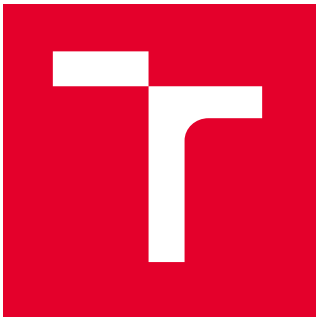


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

ÚSTAV TELEKOMUNIKACÍ

DEPARTMENT OF TELECOMMUNICATIONS

LINUXOVÝ ŠIFRÁTOR SÍŤOVÉHO PROVOZU

LINUX ENCRYPTOR OF NETWORK TRAFFIC

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. Petr Tůma**
AUTHOR

VEDOUCÍ PRÁCE — **doc. Ing. Jan Hajný, Ph.D.**
SUPERVISOR

BRNO 2023

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Diplomová práce

magisterský navazující studijní program **Informační bezpečnost**

Ústav telekomunikací

Student: Bc. Petr Tůma ***ID:*** 211816

Ročník: 2 ***Akademický rok:*** 2022/23

NÁZEV TÉMATU:

Linuxový šifrátor síťového provozu

POKYNY PRO VYPRACOVÁNÍ:

Téma je zaměřeno na problematiku šifrování síťového provozu na platformě Linux. Cílem práce je vytvořit virtuální stroj založený na OS Linux, který dokáže zašifrovat a dešifrovat příchozí síťový provoz protokolu IPv4 a odeslat jej odlišným síťovým rozhraním zpět do sítě. Student může během zpracování využít libovolné existující knihovny a výsledky předchozích bakalářských a diplomových prací, řešení by však mělo pracovat s manipulací payloadu packetu dle vlastního návrhu. Výstupem diplomové práce je funkční šifrátor a dešifrátor a jeho implementace v laboratorním prostředí včetně měření výkonnosti a stručného návodu na instalaci a zprovoznění. Systém bude napojen na generování klíčů pomocí QKD a PQC a bude umožňovat šifrování provozu přicházejícího přes virtuální rozhraní VPN..

DOPORUČENÁ LITERATURA:

[1] Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC[online]. 2007 [cit. 2022-09-08]. Dostupné z: https://csrc.nist.gov/publications/detail/sp/800-38d/final

[2] Debian Documentation [online]. [cit. 2022-09-08]. Dostupné z: https://www.debian.org/doc/

Termín zadání: 6.2.2023 ***Termín odevzdání:*** 19.5.2023

Vedoucí práce: doc. Ing. Jan Hajný, Ph.D.

doc. Ing. Jan Hajný, Ph.D.
předseda rady studijního programu

ABSTRAKT

Tato diplomová práce je zaměřena na problematiku šifrování síťového provozu na platformě Linux. Cílem práce bylo vytvořit šifrátor síťového provozu protokolu IPv4. Zvoleným způsobem šifrování je šifrování celého paketu na virtuálním rozhraní. Šifrátor využívá metody z klasické, kvantové i postkvantové kryptografie. Síťový provoz je šifrován algoritmem AES-GCM-256 s hybridním klíčem ustanoveným pomocí QKD a CRYSTALS Kyber.

KLÍČOVÁ SLOVA

AES, Šifrování, Linux, Kyber, QKD, Hybridní klíč, C++, VPN

ABSTRACT

This master's thesis is focused on network traffic encryption on Linux platform. Goal of the thesis was to create an IPv4 network traffic encryptor. The chosen encryption method is encryption of the entire packet on the virtual interface. The encryptor uses methods from classical, quantum and post-quantum cryptography. Network traffic is encrypted with the AES-GCM-256 algorithm with a hybrid key established using QKD and CRYSTALS Kyber.

KEYWORDS

AES, Encryption, Linux, Kyber, QKD, Hybrid key, C++, VPN

TŮMA, Petr. *Linuxový šifrátor síťového provozu*. Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav telekomunikací, 2023, 63 s. Diplomová práce. Vedoucí práce: doc. Ing. Jan Hajný, Ph.D.

Prohlášení autora o původnosti díla

Jméno a příjmení autora:	Bc. Petr Tůma
VUT ID autora:	211816
Typ práce:	Diplomová práce
Akademický rok:	2022/23
Téma závěrečné práce:	Linuxový šifrátor síťového provozu

Prohlašuji, že svou závěrečnou práci jsem vypracoval samostatně pod vedením vedoucí/ho závěrečné práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor uvedené závěrečné práce dále prohlašuji, že v souvislosti s vytvořením této závěrečné práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a/nebo majetkových a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů, včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb.

Brno ....

podpis autora*

*Autor podepisuje pouze v tištěné verzi.

PODĚKOVÁNÍ

Rád bych poděkoval vedoucímu diplomové práce panu doc. Ing. Janu Hajnému, Ph.D. za odborné vedení, konzultace, trpělivost a podnětné návrhy k práci.

The work was supported by the European Union's Horizon Europe project #101087529 CHESS.

Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or European Research Executive Agency. Neither the European Union nor the granting authority can be held responsible for them.

Obsah

Seznam obrázků

Seznam tabulek

Seznam výpisů

Úvod

Šifrování síťového provozu se v posledních letech stalo nedílnou součástí internetové komunikace. Šifrování zajišťuje důvěrnost dat, tedy vlastnost, že obsah zprávy může přečíst jen zamýšlený příjemce. Bez této vlastnosti by kdokoliv, kdo je schopen zachytit komunikaci, mohl přečíst obsah zpráv. Další neméně důležitou vlastností je autentičnost dat, tedy vlastnost, že data pocházejí od očekávaného odesílatele a jsou v původní podobě.

Diplomová práce je zaměřena na problematiku šifrování síťového provozu na platformě Linux. Cílem práce bylo vytvořit virtuální stroj založený na OS („Operační systém“) Linux, který dokáže zašifrovat a dešifrovat příchozí síťový provoz protokolu IPv4 a odeslat jej odlišným síťovým rozhraním zpět do sítě.

Výstupem diplomové práce byl šifrátor síťového provozu používající postkvantové mechanismy ustanovení klíče. Šifrátor je napojen na generování klíčů pomocí QKD („Kvantová distribuce klíče“) a PQC („Postkvantová kryptografie“), ze kterých pomocí hašovací funkce SHA2-256 vytvoří hybridní klíč. Šifrátor umožňuje šifrování provozu přicházejícího přes virtuální rozhraní VPN.

Šifrátor je implementován v jazyce C++ a využívá pomocné shell skripty pro získání QKD klíčů. Pakety jsou šifrovány algoritmem AES-256-GCM [1], čímž je zajištěna důvěrnost i autentičnost dat. PQC klíč je ustanoven pomocí mechanismu zapouzdření klíče CRYSTALS Kyber-512 [2]. Zvoleným způsobem šifrování bylo šifrování celého paketu.

Struktura diplomové práce je sestavena následovně. V první kapitole, *Kryptografie*, je stručný úvod do problematiky šifrování a popis algoritmů, které jsou použity v diplomové práci. Následně jsou v druhé kapitole, *Současná řešení šifrování provozu*, představena moderní řešení šifrování síťového provozu, z kterých je většina využívána ve VPN („Virtuální privátní síť“) řešeních. Ve třetí kapitole, *Základní implementace šifrátoru*, jsou popsány vyzkoušené způsoby šifrování v jazyce Python a jejich porovnání. Čtvrtá kapitola, *Optimalizace šifrátoru*, a pátá kapitola, *Rozšíření funkcionality*, popisují kroky ke zlepšení výkonu a přidané funkce vybraného šifrovacího řešení. Poslední kapitola, *Zhodnocení výsledné implementace*, popisuje návod na instalaci, obsluhu šifrátoru a jsou zde zahrnuty měření výsledné implementace.

1 Kryptografie

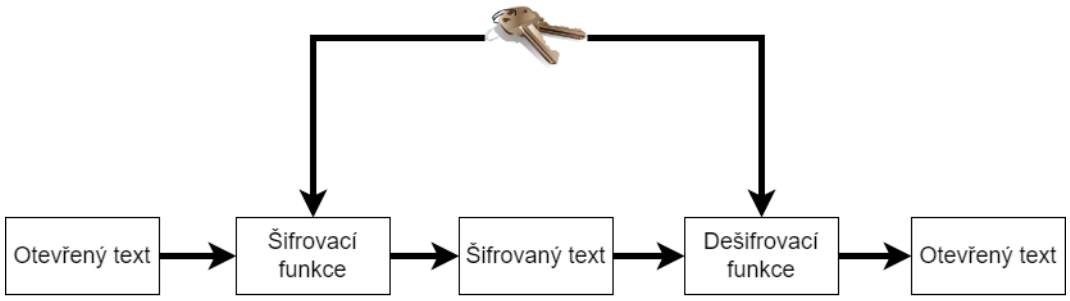
Kryptografie je obor informatiky a matematiky, který se zaměřuje na techniky pro zabezpečení komunikace mezi dvěma stranami, zatímco třetí strana je přítomna na komunikačním médiu [4]. Pomocí kryptografických primitiv lze zajistit zejména následující vlastnosti:

- **Důvěrnost** - Zajišťuje, že zprávu může přečíst pouze zamýšlený příjemce.
- **Integrita** - Ujišťuje příjemce, že se přijatá zpráva nijak neliší od originálu.
- **Autentizace** - Zajišťuje ověření identity osob, zařízení či zdroje dat.
- **Autentičnost** - Ověření, že data pocházejí od očekávaného odesílatele a jsou v původní podobě.
- **Nepopiratelnost** - Strana nemůže tvrdit, že není autorem výroku (resp. dat).

Kryptografie se dělí na symetrickou a asymetrickou podle toho, zda pro šifrování a dešifrování používá stejný klíč.

1.1 Symetrická kryptografie

Symetrická kryptografie (neboli kryptografie s tajným klíčem) představuje metody, při kterých účastníci komunikace používají k šifrování i dešifrování stejný klíč. Podstatnou výhodou symetrické kryptografie je její nízká výpočetní náročnost, tudíž je vhodná pro šifrování velkých objemů dat. Nevýhodou je, že obě strany musí spolu sdílet tajný klíč, který si musí předem vyměnit, nebo se na něm dohodnout před začátkem komunikace [3]. Princip symetrické kryptografie je zobrazen na obrázku 1.1.



Obr. 1.1: Princip symetrické kryptografie

Algoritmy symetrické kryptografie se dále dělí na proudové a blokové podle toho, zda se otevřený text šifruje po bitech, nebo po skupinách bitů, zvaných bloky.

1.1.1 AES

AES („Advanced Encryption Standard“) je kryptografický algoritmus, který lze použít k ochraně elektronických dat. Algoritmus AES je symetrická bloková šifra, která je schopna používat kryptografické klíče o délce 128, 192 a 256 bitů. AES šifruje a dešifruje data v blocích o velikosti 128 bitů [5].

Kvantové počítače představují i pro algoritmus AES hrozbu. Groverův algoritmus dokáže zkrátit dobu trvání útoku hrubou silou na druhou odmocninu původního času. Při použití Groverova algoritmu je tedy efektivní velikost klíče zkrácena na polovinu. Pro zajištění kvantové odolnosti algoritmu AES je tedy potřeba používat klíč o velikosti alespoň 256 bitů [6].

V současné době NIST („National Institute of Standards and Technology“) schválil čtrnáct režimů blokových šifer. Z toho osm režimů důvěrnosti (ECB, CBC, OFB, CFB, CTR, XTS-AES, FF1 a FF3), jeden režim autentičnost (CMAC) a pět kombinovaných režimů důvěrnosti a autentičnosti (CCM, GCM, KW, KWP a TKW) [7].

1.1.2 Módy blokové šifry - GCM

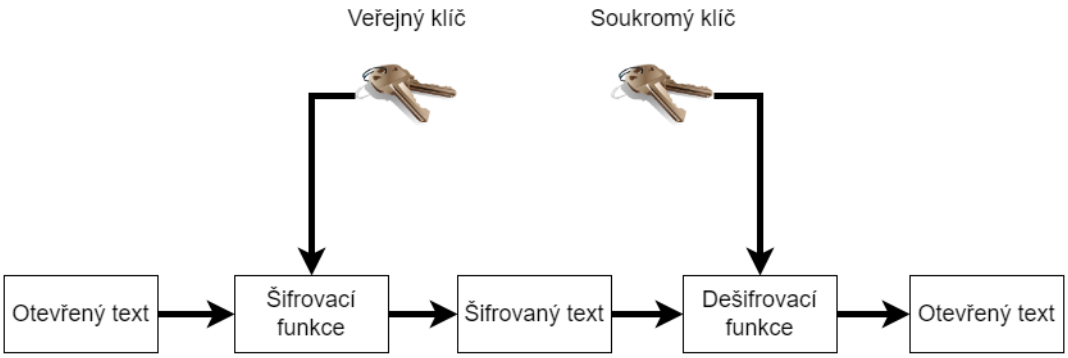
Režim GCM („Galois Counter Mode“) poskytuje vysokorychlostní autentizované šifrování. GCM používá dvě funkce - GCTR a GHASH. GCTR slouží k zajištění důvěrnosti dat a lze si ji představit jako režim CTR. Funkce GHASH slouží k zajištění autentičnosti dat pomocí univerzální hašovací funkce. GHASH používá podklíč H, který vznikne aplikací blokové šifry na blok tvořený samými nulami [1].

Režim GCM umožňuje k šifrovanému textu přidat další data, u kterých bude zajištěna autentičnost, ale nebudou šifrována. Tato přidaná data mohou představovat např. obsah IP („Internet Protocol“) hlavičky, jejíž změnou by se mohl stát paket nedoručitelným.

1.2 Asymetrická kryptografie

Asymetrická kryptografie, známá také jako kryptografie s veřejným klíčem, označuje kryptografické algoritmy, které vyžadují dva samostatné klíče, z nichž je jeden označován jako soukromý a druhý jako veřejný. Veřejný klíč se používá k šifrování zprávy a soukromý k dešifrování zprávy. Tyto klíče jsou spolu matematicky provázány a je velice výpočetně náročné odvodit jeden klíč od druhého.

Hlavní výhoda asymetrické kryptografie spočívá v tom, že odesílatel a příjemce nepotřebují sdílet tajný klíč přes nezabezpečený kanál. Celá komunikace vyžaduje jen přítomnost veřejného klíče, který je možno přenést i přes nezabezpečený kanál.



