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

ŠIFRÁTOR SÍŤOVÉHO PROVOZU NA PLATFORMĚ OS LINUX

NETWORK TRAFFIC ENCRYPTOR ON LINUX PLATFORM

BAKALÁŘSKÁ PRÁCE

BACHELOR'S THESIS

AUTOR PRÁCE — Jan Havlín

AUTHOR

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

SUPERVISOR

BRNO 2024

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Bakalářská práce

bakalářský studijní program **Informační bezpečnost**

Ústav telekomunikací

Student: Jan Havlín ***ID:*** 241030
Ročník: 3 ***Akademický rok:*** 2023/24

NÁZEV TÉMATU:

Šifrátor síťového provozu na platformě OS Linux

POKYNY PRO VYPRACOVÁNÍ:

Cílem práce je rozšířit stávající implementaci linuxového šifrátoru síťového provozu o nové funkce, zejména pro generování hybridního klíče a zajištění kompatibility s šifrátory na platformě FPGA. Výstupem práce bude funkční implementace ustanovení klíče pomocí ECDH, tvorba hybridního klíče a kompletní implementace šifrátoru včetně výkonových a funkčních testů, dokumentace a veřejného zdrojového kódu.

DOPORUČENÁ LITERATURA:

[1] TŮMA, Petr. Linuxový šifrátor síťového provozu [online]. Brno, 2023 [cit. 2023-09-05]. Dostupné z: http://hdl.handle.net/11012/210107. Diplomová práce. Vysoké učení technické v Brně. Fakulta elektrotechniky a komunikačních technologií. Ústav telekomunikací. Vedoucí práce Jan Hajný.

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

Termín zadání: 5.2.2024 ***Termín odevzdání:*** 28.5.2024

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

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

UPOZORNĚNÍ:

Autor bakalářské práce nesmí při vytváření bakalářské práce porušit autorská práva třetích osob, zejména nesmí zasahovat nedovoleným způsobem do cizích autorských práv osobnostních a musí si být plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č.40/2009 Sb.

ABSTRAKT

Tato bakalářská práce je zaměřena na zajištění šifrované komunikace mezi dvěma stanicemi s operačním systémem Linux. Jedná se o šifrování pomocí šifry AES-GCM-256, která používá hybridní klíč, odvozený od tří separátně vygenerovaných klíčů. Metodami pro tvorbu těchto klíčů jsou Quantum Key Distribution, CRYSTALS-Kyber a Elliptic Curve Diffie-Hellmann. Používáme tedy funkce tří druhů kryptografie - klasické, kvantové i postkvantové. Práce se také zabývá rozšířením funkcionality původního šifrátoru o ověření komunikujících stran pomocí certifikátů a nasazením šifrátoru v praxi.

KLÍČOVÁ SLOVA

ECDH, QKD, Kyber, Linux, C++, Šifrátor, Šifra, Socket, Hash, Certifikát

ABSTRACT

This bachelor thesis focuses on providing encrypted communication between two Linux operating system stations. It involves encryption using the AES-GCM-256 cipher, which uses a hybrid key derived from three separately generated keys. The methods for generating these keys are Quantum Key Distribution, CRYSTALS-Kyber and Elliptic Curve Diffie-Hellmann. Thus, we use functions of three types of cryptography - classical, quantum and post-quantum. The paper also deals with the extension of the functionality of the original cryptography to authenticate the communicating parties using certificates and deployment of the encryptor in practice.

KEYWORDS

ECDH, QKD, Kyber, Linux, C++, Encoder, Cypher, Socket, Hash, Certificate

Vysázeno pomocí balíčku thesis verze 4.09; https://latex.fekt.vut.cz/

HAVLÍN, Jan. Šifrátor síťového provozu na platformě OS Linux [online]. Brno, 2024 [cit. 2024-04-12]. Dostupné z: https://www.vut.cz/studenti/zav-prace/detail/159210. Bakalářská práce. Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav telekomunikací. Vedoucí práce Jan Hajný.

Prohlášení autora o původnosti díla

Jméno a příjmení autora:	Jan Havlín
VUT ID autora:	241030
Typ práce:	Bakalářská práce
Akademický rok:	2023/24
Téma závěrečné práce:	Šifrátor síťového provozu na platformě OS Linux

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 bakalářské práce panu doc. Ing. Janu Hajnému, Ph.D. za odborné vedení, konzultace, trpělivost a podnětné návrhy k práci. Dále bych rád poděkoval Ing. Petru Tůmovi a Ing. Petru Muzikantovi za přínosné rady.

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

Úvod **13**

1 Použité algoritmy **14**
1.1 Kryptografické funkce . . . 14
1.1.1 AES . . . 14
1.1.2 ECDH . . . 14
1.1.3 PQC - Kyber . . . 15
1.2 QKD . . . 15
1.3 Hašovací funkce . . . 15

2 Původní implementace **17**
2.1 Architektura . . . 17
2.2 Kryptografická funkcionalita . . . 17
2.2.1 Šifrování a dešifrování . . . 18
2.2.2 Rekeying . . . 18
2.3 Ovládání a instalace . . . 19

3 Nová implementace **20**
3.1 Cíle . . . 20
3.2 Tvorba hybridního klíče . . . 20
3.2.1 Režim bez QKD . . . 21
3.2.2 Režim s QKD . . . 21
3.3 Vlastní implementace nových funkcí . . . 22
3.3.1 ECDH klíč . . . 22
3.3.2 Hašovací a logické funkce . . . 23
3.3.3 Autentizace pomocí certifikátů . . . 24
3.4 Ostatní úpravy . . . 25
3.4.1 QKD a PQC klíče . . . 26

4 Ověření funkčnosti **27**
4.1 Round Trip Time . . . 27
4.2 Rychlost komunikace . . . 27
4.3 Vyhodnocení . . . 30

5 Praktické nasazení a další kroky **32**
5.1 EEPilot . . . 32
5.1.1 Infrastruktura . . . 32
5.1.2 Problémy a řešení . . . 32

5.1.3 Měření 33
5.2 Další kroky 34

Závěr **36**

Literatura **37**

Seznam symbolů a zkratek **39**

Seznam příloh **41**

A Přílohy **42**
A.1 Generování hybridního klíče v režimu s QKD serverem 42
A.2 Přidání certifikátu CA mezi důvěryhodné certifikační autority 44

Seznam obrázků

1.1 Příklad distribuce klíče pomocí QKD 16
2.1 Topologie zapojení virtuálních strojů [1] 17
2.2 Ukázka startu serverové části šifrátoru v režimu s QKD serverem . . 19
2.3 Ukázka startu klientské části šifrátoru v režimu s QKD serverem . . . 19
3.1 Návrh generátoru hybridního klíče 20
4.1 Odezva příkazu ping 27
4.2 Rychlost stahování pro soubor o velikosti 10 KB 28
4.3 Rychlost stahování pro soubor o velikosti 1 MB 28
4.4 Rychlost stahování pro soubor o velikosti 500 MB 29
4.5 Rychlost stahování pro soubor o velikosti 1 GB 29
4.6 Rychlost stahování pro soubor o velikosti 5 GB 30
5.1 Infrastruktura spojení projektu EEPilot 33
5.2 Výpis z programu iperf3 34

Seznam tabulek

4.1 Výsledky měření 30

Seznam výpisů

3.1 Ukázka ustanovení ECDH klíče 22
3.2 Operace s klíči a parametry 23
3.3 Funkce hmac_hashing 23
3.4 Shell skript client_cert.sh 25
3.5 Instalace knihoven v OS Debian 26
A.1 Generování hybridního klíče v režímmu s QKD serverem 42
A.2 Přidání CA mezi důvěryhodné autority 44
A.3 Výpis po proběhnutí update-ca-certificates 44

Úvod

Potřeba důvěrné komunikace mezi osobami je spojena s lidstvem od počátku prvních organizovaných společenstev. Společně s technologickým pokrokem se zvyšují i nároky na bezpečnost, mezi které patří i ochrana našich dat a jejich přenosu. Zabezpečení pomocí šifer je spolehlivé řešení tohoto problému.

Tato práce se věnuje problematice šifrování dat pro přenos přes nezabezpečený kanál, jakým je například internet. Konkrétní řešení představené v této práci je specifické pro operační systém Linux. Řešení, které je zde představeno, staví na základech, jež ve své diplomové práci navrhl Ing. Petr Tůma [1].

Výsledkem práce je šifrátor, který je schopný ověřit komunikující strany a přenášet a přijímat zašifrované pakety přes rozhraní pomocí protokolu IPv4. Tato komunikace, stejně jako samotný proces šifrování a dešifrování, je realizována pomocí virtuálního rozhraní šifrátorů. Společný klíč, použitý pro komunikaci, se derivuje ze tří samostatných klíčů, na kterých se stanice předem dohodnou. Těmito klíči jsou Eliptic Curve Diffie-Hellman (ECDH) [8] klíč, Quantum Key Distribution (QKD) klíč, vytvářený pomocí dvou QKD zařízení, se kterými šifrátory komunikují, a CRYSTALS-Kyber klíč [3], který zde funguje jako klíč postkvantové kryptografie (PQC). Z těchto tří klíčů se následně pomocí kaskády hašovacích funkcí vytváří hybridní klíč, použitý k samotnému šifrování paketů. Toto šifrování probíhá pomocí symetrické šifry AES-256-GCM [4]. Výsledný klíč celého tvořícího procesu je tedy hybridní klíč o délce 256 bitů.

Všechny funkce pro správné fungování výsledného programu jsou napsané v jazyce C++. Výjimkou jsou skripty, určené pro komunikaci s QKD simulátory a skripty pro generování certifikátů, které jsou napsané jako shell skript a simulátory samotné, které jsou realizované v jazyce PHP.

