN poskytuje kompatibilitu pre 6 OS, medzi, ktoré patrí Linux, FreeBSD, NetBSD, OpenBSD, MacOS, DragonFly. Následne na základe systému dochádza

[8] z ang. *GateWay*

k vytváraniu tunelu s prednastaveným, resp. zvoleným menom. Program ďalej nastaví maximálnu veľkosť prepravovaného paketu[9] (ďalej MTU) na 1500/9000 bajtov.

Po príprave tunelu dochádza k overeniu dostupnosti VPN servera. Tento krok vykonáva funkcia `resolve_ip`. Tá ma v sebe vnorené 2 funkcie, ktoré su menným ekvivalentom aj vo Windows knižniciach. Jedná sa o funkcie `getaddrinfo` a `getnameinfo`.

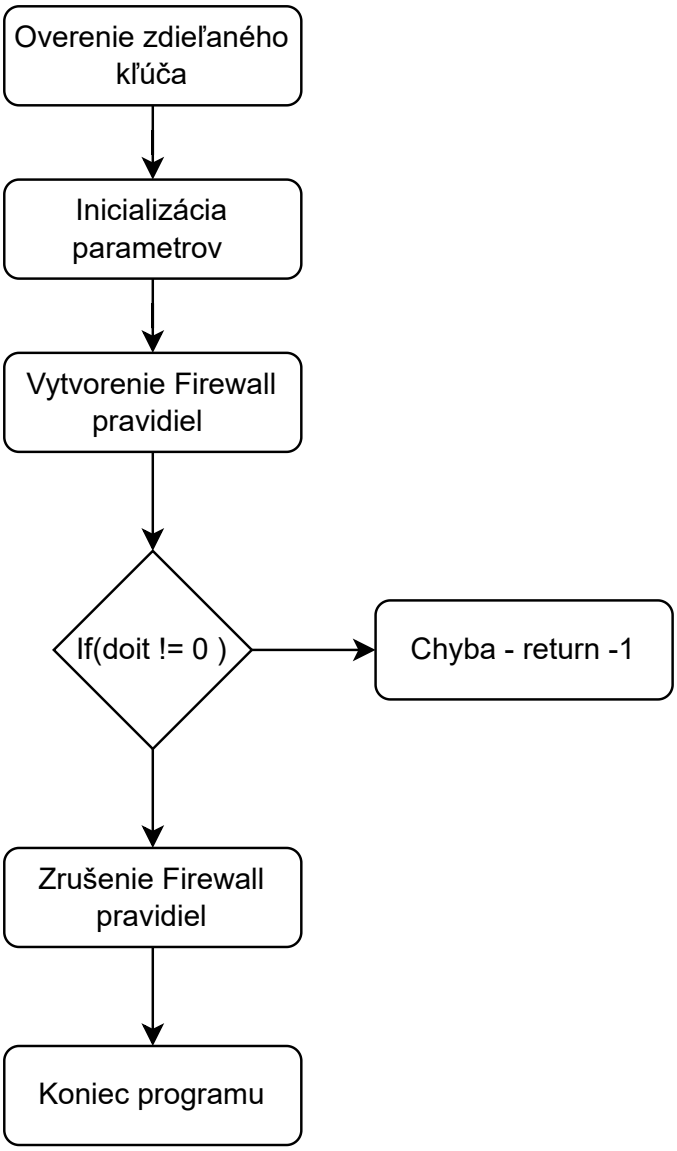
Posledným krokom súvisiacim s konfiguráciou prostredia je vytvorenie pravidla pre branu FireWall (ďalej FW). Tento úkon realizuje `firewall_rules()`. Tento proces je opäť systémovo závislý. Jeho realizácia je vykonaná pomocou globálne definovanej funkcie `Cmds firewall_rules_cmds()`. Tá obsahuje súbor prednastavených príkazov. Ich úlohou je presmerovanie celej premávky cez vzniknutý tunel.

Posledným úkonom je samotný beh VPN. Ten spúšťa funkcia `doit()`. Doterajší beh programu je jednoducho znázornený pomocou flow diagramu 4.6.

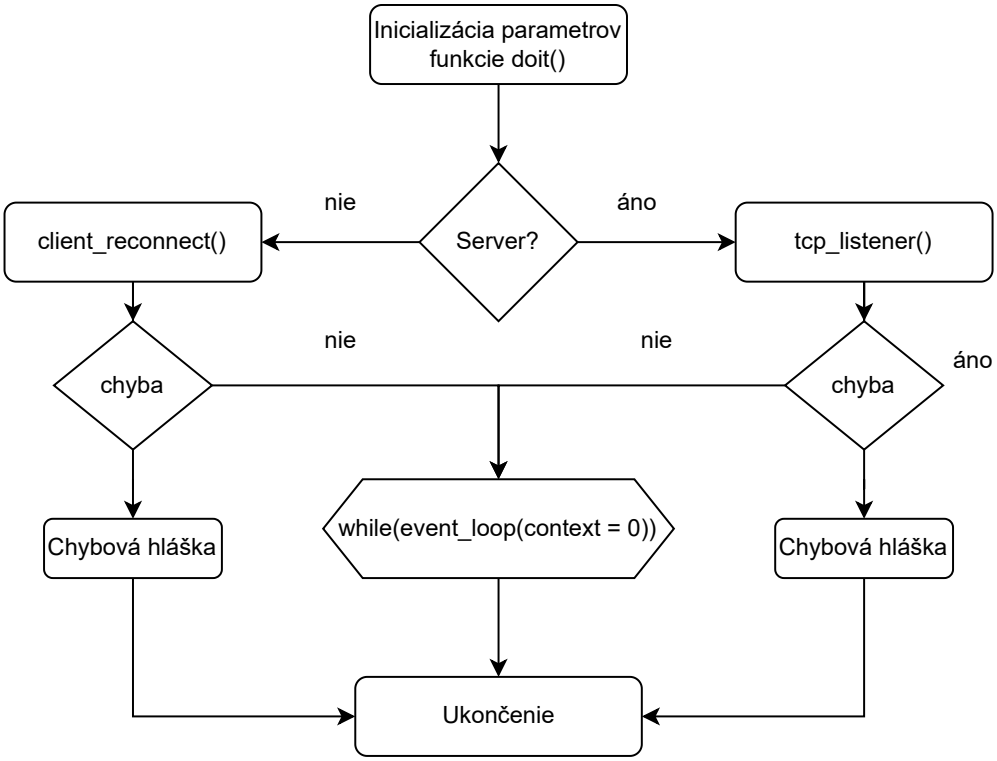
Od tohto momentu sa presúvame k postupnému vnáraniu do procesu spojenia a spracovania dát. Hlavnou úlohou funkcie `doit()` je nadviazanie soketového TCP spojenia medzi klientom a serverom. Vo vnútri `doit()` preto dochádza k vetveniu programu. V prípade, že je DSVPN spustená ako server spustí sa funkcia `tcp_listener()`. V opačnom prípade `client_reconnect()`. Listener vytvára socket a následne pomocou systemovej funkcie `bind()` sa napojí na zvolený port. Potom čaká na spojenie od klienta pomocou funkcie `listen()`. Smerník na socket je uložený do smerníka na štruktúru Context. Klient používa vetvu s `Client_reconnect()`. V nej sa snaží opakovane nadviazať spojenie so serverom. Za týmto účelom `client_connect()`. Pred týmto úkonom samozrejme dochádza k overeniu či už nedošlo k nadviazaniu spojenia. O to sa stará `client_disconnect()`, ktorý ruší aktívne spojenie.

`Client_connect()` slúži na pripojenie klienta k serveru. Vykonáva aj úpravu pravidiel v FW. Následne sa pokúša o nadviazania spojenia pomocou funkcie `tcp_client()`. Po tejto sérií úloh sa vraciame opäť do `doit()`. V cykle **`while`** sa vykonávaná funkcia `event_loop()`, v ktorej dochádza k použitiu kryptografickej knižnice charm. Jej obsahom je inicializácia pomocných premenných. viď. zdrojový kód 4.6. Schéma 4.7 znázorňuje doteraz opísané skutočnosti.

[9] z ang. *Maximum Transmission Unit*



Obr. 4.6: Schéma behu DSVPN vo funkcii main()



Obr. 4.7: Princíp fungovania funkcie doit()

Zdrojový kód 4.6: Premenné funkcie event loop

```
1  struct pollfd *const fds = context->fds;
2  Buf                  tun_buf;
3  Buf *                client_buf = &context->client_buf;
4  ssize_t              len;
5  int                  found_fds;
6  int                  new_client_fd;
```

Funckia `event_loop()`

`Event_loop()` je 111 riadkov dlhá funkcia. Jej obsah môžeme rozdeliť na overovací a výkonový. Úlohou niekoľkých **`if`**-ov vo funkcii je preverovanie signálov, spätných hodnôt a podobných premenných, ktoré by signalizovali chybu, používateľov záujem o ukončenie programu alebo signál pre čítanie dát zo soketov, respektíve tunelov.

Zaujímavé je taktiež predom definované makro `BUFFERBLOAT_CONTROL`. Jeho úlohou je zamedzenie problému zvaného **Bufferbloat**. V skratke, je to nechcený jav, ktorý je zapríčinený nadmerným ukladaním paketov do vyrovnávacej pamäte, tzv. zahltenie. To ma za následok vysokú latenciu a tzv. z ang. *Packet Delay*

Variation (ďalej PDV/Jitter), v paketovo-orientovaných sieťach. Viac o tejto problematike je možne si prečítať na [87].

Na druhej strane výkonové funkcie, ako hovorí ich názov. niečo vykonávajú. Do tejto kategórie sme zaradili funkcie:

- `tcp_accept()` – slúži na nastolenie nového soketového TCP spojenia s klientom,
- `tun_read()` – v prípade linuxového OS, volá `safe_read_partial()` na zapísanie dát do nami vytvoreného tunelovacieho rozhrania,
- `uc_encrypt()` – kryptografické šifrovanie paketov z tunelovacieho rozhrania,
- `safe_write_partial()` – používa štandardizovanú funkciu `write()` vo while cykle, zapíše zašifrované dáta do buffera určeného pre odoslanie klientovi, vracia počet zapísaných dát,
- `safe_write()` – používa sa v prípade ak došlo k zahlteniu paketmi,
- `client_reconnect()` – slúži na obnovu spojenia v prípade chyby,
- `safe_read_partial()` – obdobne ako pri `write()`, používa `read()` funkciu,
- `uc_decrypt()` – kryptografické dešifrovanie správy a následne odoslanie na tunelovacie rozhranie,
- `tun_write()` – volá `safe_write` pri OS Linux, teda klasický zápis dešifrovaných dát.