Obr. 1.2: Princip asymetrické kryptografie

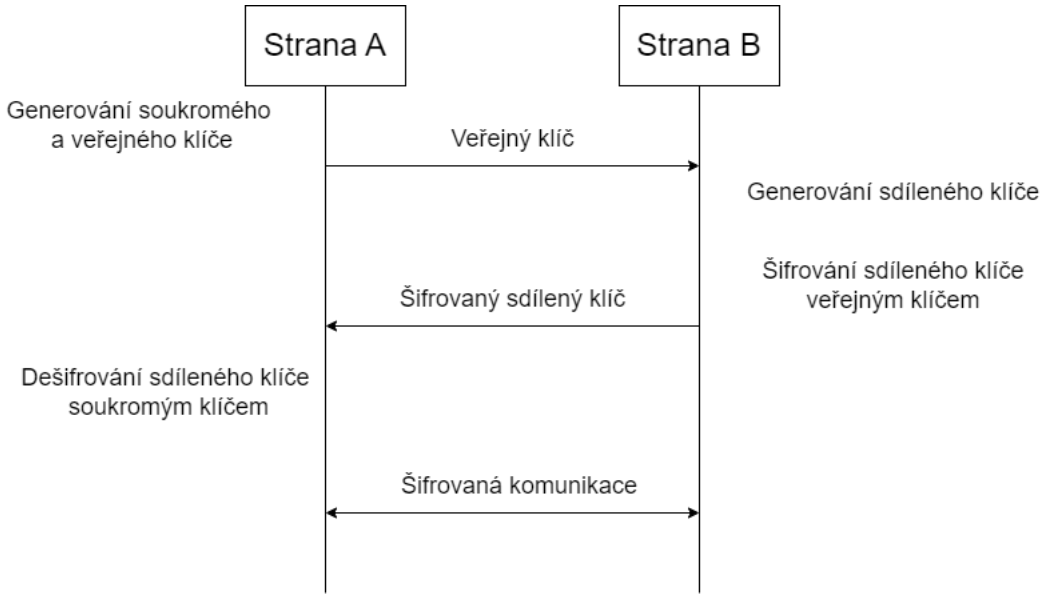
Příjemce musí mít pouze potvrzení autentičnosti veřejného klíče. Nevýhoda asymetrické kryptografie je pomalá rychlost šifrování, především kvůli délce klíče a složitosti použitých algoritmů [8].

Moderní kryptosystémy používají jak asymetrickou, tak symetrickou kryptografii. Algoritmy asymetrické kryptografie řeší problém distribuce klíčů symetrické kryptografie a symetrická kryptografie je díky svojí rychlosti používána k šifrování dat. Princip asymetrické kryptografie je zobrazen na obrázku 1.2.

1.2.1 CRYSTALS - Kyber

Kyber je KEM („mechanisnus zapouzdření klíče“), jehož bezpečnost je založena na problému LWE („learning with errors“) na mřížkách. KEM se využívá k ustanovení klíče mezi komunikujícími stranami. Kyber se doporučuje použít v hybridním režimu - tzn. v kombinaci se zavedenou klasickou kryptografií [2]. Schéma ustanovení sdíleného klíče je zobrazena na obrázku 1.3. Kyber existuje ve třech variantách:

- **Kyber-512** - Poskytuje zabezpečení podobné úrovni AES-128.
- **Kyber-768** - Poskytuje zabezpečení podobné úrovni AES-192.
- **Kyber-1024** - Poskytuje zabezpečení podobné úrovni AES-256.



Obr. 1.3: Princip zapouzdření klíče

Kyber je jedním z vybraných algoritmů pro šifrování s veřejným klíčem a ustanovení klíče v post-quantum cryptography project uskutečněným NIST[1].

[1]Seznam vybraných algoritmů je dostupný z: `https://csrc.nist.gov/projects/post-quantum-cryptography/selected-algorithms-2022`

2 Současná řešení šifrování provozu

Tato kapitola popisuje současné přístupy k šifrování síťového provozu. Přístupy jsou běžně využívány k vytváření VPN, což je způsob, jak bezpečně propojit několik počítačů pomocí nedůvěryhodné sítě.

2.1 PPTP

PPTP („Point-to-Point Tunneling Protocol“) je síťový protokol, který umožnuje zabezpečený přenos dat mezi klientem a serverem, vytvořením VPN. PPTP je rozšířením protokolu point-to-point pro vzdálený přístup, který byl definován v RFC („Request for Comments“) 1171. PPTP zapouzdřuje PPP („Point-to-Point Protocol“) pakety do IP datagramů, které jsou následně přeneseny přes internet nebo jinou síť. PPTP může být také využit pro VPN typu site-to-site [9].

PPTP je dostupný jako standard na téměř každém zařízení s podporou VPN. Nastavení PPTP je snadné, bez nutnosti instalace dalšího softwaru. PPTP používá šifru MPPE („Microsoft Point-to-Point Encryption“) s klíčem o délce 128 bitů a pro komunikaci využívá TCP port 1723 [10].

2.2 L2TP/IPsec

L2TP („Layer Two Tunneling Protocol“) je zabudován do téměř všech moderních operačních systémů a zařízení podporujících VPN. L2TP sám o sobě nešifruje data, tudíž je obvykle implementován s protokolem IPsec („Internet Protocol Security“), který šifrování zajistí. L2TP/IPsec používá buď šifru 3DES („Triple Data Encryption Standard“) nebo AES.

L2TP/IPsec dvakrát zapouzdřuje data - nejprve přidá hlavičku L2TP a pak zapouzdření IPsec. Toto dvojité zapouzdření má za následek zpomalení šifrování dat. L2TP/IPsec umožňuje multi-threading, čímž lze toto zpomalení kompenzovat [10].

2.3 SSTP

SSTP („Secure Socket Tunneling Protocol“) je protokol využívaný ve VPN, který poskytuje mechanismus pro přenos provozu prostřednictvím kanálu SSL/TLS („Secure Sockets Layer“/„Transport Layer Security“). Díky využití SSL/TLS lze posílat data na port 443, což umožňuje procházet prakticky všemi firewally.

SSTP je proprietární standard vlastněný společností Microsoft, tudíž není zdrojový kód veřejně dostupný. Protokol SSTP byl představen v OS Windows Vista service pack 1. Nyní je k dispozici i pro Linux a Mac OS [10].

2.4 OpenVPN

OpenVPN je open-source technologie, která využívá knihovnu OpenSSL a protokol TLS. Hlavní výhodou OpenVPN je její široká možnost konfigurace. OpenVPN není nativně podporována žádnou platformou, ale je k dispozici prostřednictvím softwaru třetích stran. Zdrojový kód pro OpenVPN je vyvíjen projektem OpenVPN.

OpenVPN nejlépe pracuje přes UDP („User Datagram Protocol“), ale není problém nastavit ji i přes kterýkoliv jiný port. Stejně jako SSTP lze OpenVPN nastavit na TCP („Transmission Control Protocol“) port 443 pro obcházení firewallů, což ztěžuje její blokování. Použití knihovny OpenSSL umožňuje využít řadu různých šifer, ale v praxi se běžně využívá pouze AES či Blowfish[10].

2.5 WireGuard

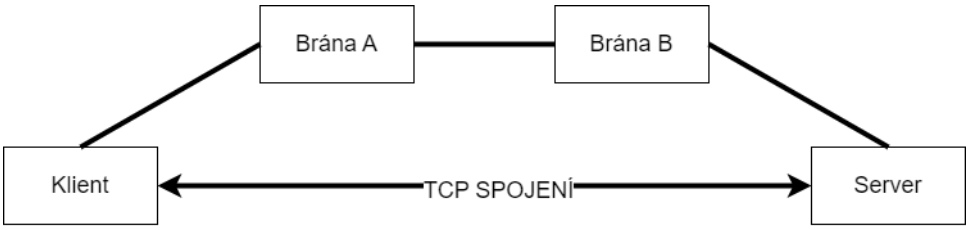
WireGuard je jednoduché, rychlé a moderní VPN řešení, které využívá současné kryptografické trendy. Cílem WireGuard je být rychlejší, jednodušší a užitečnější než IPsec. WireGuard je navržen jako obecné VPN řešení, neměl by tedy být problém jej spustit jak na vestavěných rozhraních, tak na superpočítačích. WireGuard byl původně vydán pro Linux, ale nyní je již multiplatformní (Windows, macOS, BSD, iOS, Android). WireGuard je stále intenzivně vyvíjen, ale již může být považován za nejbezpečnější, nejsnadněji použitelné a nejjednodušší VPN řešení [11].

3 Základní implementace šifrátoru

Tato část je zaměřena na popsání realizace šifrování a zhodnocení šifrátoru. Pro pochopení některých částí následujícího textu je dobré si uvědomit, že pakety jsou také posloupnost bitů.

3.1 Topologie

Prostředí bylo virtualizováno nástrojem VirtualBox. Počítač, na kterém byly stroje virtualizovány, obsahoval 4 jádrový procesor Intel Core i7 1065G7 Ice Lake frekvence 1,3 GHz. Prostředí se skládalo ze 4 počítačů - klient, server a 2 šifrovací brány. Šifrovacím branám byly přiděleny dvě virtuální CPU („Centrální procesorová jednotka“). Klient a server měli přiděleny po jednom CPU. Schéma zapojení je zobrazeno na obrázku 3.1.



Obr. 3.1: Topologie sítě

Počítače byly rozděleny do třech NAT („Network Address Translation“) sítí. Směrování mezi sítěmi bylo nastaveno staticky pomocí příkazu `ip route`. Adresace rozhraní a příslušnost v NAT sítích je shrnuta v tabulce 3.1.

Tab. 3.1: Adresace

Role	Rozhraní	Název NAT sítě	Adresa
`Klient`	`enp0s3`	`Klient`	`10.0.3.10`
`Server`	`enp0s3`	`Server`	`10.0.1.10`
`Brána A`	`enp0s3`	`Klient`	`10.0.3.254`
`Brána A`	`enp0s8`	`Spoj`	`10.0.2.10`
`Brána B`	`enp0s3`	`Server`	`10.0.1.254`
`Brána B`	`enp0s8`	`Spoj`	`10.0.2.20`

3.2 Zachycení paketů

Pro šifrování paketů je paket nejprve nutné zachytit a následně upravit jeho obsah. Vzhledem k tomu, že šifrátor není součástí jádra, je potřeba přenést paket z kernel space do user space, kde se následně provede šifrování. Tento přesun je možné vykonat například na úrovni firewallu s pomocí `iptables` a netfilter fronty (NFQUEUE) nebo pomocí virtuálních rozhraní tun.

3.2.1 NFQUEUE

NFQUEUE lze zvolit jako cíl v `iptables` a `ip6tables`, který deleguje rozhodování o paketech na program v user space. Před rozhodnutím o přijetí/zamítnutí paketu je možné paket upravit [12].

Následujícím pravidlem lze například zachytit všechny pakety, které jsou určeny ke směrování s protokolem 99[1] v IP hlavičce.

```
iptables -I FORWARD -j NFQUEUE --queue-num 1 -p 99
```

3.2.2 Virtuální rozhraní

Virtuální rozhraní zajišťuje příjem a přenos paketů pro programy v user space. Lze na něj pohlížet jako na jednoduché zařízení typu Point-to-Point nebo Ethernet. Virtuální rozhraní umožňují programům v user space vidět a upravovat síťový provoz [13].

Příklad příkazů k vytvoření virtuálního rozhraní tun0, díky kterému budou pakety s cílovou adresou 192.0.2.2 přesměrovány do programu v user space [14]:

```
sudo ip tuntap add name tun0 mode tun user $USER
sudo ip link set tun0 up
sudo ip addr add 192.0.2.1 peer 192.0.2.2 dev tun0
```

3.3 Šifrování paketů

Pro šifrování paketů byl zvolen algoritmus AES v režimu GCM s délkou klíče 256 bitů. Každý paket je šifrován zvlášť, tudíž je s každým paketem odeslána i hodnota nonce a MAC („Message Authentication Code“). Použitá nonce a MAC mají velikost 16 bytů. Díky přidání MAC a nonce se zvětší paket, což může mít za následek zahození paketu z důvodu překročení MTU („Maximum Transmission Unit“), proto je potřeba na koncových stanicích upravit MTU tak, aby po zašifrování paketů nedocházelo k zahazování paketů.

[1] Číslo protokolu 99 je vyhrazeno pro jakékoli soukromé šifrovací schéma

3.3.1 Šifrování dat s využitím NFQUEUE

Při zachycení paketů pomocí netfilter fronty se šifrují pouze data v paketu. Při modifikaci dat je potřeba změnit hodnotu protokolu v IP hlavičce, aby nedošlo k zahazování paketů na mezilehlých uzlech z důvodu chyby v kontrolním součtu. Vzhledem ke změně protokolu v IP hlavičce je potřeba k datům před šifrováním přidat 1 byte s hodnotou původního protokolu. Díky AES v režimu GCM je zajištěna autentičnost IP hlavičky bez jejího zašifrování. Pole, u kterých není zajištěna autentičnost, protože se jejich hodnota v průběhu přenosu mění, jsou TTL a checksum.

3.3.2 Šifrování dat s využitím virtuálního rozhraní

Při použití virtuálních rozhraní je potřeba zaznamenat IP adresu původního příjemce zprávy a také změnit protokol v IP hlavičce. Celkově se tedy velikost paketu zvětší o 37 bytů, z čehož jsou 4 byte původní adresa příjemce, 1 byte původní protokol, 16 byte hodnota nonce a 16 byte MAC. Nevýhodou tohoto přístupu je složitost směrování - každý uzel na trase musí znát IP adresu cílového virtuálního rozhraní.

3.3.3 Šifrování paketu s využitím virtuálního rozhraní

Poslední vyzkoušenou možností bylo zašifrovat celý paket a uložit jej jako payload do nového paketu. Původní paket se tedy zvětšil o MAC, nonce, novou IP hlavičku a UDP hlavičku. Protokol UDP je preferován oproti TCP z důvodu menší velikosti hlavičky a také proto, že není stream-oriented[2]. Vzhledem k existenci TCP spojení mezi klientem a serverem je zajištěna spolehlivost spojení i při použití UDP protokolu.

Tato možnost šifrování eliminuje potřebu složitých požadavků na směrování z předchozího přístupu.

3.4 Dešifrování paketů

Při dešifrování jsou z dat v paketu vyčteny hodnoty nonce a MAC, které slouží k dešifrování a ověření integrity. Pokud selže kontrola integrity, je paket zahozen. Za předpokladu předchozího navázání spojení pomocí protokolu TCP klient požádá o nové zaslání paketu.

[2]Při práci s TCP by bylo potřeba označit začátek a konec paketu pro správnou kontrolu autentičnosti

3.4.1 Dešifrování dat s využitím NFQUEUE

Z hlavičky se nejprve paketu vyčte číslo protokolu. Pokud číslo odpovídá šifrovanému paketu, jsou z payloadu paketu vyčteny hodnoty nonce a MAC. Paket je následně dešifrován a číslo protokolu v IP hlavičce nahrazeno původním.

3.4.2 Dešifrování dat s využitím virtuálního rozhraní

Dešifrování probíhá podobně jako v předchozím případě s tím rozdílem, že je po kontrole integrity nahrazena i cílová adresa původní hodnotou.

