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

HARDWAROVĚ AKCELEROVANÝ PŘENOS DAT S VYUŽITÍM TLS PROTOKOLU
HARDWARE ACCELERATED DATA TRANSFER USING TLS PROTOCOL

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. Adam Zugárek**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. David Smékal**
SUPERVISOR

BRNO 2020

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Diplomová práce

magisterský navazující studijní obor **Telekomunikační a informační technika**

Ústav telekomunikací

Student: Bc. Adam Zugárek ***ID:*** 173785

Ročník: 2 ***Akademický rok:*** 2019/20

NÁZEV TÉMATU:

Hardwarově akcelerovaný přenos dat s využitím TLS protokolu

POKYNY PRO VYPRACOVÁNÍ:

Cílem diplomové práce je hardwarová implementace TLS protokolu na platformě FPGA. Seznamte se s programovacím jazykem VHDL, FPGA kartami NFB, vývojovým frameworkem NDK a platformou Xilinx Vivado. Analyzujte možnosti implementace TLS protokolu na FPGA platformě. Z dostupných implementací zvolte vhodnou kryptografickou sadu pro konkrétní použití. Proveďte základní implementaci vybraných částí TLS protokolu. V závěru diskutujte dosažené výsledky.

DOPORUČENÁ LITERATURA:

[1] ISOBE, Takashi, Satoshi TSUTSUMI, Koichiro SETO, Kenji AOSHIMA a Kazutoshi KARIYA, 2010. 10 Gbps implementation of TLS/SSL accelerator on FPGA. 2010 IEEE 18th International Workshop on Quality of Service (IWQoS). IEEE, 2010, 1-6. DOI: 10.1109/IWQoS.2010.5542723. ISBN 978-1-4244-5987-2. Dostupné z: http://ieeexplore.ieee.org/document/5542723/

[2] PINKER, Jiří, Martin POUPA. Číslicové systémy a jazyk VHDL. 1. vyd. Praha: BEN - technická literatura, 2006, 349 s. ISBN 80-7300-198-5.

Termín zadání: 3.2.2020 ***Termín odevzdání:*** 1.6.2020

Vedoucí práce: Ing. David Smékal

prof. Ing. Jiří Mišurec, CSc.
předseda oborové rady

ABSTRAKT

Tato práce se zabývá implementací kompletního kryptografického protokolu TLS, včetně řídící logiky a kryptografických systémů jež využívá. Cílem je implementace aplikace v technologii FPGA, aby mohla být použita v hardwarově akcelerovaných síťových kartách. Důvodem je podpora vyšších rychlostí, kterých se již na Ethernetu dosahuje, a absence implementace tohoto protokolu na FPGA.

V první polovině práce je pospána teorie pro kryptografii následující popisem protokolu TLS, jeho vývojem, strukturou a fungováním. Druhá polovina se zabývá implementací na cílovou technologii, která je zde popsána. Pro implementaci protokolu je využito již existujících řešení daných kryptografických systémů, nebo alespoň jejich částí, které jsou dle potřeby upraveny dle požadavků TLS.

Implementováno bylo jen několik částí protokolu, a to RSA, Diffie-Hellman, SHA a část AES. Z implementace těchto částí a dalšího zkoumání problematiky vyplynul a byl vyvozen závěr, že pro implementaci protokolu TLS jeho řídící logiky je technologie FPGA nevhodná. Bylo také doporučeno použít FPGA pouze pro provádění výpočtů kryptografických systémů, které jsou řízeny řídící logikou, jenž implementuje software na standartních procesorech.

KLÍČOVÁ SLOVA

VHDL, TLS, FPGA, hardwarově akcelerovaná síťová karta, RSA, Diffie-Hellman, SHA, AES

ABSTRACT

This paper describes implementation of the whole cryptographic protocol TLS including control logic and used cryptographic systems. The goal is to implement an application in the FPGA technology, so it could be used in hardware accelerated network card. The reason for this is new supported higher transmission speeds that Ethernet is able to operate on, and the absence of implementation of this protocol on FPGA.

In the first half of this paper is described theory of cryptography followed by description of TLS protocol, its development, structure and operating workflow. The second half describes the implementation on the chosen technology that is also described here. It is used already existing solutions of given cryptographic systems for the implementation, or at least their parts that are modified if needed for TLS.

It was implemented just several parts of whole protocol, such are RSA, Diffie-Hellman, SHA and part of AES. Based on these implementations and continuing studying in this matter it was made conclusion, that FPGA technology is inappropriate for implementation of TLS protocol and its control logic. Recommendation was also made to use FPGA only for making calculations of given cryptographic systems that are controlled by control logic from software implemented on standard processors.

KEYWORDS

VHDL, TLS, FPGA, hardware accelerated network card, RSA, Diffie-Hellman, SHA, AES

Zugárek, A. *Hardwarově akcelerovaný přenos dat s využitím TLS protokolu*. Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav telekomunikací, 2020. 47 s., Diplomová práce. Vedoucí práce: Ing. David Smékal.

PROHLÁŠENÍ

Prohlašuji, že svou diplomovou práci na téma Hardwarově akcelerovaný přenos dat s využitím TLS protokolu jsem vypracoval samostatně pod vedením vedoucího diplomové 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 diplomové práce dále prohlašuji, že v souvislosti s vytvořením této diplomové 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 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.

V Brně dne ....

(podpis autora)

PODĚKOVÁNÍ

Tímto bych chtěl poděkovat svému vedoucímu práce, panu inženýrovi Davidu Smékalovi, za jeho bezednou trpělivost, pomoc a rady, díky kterým jsem mohl dokončit tuto práci.