Metóda použitá pri zápise zašifrovaných dát vo funkcii `event_loop()` je znázornená v 4.7.

Zdrojový kód 4.7: Spôsob zápisu šifrovaných dát

```
1
2  writenb = safe_write_partial(context->client_fd, tun_buf.len,
3                               2U + TAG_LEN + len);
4  if (writenb < (ssize_t) 0) {// kontrola zahltenia -- bufferbloat
5   context->congestion = 1;
6   writenb             = (ssize_t) 0;
7  }
8 // ak doslo k zahlteniu
9  if (writenb != (ssize_t)(2U + TAG_LEN + len)) {
10  writenb = safe_write(context->client_fd, tun_buf.len + writenb,
11  2U + TAG_LEN + len - writenb, TIMEOUT);
12 }
```

V princípe celá logika tejto VPN pozostáva z kontroly obsahu tunelovacieho rozhrania a aktívnych soketov. Následne ak sa na vstupoch nachádzajú dáta dochádza k ich čítaniu, dešifrovaniu a následnému odoslaniu dát aplikácií, ktorej prislúchajú.

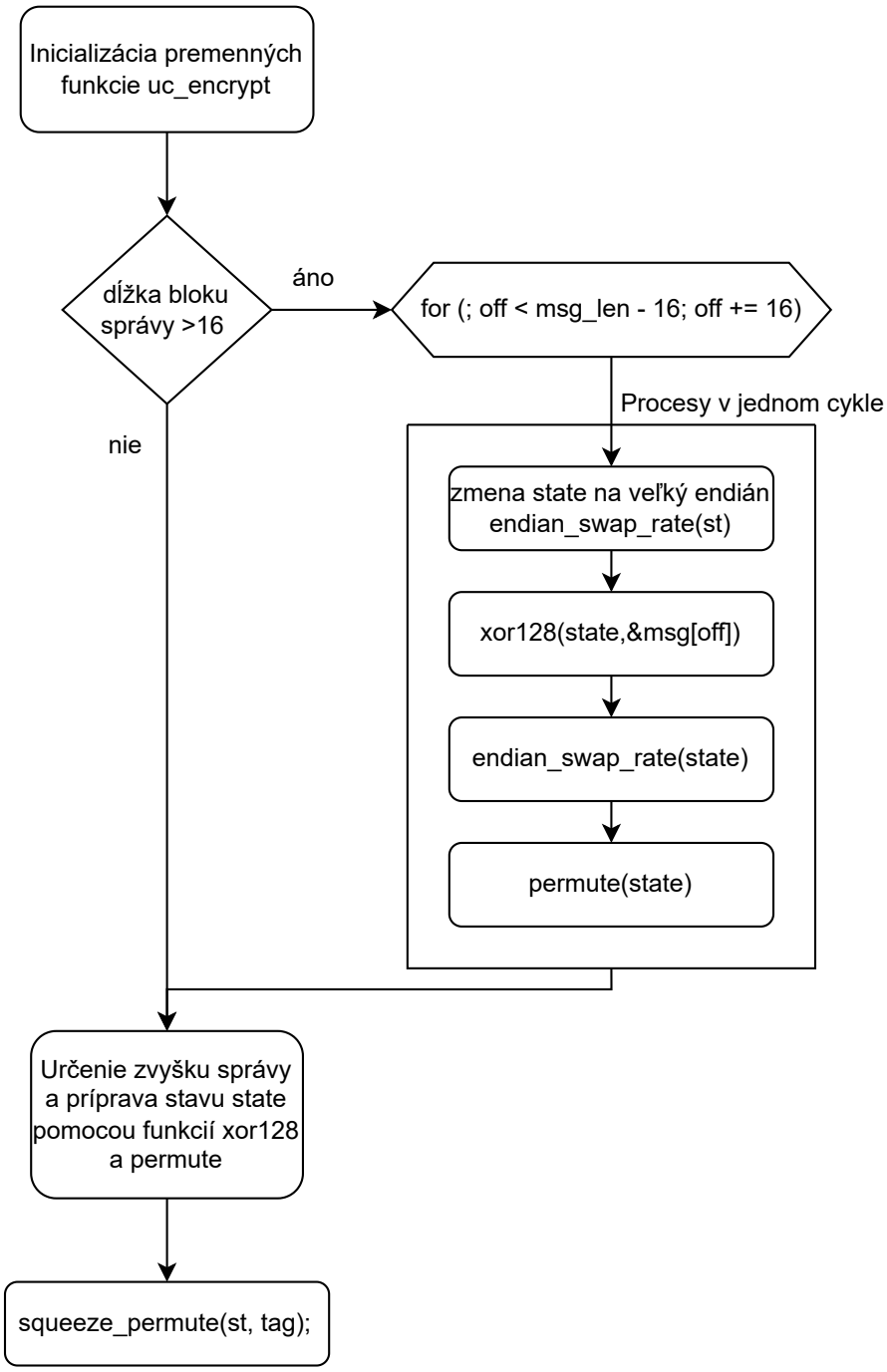
Na konci funkcie `event_loop()` sa nachádza ešte overenie pre prípad keď buffer s dátami nie je úplne plný. V tomto prípade funkcia vykonáva posun uložených bajtov na začiatok buffera. Inými slovami pripravuje dáta na ďalšie čítanie. Vyššie uvedené výroky sú znázornené v zdrojovom kóde 4.8.

Zdrojový kód 4.8: Príprava dát na ďalšie čítanie

```
1
2   if (2 + TAG_LEN + MAX_PACKET_LEN != len_with_header) {
3     unsigned char *rbuf      = client_buf->len;
4     size_t         remaining = client_buf->pos - len_with_header;
5     memmove(rbuf, rbuf + len_with_header, remaining);
6   }
7   client_buf->pos -= len_with_header;
```

Šifrovanie a dešifrovanie

Proces šifrovanie resp. dešifrovania správy nastáva na oboch stranách spojenia, teda pri klientovi aj serveri. Zdrojový kód 4.9 demonštruje šifrovanie implementované v funkcii `uc_encrypt()`. Tento proces sme sa pokúsili opísať v grafe 4.8.



Obr. 4.8: Proces šifrovania v funkcii uc_encrypt()

Zdrojový kód 4.9: Šifrovanie správy pomocou uc_encrypt

```
1
2  void uc_encrypt(uint32_t st[12], unsigned char *msg,
3                  size_t msg_len, unsigned char tag[16])
4  { //spracovanie po 16 znakov
5    unsigned char squeezed[16];
6    unsigned char padded[16 + 1];
7    size_t        off = 0;
8    size_t        leftover;
9
10   if (msg_len > 16) {
11     for (; off < msg_len - 16; off += 16) {
12       endian_swap_rate(st);
13       memcpy(squeezed, st, 16);
14       xor128(st, &msg[off]);
15       endian_swap_rate(st);
16       xor128(&msg[off], squeezed);
17       permute(st);
18     }
19   }
20   leftover = msg_len - off;
21   memset(padded, 0, 16);
22   mem_cpy(padded, &msg[off], leftover);
23   padded[leftover] = 0x80;
24   endian_swap_rate(st);
25   memcpy(squeezed, st, 16);
26   xor128(st, padded);
27   endian_swap_rate(st);
28   st[11] ^= (1UL << 24 | (uint32_t) leftover >> 4 << 25
29              | 1UL << 26);
30   xor128(padded, squeezed);
31   mem_cpy(&msg[off], padded, leftover);
32   permute(st);
33   squeeze_permute(st, tag);   //vytvorenie tagu
34 }
```

Ako je viditeľne v kóde dochádza k častému použitiu 2 funkcií. Nimi sú `xor128()` a `permute()`. Jedná sa o pomerne dôležité bloky pre správne fungovanie šifrovacieho algoritmu. Obsah prvej z uvedených je preto znázornený v zdrojovom kóde

4.10.

Zdrojový kód 4.10: Funkcia xor128

```
1  static inline void xor128(void *out, const void *in)
2  {
3    #ifdef __SSSE3__
4    _mm_storeu_si128((__m128i *) out,
5    _mm_xor_si128(_mm_loadu_si128((const __m128i *) out),
6    _mm_loadu_si128((const __m128i *) in)));
7    #else
8    unsigned char *      out_ = (unsigned char *) out;
9    const unsigned char *in_  = (const unsigned char *) in;
10   size_t               i;
11
12   for (i = 0; i < 16; i++) {  //xororvanie jednotlivych znakov
13     out_[i] ^= in_[i];    //v 16 bitovom bloku spravy
14   }
15   #endif
16 }
```

Funkcia `permute()` je pomerne rozsiahla. Jej obsah sa rozprestiera na 100 riadkoch. Jedná sa o implementáciu XOODOO permutácie, ktorá bola opísaná v kapitole číslo 2. Funkcionalita závisí od prostredia a procesorových inštrukcií. Samozrejmosťou je softvérová implementácia, ktorá je aj použitá pri behu. Dôvodom je, že naše zariadenie nemá k dispozícií uvedené procesorové inštrukcie. Blok, ktorý je použitý pri volaní sa nachádza v 4.11.