3.4.3 Dešifrování paketu s využitím virtuálního rozhraní

Pokud byl šifrovaný celý paket na virtuálním rozhraní, tak byl odeslán na předem stanovený UDP port. Lze tedy předpokládat, že všechna příchozí data na konkrétním UDP portu bylo potřeba dešifrovat. Příchozí data jsou vlastně šifrované pakety, takže ihned po dešifrování dat je získána původní podoba paketu[3].

3.5 PoC implementace v jazyce Python

Zkušební implementace šifrování byly naprogramovány v jazyce Python. Pro šifrování byla využita knihovna `PyCryptodomex`. Pro úpravu IP hlavičky a generování kontrolního součtu byla použita knihovna `Scapy`. Ve výpisu 3.1 jsou uvedeny funkce pro šifrování a dešifrování dat.

Výpis 3.1: Šifrovací a dešifrovací funkce v Pythonu

```
1   from Cryptodome.Cipher import AES
2
3   def encrypt(plaintext, key, mode):
4       encobj = AES.new(key, AES.MODE_GCM)
5       ciphertext,authTag=encobj.encrypt_and_digest(plaintext)
6       return(ciphertext,authTag, encobj.nonce)
7
8   def decrypt(ciphertext, key, mode):
9       (ciphertext, authTag, nonce) = ciphertext
10      encobj = AES.new(key, AES.MODE_GCM, nonce)
11      return(encobj.decrypt_and_verify(ciphertext, authTag))
```

3.5.1 Netfilter NFQUEUE

První z vyzkoušených implementací zachytávala pakety pomocí NFQUEUE. Největší výhodou tohoto přístupu je snadná implementace.

[3]Včetně TTL, které se přenosem mezi branami nesnižovalo

Výpis 3.2: Šifrování dat s využitím NFQUEUE

```
1  from netfilterqueue import NetfilterQueue as nfq
2
3  def sifruj(packet):
4    ...
5  queue = nfq()
6  queue.bind(1, sifruj)
7  queue.run()
```

Kód uvedený ve výpisu 3.2 pro každý paket, z fronty s číslem 1, vykoná funkci `sifruj()`. Pro šifrátor byly vytvořeny dvě fronty - jedna pro pakety s protokolem 99 určené pro dešifrování a druhá pro pakety, které ještě nebyly šifrovány.

Pro potřeby přenosu po síti je po šifrování upraven protokol v IP hlavičce, upraven záznam o délce paketu a přepočítán kontrolní součet celého paketu. Tyto úkoly byly vykonány pomocí nástroje `Scapy`[4]. Změna protokolu v IP hlavičce také slouží k rozpoznání šifrovaného provozu. Pro zajištění autentičnosti IP hlavičky bylo potřeba doplnit šifrovací a dešifrovací funkci o data IP hlavičky. Ve výpisu 3.3 je vidět upravená šifrovací funkce.

Výpis 3.3: Šifrování s přidanými daty

```
1  def encrypt(plaintext, key, mode, asociated_data):
2      encobj = AES.new(key, AES.MODE_GCM)
3      encobj.update(asociated_data)
4      ciphertext,authTag=encobj.encrypt_and_digest(plaintext)
5      return(ciphertext,authTag, encobj.nonce)
```

Nevýhodou tohoto přístupu je, že k frontě může zároveň přistoupit pouze jedno jádro procesoru. Tento přístup je tedy značně omezen možnostmi škálování. Do jisté míry lze tento problém řešit rozdělením příchozího provozu do více front a každou zpracovávat zvlášť. Ovšem i tento přístup má svá úskalí, a to zejména změnu pořadí paketů na straně příjemce.

3.5.2 Virtuální rozhraní - šifrování dat

Druhá z vyzkoušených implementací využívá k zachycení virtuální rozhraní. Výhodou tohoto přístupu je nekomplikovaná možnost škálování. Vytvoření přístupového bodu do virtuálního rozhraní lze vidět ve výpisu 3.4.

Výpis 3.4: Přístup do virtuálního rozhraní v Pythonu

```
1  import fcntl
2
3  def openTun(tunName):
4      TUNSETIFF = 0x400454ca
5      IFF_TUN = 0x0001
6      IFF_NO_PI = 0x1000
7
```

[4]Dostupný z https://scapy.net/

```
8       tun = open('/dev/net/tun', 'r+b', buffering=0)
9       ifs = struct.pack('16sH22s', tunName, IFF_TUN | IFF_NO_PI, b'')
10      fcntl.ioctl(tun, TUNSETIFF, ifs)
11      return tun
```

Pomocí funkcí `read()` a `write()` jsou pakety čteny/zapisovány do virtuálního rozhraní. Využití více jader procesoru lze zajistit pomocí knihovny `Multiprocessing`[5]. Ve výpisu 3.5 lze vidět použití knihovny `Multiprocessing`. Zobrazený kód vytvoří nový proces, ve kterém spustí funkci `sifruj()`, která agreguje funkce přečtení dat z virtuálního rozhraní, šifrování a přenos šifrovaných dat.

Výpis 3.5: Vytvoření nového procesu

```
1   from multiprocessing import Process
2
3   p = Process(target=sifruj,args=())
4   p.start()
```

V tomto přístupu nastává problém adresování na virtuálních rozhraní, kdy je potřeba nahrazovat cílovou adresu adresou cílového virtuálního rozhraní. Proto je potřeba, aby měl každý uzel na trase záznam o virtuálním rozhraní a přidat zašifrovanou originální cílovou adresu do obsahu paketu. Z tohoto důvodu je tento přístup v praxi téměř nepoužitelný - virtuální rozhraní by musela mít veřejnou IP adresu.

3.5.3 Virtuální rozhraní - šifrování paketu

Poslední přístup také využívá virtuální rozhraní a řeší problém se směrováním za použití UDP socketu. Tento přístup zašifruje celý paket a zabalí jej do UDP datagramu. Nevýhodou přístupu je nutnost zabalení šifrovaného paketu do nových hlaviček a tedy další snížení MTU na koncových bodech. Ve výpisu 3.6 je zobrazeno vytvoření UDP socketu, poslání proměnné `enc` a příjem dat o maximální velikosti 1500 bytů.

Výpis 3.6: Python - šifrování paketu

```
1   import socket
2
3   s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
4   s.bind((HOST, PORT))
5   s.sendto(enc, (HOST2, PORT))
6   s.recvfrom(1500)
```

Šifrátor tedy šifruje pakety, které jsou přijaty na virtuálním rozhraní, a odesílá je pomocí UDP socketu na předem stanovený port. Zároveň také naslouchá na stanoveném portu, dešifruje příchozí datagramy a zapíše je do virtuálního rozhraní[6].

[5] Dostupné z `https://docs.python.org/3/library/multiprocessing.html`

[6] Data zapsaná do virtuálního rozhraní se jeví jako příchozí provoz, který může být dále směrován

3.5.4 Šifrátor jako systémová služba

Vzhledem k serverové instalaci operačního systému Linux bylo vhodné nastavit šifrátor jako systémovou službu. Pro vytvoření služby je potřeba vytvořit soubor sifrator.service a uložit jej do `/etc/systemd/system/`. Ve výpisu 3.7 je zobrazen obsah souboru `sifrator.service`, ve kterém jsou uvedeny parametry služby. Při startu služby se vykoná příkaz `/usr/bin/python3 /home/xtumap02/sifrator.py`, čímž se spustí šifrátor.

Výpis 3.7: Ukázka vytvoření systémové služby

```
[UNIT]
Description=sifrator
After=multi-user.target

[Service]
type=simple
Restart=always
ExecStart=/usr/bin/python3 /home/xtumap02/sifrator.py

[Install]
WantedBy=multi-user.target
```

Následujícími příkazy je povolena a spuštěna systémová služba:

```
sudo systemctl daemon-reload
sudo systemctl enable sifrator.service
sudo systemctl start sifrator.service
```

3.6 Porovnání přístupů šifrování

V této části jsou zhodnoceny přístupy k šifrování paketů. Zkušební implementace šifrování se hodnotily podle třech kritérií - RTT („Round-trip Time“), propustnost a data přenesená navíc.

3.6.1 RTT

Prvním kritériem zhodnocení způsobů šifrování bylo změřit RTT. Měření probíhalo pomocí příkazu ping, přičemž se zaznamenal průměrný čas odezvy po 15 žádostech. V tabulce 3.2 jsou shrnuty výsledky. Z výsledků vyplývá, že přístup využívající zabalení celého paketu měl nejnižší RTT.

3.6.2 Propustnost

Další měření probíhalo pomocí nástroje `iperf`, který slouží ke generování síťového provozu. Cílem měření bylo zjistit propustnost šifrovaného kanálu pro protokol TCP.

Tab. 3.2: Doba RTT

Způsob šifrování	RTT [ms]
`Bez`	2.283
`NFQUEUE`	6.210
`Tun rozhraní`	5.979
`Tun + UDP socket`	3.574

Měření probíhalo po dobu 60 sekund s maximální rychlostí dat 100 Mbps. V tabulce 3.3 jsou shrnuty výsledky. Z výsledků vyplývá, že přístup zabalení celého paketu dokázal data šifrovat nejrychleji. Všechny přístupy maximálně vytížily CPU, ale šifrování celého paketu mělo nejméně operací s byty v paketech, proto tento přístup šifroval nejrychleji.

Tab. 3.3: Propustnost

Způsob šifrování	Rychlost [Mbps]	MTU
`Bez`	100	1500
`NFQUEUE`	6.99	1467
`Tun rozhraní`	9.93	1463
`Tun + UDP socket`	36.0	1440

3.6.3 Režijní náklady

Poslední měření mělo za úkol zjistit datové náklady spojené se šifrováním. Během měření se přenášel soubor o velikosti 100 MB a zjistilo se, kolik dat bylo potřeba přenést více pro každý způsob šifrování. Soubor byl vytvořen pomocí nástroje `fallocate` a umístěn na HTTP („Hypertext Transfer Protocol“) server. HTTP server byl vytvořen pomocí příkazu `python3 -m http.server`. Odchyt komunikace byl vykonán pomocí nástroje `tcpdump`. Odchycená data byla následně otevřena v programu Wireshark a z položky `Statistics -> Endpoints -> IPv4 -> Bytes` zjištěn celkový objem přenesených dat. V tabulce 3.4 jsou shrnuty výsledky.

Z výsledků vyplývá, že přístup šifrování celého paketu potřeboval k přenosu nejméně dat. Tento výsledek byl překvapivý, protože ke každému paketu přidával nejvíce dat. Po zvážení výsledků se dá předpokládat, že u ostatních přístupů docházelo k opakovaným přenosům. Opakované přenosy mohou být spojeny s vytížením CPU, kdy se do vyrovnávací paměti dostává více paketů, než je systém schopný

Tab. 3.4: Přenesená data

Způsob šifrování	Přenesená data [MB]	Rozdíl [%]
Bez	104	0
NFQUEUE	117	12.5
Tun rozhraní	110	5.77
Tun + UDP socket	109	4.81

zpracovat. Po naplnění vyrovnávací paměti dojde k zahazování paketů a také ke změně jejich pořadí.

3.7 Zhodnocení a výběr vítěze

Ze všech testů vyšel nejlépe způsob šifrování celého paketu na virtuálním rozhraní. Tento způsob dokáže efektivně řešit problém směrování z virtuálního rozhraní a může využít více jader procesoru.

Další způsob - šifrování dat na virtuálním rozhraní, měl oproti šifrování celého paketu mnohem menší rychlost šifrování. Tento způsob měl zásadní problém se směrováním na virtuální rozhraní, proto je v praxi téměř nepoužitelný.

Poslední způsob - šifrování dat pomocí NFQUEUE, byl ve všech testech nejhorší. Způsob sice neměl problémy se směrováním, ale wrapper pro správu front v jazyce Python umožňuje využít pouze jedno jádro procesoru na frontu. Z tohoto důvodu je škálování poměrně složitá záležitost.

Na základě výsledků testování byl pro další použití vybrán způsob šifrování celého paketu na virtuálním rozhraní.

4 Optimalizace šifrátoru - implementace v C++

Tato kapitola popisuje optimalizaci šifrátoru - především jeho implementaci v jazyce C++. Vzhledem k interpretované povaze programovacího jazyka Python nedosahoval šifrátor potřebné rychlosti k praktickému využití. Zvolenou optimalizací bylo přepsat šifrátor do jazyka C++, který je kompilovaný a je tedy zbaven mnoha výpočetních nákladů při běhu programu.

4.1 Převedení do C++

Při převedení šifrátoru do C++ se změnily všechny hlavní funkce v šifrátoru i použité knihovny. Knihovnou použitou pro šifrování byla knihovna `Crypto++`[1]. Příkazy k instalaci knihovny jsou zobrazeny ve výpisu 4.1.

Výpis 4.1: Instalace knihovny Crypto++

```
wget https://www.cryptopp.com/cryptopp870.zip
unzip -aoq cryptopp870.zip -d cryptopp
cd cryptopp
sudo make
sudo make install
```

Ve výpisu 4.2 je zobrazeno použití knihovny `Crypto++` pro šifrování dat pomocí AES-GCM. Zobrazená funkce zašifruje data obsažená ve druhém argumentu a na jejich začátek přidá hodnotu nonce. Vzhledem ke změně knihovny je MAC dosazen až za šifrovaná data.

Výpis 4.2: Šifrování v C++

```
1   #include <string>
2   using std::string;
3
4   #include "cryptopp/hex.h"
5   using CryptoPP::HexEncoder;
6   using CryptoPP::HexDecoder;
7
8   #include "cryptopp/osrng.h"
9   using CryptoPP::AutoSeededRandomPool;
10
11  #include "cryptopp/cryptlib.h"
12  using CryptoPP::BufferedTransformation;
13  using CryptoPP::AuthenticatedSymmetricCipher;
14  using CryptoPP::byte;
15
16  #include "cryptopp/filters.h"
17  using CryptoPP::Redirector;
18  using CryptoPP::StringSink;
```

[1] Dostupná z: https://cryptopp.com/

```
19  using CryptoPP::StringSource;
20  using CryptoPP::AuthenticatedEncryptionFilter;
21  using CryptoPP::AuthenticatedDecryptionFilter;
22  using CryptoPP::SecByteBlock;
23
24  #include "cryptopp/aes.h"
25  using CryptoPP::AES;
26
27  #include "cryptopp/gcm.h"
28  using CryptoPP::GCM;
29
30  string Sifrovani(SecByteBlock key, string zprava){
31  string cipher;
32  AutoSeededRandomPool prng;
33  byte iv[ AES::BLOCKSIZE ];
34  prng.GenerateBlock( iv, sizeof(iv));
35
36  GCM< AES, CryptoPP::GCM_64K_Tables >::Encryption e;
37  e.SetKeyWithIV( key, key.size(), iv, sizeof(iv) );
38
39  try
40  {
41  StringSource ss1( zprava, true,
42  new AuthenticatedEncryptionFilter( e,
43          new StringSink( cipher ), false, TAG_SIZE
44        )
45    );
46  }
47
48  catch( CryptoPP::Exception& e)
49  {
50  cerr << e.what() << endl;
51  exit(1);
52  }
53
54  string poslani((char *)iv, sizeof(iv));
55  poslani += cipher;
56
57  return poslani;
58  }
```

Funkce pro získání deskriptoru ke čtení a zápisu dat z virtuálního rozhraní je zobrazena ve výpisu 4.3. Virtuální rozhraní je nastaveno do neblokujícího režimu kvůli možnosti práce s více procesy.