OBSAH

SEZNAM OBRÁZKŮ

SEZNAM TABULEK

ÚVOD

Tato práce zkoumá možnosti, kterými je možné provést akceleraci kryptografického protokolu TLS (Transport Layer Security), jenž se nachází nad 4. vrstvou vrstvového modelu ISO/OSI, jehož hlavním úkolem je zabezpečení komunikace proti odposlechu nebo manipulaci mezi dvěma komunikujícími stranami. Jedná se o systém, jenž zastřešuje několik kryptografických systémů, pomocí kterých může zajistit výměnu kryptografických klíčů, autentizaci stran, utajení komunikace a zaručení její integrity.

Kromě popisu protokolu TLS jsou zde popsány i jednotlivé kryptosystémy podílející se na činnosti protokolu TLS. A to včetně jejich implementace na platformu FPGA, kterou využívají hardwarově akcelerované síťové karty. Důvodem implementace na FPGA jsou vyšší přenosové rychlosti v síťových technologií a absence implementace tohoto protokolu, která by ulevila jeho softwarové implementaci a tím i procesoru konkrétního počítače. Autorovým cílem je implementace protokolu TLS jako celku, s využitím již existujících řešení jednotlivých systémů jako je například RSA nebo AES, neboť toto řešení není na trhu dostupné jako hotové řešení. Na trhu se nachází pouze částečné implementace TLS, většinou se jedná o asymetrickou část celého protokolu.

V úvodní kapitole se čtenář seznámí s některými pojmy, využívající se v kryptografii, a i v této práci. Slouží čtenáři jako slovník k vysvětlení základních funkcí a dělení kryptografických systémů.

Druhá kapitola se zabývá teoretickým popisem protokolu TLS, autor zde popisuje hlavní změny protokolu, které se v něm udály při jeho vývoji. Následně je popsána jeho struktura, vlastnosti a komponenty, ze kterých se skládá, a taktéž stručně popsány procesy, které probíhají při sestavování a ukončování spojení.

Poslední kapitola je zaměřena na praktickou část, kde je popsána platforma FPGA a jednotlivé kryptografické systémy a jejich možnosti implementace pomocí číslicových systémů.

1 ÚVOD DO KRYPTOGRAFIE

Kryptografie je věda, jenž se dříve zabývala vytvářením a zkoumáním kryptografických systémů, které prováděly šifrování. Nyní pod tento obor spadá i další problematika, jako je autentizace, integrita dat, distribuce tajemství, digitální podpisy. Dle knihy Protocols for Authentication and Key Establishment jsou základními úkoly kryptografického systému následující 4 body:

- Utajení
- Integrita dat
- Ověření odesílatele dat
- Nepopiratelnost

Utajení – proces, kdy se ze zprávy (z dat) v čitelné podobě (plaintext) šifrováním (za pomoci šifry a šifrovacího klíče) převede zpráva do nečitelné podoby – kryptogramu (ciphertext). Zajištění požadavku, aby zprávu byly schopny přečíst pouze komunikující strany, je dáno znalostí šifrovacích a dešifrovacích klíčů, které by měly mít pouze komunikující strany.

Integrita dat – jedná se o ochranu dat vůči pozměnění obsahu třetími stranami. Používá se speciální funkce, která generuje MAC (Message Authentication Code) ze zprávy (v zašifrované nebo čitelné podobě), jenž se k ní pak připojuje. MAC funkce zpracovává vstupní data o libovolné délce a produkuje hodnotu MAC o přesně dané velikosti, jež je vlastnost konkrétní funkce, která výpočet zprostředkovává. Tato hodnota by měla být vytvářena takovým způsobem, aby při sebemenší modifikaci původní zprávy došlo k její změně. Tudíž by musel útočník vygenerovat pro modifikovanou zprávu novou hodnotu MAC. To ale není možné, protože do procesu vstupuje i kryptografický klíč, který mají pouze komunikující strany, a bez tohoto klíče není možné vypočítat, či jinak odvodit, validní MAC. S tímto lze spojit i další bod – **ověření odesílatele dat**. Odesílatel vytvoří zprávu, provede výpočet MACu s pomocí klíče, který zná jen on a protistrana a následně jej připojí k odesílané zprávě. Příjemce spočítá MAC přijaté zprávy s použitím stejného klíče, jaký použil odesílatel, a v případě že se vypočítaná hodnota MAC bude shodovat s tou v přijaté zprávě, vyhodnotí příjemce přijaté data jako validní (nedošlo k modifikaci zprávy – zaručena integrita zprávy) a současně se ověří i odesílatel. V případě rozdílných hodnot MAC se zpráva jednoduše zahodí. Díky podobnosti vlastností MAC funkce s hashovacími funkcemi, používá se pro generování MAC i hashovacích algoritmů. Výstupem těchto algoritmů je pak hash a tyto algoritmy se označují pojmem HMAC – key-hash MAC. (Hashovací funkce jsou popsány dále v textu této kapitoly)

Nepopiratelnost – ve smyslu, aby nebylo možné zpochybnit přijatou zprávu. Ať je to její obsah nebo ověření odesílatele zprávy. Tato skutečnost může nastat například v situaci, kdy pro ověření odesílatele dat (viz odstavec výše) je použit symetrický kryptografický systém, který má stejné kryptografické klíče pro vytváření a ověřování MAC hodnot. Tudíž komunikující strana *A* zašle zprávu, kterou zabezpečí MAC hodnotou vytvořenou symetrickou šifrou, straně *B*. Strana *B* má klíč pro ověření MAC zprávy od strany *A*, jenž je v případě symetrické šifry stejný s klíčem strany *A*. Strana *B*

tak může vytvořit zprávu s MAC zabezpečením a zaslat ji straně *C* a zároveň popřít, že jí zaslala. Strana *C* si pak nemůže být jistá, jestli zprávu zaslala strana *A* nebo *B*. Tento problém řeší asymetrické systémy, konkrétně systémy digitálních podpisů. Každá komunikující strana využívající systémy digitálních podpisů vlastní svůj privátní klíč, která nikdo jiný nezná, ani si ho nedokáže odvodit. Privátní klíč se podílí na vytváření podpisu, jenž se chová zároveň jako MAC hodnota zprávy, a zaručuje tak integritu dat, ověření odesílatele dat a také nepopiratelnost jak obsahu, tak i původce zprávy. Popis principu fungování asymetrických kryptografických systémů je popsána v textu níže.