Jak již bylo řečeno, výsledný produkt nevznikl zcela od základů. Cílem této práce rozšířit a vylepšit již dříve vytvořený šifrátor. Novým přínosem do projektu je tedy přidání ECDH klíče jako třetího klíče v pořadí k již dvěma existujícím a implementace zcela nového systému vytváření hybridního klíče. Vlastnosti šifrátoru byly také rozšířeny o schopnost autentizovat komunikující uživatele pomocí certifikátů. Tato práce se také zabývá nasazením šifrátoru na reálném spoji mezi Českou republikou a Estonskem v rámci projektu CHESS, na kterém spolupracuje Fakulta elektrotechniky a komunikačních technologií Vysokého učení technického v Brně s estonskou firmou Cybernetica.

1 Použité algoritmy

V následující kapitole si představíme algoritmy a šifry, se kterými se můžeme v této práci setkat. V šifrátoru jich je využito hned několik, pro lepší orientaci je zde uvedeno jejich použití: AES je použit pro šifrování samotné komunikace mezi stroji; ECDH, QKD a Kyber jsou algoritmy a principy využité pro získání jednotlivých dílčích klíčů, a pomocí hashovacích funkcí se posléze vypočítá hybridní klíč.

1.1 Kryptografické funkce

1.1.1 AES

AES (zkratka pro „Advanced Encryption Standard“) je symetrická bloková šifra. Využívá algoritmu Rijndael. Zpracovává data po blocích. Standardem je blok o velikosti 128 bitů. AES dokáže využít klíče o třech různých délkách, 128 bitů, 192 bitů a 256 bitů [5]. Algoritmus AES generující klíč o velikosti 256 bitů je dnes považován za velmi bezpečný algoritmus pro šifrování dat. Data šifrovaná klíčem o této délce jsou chráněna i před útoky kvantovými počítači [6].

Algoritmus AES nabízí široké spektrum módů blokových šifer (k roku 2023 jich je celkem 14 [7]). V zásadě se módy (také označované jako režimy) blokových šifer dají rozdělit do tří kategorií. První kategorií jsou režimy zajišťující důvěrnost. Mezi tyto patří ECB, CBC, OFB, CFB, CTR, XTS-AES, FF1 a FF3. Další skupinou jsou režimy zajišťující autentizaci protější strany, tím je režim CMAC. Některé z módů dokáží ale zajistit nejenom důvěrnost, ale i autentizaci zároveň. Těchto módů je celkem pět, jsou jimi CCM, GCM, KW, KWP a TKW. [7] V tomto projektu se využívá blokové šifry AES v režimu GCM („Galois Counter Mode“).

1.1.2 ECDH

Elliptic Curve Diffie-Hellman (ECDH) je algoritmus využívaný pro domluvu společného klíče mezi dvěma klienty, kteří komunikují přes nezabezpečený kanál. Tento algoritmus spojuje výhody kryptografie nad eliptickými křivkami s vlastnostmi protokolu pro ustanovení společného klíče Diffie-Hellman [8]. Pro správné fungování je důležité, aby oba uživatelé využívali stejnou eliptickou křivku. Z této křivky si obě strany získají její parametry. Dále si také vygenerují veřejný a soukromý klíč. Následuje výpočet s využitím vygenerovaných hodnot na jehož konci disponují obě strany stejným klíčem. Tento klíč se poté dá použít pro samotnou šifrovanou komunikaci, nebo k zašifrování dalšího klíče.

Eliptické křivky jsou v kryptografii často používanou algebraickou strukturou. Pro kryptografické účely se používají křivky definované nad množinou prvočísel nebo nad množinou binárních čísel. NIST („National Institute of Standarts and Technology“) definuje konkrétní křivky, které jsou považovány za bezpečné a je možné je volně použít [9]. V tomto projektu využíváme křivku P-521.

1.1.3 PQC - Kyber

Šifrovací algoritmus CRYSTALS - Kyber je algoritmus pro enkapsulaci klíče. Je považován za odolný vůči útoku pomocí kvantového počítače. Princip tohoto algoritmu je založen na učení s chybami („Learning With Errors - LWE“) nad mřížkami. Kyber je dostupný ve třech variantách: Kyber-512, Kyber-768 a Kyber-1024 [11].

1.2 QKD

Kvantová distribuce klíčů je široký pojem, který pod sebe zahrnuje mnoho řešení a protokolů, které jsou odolné proti útokům kvantových počítačů. Je založena na fyzikálních vlastnostech nosičů informace (často jsou jimi fotony). Tím se liší od „nekvantových“ standardů, které si často svou bezpečnost zakládají na matematických problémech s vysokou časovou složitostí, potřebnou k jejich vyřešení [10]. Základní vlastností je zajištění důvěrnosti přenosu šifrovacího klíče přes nezabezpečený kanál. Obecně platí, že QKD nedokáže autentizovat stroj, se kterým komunikuje, k tomuto účelu je potřeba využít jiných řešení.

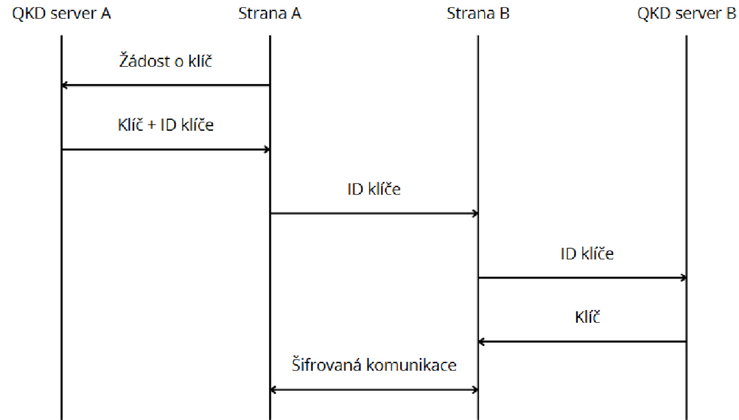
V diplomové práci Ing. Petra Tůmy[1] byl implementován simulátor kvantové distribuce klíčů, kterého tento projekt využívá.

1.3 Hašovací funkce

Hašovací funkce jsou užitečným kryptografickým nástrojem, které se mohou využít hned k několika účelům. Použít je můžeme například k ustanovení a verifikaci digitálních podpisů, derivaci klíčů nebo pseudonáhodné generování bitů [13].

K vypočítání hybridního klíče je v tomto projektu použita hašovací funkce SHA-3. Secure Hash Algorithm je jedním z rodiny algoritmů Keccak. V roce 2012 se KECCAK stal vítězem soutěže NIST pro nový standard hašovacích funkcí. Celý algoritmus je typu houba. Standard FIPS 202 rozlišuje celkem 4 verze SHA-3 v závislosti podle délky výstupu. Jsou jimi: SHA3-224, SHA3-256, SHA3-384 a SHA3-512 [13]. Tento standard umožňuje i použití modifikovaného algoritmu SHA, který doplní

[1]Dostupné z: https://www.vut.cz/studenti/zav-prace/detail/151280



Obr. 1.1: Příklad distribuce klíče pomocí QKD

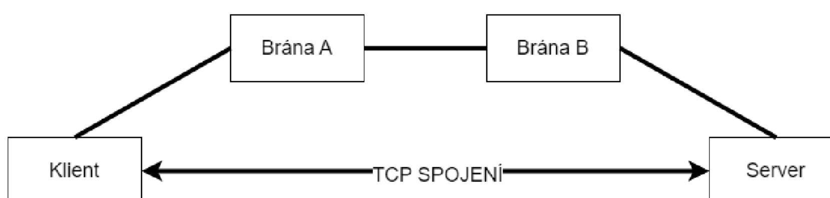
výstup z této funkce na uživatelem definovanou hodnotu. Algoritmy s těmito vlastnostmi se nazývají funkce s rozšiřitelným výstupem, anglicky Extendable-Output function (XOF). Tyto verze se nazývají SHAKE (spojením názvů SHA a KEccak). V tomto projektu se využívají funkce SHA3-256 a SHAKE-256.

2 Původní implementace

V této části si představíme původní implementaci šifrátoru. Tato kapitola také slouží jako základ pro seznámení se s funkcionalitou a strukturou tohoto programu.

2.1 Architektura

Pro snadnější vývoj a testování byl šifrátor a všechny části potřebné k jeho správnému fungování virtualizovány. Software použitý pro virtualizaci byl VirtualBox. Celkem byli vytvořeny čtyři virtuální stroje - 2 brány, klient a server. Na všech těchto strojích byla nainstalována distribuce Debian. Stroje poté byli rozděleni do třech NAT („Network Address Translation“) sítí. Topologii zařízení ilustruje obrázek 2.1.



Obr. 2.1: Topologie zapojení virtuálních strojů [1]

Původním programovacím jazykem byl Python. Ten se ovšem, vzhledem k interpretované povaze tohoto jazyka ukázal jako nevhodný. Záhy byl tedy celý projekt přetransformován do jazyka C++. Tento jazyk je kompilovaný a umožňuje mnohem snadnější práci s jádry. Výsledkem tohoto rozhodnutí tedy byl skokový nárůst výkonu [1]. C++ je zde majoritním programovacím jazykem. V projektu ale najdeme třídy a metody napsané i v jiných jazycích. Objevuje se zde PHP či skripty pro shell. Práci s šiframi v programu zajišťuje knihovna Crypto++.

Pro komunikaci mezi zařízeními se využívá protokolů TCP a UDP. TCP protokolu se využívá pro ustanovení klíče a zajištění spolehlivosti spojení. UDP je poté využíván k samotnému přenášení paketů, kvůli své rychlosti a menší velikosti hlavičky.

2.2 Kryptografická funkcionalita

Počáteční verze šifrátoru vytvářela dva klíče. Prvním byl klíč vypočítaný algoritmem CRYSTALS-Kyber. Tím druhým byl QKD klíč. Ten se získával prostřednictvím QKD simulátorů. Oba klíče měly délku 256 bitů.

Vytvořené klíče se poté použily pro tvorbu hybridního klíče. Hybridní klíč je konstrukce, ve kterém se spojí několik dílčích klíčů. Celý algoritmus je v tomto případě založen na hašovací funkci SHA-256. Oba vygenerované klíče se spojí v jeden řetězec. Tento řetězec je poté zahašován pomocí SHA-256. Výsledný hash se přeloží na byty.