Výpis 4.3: Získání deskriptoru virtuálního rozhraní

```
1   #include <linux/if.h>
2   #include <linux/if_tun.h>
3   #include <sys/ioctl.h>
4
5   int tun_open(){
6   struct ifreq ifr;
7   int fd, err;
8
9   if ( (fd = open("/dev/net/tun", O_RDWR | O_NONBLOCK)) == -1 ) {
10      perror("open␣/dev/net/tun");exit(1);
11  }
```

```
12  memset(&ifr, 0, sizeof(ifr));
13  ifr.ifr_flags = IFF_TUN | IFF_NO_PI;
14  strncpy(ifr.ifr_name, "tun0", IFNAMSIZ);
15
16  if ( (err = ioctl(fd, TUNSETIFF, (void *) &ifr)) == -1 ) {
17      perror("ioctl TUNSETIFF");close(fd);exit(1);
18  }
19
20  return fd;
21  }
```

Vytvoření UDP socketu a následná výměna úvodních zpráv je zobrazena ve výpisu 4.4. Příjem zpráv zajišťuje funkce `recvfrom()`, posílání funkce `sendto()`. Socket je také nastaven do neblokujícího režimu.

Výpis 4.4: Vytvoření UDP socketu v C++

```
1   #include <netinet/in.h>
2   #include <arpa/inet.h>
3   #include <sys/socket.h>
4   #include <fcntl.h>
5
6   int sockfd;
7   struct sockaddr_in servaddr, cliaddr;
8
9   if ( (sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0 ) {
10      perror("socket creation failed");
11      exit(EXIT_FAILURE);
12  }
13
14  memset(&servaddr, 0, sizeof(servaddr));
15  memset(&cliaddr, 0, sizeof(cliaddr));
16
17  servaddr.sin_family = AF_INET;
18  servaddr.sin_addr.s_addr = INADDR_ANY;
19  servaddr.sin_port = htons(PORT);
20
21  if ( bind(sockfd, (const struct sockaddr *)&servaddr, sizeof(servaddr)) < 0 ){
22  perror("bind failed");
23  exit(EXIT_FAILURE);
24  }
25
26  socklen_t len;
27  int n;
28  len = sizeof(cliaddr);
29  const char *hello = "Hello from server";
30
31  n = recvfrom(sockfd, (char *)buffer, MAXLINE, MSG_WAITALL, ( struct sockaddr *)
    ↪  &cliaddr, &len);
32  buffer[n] = '\0';
33  printf("Client : %s\n", buffer);
34  sendto(sockfd, (const char *)hello, strlen(hello), MSG_CONFIRM, (const struct
    ↪ sockaddr *) &cliaddr, len);
35
36  fcntl(sockfd, F_SETFL, O_NONBLOCK);
```

4.2 Práce s jádry

Práce s jádry je v šifrátoru velice jednoduchá. Šifrátor k využití dalšího jádra procesoru vytvoří nový proces pomocí funkce `fork()`. Funkce `fork()` vytvoří kopii stávajícího procesu a při úspěšném vytvoření potomka vrátí kladné celé číslo, které identifikuje proces potomka. Díky návratové hodnotě funkce `fork()` lze odlišit rodiče a potomka - rodič má kladné číslo, potomek nulu a také se lze odkázat z rodiče na potomka.

Použití funkce `fork()` v kombinaci s nekonečnou while smyčkou, použitou k šifrování a dešifrování dat, způsobí, že po ukončení rodičovského procesu bude potomek stále aktivní. Tato situace je vyřešena přidáním globální proměnné, která ukončí while smyčku při stisku kombinace kláves CTRL + C, a na konec programu je přidána funkce `kill()` s odkazem na potomka. Ve výpisu 4.5 je vidět ukončení obou procesů. Funkce `signal(SIGINT, inthand)` zajistí vykonání funkce `inthand()` při stisku kombinace CTRL + C a funkce `kill(frk, SIGTERM)` zajistí ukončení potomka.

Výpis 4.5: Vytvoření kopie procesu

```
1   #include <sys/types.h>;
2   #include <unistd.h>;
3   #include <signal.h>
4   volatile bool stop = false;
5
6   void inthand(int signum) {
7   stop = true;
8   }
9
10  int main(int argc, char* argv[]){
11  ...
12  pid_t frk = fork();
13  signal(SIGINT, inthand);
14  while (!stop){
15  ...
16  }
17  kill(frk, SIGTERM);
18  return 0;
19  }
```

4.3 Analýza výkonu, srovnání verzí

V tabulce 4.1 je vidět porovnání s implementací v jazyce Pythonu. Srovnávanými vlastnostmi bylo opět RTT, propustnost šifrovaného kanálu a přenesená data navíc.

Tab. 4.1: Srovnání Python x C++

Implementace	RTT [ms]	Rychlost [Mbps]	Přenesená data [MB]
`Python`	`3,574`	`36`	`108`
`C++`	`2,706`	`175`	`108`

Z tabulky vyplývá, že po optimalizaci se propustnost šifrovaného kanálu zvedla o 486 procent.

5 Rozšíření funkcionality

Všechny dosud zmíněné implementace používaly k šifrování statické klíče obsažené přímo v kódu. Tato kapitola pojednává o dalších funkcích přidaných do šifrátoru - především o mechanismech pracujících s klíčem. Pro potřeby ustanovení klíčů byl šifrátor rozdělen na serverovou a klientskou část.

5.1 Ustanovení klíče

Ustanovení klíče je jedna ze základních funkcí každého šifrovacího systému. V šifrátoru byly použity metody post-kvantové kryptografie k ustanovení klíče.

5.1.1 Mechanismy ustanovení klíče

Do šifrátoru byly přidány 2 mechanismy k ustanovení klíče. První využíval post-kvantový algoritmus CRYSTALS - Kyber, založený na mřížkové kryptografii, k zapouzdření klíče. Druhý mechanismus představoval QKD.

5.1.2 KEM

K implementaci zapouzdření klíče byla využita knihovna `Kyber`[1].
Průběh komunikace v implementaci má následující kroky:

- Strana A vygeneruje soukromý a veřejný klíč
- Strana A pošle straně B veřejný klíč
- Strana B vygeneruje klíč o délce 256 bitů
- Strana B zašifruje vygenerovaný klíč veřejným klíčem strany A a pošle straně A
- Strana A dešifruje zprávu strany B soukromým klíčem
- Obě strany nyní mají k dispozici shodný klíč o velikosti 256 bitů

Implementace strany A je zobrazena ve výpisu 5.1. Implementace strany B je zobrazena ve výpisu 5.2.

Výpis 5.1: KEM - strana A

```
1   #include "kyber512_kem.hpp"
2
3   uint8_t pkey[kyber512_kem::pub_key_len()];
4   uint8_t skey[kyber512_kem::sec_key_len()];
5   prng::prng_t prng;
6   kyber512_kem::keygen(prng, pkey, skey);
7
```

[1]Dostupná z: https://github.com/itzmeanjan/kyber

```
8   send(client_fd, pkey, sizeof(pkey), 0);
9   read(client_fd, buffer, MAXLINE);
10  auto rkdf = kyber512_kem::decapsulate(skey, (const unsigned char*)buffer);
11  uint8_t* shrd_key = static_cast<uint8_t*>(std::malloc(32));
12  rkdf.read(shrd_key, 32);
13  string pqc_key = kyber_utils::to_hex(shrd_key, 32);
```

Výpis 5.2: KEM - strana B

```
1   #include "kyber512_kem.hpp"
2
3   uint8_t cipher[kyber512_kem::cipher_text_len()];
4   prng::prng_t prng;
5
6   read(new_socket, buffer, MAXLINE);
7   auto skdf = kyber512_kem::encapsulate(prng, (const unsigned char*)buffer, c>
8   uint8_t* shrd_key = static_cast<uint8_t*>(std::malloc(32));
9   skdf.read(shrd_key, 32);
10  string pqc_key=kyber_utils::to_hex(shrd_key, 32);
11  send(new_socket, cipher, sizeof(cipher), 0);
```

5.1.3 QKD

Systém pro kvantovou distribuci klíče byl již sestaven na Fakultě elektrotechniky a komunikačních technologií VUT v Brně. Každá z komunikujících stran má přístup k jednomu QKD serveru - servery jsou označeny QKDA, QKDB. Klíče jsou získávány dotazem na API („Application Programming Interface“) QKD serveru. Výsledkem dotazování je klíč a identifikace klíče ve formátu JSON („JavaScript Object Notation“). Komunikace se servery probíhá šifrovaně pomocí HTTPS.

Komunikace probíhá následovně:

- Strana A pošle dotaz na server QKDA
- Server QKDA odpoví klíčem a jeho identifikací
- Strana A pošle identifikaci klíče straně B
- Strana B pošle dotaz na server QKDB s identifikací klíče
- Server QKDB pošle klíč straně B
- Strana A a B nyní disponují shodným klíčem

Pro získání klíčů byl v předchozí diplomové práci[2] Ing. Andreje Krivulčíka vytvořen shell skript. Pro účely šifrátoru byl skript upraven do podoby ve výpisu 5.3. Skript byl tedy rozšířen o uložení klíče a jeho identifikátoru do samostatných souborů. Dále byly přidány parametry režim a IP adresa QKD serveru.

[2]Dostupné z: https://www.vut.cz/studenti/zav-prace/detail/141390

Výpis 5.3: Skript pro získání QKD klíčů

```
1   #!/bin/sh
2   CURRENTDIR=$(dirname $0)
3   KMSM_IP=$2
4
5   if [ $1 = "client" ]
6   then
7   Rep=$(curl --cert ENCA-cert.pem  --key ENCA-key.pem --cacert ca-cert.pem -k
    ↪ https://$KMSM_IP/api/v1/keys/ENCB/enc_keys)
8   Key=$(echo $Rep | jq '.keys[0].key' | cut -d '"' -f 2)
9   KeyIDSlave=$(echo $Rep | jq '.keys[0].key_ID' | cut -d '"' -f 2)
10  echo -n $Key | base64 -d | hexdump -v -e '/1 "%02x" ' > klic
11  echo $KeyIDSlave > keyID
12  fi
13
14  if [ $1 = "server" ]
15  then
16  KeyIDSlave=$(cat keyID)
17  Rep=$(curl --cert ENCB-cert.pem  --key ENCB-key.pem --cacert ca-cert.pem -X
    ↪ POST -H 'Content-Type:application/json' -d "{\"key_IDs\":[{\"key_ID\":\"
    ↪ $KeyIDSlave\"}]}" -k https://$KMSS_IP/api/v1/keys/ENCA/dec_keys)
18  Key=$(echo $Rep | jq '.keys[0].key' | cut -d '"' -f 2)
19  KeyIDSlave=$(echo $Rep | jq '.keys[0].key_ID' | cut -d '"' -f 2)
20  echo -n $Key | base64 -d | hexdump -v -e '/1 "%02x" ' > klic
21  fi
```

Simulátor QKD

Během vytváření šifrátoru nebylo vždy možné zajistit přístup na QKD servery. Z toho důvodu byl vytvořen simulátor QKD. Simulátor je napsán v jazyce PHP. V topologii je simulátor umístěn na klientovi a serveru z důvodu ušetření výpočetních zdrojů. K hostování simulátoru byl použit server typu Apache. Příkazy nutné k instalaci serveru a povolení PHP jsou zobrazeny ve výpisu 5.4

Výpis 5.4: Instalace serveru apache a povolení PHP

```
apt-get install apache2
apt-get install php7.4
a2dismod mpm_event
a2enmod php7.4
systemctl restart apache2
```

Simulátor disponuje třemi statickými klíči a jejich identifikací dříve získanou z QKD serverů. Klíče se přidělují odesláním GET požadavku na adresu http://simulator1/klic/. Při příjmu GET požadavku server vygeneruje náhodné číslo 1 až 3, na základě kterého přidělí příslušný klíč. Získání shodného klíče proběhne pomocí POST požadavku s identifikací klíče na adresu http://simulator2/ID/. Implementace simulátoru je zobrazena ve výpisech 5.5 a 5.6.

Výpis 5.5: Simulátor - přidělování klíčů

```
<?php
$x = rand(1,3);
switch ($x) {
case 1:
  echo '{
    "keys":[{
    "key_ID":"8689982f-22ca-4afe-a0ce-abc43f06a2bf",
    "key":"B2s+pwhnHDjXIlXVuOC423TRZZG8Ahe6t6cR+JUnxpk="
    }]
}';
break;
case 2:
    echo '{
    "keys":[{
    "key_ID":"ebda3721-ad57-4774-937f-efb6d6ac5f46",
    "key":"5dV1r7GCIdT5mVDR2\/XoHYJ9DcPVDAV\/s3h\/TO5cnyE="
    }]
}';
break;
case 3:
    echo '{
    "keys":[{
    "key_ID":"5d78c28c-85bf-4701-8fb0-31607dcdf479",
    "key":"aOf3KnknCfd9tHOce6HUvBYxdw34bnhaRSx4f6OYq3Q="
    }]
}';
break;
}
?>
```

Výpis 5.6: Simulátor - získání klíče podle ID

```
<?php
$keyID = $_POST['keyID'];
switch ($keyID) {
case "8689982f-22ca-4afe-a0ce-abc43f06a2bf":
    echo '{
    "keys":[{
    "key_ID":"8689982f-22ca-4afe-a0ce-abc43f06a2bf",
    "key":"B2s+pwhnHDjXIlXVuOC423TRZZG8Ahe6t6cR+JUnxpk="
    }]
}';
break;
case "ebda3721-ad57-4774-937f-efb6d6ac5f46":
    echo '{
    "keys":[{
    "key_ID":"ebda3721-ad57-4774-937f-efb6d6ac5f46",
    "key":"5dV1r7GCIdT5mVDR2\/XoHYJ9DcPVDAV\/s3h\/TO5cnyE="
    }]
}';
break;
case "5d78c28c-85bf-4701-8fb0-31607dcdf479":
    echo '{
    "keys":[{
    "key_ID":"5d78c28c-85bf-4701-8fb0-31607dcdf479",
    "key":"aOf3KnknCfd9tHOce6HUvBYxdw34bnhaRSx4f6OYq3Q="
    }]
}';
```

```
27  break;
28
29  default:
30      echo "ChybneID";
31  }
32  ?>
```

5.1.4 Hybridní klíč

Myšlenka hybridního klíče spočívá v kombinování několika klíčů do jednoho. Útočník by tedy neměl být schopen zjistit hodnotu hybridního klíče bez znalosti klíčů, ze kterých je nakombinován. V šifrátoru je hybridní klíč vytvořen z PQC klíče a QKD klíče. Spojení klíčů proběhlo pomocí hašovací funkce SHA-256. Funkce k získání hybridního klíče je zobrazena ve výpisu 5.7. Klíč získaný pomocí zapouzdření je funkci předán jako parametr, QKD klíč je vyčten ze souboru. Řetězce představující klíče jsou spojeny dohromady a předány na vstup hašovací funkce. Výstupem hašovací funkce je hybridní klíč v hexadecimální podobě, který je dále převeden na byty.