V následujícím textu budou stručně popsány základní pojmy, používající se v kryptografii, a základní rozdělení kryptografických systémů, které lze dělit dle několika různých kritérií.

Kryptografický systém je soubor entit určen pro zabezpečení informace, jakéhokoli druhu, uložené na paměťovém médiu nebo vyslané po komunikačním kanále. Entita může být buď jen pouhá šifrovací funkce, část systému, jež se stará o generování provozních dat (např. klíče) pro funkci celého systému, anebo lze považovat za entitu celý další kryptografický systém, jenž je vnořený v tomto systému.

Šifrovací funkce je algoritmus – soubor pravidel (např. matematických výpočtů), který ze zprávy/dat/informace na vstupu v čitelné podobě (plaintext) vytvoří dle určitých pravidel data na výstupu v nečitelné podobě (ciphertext). Aby tento úkon měl vypovídající kryptografickou hodnotu, je zapotřebí, aby výše zmíněný proces probíhal v závislosti na dalším kusu vstupní informace – kryptografickém klíči (tajemství), který je ve většině případů utajovaná informace.

Kryptografické klíče se rozdělují podle použití. Základní rozdělení je na šifrovací a dešifrovací klíč. Pokud šifrovací klíč je stejný jako klíč použitý při dešifrování, nebo lze z jednoho odvodit druhý, jedná se o symetrickou šifru. Jestliže oba klíče jsou nestejné a nelze jeden z druhého odvodit, pak se šifra s těmito klíči nazývá asymetrická. S klíči souvisí také pojem expanze klíče. Jedná se o algoritmus, s jehož pomocí lze z krátkého klíče vytvořit mnohonásobně delší klíč, který lze využít hlavně tam, kde se například používají zřetězené šifry (aby každá šifra v řetězu měla svůj šifrovací klíč).

Asymetrické šifry (např. RSA) se nejčastěji používají v režimu, kdy jeden z klíčů je utajený – privátní klíč, a druhý je veřejný – je znám všem. Při zasílání zprávy adresátovi, si odesílatel stáhne adresátův veřejný klíč a zprávu jím zašifruje. Kdokoliv zná veřejný klíč může zašifrovat a zaslat zprávu adresátovi, ale pouze adresát může svým privátním klíčem zprávu dešifrovat. Pokud bychom upravili použití privátního a veřejného klíče, dostaneme systém digitálních podpisů. A jak již bylo zmíněno dříve v textu, digitální podpis se chová obdobně jako MAC zprávy. Odesílatel vypočítá hodnotu MAC za použití svého privátního klíče a zabezpečovací funkce MAC (na tuto zabezpečovací funkci MAC se pak vztahují podobné požadavky jako na hashovací funkci). Kdokoliv z příjemců může za použití veřejného klíče odesílatele opravdu ověřit, že konkrétní zprávu zaslal tento odesílatel. Vyvstává jen otázka, jak zaručit výměnu veřejných klíčů všech komunikujících stran.

PKI (Public-Key Infrastructure) je infrastruktura řešící právě problém výměny veřejných klíčů komunikujících stran. Bez této infrastruktury by si musely všechny komunikující strany vyměnit své veřejné klíče BEZPEČNÝM způsobem (např. fyzická výměna klíče), což při vyšším počtu komunikujících stran začíná být obtížné.

Infrastruktura PKI obsahuje tzv. certifikační autoritu (CA), což je prvek infrastruktury, kterému důvěřují všichni účastnící infrastruktury. Certifikační autorita se stará o vydávání certifikátů účastníkům. Certifikát obsahuje veřejný klíč účastníka, digitální podpis certifikační autoritou a další informace, např. datum vystavení a platnost certifikátu. Každý z účastníků si pak musí nechat digitálně podepsat svůj veřejný klíč – nechat si vystavit certifikát – certifikační autoritou. Opět musí svůj klíč doručit bezpečně certifikační autoritě a zároveň si musí od ní převzít její certifikát (veřejný klíč). A protože všichni účastnící pak vlastní certifikát certifikační autority, můžou si pak své certifikáty (veřejné klíče) mezi sebou bez problému vyměňovat i přes veřejný Internet, neboť všichni důvěřují všem vydaným certifikátům konkrétní autority.

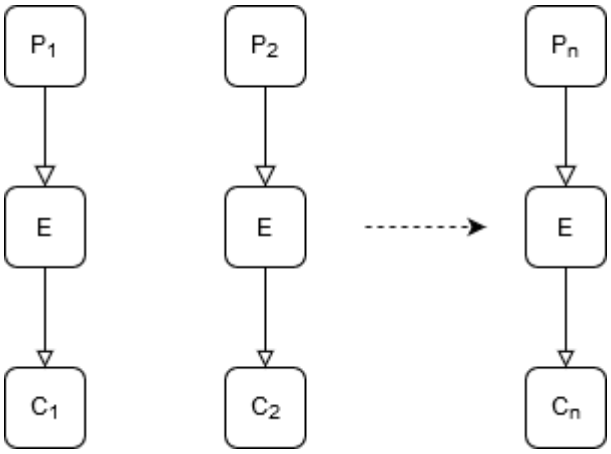
Symetrické šifry se liší od těch asymetrických, kromě stejných šifrovacích a dešifrovacích klíčů, také svou vyšší rychlostí. To je činní lepší volbou pro šifrování velkých objemů dat. Ale jedna z nevýhod oproti asymetrickým šifrám spočívá ve výměně klíčů. Protože dešifrovací klíč lze odvodit z šifrovacího a naopak, neexistuje žádný veřejný/privátní klíč. Klíče si komunikující strany nemůžou vyměnit přímo přes Internet. K tomuto účelu existují protokoly a algoritmy pro distribuci a ustanovení klíčů. Jedná se o postup, při kterém lze dosáhnout bezpečné výměny šifrovacího klíče přes nezabezpečený komunikační kanál, bez předchozí komunikace nebo výměny informací, podílející se na ustanovení klíče. Příkladem může být algoritmus Diffie-Hellman. Jiný způsob výměny klíčů může probíhat pomocí asymetrické šifry, kdy se zašifruje relativně krátký klíč symetrické šifry.