2.2.1 Šifrování a dešifrování

Funkcí zvolenou pro šifrování je bloková šifra AES-256 v režimu GCM („Galois Counter Mode“). Výhodou tohoto přístupu je vysoká rychlost bez snižování bezpečnosti šifrování. Proces šifrování spočívá v šifrování celého paketu, který chceme přenést, na virtuálním rozhraní. Zde se celý paket zašifruje vytvořeným hybridním klíčem a jako payload se vloží do paketu UDP, který je vyslán na naslouchající port příjemce. K datům odesílaného paketu je přiložena ještě hodnota nonce a MAC („Message Authentication Code“), což výrazně navyšuje jeho velikost. Vzhledem k větší velikosti paketů může dojít k překročení hodnoty MTU („Maximum Transmission Unit“), což by vedlo ke zvýšenému počtu zahozených paketů. Komunikace by se tak stala prakticky nefunkční. Řešením je úprava nastavení portů, které zašifrované pakety odesílají a přijímají.

Po přijetí paketu je odebrána část, kde se nachází nonce a MAC. Následně proběhne kontrola integrity paketu. Paket je poté dešifrován celý, tzn. že výsledkem je původní paket.

2.2.2 Rekeying

Podstata rekeyingu spočívá v pravidelné výměně klíče. Zvyšuje se tím bezpečnost komunikace, jelikož snižujeme čas, ve kterých může útočník vypočítat používaný klíč. V této verzi programu se nový klíč vypočítá po 200 000 zpracovaných zprávách (zpracovanou zprávou se rozumí odeslaný nebo přijatý paket). Přes TCP kanál si klient vyžádá od QKD simulátoru nový klíč. Poté proběhne ustanovení QKD klíče. PQC klíč (CRYSTALS-Kyber) zůstává stejný. Tím se rekeying stává výraznou slabinou tohoto programu. V dalších kapitolách jsou popsány změny, které byly provedeny s cílem tuto zranitelnost opravit. Vzhledem k tomu, že program využívá při svém fungování více jader, mezi nimiž neexistuje žádná synchronizace, je nutné před samotnou výměnou klíče zabít všechny potomky rodičovského procesu. Po výměně klíče v rodiči se opět vytvoří potomci. Tím je zajištěno že všechny instance budou pracovat se stejným klíčem a nebude docházet k zahazování paketů z důvodu desynchronizace klíčů.

2.3 Ovládání a instalace

Instalace šifrátoru a jeho ovládání jsou dělané velmi jednoduše. Jsou zde vytvořeny dva skripty, `install.sh` a `install_QKD.sh`. První z uvedených instaluje šifrátor samotný, druhý instaluje simulátory QKD. Skript `install.sh` vyžaduje argument, tím je adresa sítě, pro kterou má šifrovat provoz. Skript `install_QKD.sh` nevyžaduje žádné argumenty. Po instalaci se vytvoří dva spustitelné soubory, `encryptor_client` a `encryptor_server`. Spustíme je příkazem .\<jméno skriptu>. Strana serveru musí být spuštěna vždy jako první. V závislosti na spouštěném skriptu musíme zadat argumenty. Pokud spouštíme `encryptor_server` argumentem je adresa serveru QKD (v nové verzi je tento argument nepovinný). Pro `encryptor_clienta` je prvním argumentem adresa brány serveru (povinný) a druhým adresa QKD serveru (v nové verzi nepovinný).

```
root@brana2:/home/debian/Linux-network-traffic-encryptor# ./encryptor_server 10.0.1.8
```

Obr. 2.2: Ukázka startu serverové části šifrátoru v režimu s QKD serverem

```
root@brana1:/home/debian/Linux-network-traffic-encryptor# ./encryptor_client 10.0.2.7 10.0.3.6
```

Obr. 2.3: Ukázka startu klientské části šifrátoru v režimu s QKD serverem

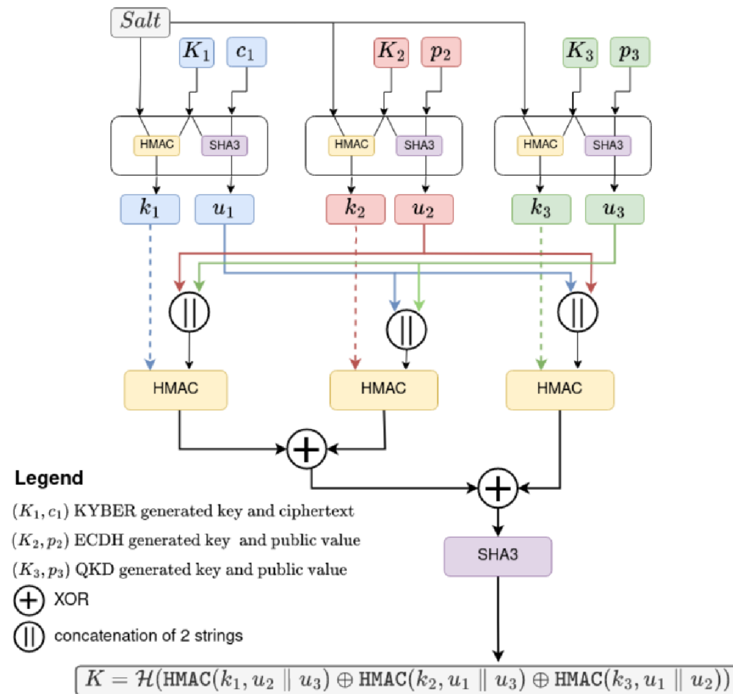
3 Nová implementace

V této kapitole si představíme úpravy provedené v projektu v rámci této práce. Podkapitoly se budou soustředit pouze na změněné části.

3.1 Cíle

Přestože šifrátor ve stavu popsaném v minulé kapitole pracuje bezchybně, můžeme si všimnout, že má pár slabých stránek. Cílem tohoto projektu bylo rozšířit program o další funkcionalitu. Primární zaměření bylo na zvýšení bezpečnosti a větší variabilitu využití. Většina nově použitých funkcí a algoritmů využívá již implementovanou knihovnu `Crypto++`, ovšem kvůli přidání funkcionality autentizace pomocí certifikátů nyní program vyžaduje i nové knihovny `libssl-dev` a `ca-certificates`.

3.2 Tvorba hybridního klíče



Obr. 3.1: Návrh generátoru hybridního klíče

Kompletním přepracováním prošla funkce `rekey`. Jejím úkolem je tvorba hybridního klíče. Hlavní změnou je implementace zcela nového návrhu tvorby hybridního klíče. Dále funkce nabízí dva režimy provozu, první pro situaci, kdy je dostupný

QKD server (počet dílčích klíčů vstupujících do funkce je 3) a druhý je pro situaci bez dostupných QKD serverů (hybridní klíč je tvořen ze 2 dílčích klíčů). Návrh implementovaného generátoru vznikl na Fakultě elektrotechniky a komunikačních technologií Vysokého učení technického v Brně. V této funkci je také definována použitá kryptografická sůl. Hybridní klíč se vytváří na začátku každého nového spojení a také po uplynutí časového intervalu 10 minut.

Bez ohledu na to, v jakém režimu funkce běží, proběhne jako první ustanovení PQC klíče, následované ustanovením ECDH klíče. Tyto klíče se použijí v obou režimech, proto je nutné je ustanovit na začátku. Předcházíme tím tak zbytečné duplicitě kódu. Funkce následně zkontroluje počet argumentů vložených uživatelem při spouštění skriptu. Pokud chce uživatel používat šifrátor, aniž by byl dostupný QKD server, jednoduše tak učiní tím, že při spouštění skriptu nezadá parametr vyčleněný pro adresu serveru QKD. Naopak pokud je tato adresa při spuštění zadána, program automaticky počítá s tím že ji chce uživatel použít. Střídání režimů je tedy možné jen při spuštění/restartu šifrátoru.

3.2.1 Režim bez QKD

Tento režim vyžaduje 5 vstupních argumentů: Sůl, vytvořenou při zavolání funkce `rekey()`, klíč ECDH, klíč PQC, souřadnice bodů X a Y (vypočítané při tvorbě ECDH klíče) a šifrový text (vzniklý při tvorbě PQC klíče). Poslední dva argumenty budeme pro jednoduchost označovat parametry. Klíče jsou ve formě textových řetězců vloženy jako vstupní argumenty funkce, parametry jsou vyčteny z globálních proměnných. Prováděné operace jsou ve výpisu 3.2.

3.2.2 Režim s QKD

Pokud je přítomen QKD server, postup je velice podobný, generátor je ale rozšířen o dvě nové proměnné: QKD klíč a parametr složený z ID QKD klíče a „QKD logtail“. Funkci generující hybridní klíč můžeme nalézt v příloze A.1. Hybridní klíč vygenerovaný kaskádou šifrovacích a logických funkcí je následně zahašován. Klíč je ovšem generován delší, než je potřebné pro funkci AES-256, zajišťující šifrování paketů. Řešením je vyjmutí prvních 32 bytů (256 bitů), které poslouží jako vstup pro šifrovací funkci.

3.3 Vlastní implementace nových funkcí

3.3.1 ECDH klíč

V rámci zajištění vyšší bezpečnosti byl oproti původní verzi šifrátoru přidán další, v pořadí již třetí dílčí klíč. Tento klíč se získává pomocí algoritmu ECDH. Křivkou používanou pro výpočet společného klíče je „NIST P-521“. Pro ustanovení klíče byla vytvořena samostatná funkce `PerformECDHKeyExchange`, která se nachází v obou šifrátorech. V této funkci si nejprve vygenerujeme křivku. V návaznosti na atributy získané z křivky si každá stanice vygeneruje vlastní soukromý a veřejný klíč. Následuje výměna potřebných informací. Tu iniciuje strana klienta (strana A). Spojení a výměna informací probíhá pomocí protokolu TCP. Z přijatých dat se následně vypočítá společný klíč. Funkce vrací společný klíč ve formě textového řetězce. Dále funkce ukládá hodnoty bodů X a Y na eliptické křivce, které posloužili jako zdroj pro výpočty. Uložené souřadnice se později využívají při výpočtu hybridního klíče.