Výpis 5.7: Vytvoření hybridního klíče

```
1   SecByteBlock VymenaKlice_srv(string pqc_key){
2
3   CryptoPP::SHA256 hash;
4   byte digest[ CryptoPP::SHA256::DIGESTSIZE ];
5   SecByteBlock key( AES::MAX_KEYLENGTH );
6
7   std::ifstream t("klic");
8   std::stringstream buffer;
9   buffer << t.rdbuf();
10
11  std::string message = buffer.str() + pqc_key;
12  hash.CalculateDigest( digest, (byte*) message.c_str(), message.length() );
13  CryptoPP::HexEncoder encoder;
14  std::string output;
15  encoder.Attach( new CryptoPP::StringSink( output ) );
16  encoder.Put( digest, sizeof(digest) );
17  encoder.MessageEnd();
18
19  int x = 0;
20  for (unsigned int i = 0; i < output.length(); i += 2) {
21  std::string bytestring = output.substr(i, 2);
22  key[x] = (char)strtol(bytestring.c_str(), NULL, 16);
23  x++;
24  }
25  return key;
26  }
```

5.1.5 Rekeying

Rekeying představuje výměnu klíče za nový. Rekeying je důležitý kvůli omezenému množství potencionálních hodnot nonce, kdy při stejné kombinaci klíče a hodnoty

nonce může dojít k prozrazení otevřeného textu, nebo v horším případě prozrazení podklíče H, sloužícímu k výpočtu MAC [15].

Rekeying probíhá po příjmu/odeslání 200 000 zpráv v rodičovském procesu klientské části šifrátoru. Počet zpráv zašifrovaných jedním klíčem by neměl přesáhnout 2^{32} [1]. Hodnota 200 000 je zvolena z důvodu nedeterministického volení nonce hodnoty a počítání zpráv pouze v jednom procesu jedné strany.

Pro změnu klíče byl vytvořen nový TCP kanál pro posílání identifikace QKD klíče serveru. Ve funkci pro výměnu klíče, v klientské části šifrátoru, bylo přidáno odesílání identifikace klíče serverové části. Rekeying tedy probíhá získáním nového QKD klíče a přepočtem hybridního klíče. PQC klíč zůstává po celou dobu šifrování stejný. Během rekeyingu dojde ke ztrátě několika paketů, kvůli paketům uloženým ve vyrovnávací paměti, zašifrovanými starým klíčem.

Synchronizace klíče mezi procesy

Při změně klíče je potřeba, aby oba procesy - jak rodič, tak potomek, pracovaly se stejným klíčem. Pokud by měli odlišný klíč, docházelo by k zahazování velkého množství paketů z důvodu selhání kontroly integrity. Šifrátor nemá implementovaný žádný způsob komunikace mezi procesy, tudíž je nutné ukončit proces potomka, vyměnit klíč a znovu vytvořit kopii procesu. Implementace celého rekeyingu je zobrazena ve výpisu 5.8.

Výpis 5.8: Synchronizace klíče mezi procesy

```
1   int counter = 0;
2   while (counter<200000){
3   ...
4   counter++
5   ...
6   }
7   if (frk > 0){
8   counter = 0;
9   kill(frk, SIGTERM);
10  key = VymenaKlice_cli(client_fd, pqc_key, qkd_ip);
11  frk = fork();
12  }
```

5.2 Řešení dynamického NAT

Po nasazení šifrátoru na mezifakultní spoj se projevily problémy kvůli dynamickému překladu adres. Šifrátor - klient (192.168.10.3) se připojoval na veřejnou adresu šifrátoru - server (147.229.177.162). Kvůli vytvoření dočasného NAT překladu pro komunikaci ze serveru na klienta byl šifrovací kanál po několika vteřinách neaktivity nepoužitelný z důvodu vypršení NAT záznamu. Problém je v šifrátoru vyřešen

posíláním výplňových zpráv ze serverové části v době nečinnosti. Ve výpisu 5.9 je zobrazeno řešení problému dynamického překladu adres. Udržení překladu adres je použito pro TCP i UDP socket.

Výpis 5.9: Posílání výplňových zpráv

```
1   #include <ctime>
2
3   const char* keepalive = "Keep Alive";
4   time_t ref = time(NULL);
5   while (){
6   ...
7   if (time(NULL)-ref>=1){
8   send(new_socket, keepalive, strlen(keepalive), 0);
9   sendto(sockfd, keepalive, strlen(keepalive), MSG_CONFIRM, (const struct
    ↪ sockaddr *) &cliaddr, len);
10   ref = time(NULL);
11  }}
```

5.3 VPN přístup

Pro vyzkoušení šifrátoru byl vytvořen přístup přes VPN, použitým VPN řešením bylo OpenVPN. VPN přístup byl vytvořen na první bráně (v obr 3.1 brána A). K vytvoření přístupu byl využit skript dostupný z: `https://raw.githubusercontent.com/Angristan/openvpn-install/master/openvpn-install.sh`.

6 Zhodnocení výsledné implementace

V této kapitole je popsána výsledná implementace - její instalace, ovládání a měření výkonu.

6.1 Instalace šifrátoru

Všechny soubory šifrátoru jsou dostupné na adrese: `https://github.com/gabsssq/sifrator`. Součástí jsou i skripty `install.sh` a `install_QKD.sh`. Pro instalaci šifrátoru stačí vytvořit kopii adresáře a spustit skript `install.sh`. Skript vyžaduje parametr sítě, pro kterou má šifrovat provoz. Skript je zobrazen ve výpisu 6.1.

Výpis 6.1: Skript pro instalaci šifrátoru

```
1   #!/bin/sh
2
3   Help()
4   {
5   echo "Pouziti: ./install.sh [Sit_druhe_brany]"
6   echo "Tvar site: x.x.x.x/y"
7   echo
8   }
9
10  if [ $# -lt 1 ] || [ $1 = "-help" ] || [ $1 = "-h" ]
11  then
12  Help
13  exit 1
14  fi
15
16  cat requirements.txt | sudo xargs apt install -y
17  Route_IP=$1
18
19  git clone https://github.com/itzmeanjan/kyber.git
20  (cd kyber && git submodule update --init)
21
22  wget https://www.cryptopp.com/cryptopp870.zip
23  unzip -aoq cryptopp870.zip -d cryptopp
24  (cd cryptopp && sudo make)
25  (cd cryptopp && sudo make install)
26
27  sudo ip tuntap add name tun0 mode tun
28  sudo ip link set tun0 up
29  sudo ip addr add 192.168.1.1 peer 192.168.1.2 dev tun0
30
31  echo "1" | sudo tee /proc/sys/net/ipv4/ip_forward
32  sudo ip route add $Route_IP via 192.168.1.2
33  chmod +x sym-ExpQKD
34  g++ -std=c++20 -O3 -pthread -I /usr/local/include/ -I ./kyber/include/ -I ./
    ↪ kyber/subtle/include/ -I ./kyber/sha3/include/ sifrator_server.cpp  /usr/
    ↪ local/lib/libcryptopp.a -o sifrator_server
35  g++ -std=c++20 -O3 -pthread -I /usr/local/include/ -I ./kyber/include/ -I ./
    ↪ kyber/subtle/include/ -I ./kyber/sha3/include/ sifrator_client.cpp  /usr/
    ↪ local/lib/libcryptopp.a -o sifrator_client
```

Skript nejprve nainstaluje závislosti ze souboru `requirements.txt`, konkrétně nástroje `unzip`, `g++`, `make`, `jq` a `curl`. Dále jsou nainstalovány knihovny `Kyber` a `Crypto++`. Následně je vytvořeno virtuální rozhraní s adresou 192.168.1.2, přidán směrovací záznam a povoleno směrování. V posledním kroku jsou zkompilovány obě části šifrátoru.

Druhý skript `install_QKD.sh` slouží k instalaci simulátoru QKD. Skript je zobrazen ve výpisu 6.2.

Výpis 6.2: Skript pro instalaci QKD simulátoru

```
1   #!/bin/sh
2
3   apt-get install apache2
4   apt-get install php7.4
5   a2dismod mpm_event
6   a2enmod php7.4
7   systemctl restart apache2
8
9   mkdir /var/www/html/klic
10  mkdir /var/www/html/ID
11  cp index-klic.php /var/www/html/klic/index.php
12  cp index-ID.php /var/www/html/ID/index.php
13  (cd /var/www/html && fallocate -l 1G test)
```

Skript nainstaluje server Apache a povolí PHP. Následně zkopíruje soubory simulátoru do příslušných složek. Nakonec vytvoří testovací soubor o velikosti 1 GB.

6.2 Ovládání

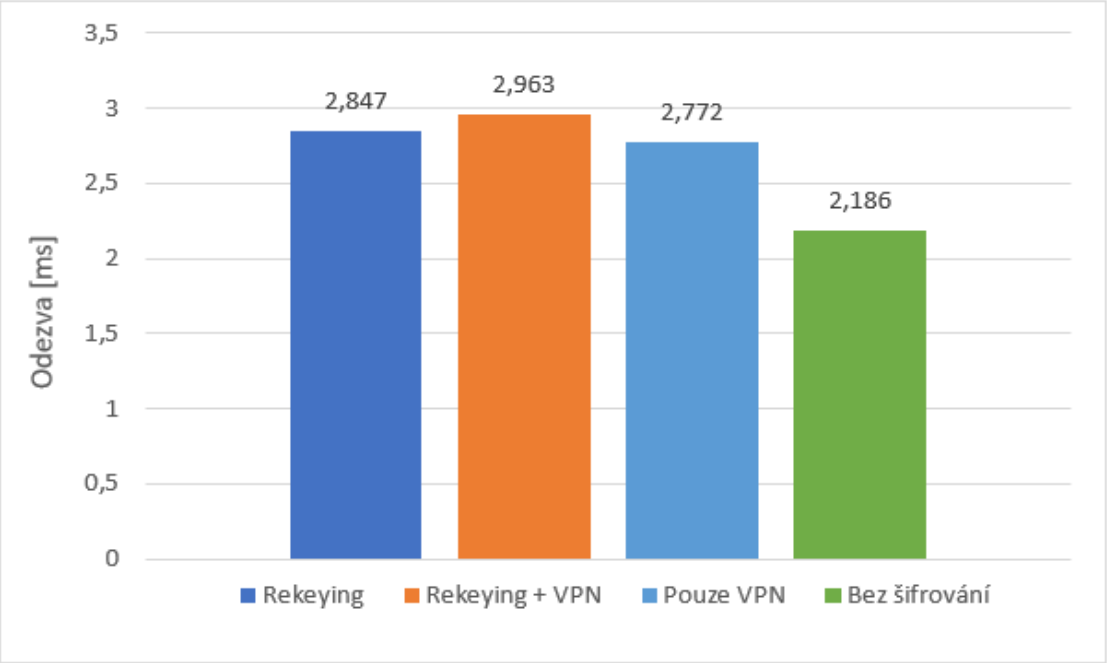
Po instalaci šifrátoru jsou vytvořeny soubory `sifrator_server` a `sifrator_client` představující serverovou a klientskou část šifrátoru. Pro vytvoření šifrovaného kanálu se nejdříve na první šifrovací bráně spustí program `sifrator_server` s parametrem IP adresy prvního QKD serveru. Na druhé šifrovací bráně se spustí program `sifrator_client` s dvěma parametry - IP adresa druhého QKD serveru a IP adresa první šifrovací brány. Šifrovaný spoj je nyní aktivní a lze vyzkoušet pomocí přenosu testovacích souborů umístěných na simulátorech.

6.3 Měření výkonu

Při měření rychlosti výsledné implementace se měřila doba RTT a přenos souborů různých velikostí z HTTP serveru. Velikosti souborů byly 10 KB, 1 MB, 500 MB, 1 GB a 5 GB. V měření je porovnán vliv rekeyingu a VPN na rychlost přenosu dat.

6.3.1 Šifrátor - RTT

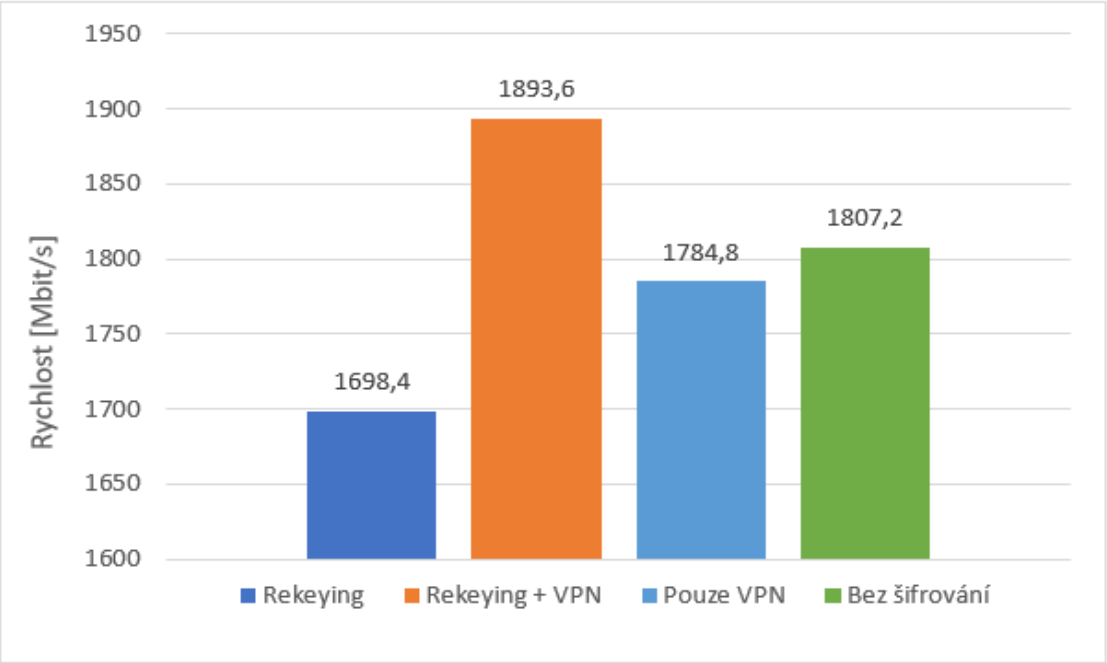
Pro měření RTT byl opět použit nástroj `ping` a zaznamenával se průměrný čas odezvy po 15 žádostech. Na obrázku 6.1 je zobrazen graf průměrné doby odpovědí na ping žádost.