Šifry se dále můžou dělit podle způsobu zacházení s daty. Proudové šifry zpracovávají proud (řetězec) vstupních dat. Jedná se o posloupnost bitů, která se v jádru šifry xoruje po jednotlivých bitech s bity šifrovacího klíče. Ten vytváří generátor pseudo-náhodných čísel (PRNG). Konkrétní bit kryptogramu je závislý pouze na konkrétním jednom bitu původní zprávy. Pro dešifrování stačí pouze vstupní zašifrované data znovu xorovat s šifrovacím klíčem – totožný generátor čísel musí být i na protější komunikující straně. Pro synchronizaci generátorů odesílatele i příjemce slouží speciální hodnota – semeno (seed), které zaručí identické generování pseudo-náhodné posloupnosti na obou stranách.

Druhou možností zpracování dat je po blocích. Blokové šifry zpracovávají na svém vstupu data o pevně dané velikosti (základní parametr blokových šifer) např. 64 bitů nebo 128 bitů. Samozřejmě je použit při procesu šifrovací klíč a na výstupu je pak vrácen stejně velký blok zašifrovaných dat, jako byl na vstupu. Na rozdíl od proudových šifer závisí každý jeden bit kryptogramu vždy na všech bitech původních dat, což značně komplikuje analýzu kryptogramu. Protože šifra zpracovává blok dat o pevné délce, tak v případě jiné délky dat musí být použit některý z režimů blokové šifry. Pokud velikost dat je menší než délka bloku šifry, je potřeba doplnit vstupní data výplní na požadovanou délku bloku. Pro vytváření této výplň existuje několik pravidel a postupů. Při délce vstupních dat větších než je délka bloku šifry, lze použít některý z následujících režimů blokové šifry, vytvoří se tak fragmenty vstupních dat o délce bloku šifry, a poslední fragment, je-li to potřeba, se doplní výplní:

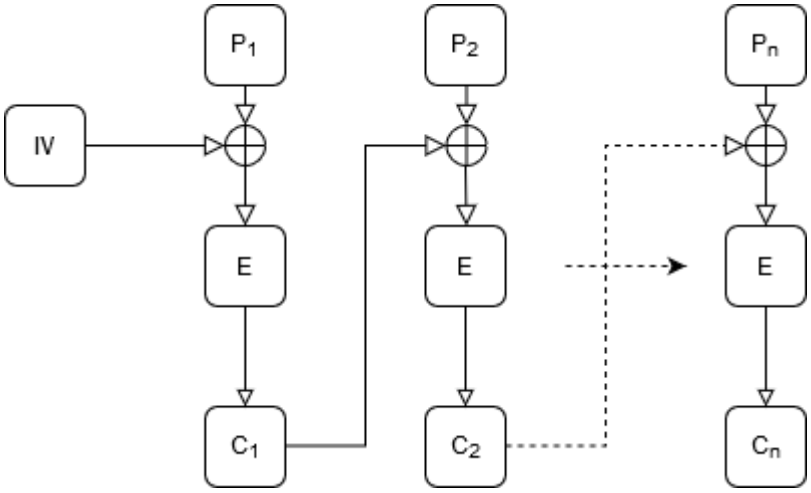
- **ECB – Electronic Code Book** – základní režim, provádí se šifrování dat blok po bloku, nezávisle jeden na druhém, bez jakékoli vazby na jiné bloky. Tento režim se nedoporučuje používat, neboť neposkytuje žádnou kryptografickou ochranu. Dva stejné bloky vstupních dat budou mít vždy stejné kryptogramy. Režim je

zobrazen na obrázku č. 1. Označení *P* je pro plaintext, tedy zprávu v čitelné podobě, *E* je blok šifry – proces šifrování, *C* je ciphertext – zašifrovaná zpráva.



Obrázek 1: Režim ECB

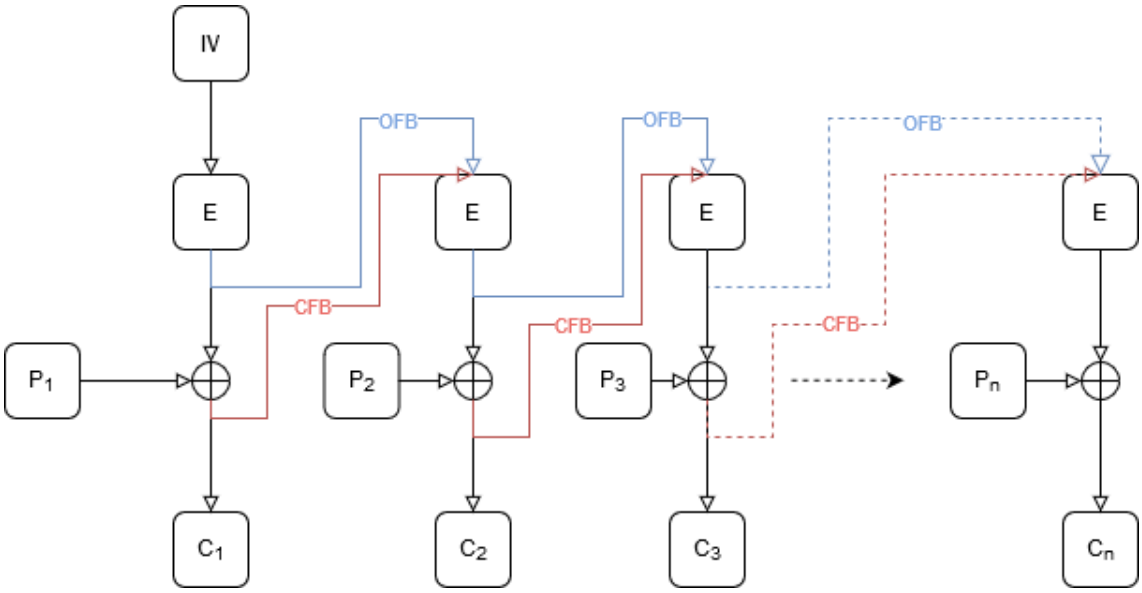
- **CBC Cipher Block Chaining** – provádí se zřetězení jednotlivých bloků. Jinými slovy výsledný kryptogram kromě klíče a vstupního bloku dat závisí také na předchozím zašifrovaném bloku. Na počátku se vstupní blok dat xoruje s hodnotou (může být i náhodná) IV, která se označuje jako inicializační vektor, a výsledek postupuje k šifrování. Vzniklý blok kryptogramu pak slouží druhému bloku vstupních dat jako hodnota IV, takže se vstupní blok dat sexoruje s kryptogramem z předchozího šifrování. Tento postup se donekonečna opakuje. Režim je vyobrazen na obrázku č. 2.