Zdrojový kód 4.11: Funkcia Permute + makrá

```
#define ROTR32(x, b) (uint32_t)(((x) >> (b)) |
                     ((x) << (32 - (b))))
#define SWAP32(s, u, v)        \
do {                           \
  t      = (s)[u];             \
  (s)[u] = (s)[v], (s)[v] = t; \
} while (0)

static void permute(uint32_t st[12])
{
  uint32_t e[4], a, b, c, t, r, i;
  for (r = 0; r < XOODOO_ROUNDS; r++) {
    for (i = 0; i < 4; i++) {
      e[i] = ROTR32(st[i] ^ st[i + 4] ^ st[i + 8], 18);
      e[i] ^= ROTR32(e[i], 9);
    }
    for (i = 0; i < 12; i++) {
      st[i] ^= e[(i - 1) & 3];
    }
    SWAP32(st, 7, 4);
    SWAP32(st, 7, 5);
    SWAP32(st, 7, 6);
    st[0] ^= RK[r];
    for (i = 0; i < 4; i++) {
      a         = st[i];
      b         = st[i + 4];
      c         = ROTR32(st[i + 8], 21);
      st[i + 8] = ROTR32((b & ~a) ^ c, 24);
      st[i + 4] = ROTR32((a & ~c) ^ b, 31);
      st[i] ^= c & ~b;
    }
    SWAP32(st, 8, 10);
    SWAP32(st, 9, 11);
  }
}
```

Na druhej strane k **dešifrovaniu** dochádza v momente ak sa v klientskom sokete nachádzajú dáta. Program overuje či sú v premennej `fds[POLLFD_CLIENT]`

dáta určené na spracovanie. `fds[POLLFD_CLIENT]` predstavuje smerník na pamäťový blok, kde je uložený soket na druhú stranu spojenia, teda akceptovaného klienta. V momente keď sa tam nachádzajú dáta dôjde k ich čítaniu pomocou funkcie `safe_read_partial()`. Následne program realizuje dešifrovanie funkciou `uc_decrypt()`, ktorá je inverznou k `uc_encrypt()`. V ukážke zdrojového kódu 4.12 a 4.13, si používateľ môže prezrieť obsah funkcie `uc_decrypt()`. Po odšifrovaní dát ich program zapisuje do tunelovacieho rozhrania prostredníctvom funkcie `tun_write()`. Následne TUN rozhranie dotvára paket a vloží ho do sieťového zásobníka[10]. OS následne vykonáva ďalšie spracovanie paketu do nižších vrstiev sieťového modelu.

[10] z ang. *network stack*

Zdrojový kód 4.12: Zdrojový kód funkcie na dešifrovanie správy (1.časť)

```
1   int uc_decrypt(uint32_t st[12], unsigned char *msg,
2                  size_t msg_len,
3                  const unsigned char *expected_tag,
4                  size_t expected_tag_len)
5   {
6     unsigned char tag[16];
7     unsigned char squeezed[16];
8     unsigned char padded[16 + 1];
9     size_t        off = 0;
10    size_t        leftover;
11
12    if (msg_len > 16) {
13      for (; off < msg_len - 16; off += 16) {
14        endian_swap_rate(st);
15        memcpy(squeezed, st, 16);
16        xor128(&msg[off], squeezed);
17        xor128(st, &msg[off]);
18        endian_swap_rate(st);
19        permute(st);
20      }
21    }
22    leftover = msg_len - off;
23    memset(padded, 0, 16);
24    mem_cpy(padded, &msg[off], leftover);
25    endian_swap_rate(st);
26    memset(squeezed, 0, 16);
27    mem_cpy(squeezed,
28            (const unsigned char *) (const void *) st,
29            leftover
30            );
31      xor128(&padded, squeezed);
32    padded[leftover] = 0x80;
33    xor128(st, padded);
34    endian_swap_rate(st);
35    st[11] ^= (1UL << 24 | (uint32_t) leftover >> 4
36    << 25 | 1UL << 26);
37    mem_cpy(&msg[off], padded, leftover);
```

Zdrojový kód 4.13: Zdrojový kód funkcie na dešifrovanie správy (2.časť)

```
38      permute(st);
39      squeeze_permute(st, tag);
40
41      if (equals(expected_tag, tag, expected_tag_len) == 0) {
42        memset(msg, 0, msg_len);
43        return -1;
44      }
45      return 0;
46    }
```

Pre čitateľa sme si pripravili aj názornú ukážku šifrovania a prenosu týchto dát cez soket. Za účelom ukážky použijeme už spomínaný nástroj Wireshark na zachytávanie premávky v sieťových rozhraniach. Prechod dát z pohľadu klienta medzi zariadeniami je znázornený na obrázku **??**. Na VPN klientovi sme spustili

25	3.043615224	10.8.2.1	10.8.1.1	ICMP	100	Echo (ping) request
26	3.043662924	192.168.88.25	192.168.88.36	TCP	160	33000 → 2340 [PSH,
27	3.059505089	192.168.88.36	192.168.88.25	TCP	160	2340 → 33000 [PSH,
28	3.059535035	192.168.88.25	192.168.88.36	TCP	68	33000 → 2340 [ACK]
29	3.066620423	10.8.1.1	10.8.2.1	ICMP	100	Echo (ping) reply

Obr. 4.9: Prenos zašifrovaných dát z VPN klienta na VPN server

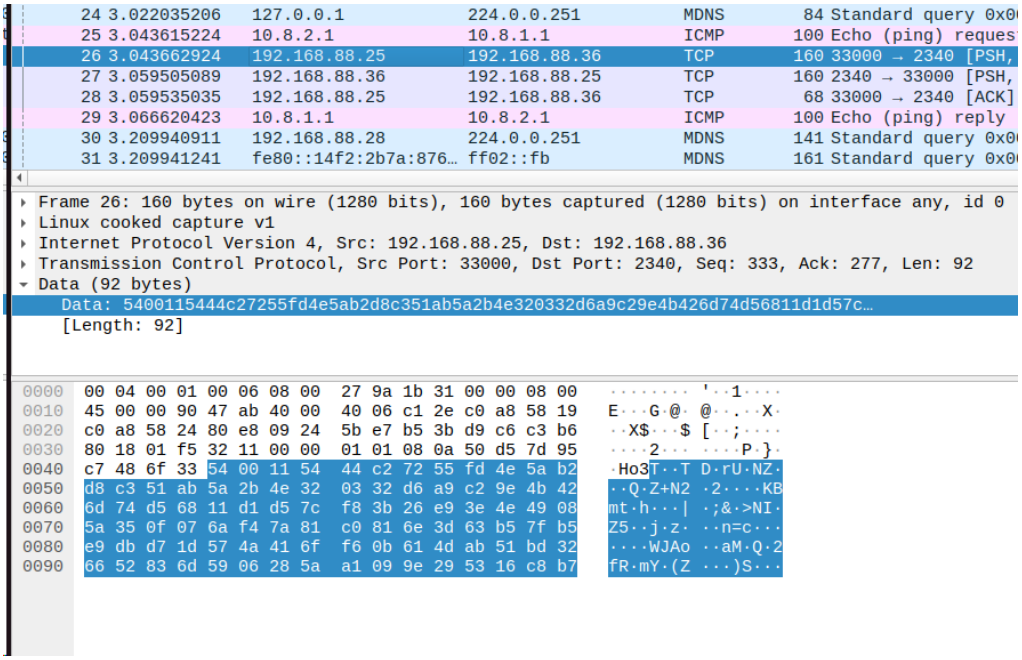
v príkazovom riadku jednoduchý príkaz `ping 10.8.1.1`. Ten nám vytvára veľmi známy ICMP paket so žiadosťou o odpoveď[11]. Obsah tohto paketu je možné si pozrieť v obrázku 4.10. Tento paket je vo VPN klinetovi po spracovaní vo DSVPN

```
00 04 ff fe 00 00 00 00  00 00 00 00 00 00 08 00   ........ ........
45 00 00 54 4f 13 40 00  40 01 d4 84 0a 08 02 01   E··TO·@· @·......
0a 08 01 01 08 00 0c 19  00 01 00 06 c0 92 35 64   ........ ......5d
00 00 00 00 30 16 07 00  00 00 00 00 10 11 12 13   ····0··· ........
14 15 16 17 18 19 1a 1b  1c 1d 1e 1f 20 21 22 23   ........ ···· !"#
24 25 26 27 28 29 2a 2b  2c 2d 2e 2f 30 31 32 33   $%&'()*+ ,-./0123
34 35 36 37                                        4567
```

Obr. 4.10: Obsah pôvodného paketu s ICMP žiadosťou

zašifrovaný pomocou XOODOO permutácie a odoslaný cez soketové TCP spojenie[12]. Obsah zašifrovaného paketu je znázornený v 4.11. Označená časť paketu v obrázku je už spomenutý zašifrovaný ICMP paket z obrázku 4.10. VPN server

[11] paket číslo 25
[12] paket číslo 26



Obr. 4.11: Zašifrovaný pôvodný ICMP paket v novom TCP pakete

ho po prijatí dešifruje a posiela na vlastné tunelovacie rozhranie. Odiaľ prevezme paket OS a vygeneruje odpoveď na pôvodnú požiadavku vo forme nového paketu. Ten je prenesený na klienta opäť cez soketové TCP spojenie[13]. Paket číslo 28 je odpoveď na úspešne prijatie. DSVPN spracuje tento paket a zapíše pôvodný na tunelovacie rozhranie.. Z hľadiska klienta už vidíme dešifrovanú odpoveď v pakete číslo 29, ktorú spracuje OS.

Ukončenie činnosti programu

Používateľ má možnosť ukončiť vzniknuté VPN spojenie medzi klientom a serverom pomocou jednoduchého príkazu `crtl + c`. Program zaznamená tento vstup a ukončí cyklus. Po návrate až do hlavnej funckie `main()` sa z OS vymažú aplikované FW pravidlá a používateľ môže používať počítač ako pred spustením DSVPN.

4.6 Analýza výpočtových nárokov DSVPN

Za účelom analýzy výpočtových nárokov sme sa v práci zamerali na meranie potrebného počtu cyklov a zároveň aj času, potrebného na vykonanie šifrovanie a dešifrovanie permutacie XOODOO. Pre linuxovú platformu sme použili niektoré zabudované funkcie ako je `clock()`. Pomocou nej sme zmerali čas vykonávania.