Výpis 3.1: Ukázka ustanovení ECDH klíče

```
string PerformECDHKeyExchange(int client_fd){
    CryptoPP::AutoSeededRandomPool rng;
    // Set up the NIST P-521 curve domain
    CryptoPP::ECDH<CryptoPP::ECP>::Domain dh(CryptoPP::ASN1::
        secp521r1());
    // Generate ECDH keys
    CryptoPP::SecByteBlock privateKey(dh.PrivateKeyLength());
    CryptoPP::SecByteBlock publicKey(dh.PublicKeyLength());
    dh.GenerateKeyPair(rng, privateKey, publicKey);
    listen(client_fd, 3);
    // Send public key to the server
    send(client_fd, publicKey.BytePtr(), publicKey.SizeInBytes()
        , 0);
    // Receive the server's public key
    CryptoPP::SecByteBlock receivedKey(dh.PublicKeyLength());
    read(client_fd, receivedKey.BytePtr(), receivedKey.
        SizeInBytes());
    // Derive shared secret
    CryptoPP::SecByteBlock sharedSecret(dh.AgreedValueLength());
    dh.Agree(sharedSecret, privateKey, receivedKey);
```

Tento výpis ukazuje klíčovou část funkce. Můžeme zde vidět celý průběh tvorby soukromého a veřejného klíče, jejich vzájemnou výměnu a derivaci společného tajemství. Pro prezentační účely byl zkrácen o servisní výpisy, které se uživateli ukazují při reálném spuštění programu.

Výpis 3.2: Operace s klíči a parametry

```
    string key_one = hmac_hashing(salt, pqc_key);
    string key_two = hmac_hashing(salt, ecdh_key);
    string param_one = sha3_hashing(pqc_key, &
       kyber_cipher_data_str);
    string param_two = sha3_hashing(ecdh_key, &xy_str);
    string second_round_param_one = param_one + param_two;
    string second_round_key_one = hmac_hashing(key_one,
       second_round_param_one);
    cout << "Second round key one: " << second_round_key_one <<
       endl;
    string second_round_key_two = hmac_hashing(key_two,
       second_round_param_one);
    cout << "Second round key two: " << second_round_key_two <<
       endl;
    string key = xorStrings(second_round_key_one,
       second_round_key_two);
    cout << "Key: " << key << endl;
```

Následně je výsledná hodnota zahašována a vzniklý hybridní klíč je převeden na byty.

3.3.2 Hašovací a logické funkce

Pokud potřebujeme zajistit integritu informací, můžeme využít například nějaké tajemství. Pokud toto tajemství spojíme s hašovací funkcí, vznikne HMAC. HMAC nebo také „Hash-Based Message Authentication Code“ je taková hašovací funkce, která dokáže ověřit integritu, a někdy i autentičnost posílané zprávy. K vytvoření HMAC funkce můžeme využít libovolnou bezpečnou hašovací funkci [12]. V tomto projektu je použita SHA3-256.

Nová funkce `hmac_hash` má za úkol vytvořit HMAC ze dvou vložených hodnot. Jako klíč nám zde slouží kryptografická sůl, jako „náklad“ pro hašovací funkci se využívají hodnoty vložené funkcí pro výpočet hybridního klíče. Funkce obsahuje i blok, který má za úkol upravit délku vložených hodnot na požadovanou délku (v tomto případě 256 bitů). Jako výsledek vrací funkce nový hash ve formě textového řetězce.

Výpis 3.3: Funkce hmac_hashing

```
string hmac_hashing(string salt, string key){
    const size_t desired_length = 216;
    string padded_key(salt);
```

```
    string padded_message(key);
    // Pad the key and message with zeros if needed
    if (padded_key.size() < desired_length){
        padded_key.resize(desired_length, '\0');
    }
    if (padded_message.size() < desired_length){
        padded_message.resize(desired_length, '\0');
    }
    CryptoPP::HMAC <CryptoPP::SHA3_256> hmac((const byte *)
       padded_key.data(), padded_key.size());
    string result;
    CryptoPP::StringSource(padded_message, true, new CryptoPP::
       HashFilter(hmac, new CryptoPP::HexEncoder(new CryptoPP::
       StringSink(result))));
    return result;
}
```

V programu je vytvořena i další funkce, která využívá hašovací funkce SHA3-256. Stejně jako předchozí funkce přijímá dva argumenty a vrací jeden textový řetězec. V tomto případě se ovšem vložené argumenty (kterými jsou textové řetězce) spojí do jednoho. Teprve následně se zahashují.

Návrh tvorby hybridního klíče používá při svém výpočtu logickou funkci XOR.

V rámci implementace byla vytvořena samostatná funkce zaměřená na tuto akci. Jako vstup přijímá dva textové řetězce. Je důležité zajistit, že řetězce budou mít stejnou délku. Jazyk C++ zvládá nativně operaci XOR pomocí operátoru „ˆ“. Funkce obsahuje cyklus, v jehož rámci se vložené řetězce znak po znaku spojují. Výsledkem je další textový řetězec.

3.3.3 Autentizace pomocí certifikátů

Pro účely ověření identity komunikujících stran byla přidána funkce, která zajišťuje vzájemnou výměnu a ověření certifikátů pojmenována `cert_authenticate()`. Kromě implementace kódu bylo přirozeně nutné vytvořit i samotné certifikáty pro strany klienta a serveru. V rámci zjednodušení testování byla vytvořena vlastní certifikační autorita, která si sama vytvořila a podepsala certifikát certifikační autority. Klíče certifikační autority, jakožto i vygenerované certifikáty mají obě strany uloženy lokálně.

Aby byl proces tvorby (či výměny) certifikátů co uživatelsky nejpříjemnější, byly vytvořeny dva shell skripty, které certifikáty vygenerují (client_cert.sh a server_cert.sh). Klíče jsou vygenerovány algoritmem RSA a zašifrovány algoritmem AES256.

Funkce umí pracovat ve dvou režimech, testovacím a běžném. Mezi těmito režimy můžeme přepínat při zapnutí šifrátoru pomocí argumentu `-t`. Je-li tento argument přítomen, funkce se spustí v testovacím režimu. V tomto režimu je na straně serveru otevřen port, na kterém server naslouchá. Na straně klienta funkce přijímá jako argument IP adresu serveru. Vzájemná výměna certifikátů je realizována pomocí TCP spojení, které se po přijmutí a ověření certifikátu opět uzavře. Vytvoření portů, výměna certifikátů a jejich ověření se provádí pomocí funkcí z knihovny `libssl-dev`. Tento způsob výměny certifikátů není ovšem není považován za bezpečný, protože je náchylný k Man-in-the-Middle útokům. Pro běžné, neexperimentální využití je proto určen druhý režim, k jehož fungování je vyžadováno, aby byli certifikáty (protější strany i certifikační autority) vyměněny libovolným bezpečným kanálem (osobně, přes zabezpečené úložiště apod.), a nahrány do složky šifrátoru. Tento režim je zvolen ve chvíli, kdy argument `-t` není použit. Funkce se v tomto režimu snaží nahrát certifikát z lokálního úložiště. Její funkcionalita je poté podobná s tím rozdílem, že jsou zvoleny funkce pro lokální autentizaci. Opět jsou využity funkce knihovny `OpenSSL`. Tento režim je také doporučen pro použití v nasazení mimo laboratoř.

V rámci vývoje také vznikla verze programu, která pracuje s PQC certifikáty. Při experimentování byly vyzkoušeny algoritmy Dilithium3, Dilithium5 a Kyber512 [1]. Pro ověřování certifikátů je využito rozšíření `OpenSSL` s názvem `OQSProvider`[2] a knihovna `liboqs`[3]. Momentálně práce v tomto směru narazila na problém s důvěryhodností vytvořené certifikační autority. Existují potenciální řešení, která jsou v plánu implementovat a otestovat, práce tedy stále probíhají. Rozšíření funkcionality tímto směrem má ovšem vysoký potenciál do budoucnosti.

Výpis 3.4: Shell skript client_cert.sh

```
rm -rf client.crt client.csr client.key
echo "Removed old client certificates"
openssl genpkey -algorithm RSA -out client.key -aes256
openssl req -new -key client.key -out client.csr
openssl x509 -req -in client.csr -CA ca.crt -CAkey ca.key -
    CAcreateserial -out client.crt -days 365
```

3.4 Ostatní úpravy

V průběhu vývoje došlo i na menší úpravy v různých částech programu. Výměna klíče nově neprobíhá po odeslání/přijetí určitého počtu zpráv, ale periodicky, jednou

[1] Dostupné z: https://pq-crystals.org/dilithium/

[2] Dostupné z: https://github.com/open-quantum-safe/oqs-provider

[3] Dostupné z: https://github.com/open-quantum-safe/liboqs

za hodinu. Úprav doznala funkce `main()`, ve které bylo upraveno pořadí volání různých funkcí. Také bylo upraveno chování portů. Funkce nově, ve spojitosti s novou funkcionalitou funkce `rekey()`, nenutí uživatele zadat všechny potřebné argumenty.

Z důvodů rozšíření šifrátoru bylo nutné upravit požadavky pro běh a instalaci šifrátoru. Pro správné fungování šifrátoru je doporučeno používat účet `root` či příkaz `sudo` při zadávání příkazů. Nově je nutné nainstalovat dvě další knihovny, které využívá funkce `cert_authenticate()`. První z nich je libssl-dev. Jedná se vývojářskou verzi knihovny OpenSSL. Druhou knihovnou je ca-certificates, jež se stará o správu certifikátů a certifikačních autorit. Poté, co máme k dispozici certifikát certifikační autority (který byl předán bezpečnou cestou, např. při osobní schůzce), je nutné jej nahrát do úložiště certifikátů. Návod jak uložit vlastní certifikační autoritu mezi důvěryhodné autority obsahuje příloha A.2.