Obrázek 2: Režim CBC

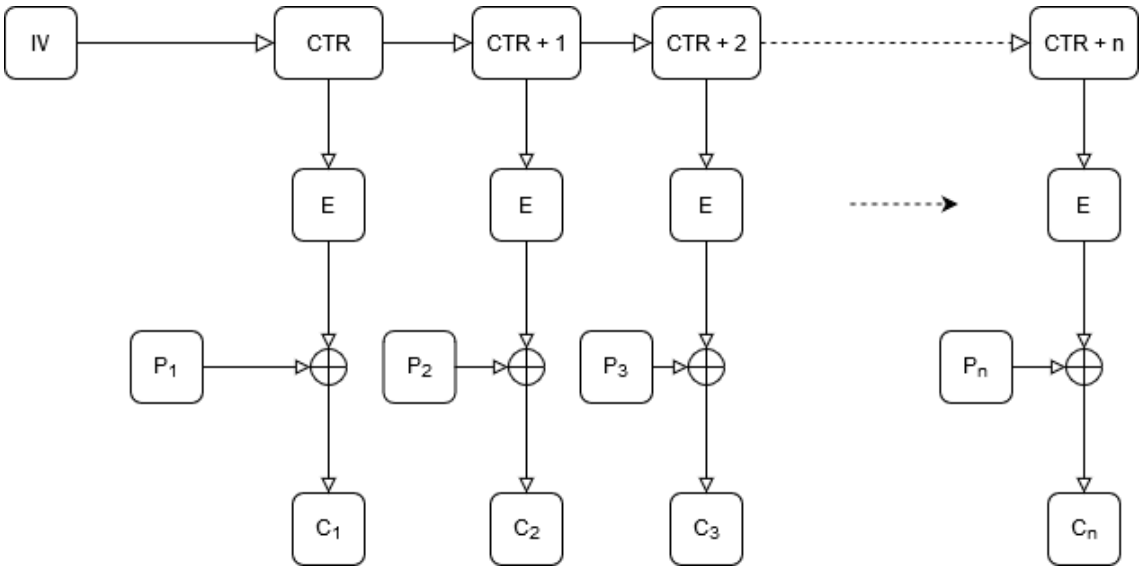
- **OFB – Output feedback** – dalo by se říct, že se jedná o hybridní režim mezi proudovou a blokovou šifrou. OFB režim šifruje vstupní data jako proudová šifra. Xoruje data s klíčem. Tento klíč ale vytváří právě bloková šifra. Opět se zde používá inicializačního vektoru IV, který vstupuje do blokové šifry. Výstupem blokové šifry je pak onen klíč pro proudovou šifru (zašifrovaný výstup blokové

šifry se xoruje se zprávou) a zároveň tvoří tento klíč nový vstup pro blokovou šifru. Existuje o verze **CFB – Cipher Feedback**, kdy v druhém kole šifrování tvoří vstup blokové šifry kryptogram zprávy z výstupu proudové šifry (až po xorování). Rozdíly mezi oběma verzemi jsou vyobrazeny na obrázku č. 3.