[13] paket číslo 27

Ukážka ako bol kód na meranie implementovaný do DSVPN je možné vidieť v ukážke zdrojového kódu

Zdrojový kód 4.14: Ukážka spôsobu merania pri (de)šifrovaní XOODOO

```
1     uint64_t time,tick; //inicializacia premennych
2     clock_t t = clock();
3
4     tick = cpucyclesS();
5     codeTomeassure();   //funkcia urcena na meranie
6     time = cpucyclesE()-tick;
7
8     t = clock() - t;    //vyhodnotenie
9     double time_taken = ((double)t)/CLOCKS_PER_SEC;
10    printf(Size(%ld B): "%llu␣cycles␣in␣%f␣s\n", msg_length,
11           time, time_taken);
```

V prípade, ak by používateľ potreboval vykonať podobné meranie času v prostredí OS Windows, tak dávame do pozornosti tento variant. Viď zdrojový kód 4.15. Viac o použití a limitáciách je možné nájsť v práci [76].

Zdrojový kód 4.15: Meranie času vykonávania na OS Windows

```
#include <windows.h>

#define TIMER_INIT \  //inicializacia makra
LARGE_INTEGER frequency; \
LARGE_INTEGER t1,t2; \
double elapsedTime; \
QueryPerformanceFrequency(&frequency);

#define TIMER_START QueryPerformanceCounter(&t1);
#define TIMER_STOP \
QueryPerformanceCounter(&t2); \
elapsedTime=(double)(t2.QuadPart-t1.QuadPart)
            /frequency.QuadPart;

TIMER_INIT    \\pouzitie
{
  TIMER_START
  codeToMeasure()
  TIMER_STOP
}
```

Spojenie medzi klientom a serverom je spustené na VM. Šifrovanie bolo merané na strane Klienta. Dešifrovanie sme merali v OS servera. Počas spojenia sme sa snažili naviesť stav väčšieho sieťového zataženia. Tento úkon sme zrealizovali otvorením viacerých rôznych kariet vo webovom prehliadači. Týmto spôsobom sme odsimuloval potrebu prenosu vačšieho množstva dát naprieč sieťou. Najčastejšie však do šifrovania vstupovali TCP pakety, ktoré majú za úlohu udržať soketové TCP spojenie. Ich veľkosť je 52bajtov. Z výpisu sme ešte na základe pozorovania sledovali aké veľkosti sa najčastejšie vyskytujú. Na základe takto získaných dát sme vypracovali tabuľku 4.1. Hodnoty pre jednotlivé stĺpce boli získane ako aritmetický priemer z meraní. Pakety, ktoré vstupovali do šifrovania, resp. dešifrovania menej ako 20-krát sme z meraní vypustili. Dôvodom je snaha o získanie lepších štatistických údajov. Obdobne sme pri výpočte aritmetického priemeru nebrali do úvahy príliš veľké výchylky pri meraní počtu cyklov. Uvedené chyby merania spôsobil samotný OS, kvôli prerušovania činnosti programu DSVPN.

Súčasťou príloh sú súbory s logmi, z ktorých bola tabuľka vytvorená. Zároveň prikladáme aj zdrojový kód programu, ktorý bol použitý na spracovanie výsled-

Veľkosť dát[B] X **Počet meraní**	**Názov meranej funkcie**			
	uc_encrypt()		**uc_decrypt()**	
	Počet cyklov	Čas vykonávania [s]	Počet cyklov	Čas vykonávania [s]
52 X 1376	1 648	0.000 076	2 377	0.000 062
60 X 79	2 527	0.000 049	2 769	0.000 044
64 X 47	1 799	0.000 048	2 341	0.000 043
80 X 145	2 853	0.000 053	3 343	0.000 057
91 X 128	2 944	0.000 046	2 965	0.000 042
116 X 35	4 083	0.000 049	2 878	0.000 049
222 X 23	4 220	0.000 345	3 976	0.000 040
569 X 51	10 245	0.000 950	8 574	0.000 043
1 385 X 147	21 232	0.000 165	19 046	0.000 045
1 452 X 14	22 599	0.000 077	20 083	0.000 532
Celkom [B]	-	-	-	-
364 656	74 150	0.001 858	68 352	0.000 957

Tabuľka 4.1: Výsledky získaných meraní

kov. Viď priečinok `Measurements`.

4.7 Windows kompatibilita

V tejto práci sme sa zamerali aj na rozšírenie pôvodnej funkcionality DSVPN o kompatibilitu na OS Windows. Rozšírenie by slúžilo vo veľkej miere ako edukatívny vzor pri jednoduchom vysvetlení problematiky VPN na rôznych platformách. Samozrejmosťou ostáva zachovanie pôvodnej kompatibility. V rámci tejto činnosti sme museli pridať, resp. zmeniť niektoré funkcionality. V priebehu tejto podkapitoly stručne opíšeme aplikované zmeny. Rád by som ešte upozornil čitateľa na použitie prepocesorov **`#ifdef`** a **`#ifndef`**. Vďaka ním sme dokázali podmienene určiť čo má program realizovať v danom OS.

4.7.1 Hlavičkové súbory

Prvotnou činnosťou bolo získanie vedomostí o potrebných knižniciach, ktoré v prípade soketového programovania potrebujeme. Priebežnou analýzou pôvodného riešenie sme zároveň vyhľadávali dané realizácie aj pre OS Windows. Obdobne pri snahe o priebežné ladenie programu, nám prekladač vypísal niektoré problémy týkajúce sa nedostupných knižníc pre Windows. Vo výsledku sme pre Windows platformu potrebovali vložiť 13 hlavičkových knižníc a zadefinovať niektoré makrá, ktoré sa vo Windows volajú inak. Aplikované zmeny je možné si

pozrieť v ukážke zdrojového kódu 4.17.

4.7.2 Funkcia generovania náhodných čísel

Implementácia XOODOO používa pri svojej činnosti aj náhodné dáta. V pôvodnej implementácií DSVPN sa tieto dáta získavajú pomocou systémových generátorov náhodných čísel. V prípade OS Linux sa tento úkon realizuje tradične systémovým volaním funkcie `SYS_getrandom()`. V prípade ostatných podporovaných OS sa volá funkcia `arc4random_buf()`. Na základe odporúčaní z [76] sme do DSVPN zakomponovali pre prípad OS Windowsu volanie funkcie `BCryptGenRandom()`. Toto rozhranie je aktuálne na Windowse odporúčaný spôsob generovania náhodných dát pre kryptografické účely. Výstupom volania sú dáta z kryptograficky bezpečného pseudonáhodného generátora náhodných čísel. V ukážke zdrojového kódu 4.16 je znázornená modifikovaná funkcia na získavanie náhodných dát.

Zdrojový kód 4.16: Ukážka zdrojového kódu na získanie náhodných dát

```
1  void uc_randombytes_buf(void *buf, size_t len)
2    {
3      #ifdef _WIN32
4      if (STATUS_SUCCESS!=BCryptGenRandom(NULL,buf,len,
5                           BCRYPT_USE_SYSTEM_PREFERRED_RNG))
6      {
7        puts("BCRYPTGENRANDOM␣ERROR");
8      }
9      #elif defined(__linux__)
10     if ((size_t) syscall(SYS_getrandom, buf, (int) len, 0)
11                           != len)
12     {
13         abort();
14     }
15     #else
16     arc4random_buf(buf, len);
17     #endif
18   }
```

Viac o problematike generovania náhodných čísel je možné si prečítať v [76].

4.7.3 Soketová kompatibilita

Windows v porovnaní s Linuxom pracuje s iným typom soketov. Ako sme videli v útržkoch kódu vyššie, linux pri vytváraní soketu vracia hodnotu typu **int**, ktorá odkazuje na vytvorený soket. Windows na druhej strane vracia štruktúru typu SOCKET. Pretypovanie z jedného typu na druhý bohužiaľ nie je možné jednoduchým riešením. Elegantné riešenie tohto problému bolo vytvorenie makra, ktoré prekladaču povedalo aký typ premennej má použiť. Za týmto účelom sme zadefinovali nové makro SCK. Viď. ukážku zdrojového kódu 4.17.

Následne postačilo len zmeniť deklarácie a definície všetkých soketových premenných vstupujúcich do funkcie. Rovnaký scenár platil aj pri vytváraní soketových premennych. Jednoducho sme nahradili každý soket typu **int** za SCK.

Zdrojový kód 4.17: Makrá a hlavičkové súbory pridané do vpn.h

```
#ifdef _WIN32
#ifndef _WIN32_WINNT
#define _WIN32_WINNT 0x0600
#endif
#include <stdbool.h>
#include <winsock2.h>
#include <ws2tcpip.h>
#include <winioctl.h>
#include <windows.h>
#include <io.h>
#include <tchar.h>
#include <ws2ipdef.h>
#include <iphlpapi.h>
#include <mstcpip.h>
#include <winternl.h>
#include <stdarg.h>
#include "wintun.h"
#define SCK   SOCKET
#define SCKERR INVALID_SOCKET
#define INVSCK INVALID_SOCKET
#define nfds_t ULONG
#define TCP_DEFER_ACCEPT SO_ACCEPTCONN
#define SOL_TCP IPPROTO_TCP
#define IFNAMSIZ 256
#define SIOCSIFMTU 0x8922
#else
#include <sys/wait.h>
#include <sys/ioctl.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <sys/uio.h>
#include <net/if.h>
#include <netinet/in.h>
#include <netinet/tcp.h>
#include <netdb.h>
#include <poll.h>
#define SCK   int
#define INVSCK -1
#define SCKERR 0
#endif
```

Nasledovala práca so soketmi. Princíp fungovania soketov na strane klienta a servera ostáva v svojej podstate rovnaký. Odlišný je spôsob inicializácie, zápisu, čítania, konfigurácie a manipulácie s nimi. V tomto smere nám vo veľkej miere pomohol návod v [88]. Autor názorným spôsobom vysvetľuje princíp fungovania, práce a manipulácie so soketmi. Ak používateľ nie je zorientovaný v oblasti soketového programovania, tak by si mál túto krátku prezentáciu prečítať.

Pred samotným používaním soketov je potrebné inicializovať tzv. Winsock. Tento úkon sme realizovali v `main()` funkcii. Viď ukážku zdrojového kódu 4.18.