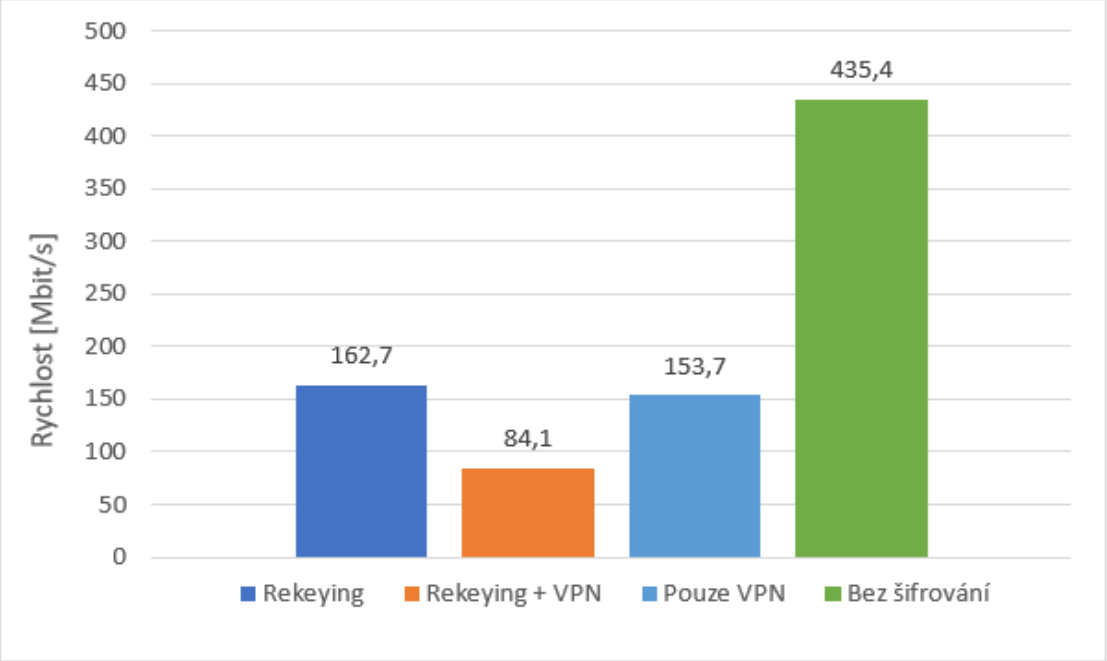
Obr. 6.1: Porovnání RTT

6.3.2 Šifrátor - přenos souborů

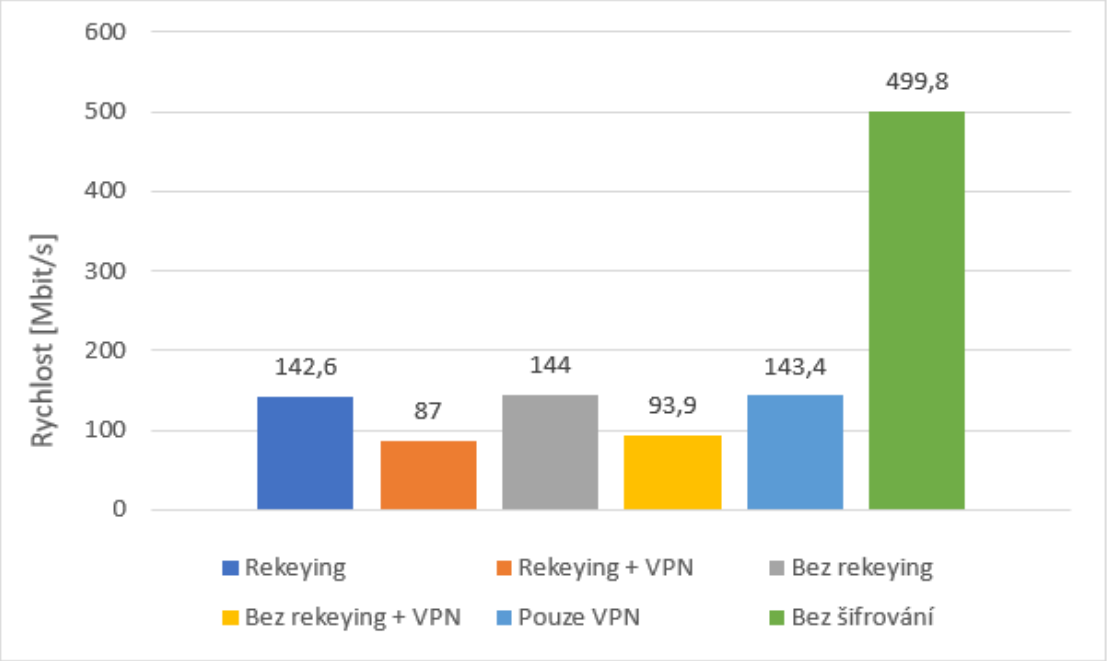
Pro přenos souborů byl použit nástroj `wget`. Měření probíhalo v 10 instancích, z nichž byla vypočtena průměrná hodnota rychlosti šifrování. Na obrázcích 6.2, 6.3, 6.4, 6.5 a 6.6 jsou zobrazeny porovnání pro různé velikosti souborů. Na obrázcích 6.7, 6.8, 6.9 a 6.10 jsou dále vidět grafy vývoje rychlosti pro soubory velikosti 1 GB a 5 GB pro demonstraci vlivu rekeyingu na přenosovou rychlost.