Výpis 3.5: Instalace knihoven v OS Debian

```
apt-get install libssl-dev
apt-get install ca-certificates
```

3.4.1 QKD a PQC klíče

Funkce pro vytváření klíčů postkvanotvé a kvantové kryptografie obsahují jen pár drobných změn. Nejzřetelnější změnou je rozšíření funkcionality o ukládání parametrů, použitých při tvorbě klíče, do nových proměnných. U algoritmu pro získání QKD klíče program nově ukládá ID klíče doplněný o SHAKE-128 tohoto ID. U PQC klíče je ukládán šifrový text. Z obou těchto hodnot je uloženo vždy 216 bytů (1728 bitů).

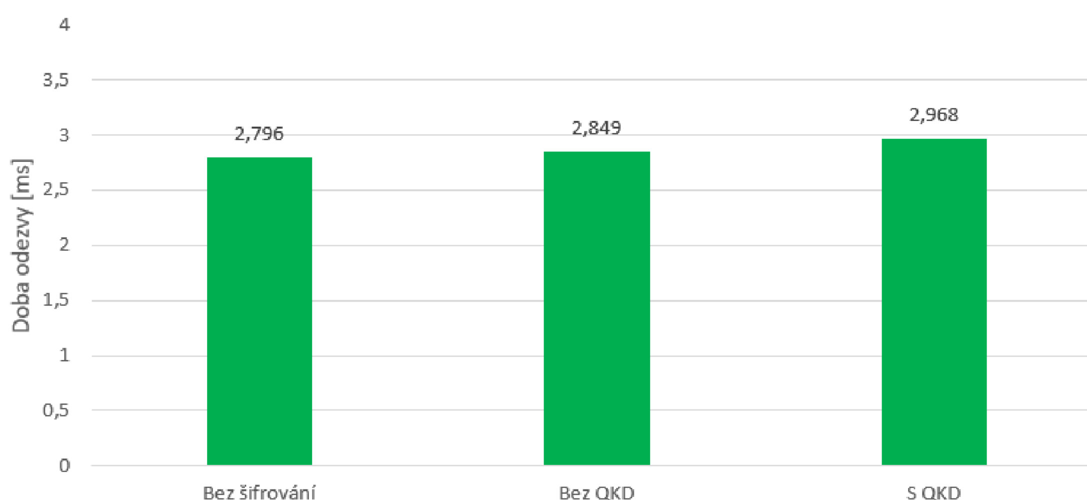
4 Ověření funkčnosti

Pro ověření funkčnosti a výkonnosti byla v laboratoři provedena série měření. První test měřil RTT („Round Trip Time“). Druhý test měřil dobu stažení souborů o různé velikosti. Soubory byly velké 10 KB, 1 MB, 500 MB, 1 GB a 5 GB. V testech jsou porovnávány tři varianty provozu: bez šifrování, rekeying s QKD serverem a rekeying bez QKD serveru.

Pokud by vznikla potřeba ověřit funkčnost bez měření, můžeme tak učinit pomocí porovnání. Pokud oba šifrátory naváží spojení, zobrazí se uživateli o tomto spojení informace. Mezi nimi jsou jednotlivé hodnoty dílčích klíčů, parametrů a i konečný hybridní klíč. Prostým porovnáním těchto výpisů můžeme zjistit, zda zařízení disponují stejným klíčem a spoj je tak funkční.

4.1 Round Trip Time

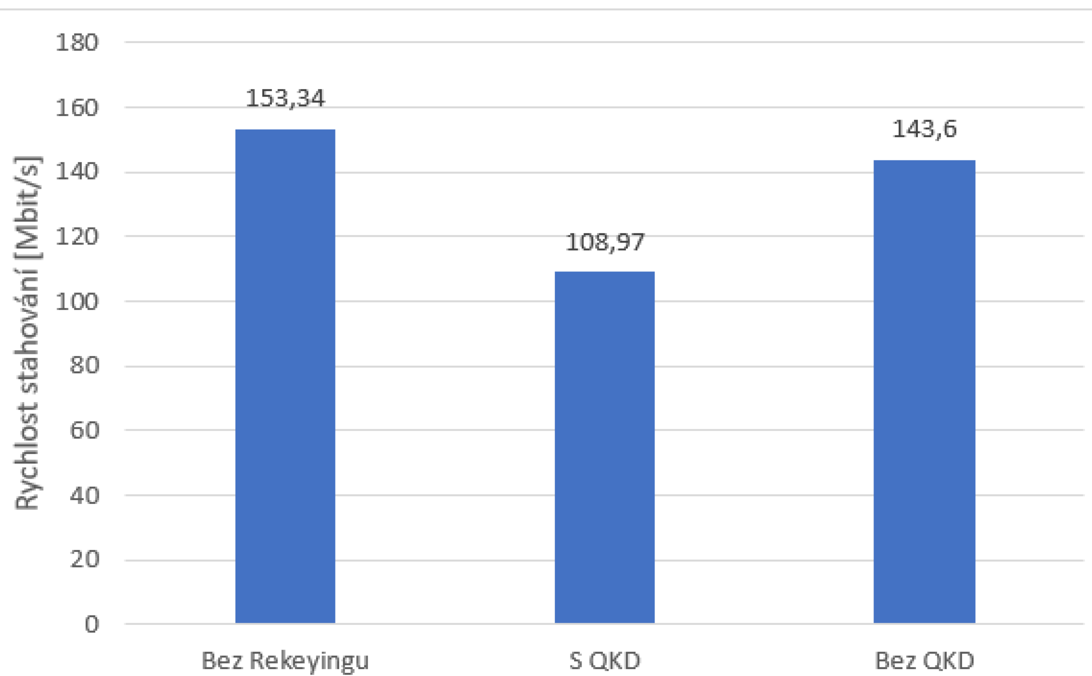
K měření RTT byl použit příkaz `ping`. Jako výsledek se zaznamenává průměrný čas po 20 dotazech `ping`. Naměřené výsledky jsou k dispozici v grafu na obrázku 4.1.



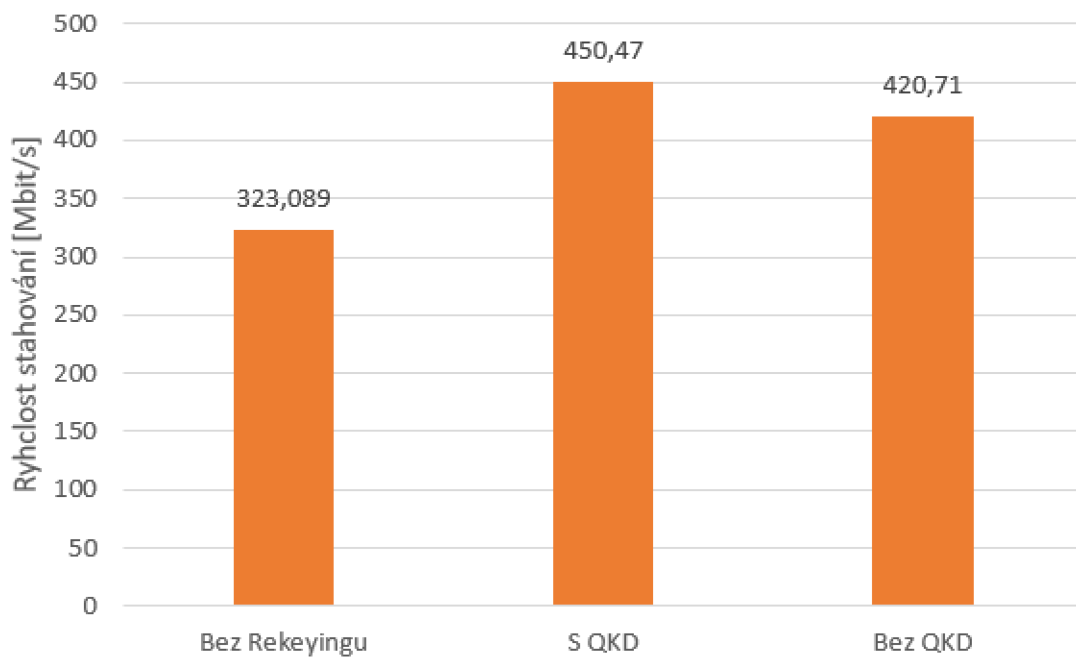
Obr. 4.1: Odezva příkazu ping

4.2 Rychlost komunikace

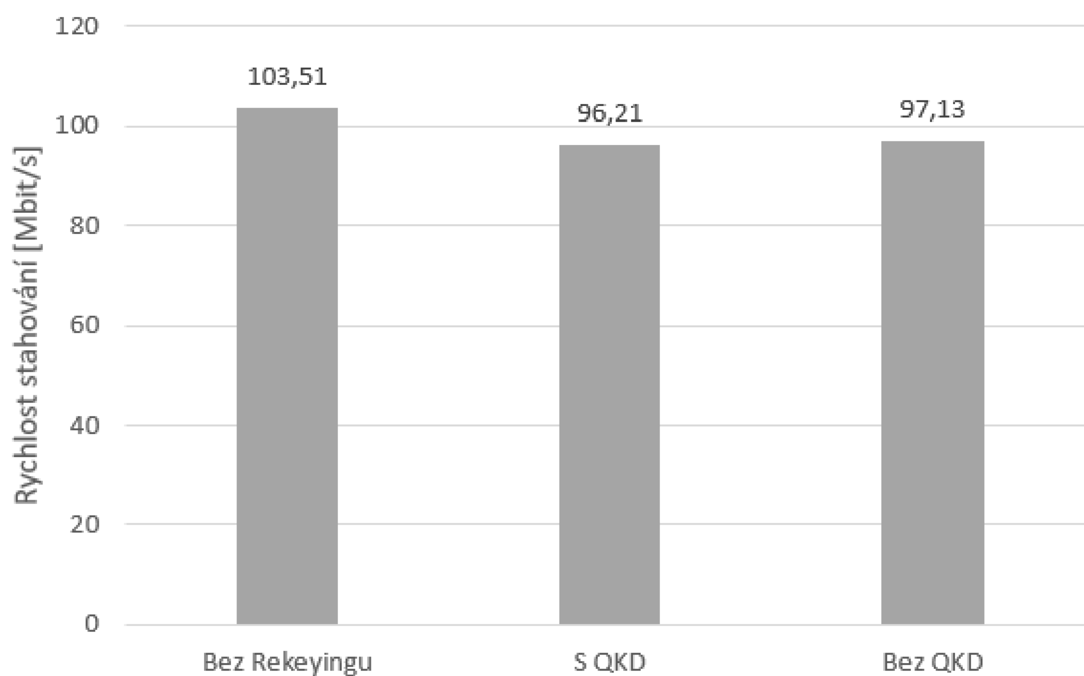
Testování rychlosti komunikace probíhalo v každém režimu (bez šifrování, s QKD, bez QKD) formou 10 opakování, po kterých byla vypočítána průměrná rychlost stahování. Přenos souboru byl uskutečněn programem `wget`. Na obrázcích 4.2, 4.3, 4.4, 4.5, 4.6 můžeme sledovat výsledky měření.