Zdrojový kód 4.18: Inicializácie knižnice na prácu so soketmi

```
1  #ifdef _WIN32
2  WSADATA wsa;
3  // Initialize Winsock
4  if (WSAStartup(MAKEWORD(2,2), &wsa) != 0) {
5    printf("WSAStartup␣failed.␣Error␣Code␣:␣%d",
6            WSAGetLastError());
7    return 1;
8  }
9  #endif
```

V kóde sme potrebovali zmeniť spôsob čítania a zápisu dát do soketov. Za týmto účelom sme vytvorili viacero funkcií, ktoré danú činnosť realizujú. Ich volanie sa vykonáva v pôvodnej funkcionalite pomocou preprocesora **`#ifdef _WIN32`**. Ak je teda potrebné aby pôvodná implementácia bola vykonaná odlišne, tak sme pomocou tohto makra volali nami vytvorenú funkciu. Celkovo sme takýmto spôsobom vytvorili 9 funkcií, ktoré sú obsahom os.c. Do os.h sme pridali deklarácie:

- `int SCK_close(SCK fd);`,
- `int w_open(const char *filename, int flag);`,
- `int w_read(int file, unsigned char *buf, size_t count);`,
- `int w_close(int file);`,
- `ssize_t w_read_file(const int fd, void *const buf_,`
  `size_t count);`,
- `ssize_t w_safe_read(const SOCKET fd, void *const buf_,`
  `size_t count, const int timeout);`,

- `ssize_t w_safe_write(const SOCKET fd, const void *const buf_, size_t count, const int timeout);`,
- `ssize_t w_safe_read_partial(const SOCKET fd, void *const buf_, const size_t max_count);`,
- `ssize_t w_safe_write_partial(const SOCKET fd, void *const buf_, const size_t max_count);`.

Z názvov je jednoznačne jasné čo dané funkcie vykonávajú.

Pri Windows OS na konci každého programu ešte potrebujeme zavolať `WSACleanup();`. Za spomenutie ešte stojí získavanie chýbových hlášok zo soketov. V Linuxe sa v prípade cchyby ukladá chybová hláška do globálnej premennej `errno`. Z nej následne programátor dokáže určiť kde mohol nastať problém. V prípade Windowsu potrebuje používateľ zavolať `WSAGetLastError()`. Následne má k dispozícií kód chybovej hlášky. Zoznam a popis chybových hlášok je dostupný na internete[14].

4.7.4 Tunelovacie rozhranie

Windows nemá natívnu podporu na vytváranie virtuálnych sieťových adaptérov. Na vyriešenie problému sme mali niekoľko možností. Prvá bola implementácia vlastného ovládača, ktorý by pracoval s jadrom OS. Drúha realnejšia možnosť bola použitie už vytvorenej implementácie. Tu sa nám naskytovali dve možné riešenie. OpenVPN TUN/TAP a Wintun adaptér. Z hľadiska použiteľnosti sa však Wintun adaptér pozdáva ako omnoho lepšie riešenie. Dôvodom je jeho použitie bez potreby inštalácie dodatočného softvéru. Do kódu DSVPN sme implementovali funkcie na inicializáciu, čitanie, zápis a ukončenie ovládača. Jedná sa o funkcie:

- `int wtunInit(Context *context);` – realizuje načítanie wintun.dll knižnice, vytvorenie adaptéru, spustenie prevádzky a jeho uloženie do štruktúry.
- `int tun_reader(Context *context, unsigned char *outgoingData)` – číta pakety, ktoré dorazia na rozhranie a ukladá ich do premennej outgoingData,

[14] `https://learn.microsoft.com/en-us/windows/win32/winsock/windows-sockets-error-codes-2`

- `int tun_writer(Context *context, BYTE *PacketData, ssize_t PacketDataSize)` – zapisuje dáta v premennej PacketData s veľkosťou PacketDataSize do tunelovacieho rozhrania,
- `void w_cleanUp(Context *context)` – obsahuje funkcie z wintun.dll na bezpečné ukončenie prevádzky a vymazanie adaptéra.

Tieto funkcie sme vložili v DSVPN do súbora vpn.c. Dôvodom bolo posielanie celej štuktúry do týchto funkcií. Na správne fungovanie potrebuje premennú typu `WINTUN_SESSION_HANDLE`. Pri vytváraní adaptéra sa do nej uložia informácie o tunelovacom rozhraní. Na základe nich dokážeme následne čítať a zapisovať dáta na rozhranie. V budúcnosti by sa kvôli optimalizácií využitia pamäte odporúčalo zmeniť túto funkciu aby pracovalo len s premennou `WINTUN_SESSION_HANDLE` a nie celou štruktúrou.

Na rozdiel od pôvodnej implementácie nastal pri tomto adaptéry problém. V DSVPN na Linuxe dokážeme monitorovať stav rozhrania. Dokáže teda určiť či sa v ňom nachádzajú dáta na čítanie, nastala chyba alebo môžeme zapisovať dáta. V prípade wintunu sme však k takejto možnosti monitorovania nenašli informácie. Z uvedeného dôvodu sme museli pozmeniť logiku v kóde. Pôvodne sa z tunelovacieho rozhrania číta len ak rozhranie obsahuje nejaké dáta určené na čítanie. V aktualizovanej verzii čítame dáta z rozhrania pri každom cykle vo funkcii `event_loop()`. OS Windows podporuje možnosť počkať na ukončenie procesu. Zároveň wintun implementuje funkcionalitu čakania na prichádzajúce dáta ak momentálne na rozhraní žiadne nie sú. Do `tun_reader()` sme teda implementovali túto logiku. Vo výsledku funkcia skončí až keď nastane prečítanie nejakých dát. Tento úkon bude v prípade bežnej prevádzky potrebné zmeniť. Dôvodom je, že môže nastať blokovanie DSVPN v stave čítania z rozhrania. Pri experimentovaní sme nenastavovali filtrovacie pravidlá a ani raz spomenutý scenár nestal.

4.7.5 Smerovanie a Firewall pravidlá

Posledný z krokov, ktorý je dôležitý, je aplikovanie firewall pravidiel a smerovacích ciest. V pôvodnej implementácií sa nachádzajú niektoré pravidlá, ktoré sa na bežnom Windowse nemožno uskutočniť. Pri experimentoch sme sa rozhodli preto Firewall pravidla na vypustiť. Pomocou príkazov ho vypíname. Okrem tejto zmeny je rozdiel aj v nastavovaní prednastavených pravidiel smerovania. Pri experimentoch sme aplikovali rôzne pravidla. Obsah príkazov pre príkazový riadok je možné vidieť v ukážke kódov 4.19 a **??**.

Zdrojový kód 4.19: CMD pravidla použité na VPN Serveri

```
1  //SERVER
2  *set_cmds[] ={
3  //nastavuje ipv4 adresu
4  "netsh␣interface␣ipv4␣add␣address␣$IF_NAME
5  $LOCAL_TUN_IP␣255.255.255.255",
6
7  //nastavuje ipv6 adresu na wtun adapter
8  "netsh␣interface␣ipv6␣add␣address␣$IF_NAME␣$LOCAL_TUN_IP6/96",
9
10 //vypina firewall
11 "netsh␣advfirewall␣set␣allprofiles␣state␣off",
12
13 //default MTU je 9000
14 "netsh␣interface␣ipv4␣set␣subinterface
15 $IF_NAME␣mtu=9000␣store=persistent",
16
17 //aby mohlo z jedneho interfacu na druhy preposielat pakety
18 "netsh␣interface␣ipv4␣set␣interface␣$IF_NAME
19 forwarding=enabled",
20
21 //(powershell) globalny pre vsetky vyssie
22 //"Set-NetIPInterface -Forwarding disabled",
23
24 //zapina interface ak by bol vypnuty
25 "netsh␣interface␣set␣interface␣$IF_NAME␣admin=enable",
26
27 //ak je destinacia ip remote tunela,
28 //tak posli na local tunel rozhranie
29 "route␣add␣$REMOTE_TUN_IP␣mask␣255.255.255.255␣$LOCAL_TUN_IP",
30
31 NULL },
```

Zdrojový kód 4.20: CMD pravidla použité na VPN Klientovi

```
1  //KLIENT
2  *set_cmds[] = {
3  "netsh␣advfirewall␣set␣allprofiles␣state␣off",
4  "netsh␣interface␣ipv4␣set␣subinterface␣$IF_NAME␣mtu=9000
5  store=persistent",
6  "netsh␣interface␣ipv4␣set␣interface␣$IF_NAME␣forwarding=enabled",
7  "netsh␣interface␣ipv4␣add␣address␣$IF_NAME
8  $LOCAL_TUN_IP␣255.255.255.255",
9
10 //pri testovani vypnute
11 //"netsh interface ipv6 add address $IF_NAME $LOCAL_TUN_IP6/96",
12
13 "netsh␣interface␣set␣interface␣$IF_NAME␣admin=enable",
14
15 //paket so s cielom server ip ma ist cez povodny gw
16 "route␣add␣$EXT_IP␣mask␣255.255.255.255␣$EXT_GW_IP",
17
18 "route␣delete␣0.0.0.0", //maze defaultnu rutu
19
20 //presmeruje kazdu IP, ktoru nepozna na lokalny tunel
21 "route␣add␣0.0.0.0␣mask␣0.0.0.0␣$LOCAL_TUN_IP",
22
23 NULL },
24
25 //$IF_NAME = wtun0
26 //$LOCAL_TUN_IP = ip lokalneho tunela
27 //$REMOTE_TUN_IP = ip tunela na druhej stane
28 //$EXT_IP = ip VPN servera
29 //$EXT_GW_IP = povodna Gateway na zariadeni
```

4.7.6 Preklad a ladenie zdrojových kódov

Zmeny vykonané s cieľom vytvorenia kompatibility si vyžadovali použitie iného balíka Make, ktorý by fungoval správne vo Windowse. Prepisovaný kód sme natvrdo premiestnili do jedného priečinka. Teda všetky pôvodné hlavičkové súbory a zdrojové kódy z DSVPN sme vložili do jedného priečinka. K nim sme priložili `wintun.h` a `wintun.dll`. Následne sme vytvorili jednoduchý Makefile

na kompiláciu programu do funkčného celku. Ukážku jednoduchého zdrojového kódu je možné vidieť v 4.21.