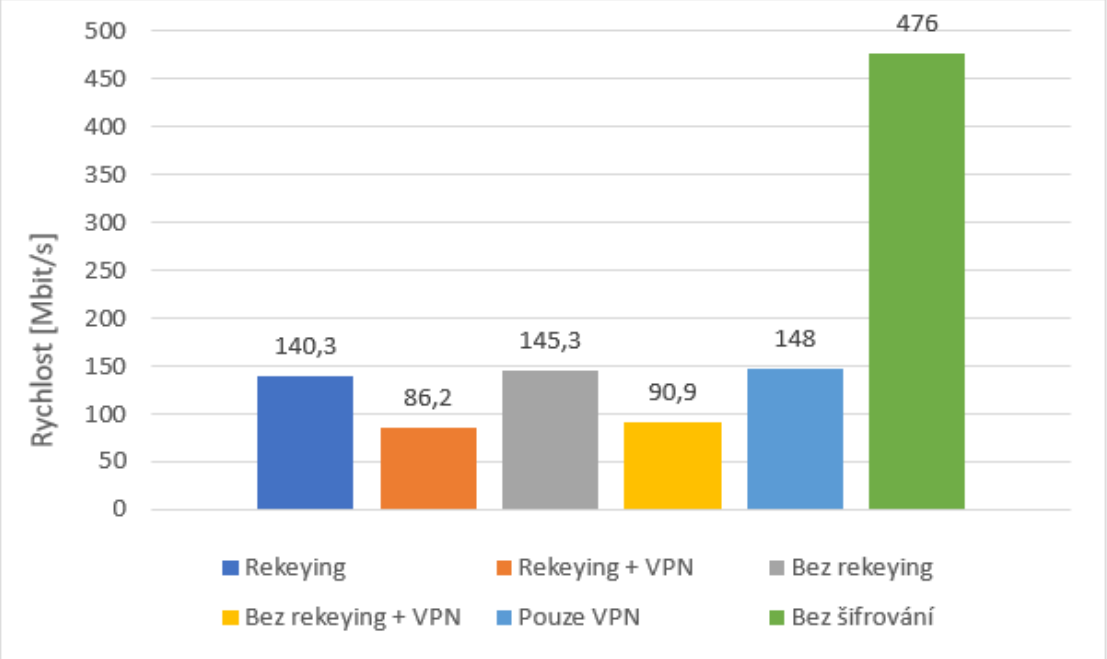
Obr. 6.2: Přenos souboru 10 KB



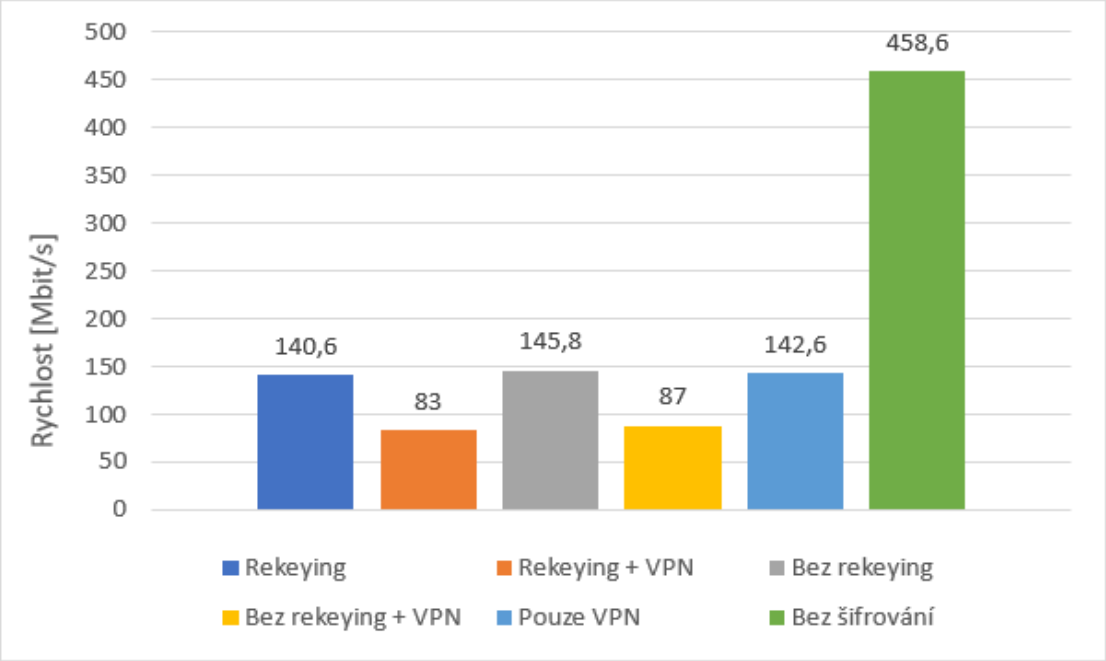
Obr. 6.3: Přenos souboru 1 MB



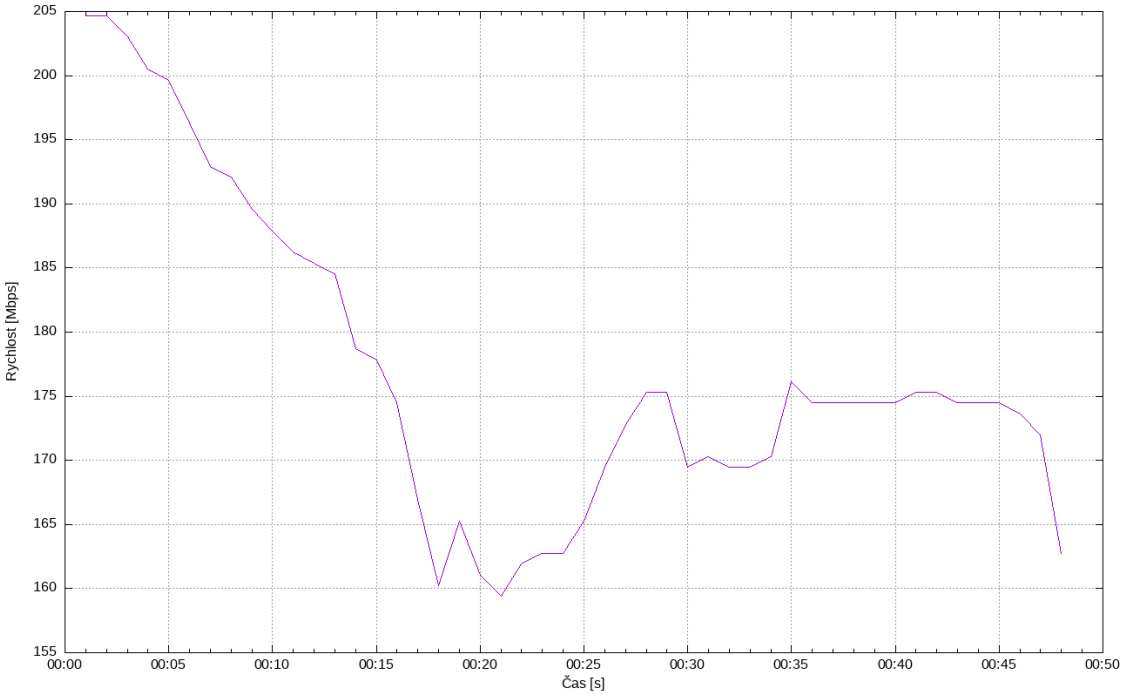
Obr. 6.4: Přenos souboru 500 MB



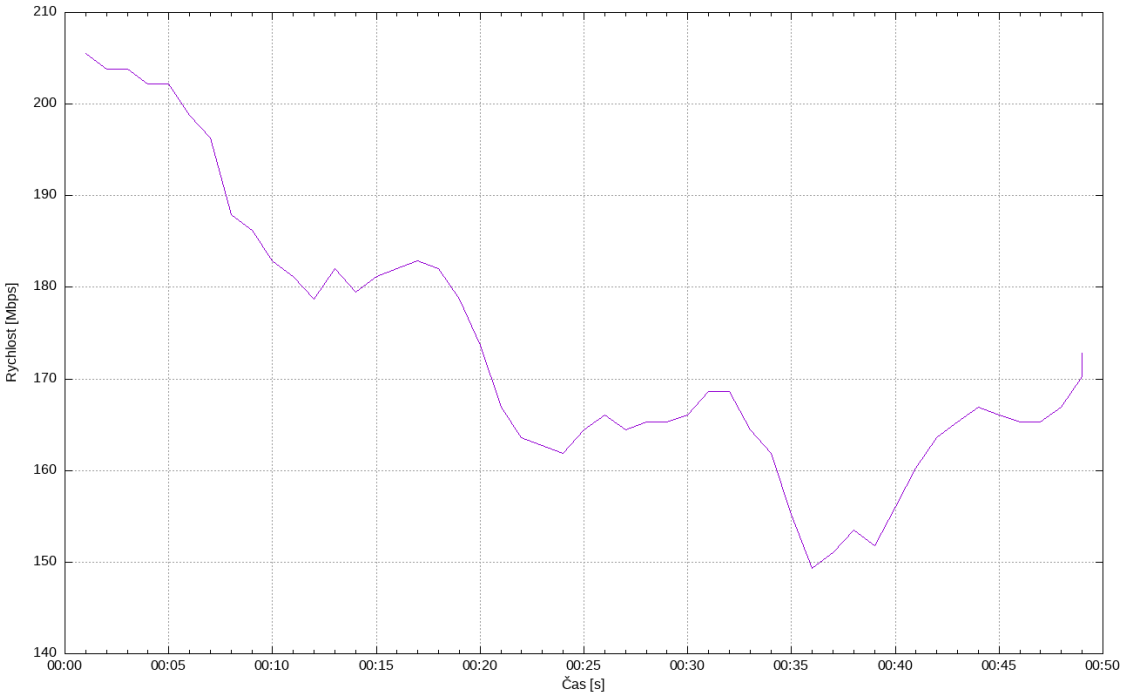
Obr. 6.5: Přenos souboru 1 GB



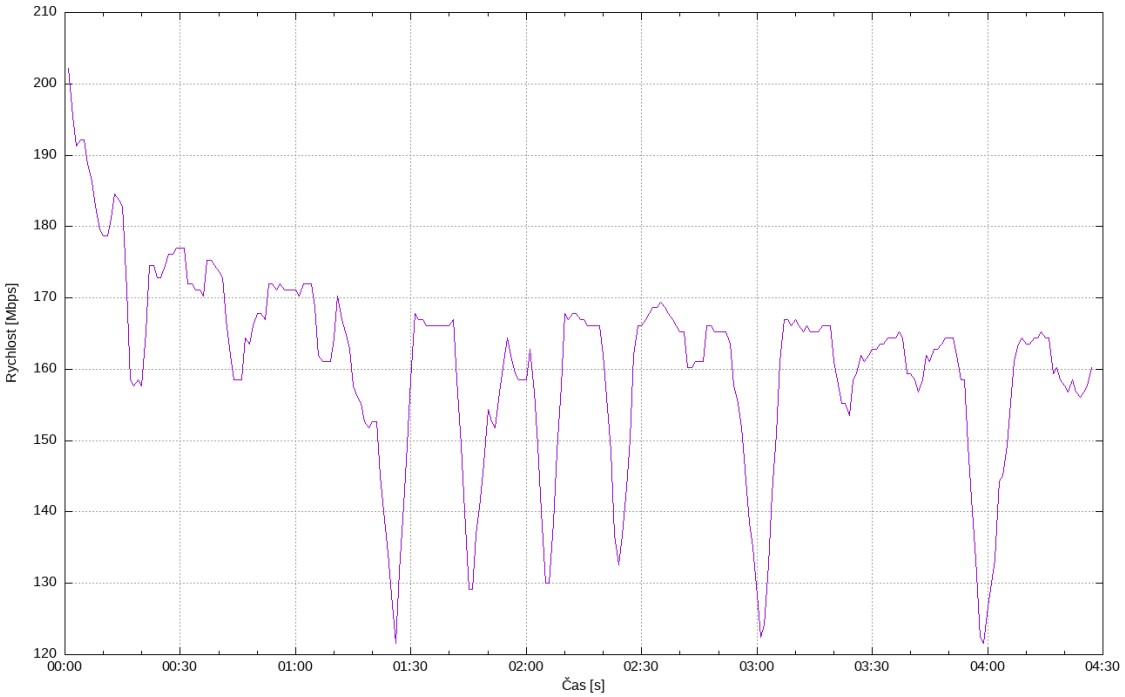
Obr. 6.6: Přenos souboru 5 GB



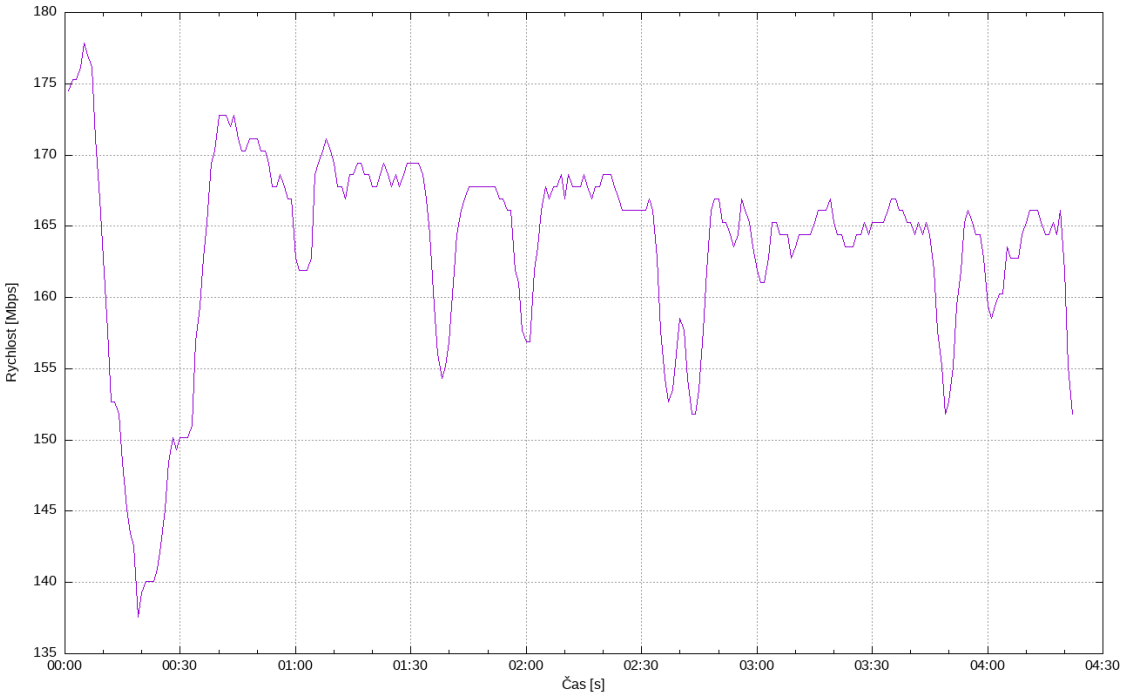
Obr. 6.7: Průběh přenosu souboru 1 GB s rekeyingem



Obr. 6.8: Průběh přenosu souboru 1 GB bez rekeyingu



Obr. 6.9: Průběh přenosu souboru 5 GB s rekeyingem



Obr. 6.10: Průběh přenosu souboru 5 GB bez rekeyingu

6.3.3 Zhodnocení výsledků měření

Z výsledků vyplývá předpokládané - nejrychlejší je nešifrovaný spoj, dále spoj šifrovaný mezi branami a nejpomalejší je varianta připojení přes VPN a šifrovaného spoje mezi branami. V tabulce 6.1 jsou přehledně shrnuty výsledky měření.

Tab. 6.1: Výsledky měření

Šifrování	RTT [ms]	10 KB [Mbit/s]	1 MB [Mbit/s]	500 MB [Mbit/s]	1 GB [Mbit/s]	5 GB [Mbit/s]
Rekeying	2,847	1698,4	162,7	142,6	140,3	140,6
Rekeying + VPN	2,963	1893,6	84,1	87	86,2	83
Bez rekeying	-	-	-	144	145,3	145,8
Bez rekeying + VPN	-	-	-	93,9	90,9	87
Pouze VPN	2,772	1784,8	153,7	143,4	148	142,6
Bez šifrování	2,186	1807,2	435,4	499,8	476	458,6

V tabulce nejsou vyplněny všechny hodnoty, protože v některých případech nemá přítomnost výměny klíče vliv na výsledek měření.

Při přenosu souboru o velikosti 10 KB se zdají být všechny varianty spoje extrémně rychlé. Tyto výsledky jsou zapříčiněny způsobem měření průměrné rychlosti nástroje `wget`, kdy pro velmi malé soubory nedokáže přesně změřit přenosovou rychlost. Pro větší soubory 500 MB, 1 GB a 5 GB byly rozdíly mezi naměřenými hodnotami nepatrné.

Z výsledků dále vyplývá, že šifrátor má na zpomalení provozu téměř shodný vliv jako VPN spoj. VPN spoj[1] ale pro svou funkci využívá pouze 1 jádro CPU oproti šifrátoru, který využívá 2.

V porovnání průběhu přenosu souborů je vidět vliv rekeyingu na přenosovou rychlost. Především u přenosu souboru o velikosti 5 GB jsou zaznamenány větší propady rychlosti, a to z důvodu zahazování paketů při výměně klíče. Změna klíče byla vyvolána zhruba každých 16 sekund. U přenosu souboru o velikosti 1 GB není vliv změny klíče v grafu natolik patrný - ke změně došlo během přenosu celkem třikrát.

[1] Realizovaný pomocí OpenVPN

Závěr

Cílem diplomové práce bylo vytvořit virtuální stroj založený na OS Linux, který dokáže zašifrovat a dešifrovat příchozí síťový provoz protokolu IPv4 a odeslat jej odlišným síťovým rozhraním zpět do sítě.

V teoretické části diplomové práce byly popsány vlastnosti přenosu, které lze zajistit kryptografií. Poté byly krátce představeny použité algoritmy - konkrétně AES-GCM a CRYSTALS Kyber.

V praktické části diplomové práce byly realizovány tři způsoby šifrování v OS Linux. Všechny způsoby byly naprogramovány v jazyce Python. První způsob k odchycení paketů využíval netfilter modul NFQUEUE. Zbylé dva odchytávaly pakety pomocí virtuálních rozhraní. Šifrování používalo knihovnu `PyCryptodomex`. Prostředí šifrovacího systému se skládalo ze čtyř počítačů - klient, server a dvě šifrovací brány.

Pro porovnání přístupů šifrování byly vybrány tři testy - doba RTT, propustnost a přenesená data pro soubor o velikosti 100 MB. Nejlépe z testů vyšel přístup šifrování celého paketu na virtuálním rozhraní, proto byl tento způsob vybrán pro použití v šifrátoru.

Kvůli nízké rychlosti šifrování bylo potřeba šifrátor optimalizovat. Nejvýznamnější nárust rychlosti šifrování byl způsoben změnou z interpretovaného programovacího jazyka na kompilovaný, změna z Pythonu na C++. Knihovnou pro šifrování byla `Crypto++`. Optimalizace způsobila nárust výkonu o 486%.

Do šifrátoru byly dále přidány dva mechanismy ustanovení klíče. Přidanými mechanismy byl KEM, realizovaný algoritmem CRYSTALS Kyber, a QKD. Z klíčů získaných pomocí KEM a QKD byl vytvořen hybridní klíč, sloužící k šifrování pomocí symetrické šifry AES-GCM. Další přidanou vlastností byla změna klíče po zašifrování 200 000 zpráv. Výměna klíče proběhne vyžádáním nového QKD klíče, který zkombinuje se stávajícím PQC klíčem, získaným KEM.

Dalšími návrhy na vylepšení šifrátoru jsou přidání možnosti výběru použitých šifer, deterministické volení nonce hodnoty, vytvoření virtuálního rozhraní v rámci programu, přidání komunikace mezi procesy a práce s vlákny, přidání opětovného připojení při ztrátě spojení mezi sockety, nezahazovat pakety při rekeyingu.

Výsledkem diplomové práce je funkční šifrátor síťového provozu, používající kvantově odolné kryptografické algoritmy, na platformě Linux.

Literatura

[1] DWORKIN, Morris. Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC [online]. November 2007 [cit. 2022-12-02]. Dostupné z: doi:https://doi.org/10.6028/NIST.SP.800-38D

[2] SCHWABE, Peter. Kyber. CRYSTALS: Cryptographic Suite for Algebraic Lattices [online]. Dec 23, 2020 [cit. 2023-04-23]. Dostupné z: https://pq-crystals.org/kyber/index.shtml

[3] Kryptografie v základním kontextu. TechPedia [online]. [cit. 2023-05-09]. Dostupné z: https://techpedia.fel.cvut.cz/html/frame.php?oid=51&pid=1002&finf=

[4] BARAKAT, Mohamed, Christian EDER a Timo HANKE. An Introduction to Cryptography [online]. 2018 [cit. 2022-12-02]. Dostupné z: https://www.mathematik.uni-kl.de/~ederc/download/Cryptography.pdf

[5] DWORKIN, Morris, Elaine BARKER, James NECHVATAL, James FOTI, Lawrence BASSHAM, E. ROBACK a James DRAY. FIPS 197: ADVANCED ENCRYPTION STANDARD (AES) [online]. Gaithersburg, 2001 [cit. 2022-11-30]. Dostupné z: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf

[6] O'SHEA, Dan. AES-256 joins the quantum resistance. Fierceelectronics.com [online]. Apr 29 2022 [cit. 2023-04-23]. Dostupné z: https://www.fierceelectronics.com/electronics/aes-256-joins-quantum-resistance

[7] DWORKIN, Morris. Block Cipher Modes. NIST [online]. 4. 1. 2017 [cit. 2022-11-30]. Dostupné z: https://csrc.nist.gov/projects/block-cipher-techniques/bcm

[8] HUGHES, Alan. The Disadvantages of Asymmetric Encryption. Techwalla [online]. [cit. 2022-12-02]. Dostupné z: https://www.techwalla.com/articles/the-disadvantages-of-asymmetric-encryption

[9] MICROSOFT CORPORATION. Understanding Point-to-Point Tunneling Protocol (PPTP) [online]. January 1997 [cit. 2022-12-07]. Dostupné z: https://wwwdisc.chimica.unipd.it/luigino.feltre/pubblica/unix/winnt_doc/pppt/understanding_pptp.html

[10] CRAWFORD, Douglas. OpenVPN vs IKEv2 vs PPTP vs L2TP/IPSec vs SSTP - Ultimate Guide to VPN Encryption. ProPrivacy [online]. June 30, 2020 [cit. 2022-12-07]. Dostupné z: https://proprivacy.com/vpn/guides/vpn-encryption-the-complete-guide#vpn-encryption-protocols

[11] DONENFELD, Jason. Wireguard: fast, modern, secure VPN tunnel. Wireguard [online]. [cit. 2022-12-08]. Dostupné z: https://www.wireguard.com/

[12] LEBLOND, Éric. Using NFQUEUE and libnetfilter_queue. Home.regit.org [online]. [cit. 2022-12-05]. Dostupné z: https://home.regit.org/netfilter-en/using-nfqueue-and-libnetfilter_queue/

[13] KRASNYANSKY, Maxim a Maksim YEVMENKIN. Universal TUN/TAP device driver. Kernel.org [online]. [cit. 2022-12-05]. Dostupné z: https://www.kernel.org/doc/html/v5.8/networking/tuntap.html

[14] EVANS, Julia. How to send raw network packets in Python with tun/tap. Julia Evans [online]. [cit. 2022-12-05]. Dostupné z: https://jvns.ca/blog/2022/09/06/send-network-packets-python-tun-tap/

[15] JOUX, Antoine. Authentication failures in NIST version of GCM [online]. In: . January 2006 [cit. 2023-04-22]. Dostupné z: https://csrc.nist.gov/csrc/media/projects/block-cipher-techniques/documents/bcm/joux_comments.pdf

Seznam symbolů a zkratek

AES	Advanced Encryption Standard
API	Application Programming Interface
CPU	Centrální procesorová jednotka
GCM	Galois Counter Mode
HTTP	Hypertext Transfer Protocol
IP	Internet Protocol
IPsec	Internet Protocol Security
JSON	JavaScript Object Notation
KEM	Zapouzdření klíče
LWE	Learning With Errors
L2TP	Layer Two Tunneling Protocol
MAC	Message Authentication Code
MMPE	Microsoft Point-to-Point Encryption
MTU	Maximum Transmission Unit
NAT	Network Address Translation
NIST	National Institute of Standards and Technology
OS	Operační systém
PPP	Point-to-Point Protocol
PPTP	Point-to-Point Tunneling Protocol
PQC	Postkvantová kryptografie
QKD	Kvantová distribuce klíče
RFC	Request for Comments
RTT	Round-trip Time
SSL	Secure Sockets Layer

SSTP	Secure Socket Tunneling Protocol
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UDP	User Datagram Protocol
VPN	Virtuální privátní síť
3DES	Triple Data Encryption Standard

Seznam příloh

A Zprovoznění šifrátoru

Nejprve je nutné nainstalovat QKD simulátor na server a klient, omezit MTU na 1440 a nastavit výchozí cestu na šifrovací bránu. Příkazy k instalaci a omezení MTU jsou vidět ve výpisu A.1. V příkladové architektuře se používají sítě 10.0.1.0/24 pro server, 10.0.3.0/24 pro klienta a 10.0.2.0/24 pro spoj mezi branami.