Obrázek 3: Režim OFB/CFB

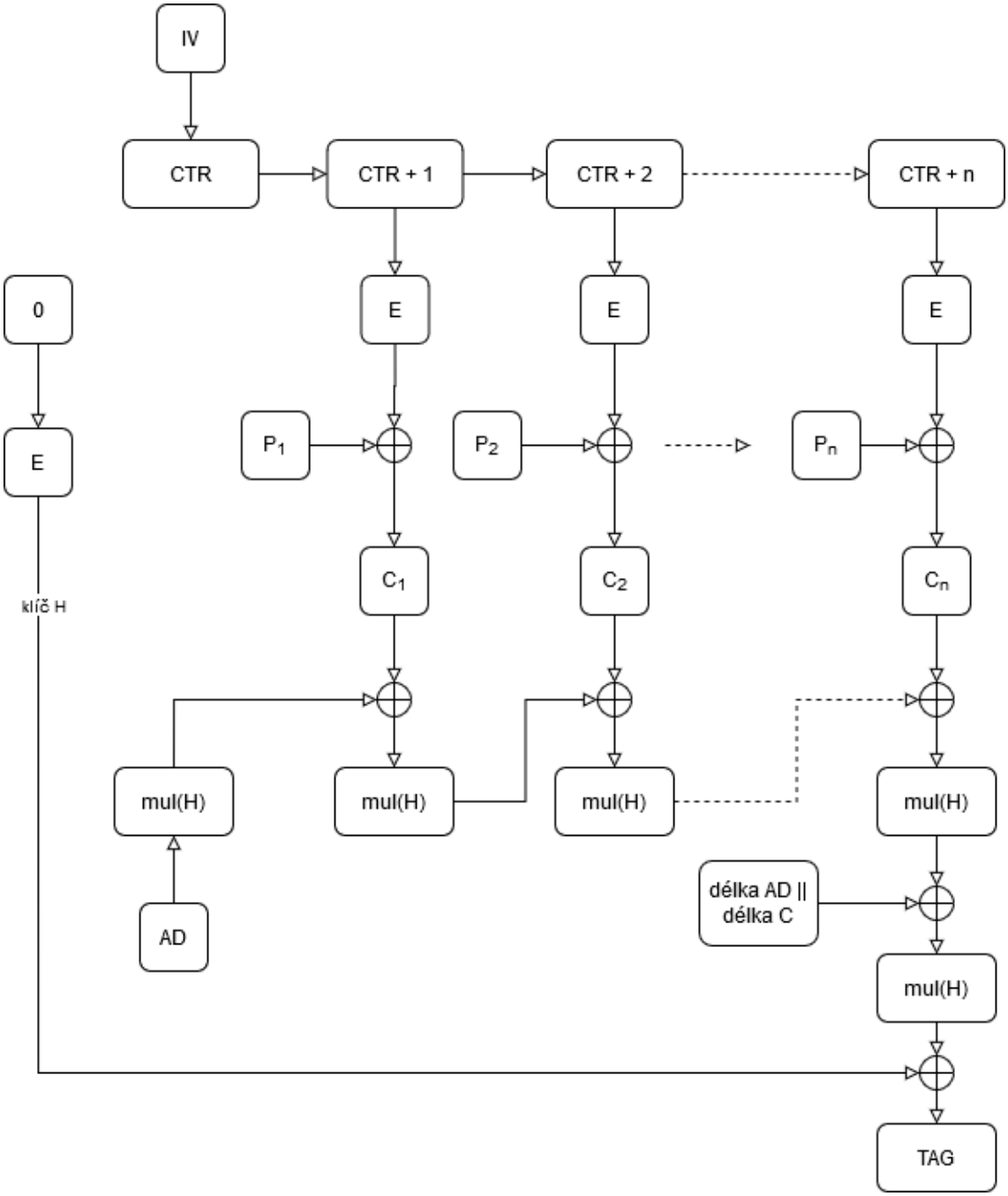
- **CTR – Counter** – režim totožný s OFB s tím rozdílem, že vstupem blokové šifry, která generuje klíč pro proudovou šifru, je sice na počátku také inicializační vektor IV, nicméně do dalšího kola šifrování nevstupuje kryptogram blokové šifry z minulého kola, ale vektor IV inkrementovaný o 1. V každém kole dochází k jeho inkrementaci. Režim je zobrazen na obrázku č. 4, kde blok *CTR* je hodnota čítače, který se postupně inkrementuje o hodnotu +1.



Obrázek 4: Režim CTR

Všechny výše zmíněné režimy jsou citlivé na volbu a použití vektoru IV. Tento vektor by se neměl nikdy opakovat při použití stejného šifrovacího klíče [1]. Tyto režimy jsou základní a obsahují různé nedostatky, například generování výše zmíněného IV vektoru (aby byl opravdu náhodný), a taky by se neměl ideálně opakovat. Z tohoto důvodu existuje velké množství dalších režimů, jenž jsou často upravenými variantami těchto základních režimů, aby odstranily některé (nejlíp všechny) nedostatky základních režimů nebo se aspoň snaží vylepšit jejich rychlost. Zmiňovat je v textu nebudu, ale zmíním dva, které se používají v nejnovější verzi TLS, a to jsou režimy **GCM – Galois Counter Mode** a **CCM – CBC-MAC with Counter**. GCM režim spadá pod jiné rozdělení režimů, a to AEAD (Authentication-encryption with associated data). To je jedna ze skupin režimů, které rozlišuje americký institut pro standardy a technologie (NIST – US-American National Institute of Standards and Technology). NIST dělí skupiny podle účelu na šifrovací režimy, autentizační režimy, šifrovací a autentizační režimy, a autentizační a šifrovací s připojenými daty. Poslední skupina režimů – AEAD – tak dokáže zaslat zprávu, která je rozdělená na 2 části. První část je v čitelné podobě (může být hlavička protokolu) a druhá část je zašifrovaná. Zároveň zajišťuje integritu obou částí. [28]

GCM sestává ze 2 částí (obrázek č. 5). První část – GCTR – se stará o šifrování v režimu CTR. Čítač začíná na hodnotě inicializační vektor IV zvýšeného o 1. Základní CTR, tak jak bylo popsáno dříve v textu. Druhá část – GHASH – se stará o vytvoření MAC zprávy označované jako TAG. TAG je utvářen zřetězeným násobení (násobení nad konečnou množinou – Galoisovo pole). Každý vzniklý kryptogram je násoben s klíčem H (na obrázku blok *mul*), který je vytvořen blokovou šifrou, kde na vstupu je řetězec nul. Výsledek násobení je pak xorován k dalšímu kryptogramu následujícího kola a tento součet je opět násoben klíčem H. Tímto způsobem se pokračuje až k poslednímu kryptogramu, ten je také zpracován a dále se k tomuto výsledku přičte délka všech vzniklých kryptogramů a vynásobí se. Nakonec se xoruje ještě se samotným klíčem H a znovu vynásobí. Vznikne TAG, který zahrnuje všechny kryptogramy, jejich délku a klíč H. Pokud mají být připojeny i dodatečná data (na obrázku blok *AD*), které jsou v čitelné podobě, tak se tyto data nejprve vynásobí s klíčem H a následně se xorují k prvnímu vzniklému kryptogramu těsně před jeho násobením s klíčem H. Délka dodatečných dat se také připojí k délce všech kryptogramů. GCM lze využít také pouze pro autentizaci, kdy odpadá celá první část s blokovou šifrou. [26]