Zdrojový kód 4.21: Ukážka zdrojového kódu v balíku Make

```
1  CC=gcc
2  CFLAGS= -Wall -Wextra -g
3  WINFLAGS = -lbcrypt -liphlpapi -lws2_32
4  SRCS= dsvpn
5  CHARM=charm.c
6  OS=os.c
7  VPN=vpn.c
8  all: $(SRCS)
9
10 $(SRCS): %:
11 $(CC) -o $< $(SRCS) $(CHARM) $(OS) $(VPN) $(CFLAGS) $(WINFLAGS)
12
13 clean:
14 $(RM) *.exe
```

Pri vývoji softvéru je veľmi dôležitým priateľom programátora aj nástroj na ladenie programov. Z uvedeného dôvodu prikladám aj konfiguráciu pre ladenie v prostredí programu Visual Studio C. Konfiguráciu bolo potrebné pripraviť. V nej ma čitateľ možnosť prečítať si ako daný program spúšťať za účelom ladenia. Dôležité je taktiež, že program Visual Studio Code musí byť pri ladení spustený ako administrátor. Ak tento scénar nenastane, tak nebude možné vytvoriť tunelovacie rozhranie a DSVPN spadne. V ukážke zdrojového kódu 4.22, je možné vidieť potrebnú konfiguráciu `launch.json` na spustenie DSVPN ako klient v režime ladenia.

Zdrojový kód 4.22: Konfigurácia launch.json pre ladenie DSVPN

```
{
  "version": "0.2.0",
  "configurations": [
  {
    "name": "C/C++:␣g++.exe␣build␣and␣debug␣active␣file",
    "type": "cppdbg",
    "request": "launch",
    "program": "${workspaceFolder}/dsvpn.exe",
    "args": ["client", "C:/Users/marro/Desktop/xy/m.key",
    "192.168.88.10", "2340", "auto", "10.8.2.1", "10.8.1.1"],
    "stopAtEntry": false,
    "cwd": "${fileDirname}",
    "environment": [],
    "externalConsole": false,
    "MIMode": "gdb",
    "miDebuggerPath": "C:\\mingw64\\bin\\gdb.exe",
    "setupCommands": [
    {
      "description": "Enable␣pretty-printing␣for␣gdb",
      "text": "-enable-pretty-printing",
      "ignoreFailures": true
    }
    ]
  }
  ]
}
```

5 Vyhodnotenie dosiahnutých výsledkov

V rámci tejto kapitoly zhrnieme dosiahnuté výsledky z analýzy, experimentálnych meraní a rozšírenia kompatibility pre OS Windows.

5.1 Analýza jednoduchej VPN siete

V úvode praktickej časti sme postupnou analýzou prešli jednotlivé časti zdrojového kódu DSVPN. Demonštratívne sme opisovali dôležité bloky a funkcionalitu implementovanú v tejto jednoduchej VPN. Na základe nadobudnutých znalostí sme vypracovali stavové diagramy, ktoré čitateľovi jednoznačne vysvetľujú čo sa v programe realizuje. Na základe týchto informácií, dokáže aj laická verejnosť pochopiť ako sa v praxi realizuje VPN sieťové spojenie.

5.2 Experimentálne meranie autentizovaného šifrovania pomocou permutácie XOODOO

Popri analýze zdrojového kódu sme experimentálne overili funkcionalitu autentizovaného šifrovania pomocou permutácie XOODOO v jednoduchej VPN sieti. Z výsledkov experimentálnych meraní implementácia XOODOO algoritmu, môžeme zhodnotiť, že autentizované šifrovanie a dešifrovanie dokáže držať na úrovní 40-70 mikrosekúnd. Čas vykonávania bol priamo úmerný veľkosti vstupujúcich dát a prostredia, na ktorom aplikácia beží. Na základe meraní môžeme tvrdiť, že sa jedná o extrémne rýchlu a výpočtovo nenáročnú implementáciu. Rád by som však oboznámil čitateľa s faktom, že do meraní nám vo veľkej miere zasahoval OS. Simuláciu bežného prostredia sme uskutočnili za pomoci 2 virtuálnych OS a jedného natívneho OS, na ktorom virtuálizácia prebiehala. Virtualizované zariadenia mali obdobne aj obmedzený výpočtový výkon a pridelené prostriedky. To

sa prejavovalo v pomalších odozvách systémov na požiadavky používateľa. Na základe uvedených faktov konštatujeme, že výsledná rýchlosť na bežnom používateľskom prostredí by mala byť ešte rýchlejšia.

5.3 Výsledky dosiahnuté pri tvorbe kompatibility DSVPN pre OS Windows

Jedným z cieľov našej práce bolo aj rozšírenie pôvodného riešenia DSVPN o kompatibilitu pre OS Windows. Samozrejmosťou bolo zachovanie pôvodnej kompatibility v rámci ostatných OS. V princípe sa nám podarilo vytvoriť tunelové spojenie medzi VPN klientom a VPN serverom so zabezpečeným prenosom dát. Taktiež je možné nadviazať spojenie nie len medzi zariadeniami s OS Windows, ale vieme spojiť ľubovoľné OS z množiny podporovaných DSVPN. Problém riešenia nastáva pri smerovaní paketov naspäť k pôvodnému vlastníkovi. Pomocou sieťových príkazov a monitoringu premávky sa nám nepodarilo doteraz zistiť, prečo sa odpoveď na pôvodnú požiadavku nedokáže vrátiť na zariadenie. Tento úkon sa deje aj napriek tomu, že pri monitoringu vidíme ako paket s odpoveďou opúšťa zariadenie a prichádza na to, ktoré má dostať odpoveď. Súčasná hypotéza je, že nám uniká niečo pri smerovaní paketu z pozície OS. Tento problém sa budeme snažiť ešte v blízkej budúcnosti vyriešiť, nakoľko by sa nemalo jednať o veľký problém. Čitateľ bude mať prístup ku aktuálnemu kódu pomocou stránky github.

6 Záver

Cieľom našej práce bolo uvedenie čitateľa do problematiky VPN sieti so zameraním na použitie algoritmu XOODOO za účelom zabezpečenia komunikácie. V prvej časti práci sme zadefinovali potrebné pojmy spojené s problematikou VPN sietí. Následne sme postupným opisom charakterizovali čo sa deje v sieťach typu VPN. Na základe naštudovanej literatúry sme vytvorili klasifikácie VPN sieti. Rozdelenie bolo realizované podľa logickej topológie a dát vstupujúcich do šifrovania. Pomocou nadobudnutých informácií sme charakterizovali a špecifikovali činnosti implementované v známych VPN protokoloch. V závere prvej kapitoly sme zhrnuli rozdelenie VPN sieti pomocou grafu.

Druhá kapitola sa venuje problematike ľahkej kryptografie. V nej sme charakterizovali tento pomerné nový odbor v oblasti kryptografie. Následne sme pokračovali s opisom XOODOO permutácie, ktorú zaraďujeme do tejto kategórie. XOODOO permutácia je súčasťou kryptografického balíka Xoodyak, ktorý je jedným z finalistov štandardizačného procesu NIST v ľahkej kryptografie. Opísali sme činnosti v algoritme a vysvetlili možnosti použitia.

Tretia kapitola obsahuje postupy prípravy prostredí na experimentovanie. Konfiguráciu virtualizovaných OS a opis jednotlivých nástrojov, ktoré boli pri práci použité. Opísali sme Wintun adaptér na tvorbu virtuálnych tunelovacích rozhraní. Vytvorili sme jednoduché demo, ktoré demonštruje odosielanie a prijímanie ICMP paketov pomocou tohto adaptéru.

Štvrtá kapitola sa zameriava na praktickú demonštráciu jednoduchej VPN siete s využitím XOODOO permutácie. V úvode sme charakterizovali program DSVPN. Následne sme praktickým experimentom overili jeho funkčnosť. Z tejto činnosti sme vypracovali článok do Zborníka vedeckých prác TUKE FEI [89]. Podrobne sme analyzovali zdrojový kód DSVPN. Popri analýze sme názorným spôsobom ukázali čo sa deje s paketmi. Experimentálne sme zmerali rýchlosť a počet cyklov implementácie XOODOO permutácie v DSVPN. Súčasťou našej práce bolo aj vytvorenie kompatibility DSVPN o OS Windows a opis vykonaných zmien.

V piatej kapitole sme zhrnuli dosiahnuté výsledky z praktickej časti práce. Celý obsah prílohy A vrátane samotnej práce sme uverejnili na git stránku Github pod profilom mr171hg. Odkaz je dostupný v prílohe A. Dôvodom je dostupnosť týchto informácií pre každého, kto sa potrebuje v problematike VPN zorientovať.

Pre účely rozšírenia práce by sme navrhovali pokračovať v práci na kompatibilite DSVPN. Rozšíriť implementáciu o viacero kryptografických algoritmov a následne experimentálne merať. Aplikovať program DSVPN na sieťové zariadenie a vyskúšať reálnu prevádzku. Refaktorovať a optimalizovať zdrojový kód. Rozšíriť program o možnosti pripojenia viacerých klientov na jeden server s využitím paralelného programovania s viac vláknami.

Literatúra

1. Generic Routing Encapsulation. [B.r.]. Dostupné tiež z: https://en.wikipedia.org/wiki/Generic_Routing_Encapsulation. [Online; Citované: 31.3.2023].
2. Generic Routing Encapsulation (GRE). [B.r.]. Dostupné tiež z: https://www.rfc-editor.org/info/rfc1701. [Online; Citované: 31.3.2023].
3. Tunneling protocol. [B.r.]. Dostupné tiež z: https://en.wikipedia.org/wiki/Tunneling_protocol. [Online; Citované: 31.3.2023].
4. TUN/TAP. [B.r.]. Dostupné tiež z: https://en.wikipedia.org/wiki/TUN/TAP. [Online; Citované: 31.3.2023].
5. Universal TUN/TAP device driver. [B.r.]. Dostupné tiež z: https://www.kernel.org/doc/Documentation/networking/tuntap.txt. [Online; Citované: 31.3.2023].
6. DRUTAROVSKÝ, Miloš. Bezpečnosť v architektúre TCP/IP II (BIKS pr6). [B.r.]. [Online; Citované:6.2.2022].
7. WIKIPEDIA. Transmission Control Protocol. [B.r.]. Dostupné tiež z: https://en.wikipedia.org/wiki/Transmission_Control_Protocol. [Online; Citované:26.1.2022].
8. Error detection and correction. [B.r.]. Dostupné tiež z: https://en.wikipedia.org/wiki/Error_detection_and_correction. [Online; Citované: 20.3.2023].
9. Segment. [B.r.]. Dostupné tiež z: https://en.wikipedia.org/wiki/Segment. [Online; Citované: 20.3.2023].
10. Flow Control (date). [B.r.]. Dostupné tiež z: https://en.wikipedia.org/wiki/Flow_control_(data). [Online; Citované: 20.3.2023].
11. Network congestion. [B.r.]. Dostupné tiež z: https://en.wikipedia.org/wiki/Network_congestion. [Online; Citované: 20.3.2023].