Výpis A.1: Nastavení QKD simulátoru

```
xtumap02@server:~$ sudo git clone https://github.com/gabsssq/sifrator.git
xtumap02@server:~$ cd sifrator
xtumap02@server:~$ sudo bash install_QKD.sh
xtumap02@server:~$ sudo ip link set enp0s3 mtu 1440
xtumap02@server:~$ sudo ip route del 0.0.0.0/0 via 10.0.1.1
xtumap02@server:~$ sudo ip route add 0.0.0.0/0 via 10.0.1.254
```

Dále se nastaví šifrátory na branách. Příkazy k nastavení šifrátoru jsou vidět ve výpisu A.2. Skript `install.sh` potřebuje argument sítě, pro kterou má šifrovat.

Výpis A.2: Nastavení QKD simulátoru

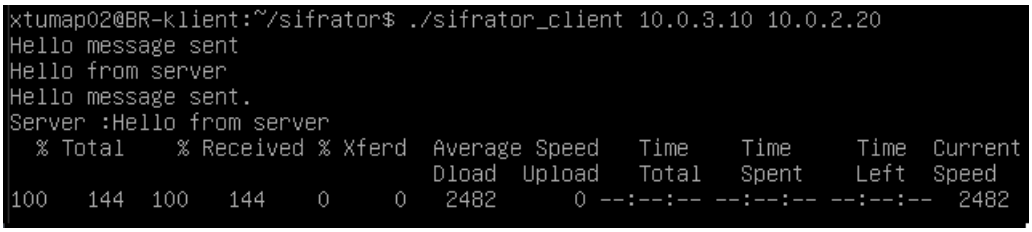
```
xtumap02@BR-server:~$ sudo git clone https://github.com/gabsssq/sifrator.git
xtumap02@BR-server:~$ cd sifrator
xtumap02@BR-server:~$ sudo bash install.sh 10.0.3.0/24
```

Posledním krokem ke zprovoznění je samotné spuštění šifrátorů. Na obrázku A.1 je vidět spuštění pro serverou bránu a na obrázku A.2 pro klientskou bránu. Serverovou část `sifrator_server` je potřeba spustit vždy jako první. Ke své funkci potřebuje argument IP adresy QKD simulátoru (instalovaný v 1. kroku). Klientská část `sifrator_client` potřebuje argumenty IP adresa QKD serveru a IP adresa serverové brány. Výpis po spuštění obou stran šifrátoru znamená úspěšné navázání TCP, UDP spojení a ustanovení QKD klíče.

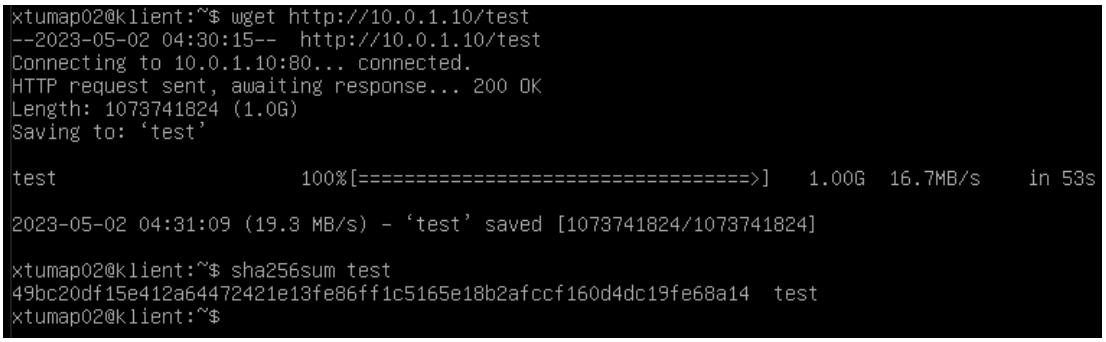
```
xtumap02@BR-server:~/sifrator$ ./sifrator_server 10.0.1.10
Hello from client
Hello message sent
Client : Hello from client
Hello message sent.
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   186  100   144  100    42  48000  14000 --:--:-- --:--:-- --:--:-- 62000
```

Obr. A.1: Serverová část šifrátoru

Stažení testovacího souboru a kontrola integrity je zobrazena na obrázcích A.3 a A.4.



Obr. A.2: Klientská část šifrátoru



Obr. A.3: Stažení testovacího souboru

Obr. A.4: Ověření integrity souboru

B PoC - NFQUEUE

Výpis B.1: Šifrování dat s využitím NFQUEUE

```
#!/usr/bin/python3
from netfilterqueue import NetfilterQueue as nfq
from scapy.all import *
from Cryptodome.Cipher import AES

def encrypt(plaintext, key, mode, asociated_data):
        encobj = AES.new(key, AES.MODE_GCM)
        encobj.update(asociated_data)
        ciphertext,authTag=encobj.encrypt_and_digest(plaintext)
        return(ciphertext,authTag, encobj.nonce)

def listener(packet):
        scapy_packet = IP(packet.get_payload())
        ip_len = scapy_packet[IP].ihl * 4

    #   zabezpeci celou hlavicku krome checksum a ttl proti zmene utocnikem
    #   pouzije se protokol 99 a delka se zvetsi o 33
        ciphertext = encrypt(packet.get_payload()[9].to_bytes(1,"big")+packet.
            ↪ get_payload()[ip_len:], key, AES.MODE_GCM, packet.get_payload()
            ↪ [:2]+(int.from_bytes(packet.get_payload()[2:4],"big")+33).to_bytes
            ↪ (2,"big")+packet.get_payload()[4:8]+(99).to_bytes(1,"big")+packet.
            ↪ get_payload()[12:ip_len])

        res_packet = IP(packet.get_payload()[:ip_len])
        res_packet = res_packet/(ciphertext[2]+ciphertext[1]+ciphertext[0])
        res_packet[IP].proto = "mujPr"
        del res_packet[IP].len
        del res_packet[IP].chksum


        packet.set_payload(bytes(res_packet))
        packet.accept()

key = bytes.fromhex("7092aeb52161089b86c5b5f2824cb529e33764a1294b7ee810
    ↪ b8226fc650e86b")

queue = nfq()
queue.bind(1, listener)
queue.run()
```

Výpis B.2: Dešifrování s využitím NFQUEUE

```
#!/usr/bin/python3
from netfilterqueue import NetfilterQueue as nfq
from scapy.all import *
from Cryptodome.Cipher import AES

def decrypt(ciphertext, key, mode, asociated_data):
        (ciphertext, authTag, nonce) = ciphertext
        encobj = AES.new(key, AES.MODE_GCM, nonce)
        encobj.update(asociated_data)
        return(encobj.decrypt_and_verify(ciphertext, authTag))

def listener(packet):
        try:
                scapy_packet = IP(packet.get_payload())
                ip_len = scapy_packet[IP].ihl*4
                ciphertext = (scapy_packet[Raw].load[32:], scapy_packet[Raw].load
                    ↪ [16:32], scapy_packet[Raw].load[:16])

            #   z hlavicky se do pridanych dat nebere byte 8,10,11 ktere znaci
                ↪ checksum a ttl
                plaintext = decrypt(ciphertext, key, AES.MODE_GCM, packet.
                    ↪ get_payload()[:8]+packet.get_payload()[9:10]+packet.
                    ↪ get_payload()[12:ip_len])
                res_packet = IP(packet.get_payload()[:ip_len])
                res_packet = res_packet/Raw(plaintext[1:])

                res_packet[IP].proto = plaintext[0]

                del res_packet[IP].len
                del res_packet[IP].chksum

                packet.set_payload(bytes(res_packet))
                packet.accept()

        except:
                print ("Kontrola GMAC selhala, zahazuji paket...")
                packet.drop()

key = bytes.fromhex("7092aeb52161089b86c5b5f2824cb529e33764a1294b7ee810
    ↪ b8226fc650e86b")

queue = nfq()
queue.bind(2, listener)
queue.run()
```

C PoC - šifrování dat na virtuálním rozhraní

Výpis C.1: Šifrování dat paketu s využitím virtuálního rozhraní

```
#!/usr/bin/python3
from scapy.all import *
from Cryptodome.Cipher import AES
import fcntl
from multiprocessing import Process

def encrypt(plaintext, key, mode):
        encobj = AES.new(key, AES.MODE_GCM)
        ciphertext,authTag=encobj.encrypt_and_digest(plaintext)
        return(ciphertext,authTag, encobj.nonce)

def decrypt(ciphertext, key, mode):
        (ciphertext, authTag, nonce) = ciphertext
        encobj = AES.new(key, AES.MODE_GCM, nonce)
        return(encobj.decrypt_and_verify(ciphertext, authTag))

def listener(packet):
        scapy_packet = IP(packet)
        ip_len = scapy_packet[IP].ihl * 4

        if (scapy_packet[IP].proto == 99):
                try:
                        ciphertext = (scapy_packet[Raw].load[32:], scapy_packet[Raw
                            ↪ ].load[16:32], scapy_packet[Raw].load[:16])
                        plaintext = decrypt(ciphertext, key, AES.MODE_GCM)
                        res_packet = IP(bytes(scapy_packet)[:ip_len])
                        res_packet = res_packet/Raw(plaintext[5:])
                        res_packet[IP].proto = plaintext[4]
                        res_packet[IP].dst = str(plaintext[0])+"."+str(plaintext
                            ↪ [1])+"."+str(plaintext[2])+"."+str(plaintext[3])

                        del res_packet[IP].len
                        del res_packet[IP].chksum

                        return(bytes(res_packet))

                except:
                        print ("ddddd")

        else:

                ciphertext = encrypt(bytes(scapy_packet)[16:20]+bytes(scapy_packet)
                    ↪ [9:10]+bytes(scapy_packet)[ip_len:], key, AES.MODE_GCM)

                res_packet = IP(bytes(scapy_packet)[:ip_len])
                res_packet = res_packet/(ciphertext[2]+ciphertext[1]+ciphertext[0])
                res_packet[IP].proto = "mujPr"
                res_packet[IP].dst="192.168.1.2"
                del res_packet[IP].len
                del res_packet[IP].chksum
                return bytes(res_packet)

def sifruj():
        while True:
```

```
                packet =  tun.read(2048)
                packet = listener(packet)
                try:
                        tun.write(bytes(packet))
                except:
                        print ("Kontrola GMAC selhala, zahazuji paket...")

def openTun(tunName):
        TUNSETIFF = 0x400454ca
        IFF_TUN = 0x0001
        IFF_NO_PI = 0x1000

        tun = open('/dev/net/tun', 'r+b', buffering=0)
        ifs = struct.pack('16sH22s', tunName, IFF_TUN | IFF_NO_PI, b'')
        fcntl.ioctl(tun, TUNSETIFF, ifs)
        return tun


if __name__ == '__main__':
        key = bytes.fromhex("7092aeb52161089b86c5b5f2824cb529e33764a1294b7ee810
            ↪ b8226fc650e86b")
        tun = openTun(b'tun0')
        p = Process(target=sifruj, args=())
        p.start()
        sifruj()
```

D PoC - šifrování paketu na virtuálním rozhraní

Výpis D.1: Šifrování paketu s využitím virtuálního rozhraní

```
#!/usr/bin/python3
import socket
from scapy.all import *
from Cryptodome.Cipher import AES
import logging as LOGGER
from concurrent.futures.thread import ThreadPoolExecutor
from ipaddress import IPv4Address
import fcntl, time
import os
import struct
import subprocess
from array import array
from multiprocessing import Queue, Process

HOST = "10.0.2.20"
HOST2 = "10.0.2.10"
PORT = 42069

def encrypt(plaintext, key, mode):
  encobj = AES.new(key, AES.MODE_GCM)
  ciphertext,authTag=encobj.encrypt_and_digest(plaintext)
  return(ciphertext,authTag, encobj.nonce)

def decrypt(ciphertext, key, mode):
  (ciphertext, authTag, nonce) = ciphertext
  encobj = AES.new(key, AES.MODE_GCM, nonce)
  return(encobj.decrypt_and_verify(ciphertext, authTag))

def sifrfun(packet):
  ciphertext = encrypt(packet, key, AES.MODE_GCM)
  enc = ciphertext[2] + ciphertext[1] + ciphertext[0]
  s.sendto(enc, (HOST2,PORT))

def desifrfun(packet):
  try:
    ciphertext = (packet[32:], packet[16:32], packet[:16])
    tun.write(bytes(decrypt(ciphertext, key, AES.MODE_GCM)))
  except:
    print("Kontrola selhala")

def sifruj():
  while 1:
    packet = tun.read(1428)
    sifrfun(packet)

def desifruj():
  while 1:
    data, addr = s.recvfrom(1500)
    desifrfun(data)

def openTun(tunName):
  TUNSETIFF = 0x400454ca
```

```
  IFF_TUN = 0x0001
  IFF_NO_PI = 0x1000
  tun = open('/dev/net/tun', 'r+b', buffering=0)
  ifs = struct.pack('16sH22s', tunName, IFF_TUN | IFF_NO_PI, b'')
  fcntl.ioctl(tun, TUNSETIFF, ifs)
  return tun


if __name__ == '__main__':
  key = bytes.fromhex("7092
      ↪ aeb52161089b86c5b5f2824cb529e33764a1294b7ee810b8226fc650e86b")
  tun = openTun(b'tun0')
  s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
  s.bind((HOST,PORT))
  print("Pripojeno")
        p = Process(target=sifruj, args=())
  p.start()
  desifruj()
```