Obrázek 5: Režim GCM

Předloha převzata ze zdroje:

https://en.wikipedia.org/wiki/Galois/Counter_Mode#/media/File:GCM-Galois_Counter_Mode_with_IV.svg

GCM se využívá kvůli své vysoké rychlosti/propustnosti a nenáročné implementaci, hlavně na poli hardwarové implementace. Dle publikace [27], kde autoři mimo jiné testovali i rychlost zpracování s jinými režimy i různou velikostí dat, byl režim GCM vždy nejrychlejší, jak v hardwarové, tak i v softwarové implementaci po určitou hranici velikosti dat. Poté začal být rychlejší režim OCB. Autoři testu prováděli test na datech o velikosti 16–8192 bytů, jelikož posuzovali chování v komunikačních sítí TCP/IP.

Druhý režim CCM náleží taktéž do skupiny šifer AEAD. V základní struktuře jsou si s režimem GCM podobné, ale detaily, ve kterých se liší, mění jejich kryptografické vlastnosti. CCM režim obsahuje také 2 části – šifrovací (CTR režim) a autentizační (CBC-MAC režim). MAC zprávy jsou zde ale počítány ze zpráv v čitelné podobě. V základu je to stejné jako u GCM, ale místo násobení v Galoisově poli se zde používá bloková šifra. Lze usuzovat, že implementace a výkon blokové šifry bude náročnější než u násobení v Galoisově poli.

Druhou část CCM tvoří bloková šifra v režimu CTR, která už bloky zpráv společně s autentizačním tagem (MAC) zašifruje dle známého způsobu činnosti režimu CTR.

Vedle šifrovacích algoritmů, které fungují oběma směry (šifrování a dešifrování), se používají v kryptografii i tzv. hashovací algoritmy. Tyto algoritmy zpracovávají vstupní data libovolné velikosti a vytváří z nich tzv. hash, nebo otisk dat, o pevně stanovené velikosti. Velikost hashe je dána konkrétním algoritmem.

Na hashovací algoritmy jsou kladené také kryptografické požadavky. Jedním z nich je jednosměrnost algoritmu. To znamená, že vytvoření hashe zprávy je jednoduché, ale zjistit z hashe a znalosti algoritmu původní zprávu je takřka nemožné. Dále se požaduje, aby ideálně nevznikaly kolize, tzn. že 2 různé zprávy nesmí mít stejný hash. Bohužel tento požadavek nejde úplně splnit, když počet a délka vstupních zpráv je nekonečné, a počet výstupních hashů je konečný.

Nejpoužívanější 2 hashovací algoritmy jsou MD5 (Message Digest) a SHA (Secure Hash Algorithm). Oba algoritmy (MD5 a SHA-0) jsou v prvotních verzích rychlé, ale jejich zabezpečení bohužel při dnešním výpočetním výkonu počítačů je už nedostačující, oba mají délku hashe 128 bitů. Navíc v SHA-0 byla objevena slabina, kterou později NSA (National Security Agency) opravila. Vznikl tak SHA-1, který měl délku 160 bitů. Bohužel i to je na dnešní poměry málo, a tak vznikly později verze s délkami 256, 384 a 512, označované jako SHA-256, SHA-384 a SHA-512. Nutno podotknout, že výpočet hashe těmito novými algoritmy je už náročnější a trvá déle. Dokonce i jako samotné šifrování šifrou AES. Zajímavostí je, že výpočetně jsou algoritmy SHA-384 a SHA-512 totožné. Při výpočtu SHA-384 se počítá hash o délce 512 bitů a část výpočtu se pak zahodí, takže pokud není kladen důraz na délku hashe, je lepší použít SHA-512. [1]

2 TLS

TLS (Transport Layer Security) je důležitým a standardizovaným protokolem pro zabezpečení komunikace v digitálních komunikačních sítí, jakou je například Internet, jenž je veřejný nezabezpečený komunikační prostředek. Protokol publikuje v jednotlivých RFC organizace IETF (Internet Engineering Task Force) a k datu publikování této práce existují již čtyři verze tohoto protokolu:

- TLS verze 1.0 – RFC 2246 [56] z roku 1999
- TLS verze 1.1 – RFC 4346 [62] z roku 2006
- TLS verze 1.2 – RFC 5246 [55] z roku 2008
- TLS verze 1.3 – RFC 8446 [51] z roku 2018

V komunikačním modelu ISO/OSI se protokol TLS nachází nad čtvrtou vrstvou (L4). A protože pro svou funkci vyžaduje spolehlivý přenos dat, který musí navíc splňovat ty vlastnosti, že přenesená data musí být doručena v pořadí, v jakém byla odeslána, a musí být přenesena na protistranu úplně všechna. Tento požadavek zaručuje v IP (Internet Protocol) sítích protokol TCP (Transport Control Protocol). Avšak existuje taktéž verze TLS pro nespolehlivý přenos dat – DTLS (Datagram Transport Layer Security). DTLS využívá pro svou činnost protokol vrstvy L4 UDP (User Datagram Protocol).