12. Port (computer networking). [B.r.]. Dostupné tiež z: https://en.wikipedia.org/wiki/Port_(computer_networking). [Online; Citované: 20.3.2023].

13. COMPUTER SCIENCE UPJS, Institute of. 5. Prednáška - Transportná vrstva: Protokol TCP. [B.r.]. Dostupné tiež z: https://siete.ics.upjs.sk/prednaska-5/. [Online; Citované:6.2.2022].

14. S., Sridevi; D.H, Dr.Manjaiah. Technical Overview of Virtual Private Networks (VPNs). *International Journal of Scientific Research*. 2013, roč. 2, č. 7, s. 93–96. Dostupné tiež z: https://www.worldwidejournals.com/international-journal-of-scientific-research-(IJSR)/file.php?val=July_2013_1372777193_d5c86_32.pdf. [Online; Citované: 2.2.2023].

15. ZORN, Glen; PALL, Gurdeep-Singh; HAMZEH, Kory. *Point-to-Point Tunneling Protocol (PPTP)* [RFC 2637]. RFC Editor, 1999. Request for Comments, č. 2637. Dostupné z DOI: 10.17487/RFC2637. [Online; Citované: 2.2.2023].

16. Understanding Point-to-Point Tunneling Protocol (PPTP). 1997. Dostupné tiež z: https://wwwdisc.chimica.unipd.it/luigino.feltre/pubblica/unix/winnt_doc/pppt/understanding_pptp.html. [Online; Citované: 2.2.2023].

17. Challenge-handshake authentication protocol. [B.r.]. Dostupné tiež z: https://cs.wikipedia.org/wiki/Challenge-handshake_authentication_protocol. [Online; Citované: 2.2.2023].

18. MS-CHAP. [B.r.]. Dostupné tiež z: https://en.wikipedia.org/wiki/MS-CHAP. [Online; Citované: 2.2.2023].

19. Password Authentication Protocol. [B.r.]. Dostupné tiež z: https://en.wikipedia.org/wiki/Password_Authentication_Protocol. [Online; Citované: 2.2.2023].

20. RC4. [B.r.]. Dostupné tiež z: https://en.wikipedia.org/wiki/RC4. [Online; Citované: 2.2.2023].

21. TOWNSLEY W. Valencia A., Rubens A.; G., Pall; G., Zorn; PALTER, B. *Layer Two Tunneling Protocol "L2TP"* [RFC 2661]. RFC Editor, [b.r.]. Request for Comments, č. 2661. Dostupné z DOI: 10.17487/RFC2661. [Online; Citované: 2.2.2023].

22. VALENCIA A., Littlewood M.; KOLAR, T. *Cisco Layer Two Forwarding (Protocol) "L2F"* [RFC 2341]. RFC Editor, [b.r.]. Request for Comments, č. 2341. Dostupné z DOI: 10.17487/RFC2341. [Online; Citované: 2.2.2023].

23. Network address translation. [B.r.]. Dostupné tiež z: https://sk.wikipedia.org/wiki/Network_address_translation. [Online; Citované: 4.2.2023].

24. AR500, AR510, AR531, AR550, and AR2500 V200R008 CLI-based Configuration Guide - VPN. [B.r.]. Dostupné tiež z: https://support.huawei.com/enterprise/en/doc/EDOC1000154777/83e5177f/overview. [Online; Citované: 4.2.2023].

25. TOWNSLEY, Mark; GOYRET, Ignacio; LAU, Jed. *Layer Two Tunneling Protocol - Version 3 (L2TPv3)* [RFC 3931]. RFC Editor, 2005. Request for Comments, č. 3931. Dostupné z DOI: 10.17487/RFC3931.

26. Layer 2 Tunneling Protocol. [B.r.]. Dostupné tiež z: https://en.wikipedia.org/wiki/Layer_2_Tunneling_Protocol. [Online; Citované: 4.2.2023].

27. Replay attack. [B.r.]. Dostupné tiež z: https://en.wikipedia.org/wiki/Replay_attack. [Online; Citované: 20.2.2023].

28. Advanced Encryption Standard. [B.r.]. Dostupné tiež z: https://en.wikipedia.org/wiki/Advanced_Encryption_Standard. [Online; Citované: 8.5.2021].

29. RSA (cryptosystem). [B.r.]. Dostupné tiež z: https://en.wikipedia.org/wiki/RSA_(cryptosystem). [Online; Citované: 20.2.2023].

30. Diffie–Hellman key exchange. [B.r.]. Dostupné tiež z: https://en.wikipedia.org/wiki/Diffie%E2%80%93Hellman_key_exchange. [Online; Citované: 20.2.2023].

31. Elliptic Curve Digital Signature Algorithm. [B.r.]. Dostupné tiež z: https://en.wikipedia.org/wiki/Elliptic_Curve_Digital_Signature_Algorithm. [Online; Citované: 20.2.2023].

32. Elliptic-curve Diffie–Hellman. [B.r.]. Dostupné tiež z: https://en.wikipedia.org/wiki/Elliptic-curve_Diffie%E2%80%93Hellman. [Online; Citované: 20.2.2023].

33. HTTPS. [B.r.]. Dostupné tiež z: https://en.wikipedia.org/wiki/HTTPS. [Online; Citované: 20.2.2023].

34. [MS-SSTP]: Secure Socket Tunneling Protocol (SSTP). [B.r.]. Dostupné tiež z: https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-sstp/c50ed240-56f3-4309-8e0c-1644898f0ea8. [Online; Citované: 20.2.2023].

35. 2.2.3 SSTP Data Packet. [B.r.]. Dostupné tiež z: https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-sstp/1e71add9-7243-490a-b816-2174d5b3c179. [Online; Citované: 20.2.2023].

36. 2.2.2 SSTP Control Packet. [B.r.]. Dostupné tiež z: https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-sstp/a961ef2b-daeb-4d1e-bf28-45f10c8ba565. [Online; Citované: 20.2.2023].

37. Transport Layer Security. [B.r.]. Dostupné tiež z: https://en.wikipedia.org/wiki/Transport_Layer_Security#TLS_1.3. [Online; Citované: 21.5.2021].

38. OpenVPN. [B.r.]. Dostupné tiež z: https://en.wikipedia.org/wiki/OpenVPN. [Online; Citované: 20.2.2023].

39. List of BSD operating systems. [B.r.]. Dostupné tiež z: https://en.wikipedia.org/wiki/List_of_BSD_operating_systems. [Online; Citované: 20.2.2023].

40. Curve25519. [B.r.]. Dostupné tiež z: https://en.wikipedia.org/wiki/Curve25519. [Online; Citované: 7.3.2023].

41. Elliptic curve cryptography. [B.r.]. Dostupné tiež z: https://en.wikipedia.org/wiki/Elliptic-curve_cryptography. [Online; Citované: 7.3.2023].

42. ChaCha Variant. [B.r.]. Dostupné tiež z: https://en.wikipedia.org/wiki/Salsa20#ChaCha_variant. [Online; Citované: 7.3.2023].

43. Poly1305. [B.r.]. Dostupné tiež z: https://en.wikipedia.org/wiki/Poly1305. [Online; Citované: 7.3.2023].

44. SipHash. [B.r.]. Dostupné tiež z: https://en.wikipedia.org/wiki/SipHash. [Online; Citované: 7.3.2023].

45. BLAKE2. [B.r.]. Dostupné tiež z: https://en.wikipedia.org/wiki/BLAKE_(hash_function)#BLAKE2. [Online; Citované: 7.3.2023].

46. WireGuard. [B.r.]. Dostupné tiež z: https://en.wikipedia.org/wiki/WireGuard. [Online; Citované: 20.2.2023].

47. WireGuard. [B.r.]. Dostupné tiež z: https://www.wireguard.com/papers/wireguard.pdf. [Online; Citované: 20.2.2023].

48. SRIDEVI, Sridevi; D H, Manjaiah. Technical Overview of Virtual Private Networks(VPNs). *International Journal of Scientific Research*. 2012, roč. 2, s. 93–96. Dostupné z DOI: 10.15373/22778179/JULY2013/32. [Online; Citované: 22.1.2022].

49. Virtual Private Networks Simplified. [B.r.]. Dostupné tiež z: https://www.cisco.com/c/dam/en_us/training-events/le21/le34/downloads/689/academy/2008/sessions/BRK-134T_VPNs_Simplified.pdf. [Online; Citované:26.1.2022].

50. LEVICKÝ, Dušan. *Kryptografia v informačnej bezpečnosti*. Elfa, 2005.

51. PAAR, Christof; POSCHMANN, Axel; SCHMIDT, Markus D. Efficient hardware implementation of the advanced encryption standard (AES). 2005, s. 319–333.

52. DAEMEN, Joan; HOFFERT, Seth; MELLA, Silvia; PEETERS, Michaël; VAN ASSCHE, Gilles; VAN KEER, Ronny. *Xoodyak*. Team Keccak, 2020. Dostupné tiež z: https://keccak.team/xoodyak.html. [Online; Citované:6.2.2022].

53. BASSHAM, Lawrence; ÇALIK, Çağdaş; MCKAY, Kerry; TURAN, Meltem Sönmez. Submission requirements and evaluation criteria for the lightweight cryptography standardization process. *US National Institute of Standards and Technology*. 2018. Dostupné tiež z: https://csrc.nist.gov/CSRC/media/Projects/Lightweight-Cryptography/documents/final-lwc-submission-requirements-august2018.pdf. [Online; Citované: 20.2.2023].