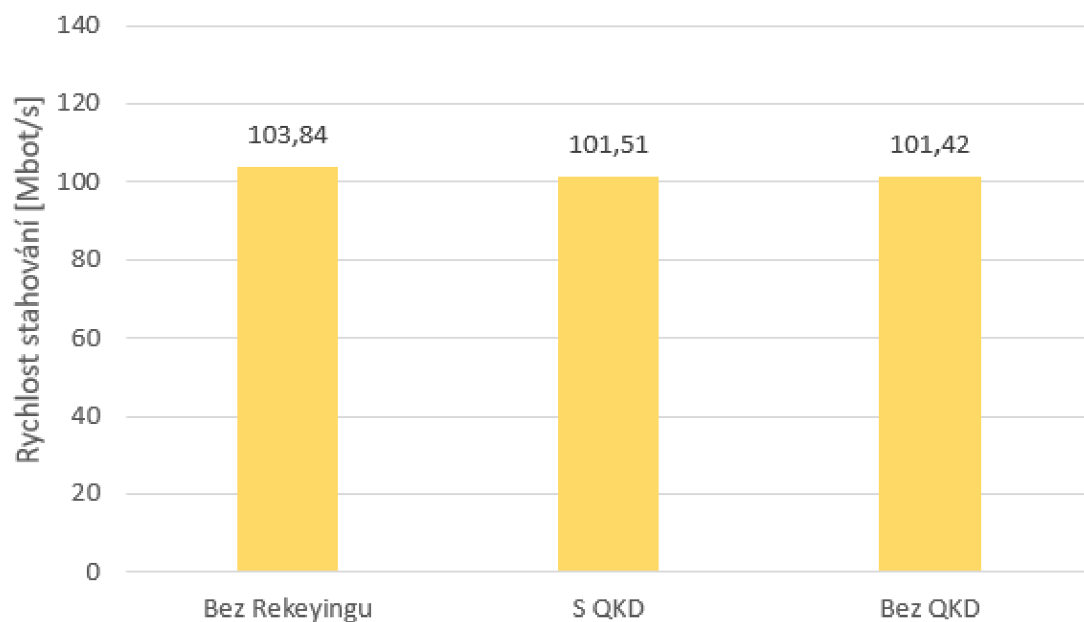
Obr. 4.2: Rychlost stahování pro soubor o velikosti 10 KB



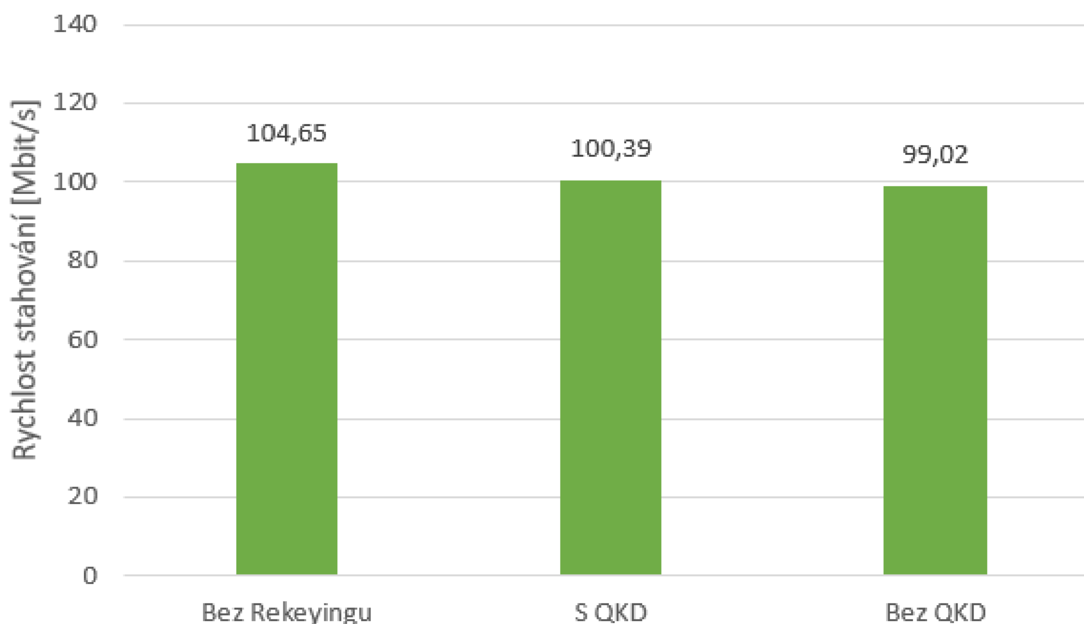
Obr. 4.3: Rychlost stahování pro soubor o velikosti 1 MB



Obr. 4.4: Rychlost stahování pro soubor o velikosti 500 MB



Obr. 4.5: Rychlost stahování pro soubor o velikosti 1 GB



Obr. 4.6: Rychlost stahování pro soubor o velikosti 5 GB

4.3 Vyhodnocení

Z měření vyplývá, že nejnižší odezvu, i nejvyšší rychlost přenosu souborů zaznamenáme při přenosu bez šifrování. Při porovnání rychlosti přenosu v režimech s QKD a bez QKD nelze jednoznačně určit, který z nich je rychlejší. V tabulce jsou vidět všechny výsledky měření.

Tab. 4.1: Výsledky měření

Režim šifrátoru	RTT [ms]	10 KB [Mbit/s]	1 MB [Mbit/s]	500 MB [Mbit/s]	1 GB [Mbit/s]	5 GB [Mbit/s]
Bez šifrování	2,796	153,3	323,1	103,5	103,8	104,7
S QKD	2,849	109	450,5	96,2	101,5	100,4
Bez QKD	2,968	143,6	420,7	97,1	101,4	99

Z výsledků lze vidět, že u menších souborů jsou rozdíly mezi jednotlivými režimy větší. S přibývající velikostí souboru jsou ale rozdíly čím dál méně znatelné, až téměř vymizí. Tento jev může být ovlivněn sníženou schopností příkazu `wget` měřit rychlost stahování pro soubory menší jak 500 MB. Tento nedostatek je vidět zejména při testu souboru o velikosti 1 MB.

V průběhu testování souborů o velikosti 1 GB a 5 GB proběhlo vždy jedna výměna klíče, jelikož délka jednotlivých testů přesáhla deset minut, ve výsledcích testování se však tato akce nijak výrazně neprojevila.

V porovnání s výsledky měření původní verze[1] můžeme sledovat, že šifrátor je

[1]Dostupné z: `https://www.vut.cz/studenti/zav-prace/detail/151280`

pomalejší v přenosu souborů. Zde je možné sledovat zpomalení o zhruba 36 Mbit/s při přenosu velkých souborů. K tomuto jevu může přispívat i rozdílný testovací hardware, na kterém testování probíhalo. V tomto případě se jednalo o zařízení s procesorem Intel i7-12700H.

V měření doby odezvy se žádné zdržení oproti původní verzi neprojevilo. Naopak, lze konstatovat drobné zkrácení doby odezvy při šifrování v režimu s QKD serverem.

5 Praktické nasazení a další kroky

Algoritmus šifrátoru je implementován ve dvou projektech probíhajících na Fakultě elektrotechniky a komunikačních technologií Vysokého učení technického. Pro tento účel musely být provedeny v šifrátoru drobné úpravy „na míru“ podle potřeb konkrétních případů použití.

5.1 EEPilot

Projekt s názvem EEPilot probíhá ve spolupráci s estonskou soukromou společností Cybernetica. Klade si za cíl nasadit a ověřit funkčnost celého systému v prostředí mimo laboratorní podmínky a vytvořit kvantově odolný spoj.