54. KATAGI, Masanobu; MORIAI, Shiho et al. Lightweight cryptography for the internet of things. *sony corporation*. 2008, roč. 2008, s. 7–10. Dostupné tiež z: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=9595b5b8db9777d5795625886418d38864f78bb3. [Online; Citované: 20.2.2023].

55. Cryptographic primitive. [B.r.]. Dostupné tiež z: https://en.wikipedia.org/wiki/Cryptographic_primitive. [Online; Citované: 20.3.2023].

56. DAEMEN, Joan; HOFFERT, Seth; MELLA, Silvia; PEETERS, Michaël; VAN ASSCHE, Gilles; VAN KEER, Ronny. *The Keccak-p Permutations*. Team Keccak, 2020. Dostupné tiež z: https://keccak.team/specifications.html. [Online; Citované:6.2.2022].

57. DAEMEN, Joan; HOFFERT, Seth; MELLA, Silvia; PEETERS, Michaël; VAN ASSCHE, Gilles; VAN KEER, Ronny. *Kravatte*. Team Keccak, 2020. Dostupné tiež z: https://keccak.team/kravatte.html. [Online; Citované:6.2.2022].

58. DAEMEN, Joan; HOFFERT, Seth; MELLA, Silvia; PEETERS, Michaël; VAN ASSCHE, Gilles; VAN KEER, Ronny. *The Keccak-p Permutations*. Team Keccak, 2020. Dostupné tiež z: https://keccak.team/keccakp.html. [Online; Citované:6.2.2022].

59. BERNSTEIN, Daniel J; KÖLBL, Stefan; LUCKS, Stefan; MASSOLINO, Pedro Maat Costa; MENDEL, Florian; NAWAZ, Kashif; SCHNEIDER, Tobias; SCHWABE, Peter; STANDAERT, François-Xavier; TODO, Yosuke et al. Gimli: a cross-platform permutation. In: *International Conference on Cryptographic Hardware and Embedded Systems*. Springer, 2017, s. 299–320.

60. DAEMEN, Joan; HOFFERT, Seth; MELLA, Silvia; PEETERS, Michaël; VAN ASSCHE, Gilles; VAN KEER, Ronny. *Farfalle: parallel permutation-based cryptography*. Team Keccak, 2020. Dostupné tiež z: `https://keccak.team/farfalle.html`. [Online; Citované:6.2.2022].

61. DAEMEN, Joan; HOFFERT, Seth; MELLA, Silvia; PEETERS, Michaël; VAN ASSCHE, Gilles; VAN KEER, Ronny. *The sponge and duplex constructions*. Team Keccak, 2020. Dostupné tiež z: `https://keccak.team/sponge_duplex.html`. [Online; Citované:6.2.2022].

62. JOAN DAEMEN Seth Hoffert, Gilles Van Assche; KEER, Ronny Van. The design of Xoodoo and Xoofff. 2018, roč. 2018. Dostupné z DOI: `10.13154/tosc.v2018.i4.1-38`. [Online; Citované:6.2.2022].

63. DAEMEN, Joan; HOFFERT, Seth; MELLA, Silvia; PEETERS, Michaël; VAN ASSCHE, Gilles; VAN KEER, Ronny. *Xoodoo cookbook*. Team Keccak, 2020. Dostupné tiež z: `https://eprint.iacr.org/2018/767.pdf`. [Online; Citované:6.2.2022].

64. DAEMEN, Joan; HOFFERT, Seth; PEETERS, Michaël; ASSCHE, Gilles Van; KEER, Ronny Van. *Xoodoo cookbook (2.revision)* [Cryptology ePrint Archive, Report 2018/767]. 2019. Dostupné tiež z: `https://eprint.iacr.org/2018/767.pdf`. [Online; Citované:6.2.2022].

65. DAEMEN, Joan; HOFFERT, Seth; MELLA, Silvia; PEETERS, Michaël; VAN ASSCHE, Gilles; VAN KEER, Ronny. *Xoodyak, an update*. Publication to NIST Lightweight Cryptography Standardization Process (round …, 2020. Dostupné tiež z: `https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/round-2/status-update-sep2020/Xoodyak-update.pdf`. [Online; Citované:6.2.2022].

66. DAEMEN, Joan; HOFFERT, Seth; MELLA, Silvia; PEETERS, Michaël; VAN ASSCHE, Gilles; VAN KEER, Ronny. *Xoodyak*. Publication to NIST Lightweight Cryptography Standardization Process, 2020. Dostupné tiež z: `https://csrc.nist.gov/CSRC/media/Projects/Lightweight-Cryptography/documents/round-1/spec-doc/Xoodyak-spec.pdf`. [Online; Citované:6.2.2022].

67. DAEMEN, Joan; HOFFERT, Seth; MELLA, Silvia; PEETERS, Michaël; VAN ASSCHE, Gilles; VAN KEER, Ronny. *The Keyak authenticated encryption scheme*. Team Keccak, 2020. Dostupné tiež z: `https://keccak.team/keyak.html`. [Online; Citované:6.2.2022].

68. What is a ratchet? [B.r.]. Dostupné tiež z: `https://crypto.stackexchange.com/questions/39762/what-is-a-ratchet`. [Online; Citované: 1.1.2023].

69. Forward secrecy. [B.r.]. Dostupné tiež z: `https://en.wikipedia.org/wiki/Forward_secrecy`. [Online; Citované: 1.1.2023].

70. DAEMEN, Joan; HOFFERT, Seth; MELLA, Silvia; PEETERS, Michaël; VAN ASSCHE, Gilles; VAN KEER, Ronny. *Xoodyak, a lightweight cryptographic scheme*. Publication to NIST Lightweight Cryptography Standardization Process, 2022. Dostupné tiež z: `https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-round/updated-spec-doc/xoodyak-spec-final.pdf`. [Online; Citované:19.3.2023].

71. Security level. [B.r.]. Dostupné tiež z: `https://en.wikipedia.org/wiki/Security_level`. [Online; Citované: 1.1.2023].

72. Collision resistance. [B.r.]. Dostupné tiež z: `https://en.wikipedia.org/wiki/Collision_resistance`. [Online; Citované: 20.3.2023].

73. Preimage attack. [B.r.]. Dostupné tiež z: `https://en.wikipedia.org/wiki/Preimage_attack`. [Online; Citované: 20.3.2023].

74. What is a multi-target attack? [B.r.]. Dostupné tiež z: `https://crypto.stackexchange.com/questions/75880/what-is-a-multi-target-attack`. [Online; Citované: 20.3.2023].

75. Power analysis. [B.r.]. Dostupné tiež z: `https://en.wikipedia.org/wiki/Power_analysis`. [Online; Citované: 20.3.2023].

76. ROHAČ MAREK, Drutarovsky Miloš. *Generovanie náhodných čísel na platforme Windows*. TUKE, 2021. Dostupné tiež z: `https://git.kpi.fei.tuke.sk/marek.rohac/bc/-/blob/master/thesis.pdf`. [Online; Citované: 31.3.2023].

77. Oracle VM Virtual Box. [B.r.]. Dostupné tiež z: `https://en.wikipedia.org/wiki/VirtualBox`. [Online; Citované: 20.1.2022].

78. MIKETHETECH. Installing Windows 10 on Virtualbox 6.1.12 – FULL PROCESS, 2020 – video tutorial. 2020. Dostupné tiež z: `https://www.youtube.com/watch?v=gKQvaPejxpc&ab_channel=MikeTheTech`. [Online; Citované: 17.5.2021].

79. ORACLE. 6.2. Introduction to Networking Modes. [B.r.]. Dostupné tiež z: https://www.virtualbox.org/manual/ch06.html. [Online; Citované:26.1.2022].

80. BOSE, Michael. VirtualBox Network Settings: Complete Guide. 2019. Dostupné tiež z: https://www.nakivo.com/blog/virtualbox-network-setting-guide/. [Online; Citované:26.1.2022].

81. NIST LIGHTWEIGHT CRYPTOGRAPHY STANDARDIZATION PROCESS, Publication to. Gimli 2019-03-29. [B.r.]. Dostupné tiež z: https://csrc.nist.gov/CSRC/media/Projects/Lightweight-Cryptography/documents/round-1/spec-doc/gimli-spec.pdf. [Online; Citované:6.2.2022].

82. SHAY GUERON, Nicky Mouha. Simpira v2: A Family of Efficient Permutations Using the AES Round Function. [B.r.]. Dostupné tiež z: https://hal.inria.fr/hal-01403414/document. [Online; Citované:6.2.2022].

83. ORG., GNU. GNU Make Manual. [B.r.]. Dostupné tiež z: https://www.gnu.org/software/make/manual/html_node/index.html. [Online; Citované:20.2.2022].

84. WIKIPEDIA. Endianita. [B.r.]. Dostupné tiež z: https://sk.wikipedia.org/wiki/Endianita. [Online; Citované:20.2.2022].

85. FREECODECAMP.ORG. Ternary Operator in C Explained. [B.r.]. Dostupné tiež z: https://www.freecodecamp.org/news/c-ternary-operator/. [Online; Citované:20.2.2022].

86. MAN7ORG. Poll. [B.r.]. Dostupné tiež z: https://man7.org/linux/man-pages/man2/poll.2.html. [Online; Citované:20.2.2022].

87. Bufferbloat. [B.r.]. Dostupné tiež z: https://en.wikipedia.org/wiki/Bufferbloat. [Online; Citované: 1.3.2021].

88. FATOUROU, Panagiota. *Introduction to Sockets Programming in C using TCP/IP*. Eleftherios Kosmas, 2012. Dostupné tiež z: https://www.csd.uoc.gr/~hy556/material/tutorials/cs556-3rd-tutorial.pdf. [Online; Citované: 13.4.2023].

89. ROHAČ MAREK, Drutarovsky Miloš. *KryptograĄcký balík Xoodyak a jeho použitie v jednoduchej VPN sieti*. TUKE FEI, 2022. Dostupné tiež z: http://eei.fei.tuke.sk/data/EEI-13.pdf. [Online; Citované: 13.4.2023].

Zoznam príloh

Príloha A CD médium – viď. obsah CD média

A Obsah CD Média

Obsah tohto média je dostupný na gite:

- `https://github.com/mr171hg/DiplomaProject`

Pouzite obrazy, zdrojove kody...