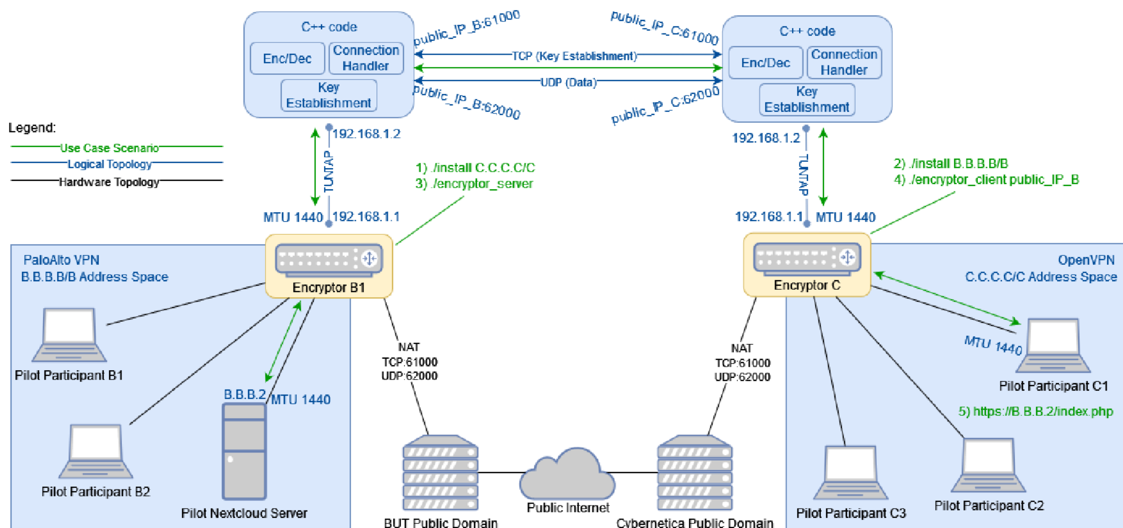
5.1.1 Infrastruktura

Na obou stranách plánovaného spojení byl šifrátor nasazen na virtuální stroje sloužící jako výchozí brána. Za touto branou byli umístěny zařízení sloužící pro komunikaci přes zabezpečený spoj. Na straně VUT jím byli server, na němž běžela cloudová služba NextCloud Talk pro uskutečňování hlasové, ale i video komunikace. Dále jsou v této síti umístěny počítače jednotlivých potenciálních uživatelů na straně VUT. Přístup do této sítě je řešen přes VPN (Virtual Private Network) od firmy PaloAlto. Na druhé straně spoje byli do sítě za výchozí bránou připojeni uživatelé spoje z firmy Cybernetica. Ti se do této sítě připojovali skrze službu OpenVPN. Pro přístup virtuálních strojů na internet byla upravena příslušná politika NAT a povoleny potřebné porty (v našem případě 61000 a 62000). Na obrázku 5.1 je detailně znázorněna celá infrastruktura v době nasazení. [1]

5.1.2 Problémy a řešení

V průběhu nasazení se vyskytlo několik problémů. Některé byli předpokládané a potřebné úpravy byli snadné a plánované. Mezi takové patřila například úprava maximální velikosti rámce (MTU), která musela být upravena na hodnotu 1440 z původních 1500, aby nedocházelo k zahazování paketů na některých síťových prvcích po cestě. Po naklonování a instalaci šifrátoru byli nastaveny patřičné hodnoty pro statické routování a otestována konektivita pomocí příkazu `ping`. První spuštění šifrátoru ovšem odhalilo další problém. Po úspěšném ustanovení klíčů nedocházelo k úspěšnému přijímání další komunikace na strojích Cybernetiky. Pomocí programu

[1] Autorem obrázku 5.1 je Ing. Petr Muzikant



Obr. 5.1: Infrastruktura spojení projektu EEPilot

`tcpdump` a `WireShark` bylo odhaleno, že šifrované pakety (které jsou posílány protokolem UDP) přicházejí na virtuální stroj Cyberneticy v přeházeném pořadí a ta není schopná je jakkoliv poskládat do pořadí správného. Tím docházelo k zahlcení kanálu, zahazování paketů a důsledkem tedy i nefunkčnosti spoje. Toto chování mohlo být způsobeno dvěma aspekty. První možností byla chyba přenosu UDP paketů někde v síti mezi šifrátory. Taková chyba by se těžko opravovala, jelikož nemáme žádnou kontrolu nad síťovými prvky na cestě. Druhou verzí bylo pomíchání paketů z důvodu vícevláknového zpracování. Myšlenkou bylo, že tato situace je způsobena rozdílnou rychlostí, kterou jednotlivá vlákna šifrují. Vzhledem k tomu, že doposud nebyl nebyl implementován žádný způsob kontroly pořadí, každé vlákno fungovalo jednoduchým způsobem„vezmi, zašifruj, pošli“. Tato skutečnost se ukázala jako pravděpodobná příčina. Řešením bylo přidání funkcionality číslování dat při přijímání na virtuální rozhraní a kontrola jejich správného pořadí před odesláním do internetu.

Dalším problémem, který si vyžadoval okamžité řešení, byla příliš vysoká latence při připojování na NextCloud Talk server přes zašifrované spojení ze sítě firmy Cybernetica. Ačkoliv `ping` žádnou velkou latenci nezobrazoval, načtení úvodní stránky cloudové služby se nikdy nepodařilo, jelikož byl vždy překročen maximální čas pro ustanovení spojení. K vyřešení tohoto problému satčilo změnit MTU na hodnotu 1440 i na rozhraní NextCloud serveru a ne pouze na portech šifrátorů.

5.1.3 Měření

V rámci testování spoje byla provedena série měření se zaměřením na síťovou propustnost, latenci a také na rychlost ustanovení společného hybridního klíče. Bylo

také provedeno několik hlasových hovorů a video hovorů, jejichž uskutečnění bylo jedním z hlavních cílů, kterého jsme chtěli s touto implementací dosáhnout. Výsledkem byl čistý a nepřerušovaný hovor bez zřetelných kazů jak v přenosu hlasu, tak přenosu videa. Hovor probíhal mezi 2 klienty. Hovor s více klienty je plánován v blízké budoucnosti.

Pro měření síťové propustnosti byl použit program `iperf3` [2]. Měření probíhalo paralelně s videohovorem. Průměrná naměřená hodnota byla 121 Mbit/s.[3] Pro měření latence byl využit příkaz `ping`, z jehož výstupu vyplývá průměrná doba RTT (Round Trip Time) 48.3 ms se standardní odchylkou 0,6 ms. Rychlost ustanovení hybridního klíče se měřila pomocí proměnných, ze kterých se odečítal čas, přidaných do kódu speciálně pro tyto účely. Šifrátor byl v době měření spouštěn v módu bez QKD. Doba ustanovení hybridního klíče se mezi stroji lišila, pravděpodobnou příčinou je rozdílný hardware na jednotlivých strojích. Průměrná doba ustanovení se pohybovala okolo 213,1 ms na stroji VUT, se standardní odchylkou 5,4 ms. Na stroji Cyberneticy ustanovení trvalo 163,2 ms se standardní odchylkou 2,1 ms.

```
iperf3 -B 10.8.0.3 --set-mss 1440 -c 172.25.37.104 -p 5201
Connecting to host 172.25.37.104, port 5201
[  5] local 10.8.0.3 port 34047 connected to 172.25.37.104 port 5201
[ ID] Interval           Transfer     Bitrate         Retr  Cwnd
[  5]   0.00-1.00   sec  14.9 MBytes   125 Mbits/sec  2024    679 KBytes
[  5]   1.00-2.00   sec  13.8 MBytes   115 Mbits/sec    4    660 KBytes
[  5]   2.00-3.00   sec  13.8 MBytes   115 Mbits/sec    0    699 KBytes
[  5]   3.00-4.00   sec  13.8 MBytes   115 Mbits/sec    0    727 KBytes
[  5]   4 00-5.00   sec  15.0 MBytes   126 Mbits/sec    0    740 KBytes
[  5]   5.00-6.00   sec  15.0 MBytes   126 Mbits/sec    0    746 KBytes
[  5]   6 00-7.00   sec  13.8 MBytes   115 Mbits/sec    0    747 KBytes
[  5]   7.00-8.00   sec  15.0 MBytes   126 Mbits/sec    0    747 KBytes
[  5]   8.00-9.00   sec  15.0 MBytes   126 Mbits/sec    0    748 KBytes
^C[  5]  10.00-22.23  sec  0.00 Bytes  0.00 bits/sec    5   1.36 KBytes
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bitrate         Retr
[  5]   0.00-22.23  sec   140 MBytes  52.8 Mbits/sec  2033             sender
[  5]   0.00-22.23  sec  0.00 Bytes  0.00 bits/sec                  receiver
iperf3: interrupt - the client has terminated
```

Obr. 5.2: Výpis z programu iperf3

5.2 Další kroky

V rámci projektu EEPilot je v plánu uskutečnění videohovoru, kterého by se účastnil větší počet účastníků. Cílem je otestovat šifrátor pod větší zátěží a demonstrovat

[2] Dostupné z: `https://iperf.fr/`

[3] Autorem obrázku 5.2 je Ing. Petr Muzikant

jeho vlastnosti. Návrhem, vývojem, testováním a nasazením šifrátoru v reálných podmínkách se také zabývá odborný článek, který je v této době v recenzním řízení, psaný pod vedením Doc. Ing. Jana Hajného, PhD. Cílem tohoto snažení by měl být funkční, bezpečný a pro širokou veřejnost dostupný kvantově odolný šifrátor síťového provozu. V budoucnu se také počítá s intenzivnější spoluprací s výzkumným týmem vyvíjející hardwarovou verzi šifrátoru (běžící na platformě FPGA karet), a to zejména v oblasti kompatibility těchto dvou typů. Aktuálně probíhají práce na úpravách softwarové verze šifrátoru a to hlavně v oblasti síťové komunikace. Dále je také nutné upravit šifrátor v oblasti generování klíčů, jelikož zde narážíme na rozdílné řešení u obou platforem. Softwarová verze generuje vždy pouze jeden klíč za určitou časovou jednotku (v základním nastavení jednou za hodinu), naproti tomu verze hardwarová využívá frontu pro ukládání klíčů a přestává generovat až když je daná fronta zaplněna. V rámci adopce tohoto konceptu je potřeba poměrně široký a komplexní zásah do struktury kódu. Toto přepracování se chystá do blízké budoucnosti.

Závěr

Cílem semestrální práce bylo rozšířit stávající šifrátor síťového provozu na platformě Linux o algoritmus ustanovení klíče ECDH a implementovat nový generátor hybridních klíčů.

V teoretické části byly představeny algoritmy použité v rámci této práce. Jmenovitě jimi jsou AES, ECDH, SHA3 a SHAKE-256. Dále bylo vysvětleny koncepty QKD a PQC.

V druhé kapitole byl popsán stav původního šifrátoru před uskutečněním změn. Stejně tak bylo vysvětleno jeho vnitřní fungování a topologie. Nalézt zde můžeme i postup pro instalaci a spuštění šifrátoru. Ten se vztahuje jak na původní, tak na novou verzi.

V třetí kapitole jsou vysvětleny změny, které byly provedeny v rámci této práce. Knihovnou využívanou pro správnou funkčnost zůstává `Crypto++`. Pro šifrování paketů je stále používána šifra AES-256. K PQC a QKD klíči nyní šifrátor vytváří i ECDH klíč. Používanou eliptickou křivkou je P-521. Funkce pro výpočet hybridního klíče je vytvořena podle zcela nového návrhu. Umožňuje dva režimy práce: s QKD klíčem, bez QKD klíče. Výměna klíčů se nyní iniciuje po časovém intervalu, nikoliv v závislosti na počtu zpracovaných zpráv. Po uplynutí časového okna se nově vymění všechny klíče. Ve starší verzi to byl vždy jen QKD klíč.

V následné kapitole bylo provedeno ověření funkčnosti šifrátoru a série měření. Z měření vyplývá, že mezi módy s QKD klíčem a bez QKD klíče je minimální rozdíl, co se týče rychlosti přenosu souborů. Z výsledků testu RTT vyplývá, že nejrychlejší variantou je nešifrované spojení, poté spojení v režimu bez QKD klíče a nejpomalejším je spojení v režimu s QKD klíčem.

Poslední kapitola se zaměřila na představení výsledků praktického nasazení šifrátoru v reálných podmínkách mimo laboratoř. Byli zde představeny reálie technického provedení, problémy, které vyvstali při testování a výsledky měření konaného v realistickém provozu. na závěr kapitoly byl nabínut i pohled do budoucna a představeny další ambice tohoto projektu. Tento program má stále mnoho aspektů, ve kterých může být vylepšen. Jedním z nich je například proces výměny klíče, kde by bylo vhodné vylepšit komunikaci mezi procesy. Dalším návrhem je například tvorba lepšího procesu získání kryptografické soli.

Výsledkem je funkční šifrátor pro operační systém Linux, implementující nové šifrovací algoritmy, mechanismus vypočítání hybridního klíče a další rozšířenou funkcionalitu.

Literatura

[1] TŮMA, P. 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.

[2] RFC6278 - Use of Static-Static Elliptic Curve Diffie-Hellman Key Agreement in Cryptographic Message Syntax [online]. [cit. 2023-11-21]. Dostupné z: https://datatracker.ietf.org/doc/html/rfc6278

[3] SCHWABE, Peter. Kyber. CRYSTALS: Cryptographic Suite for Algebraic Lattices [online]. Dec 23, 2020 [cit. 2023-11-21]. Dostupné z: https:// pq-crystals.org/kyber/index.shtml

[4] DWORKIN, Morris. Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC [online]. November 2007 [cit. 2023-11-21]. Dostupné z: doi:https://doi.org/10.6028/NIST.SP.800-38D

[5] Dworkin, M. , Barker, E. , Nechvatal, J. , Foti, J. , Bassham, L. , Roback, E. and Dray, J. (2001), Advanced Encryption Standard (AES), Federal Inf. Process. Stds. (NIST FIPS), National Institute of Standards and Technology, Gaithersburg, MD, [online], https://doi.org/10.6028/NIST.FIPS.197

[6] Xavier Bonnetain, María Naya-Plasencia, André Schrottenloher. Quantum Security Analysis of AES. IACR Transactions on Symmetric Cryptology, 2019, 2019 (2), pp.55-93. ff10.13154/tosc.v2019.i2.55- 93ff. ffhal-02397049f

[7] DWORKIN, Morris. Block Cipher Modes. NIST [online]. 4. 1. 2017 [cit. 2023-11-27]. Dostupné z: https://csrc.nist.gov/projects/ block-cipher-techniques/bcm

[8] BAKER, Elaine, Lily CHEN, Allen ROGINSKY, Apostol VASSILEV a Richard DAVIS. Recommendation for Pair-Wise Key-Establishment Schemes Using Discrete Logarithm Cryptography [online]. 2018 [cit. 2023-11-27]. Dostupné z: https://csrc.nist.gov/pubs/sp/800/56/a/r3/final

[9] Chen L, Moody D, Regenscheid A, Robinson A, Randall K (2023) Recommendations for Discrete Logarithm-based Cryptography: Elliptic Curve Domain Parameters. (National Institute of Standards and Technology, Gaithersburg, MD), NIST Special Publication (SP) NIST SP 800-186. https://doi.org/10.6028/NIST.SP.800-186

[10] Mink, A. (2009), Quantum Key Distribution (QKD) and Commodity Security Protocols: Introduction and Integration, IEEE Security and Privacy Magazine, [online], https://www.nist.gov/publications/quantum-key-distribution-qkd-and-commodity-security-protocols-introduction-and

[11] Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-Kyber (version 3.02) – Submission to round 3 of the NIST post-quantum project. Specification document (update from August 2021). [online], https://pq-crystals.org/kyber/index.shtml 2021-08-04

[12] NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY. The Keyed-Hash Message Authentication Code (HMAC) [online]. [cit. 2023-12-02]. Dostupné z: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.198-1.pdf

[13] NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY. SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions [online]. [cit. 2023-12-02]. Dostupné z: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf

Seznam symbolů a zkratek

AES	Advanced Encryption System
CA	Certifikační autorita
CPU	Central Processing Unit
ECDH	Elliptic Curve Diffie-Hellman
GCM	Galois Counter Mode
HMAC	Hash-Based Message Authentication Code
ID	Identifikace
IPv4	Internet protocol version 4
KEM	Zapouzdření klíče
LWE	Learning With Errors
MAC	Message Authentication Code
MTU	Maximum Transmission Unit
NAT	Network Address Translation
NIST	National Institute of Standards and Technology
OS	Operační systém
PQC	Postkvantová kryptografie
QKD	Kvantová distribuce klíče
RFC	Request For Comments
RTT	Round Trip Time
SHA	Secure Hash Algorithm
SHAKE	Secure Hash Algorithm KEccak
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
VPN	Virtual Private Network

XOF Extendable Output Function

XOR Excluded OR (logická funkce)

Seznam příloh

A Přílohy **42**
A.1 Generování hybridního klíče v režimu s QKD serverem 42
A.2 Přidání certifikátu CA mezi důvěryhodné certifikační autority 44

A Přílohy

A.1 Generování hybridního klíče v režimu s QKD serverem

Výpis A.1: Generování hybridního klíče v režímmu s QKD serverem

```
// all parameters set, starting to creating hybrid key
        string key_one = hmac_hashing(salt, pqc_key);
        string key_two = hmac_hashing(salt, ecdh_key);
        string key_three = hmac_hashing(salt, buffer_str);
        cout << "Key␣one:␣" << key_one << endl;
        cout << "Key␣two:␣" << key_two << endl;
        cout << "Key␣three:␣" << key_three << endl;

        string param_one = sha3_hashing(pqc_key, &
           kyber_cipher_data_str);
        string param_two = sha3_hashing(ecdh_key, &xy_str);
        string param_three = sha3_hashing(buffer_str, &
           qkd_parameter);
        cout << "Param␣one:␣" << param_one << endl;
        cout << "Param␣two:␣" << param_two << endl;
        cout << "Param␣three:␣" << param_three << endl;

        string second_round_param_one = param_two +
           param_three;
        string second_round_param_two = param_one +
           param_three;
        string second_round_param_three = param_one +
           param_two;
        string second_round_key_one = hmac_hashing(key_one,
           second_round_param_one);
        string second_round_key_two = hmac_hashing(key_two,
           second_round_param_two);
        string second_round_key_three = hmac_hashing(
           key_three, second_round_param_three);
        cout << "Second␣round␣key␣one:␣" <<
           second_round_key_one << endl;
        cout << "Second␣round␣key␣two:␣" <<
           second_round_key_two << endl;
```

```
24        cout << "Second␣round␣key␣three:␣" <<
              second_round_key_three << endl;
25
26        string third_round_key_one = xorStrings(
              second_round_key_one, second_round_key_two);
27        string fourth_round_key_one = xorStrings(
              third_round_key_one, second_round_key_three);
28        cout << "Third␣round␣key␣one:␣" <<
              third_round_key_one << endl;
29        cout << "Fourth␣round␣key␣one:␣" <<
              fourth_round_key_one << endl;
30
31        string key = xorStrings(third_round_key_one,
              fourth_round_key_one);
32        cout << "Key:␣" << key << endl;
33
34        // hash final key with SHA3_256
35        hash.CalculateDigest(digest, (byte *)key.c_str(), key
              .length());
36        CryptoPP::HexEncoder encode_key;
37        string output_key;
38        encode_key.Attach(new CryptoPP::StringSink(output_key
              ));
39        encode_key.Put(digest, sizeof(digest));
40        encode_key.MessageEnd();
41
42        int x = 0;
43        for (unsigned int i = 0; i < output_key.length(); i
              += 2)
44        {
45            string bytestring = output_key.substr(i, 2);
46            key[x] = (char)strtol(bytestring.c_str(), NULL,
                  16);
47            x++;
48        }
49
50        cout << "Key␣established:␣" << output_key << endl;
51        //take the first 32 signs
52        output_key = output_key.substr(0, 32);
53        CryptoPP::SecByteBlock sec_key(reinterpret_cast<
              const byte *>(output_key.data()), output_key.size
```

```
        ());
54      //output length of sec_key
55      cout << "Sec␣key␣length:␣" << sec_key.size() << endl;
56      return sec_key;
57  }
```

A.2 Přidání certifikátu CA mezi důvěryhodné certifikační autority

Výpis A.2: Přidání CA mezi důvěryhodné autority

```
1 sudo cp <názevcertifikátu>.crt /usr/local/share/ca-
     certificates/
2 sudo update-ca-certificates
```

Výpis po proběhnutí skriptu by měl vypadat takto:

Výpis A.3: Výpis po proběhnutí update-ca-certificates

```
1 Updating certificates in /etc/ssl/certs...
2 1 added, 0 removed; done.
3 Running hooks in /etc/ca-certificates/update.d...
4 done.
```