Úkol TLS protokolu spočívá ve vytvoření zabezpečeného komunikačního kanálu mezi dvěma komunikujícími účastníky (nejčastěji to bývá přes nezabezpečenou síť, jakou je například veřejný Internet – dále v textu bude pojem Internet popisovat právě nezabezpečenou veřejnou komunikační síť/médium). Tento zabezpečený komunikační kanál by měl zajistit autentičnost zasílaných dat, jejich integritu a také jejich utajení před odposlouchávající třetí stranou. Výše uvedené požadavky zajišťuje pak ve třech krocích:

- Ustanovení a výměna klíčů a parametrů pro zahájení zašifrované komunikace
- Ustanovení a výměna dalších potřebných parametrů – např. způsob provedení autentizace
- Autentizace komunikujících stran

Po vytvoření zabezpečeného komunikačního kanálu disponují obě komunikující strany hlavními kryptografický klíči, ze kterých si následně obě strany odvodí dílčí klíče, a kryptografickou skupinou/sestavou, která definuje použité kryptosystémy pro jednotlivé operace (podpisy, hash, šifrování). Pokud bylo požadováno, obě strany si vyměnily informace, které jim umožnily se navzájem autentizovat. Nyní již může dojít k zahájení komunikace po zabezpečeném komunikačním kanále, jenž by měl splňovat výše zmíněné vlastnosti (utajení, autentizace a integrita dat).

2.1 Vývoj verzí protokolu TLS

Předchůdcem TLS je, dnes již zastaralý, protokol SSL (Secure Sockets Layer), jehož první verzi vyvinula firma Netscape Communications Corporation jako součást svého webového prohlížeče v devadesátých letech minulého století a celkem byly vyvinuty tři verze protokolu: 1.0, 2.0 a 3.0. Protokol působil hlavně na poli webových služeb (HTTP, IMAP, a jiné). Poslední publikace SSL verze 3.0 byla v roce 1996, ze které se později stal základ pro první verzi protokolu TLS 1.0 v roce 1999. I přes podobnost obou protokolů

jsou zde odlišnosti, které nedovolují vzájemnou spolupráci obou protokolů mezi sebou, avšak je možné, aby protokol TLS 1.0 přešel do režimu zpětné kompatibility a pracoval tak s protokolem SSL 3.0. A protože SSL, které je odrazovým můstkem TLS, dlouho poté bylo velmi používaným protokolem a nemělo oficiální verzi, jenž by někdo spravoval, cítilo IETF potřebu tuto skutečnost napravit, a proto publikovalo v roce 2011 RFC 6101, ve kterém popisuje protokol SSL 3.0 ve stejné podobě jako v roce 1996.

Základní principy fungování napříč verzemi protokolu TLS (tj. až po verzi 1.3) zůstaly víceméně stejné. Proto, pokud to nebude přímo uvedeno, se veškerý popis v následujících kapitolách bude týkat výhradně verze TLS 1.3. Ačkoli drobnějších změn je za ta léta hodně, jedná se o sekundární změny, které nemění významně jádro TLS. Takovými změnami jsou na mysli například: změny týkající se chybový hlášek a stavů, na které protokol reaguje, nebo zavedení režimu 0-RTT (popsáno dále v textu). Výjimku tvoří seznam podporovaných kryptografických systémů, jenž se znatelně postupně zkracoval. Pro představu byly v následujícím textu některé tyto změny popsány.

SSL 3.0 využívalo pro distribuci klíčů systémy:

- RSA
- DHE (Ephemeral Diffie-Helman) s DSS (Digital Signature Standard) podpisy nebo RSA podpisy
- DH s DSS nebo s RSA certifikáty
- Anonymní DH bez podpisů
- Fortezza

Šifrovací algoritmy byly následující:

- Fortezza v režimu CBC s 96-bitovým klíčem
- IDEA v režimu CBC s 128-bitovým klíčem
- RC2 v režimu CBC s 40-bitovým klíčem
- RC4 s 40-bitovým klíčem
- RC4 s 128-bitovým klíčem
- DES v režimu CBC s 40-bitovým klíčem
- DES v režimu CBC s 56-bitovým klíčem
- 3DES-EDE v režimu CBC s 168-bitovým klíčem

Hashovací funkce byly dvě: MD5 – 128 bitů, a SHA-1 – 160 bitů

TLS 1.0 přišlo se změnou v zarovnávání, kdy velikost bloku dat v čitelné podobě musela být nejmenším násobkem délky bloku dané šifry SSL. TLS už tento fakt nevyžaduje, lze použít až 255 bytů výplně při zarovnávání. Tímto způsobem lze ztížit útoky, které provádějí analýzu délky zprávy. Změna v podporovaných kryptografických systémech je v podobě úplného vyřazení Fortezzy z distribuce klíčů i z šifrovacích algoritmů.

TLS 1.1 verze vyřadila další kryptografické systémy. Těmi jsou RC2 a DES s délkou klíče 40 bitů.

TLS 1.2 verze pokračovala s dalším vyřazováním. Tentokrát byla vyřazena šifra IDEA a DES. Další změny byly v bloku pseudonáhodné funkce, kde byla vyřazena kombinace hashovacích funkcí SHA-1/MD5. Pseudonáhodná funkce se využívá pro expanzi různých kryptografických klíčů, jenž jsou potřeba pro správný chod protokolu TLS. Jádrem této funkce je kryptografický systém HMAC. Přibyla podpora hashovací funkce SHA256 s délkou hashe 256 bitů, a podpora šifry AES v režimu CBC s délkou klíčů 128 a 256 bitů. Tímto zbyly 3 šifry: RC4, 3DES a AES. A další změny (např. není vyžadována podpora s SSL 2.0)

TLS 1.3 přichází se změnami jako jsou:

- Vyřazení statických klíčů u RSA a DH
- Podpora algoritmů využívající eliptické křivky
- Zprávy při sestavování spojení jsou po zprávě ServerHello již šifrované
- Podpora režimu 0-RTT, kdy lze zaslat již data druhé straně během procesu sestavování a vyjednávání parametrů spojení.

Nyní jsou podporovány pouze tyto šifry:

- AES v režimech CCM a GCM s klíči 128 nebo 256 bitů
- CHACHA20 POLY1305

Pro distribuci klíčů je použito:

- (EC)DHE
- Pouze PSK
- PSK s (EC)DHE

2.2 Struktura TLS protokolu

Protokol TLS lze z pohledu funkcionality rozdělit na několik funkčních bloků, z nichž jsou dva nejdůležitější: sestavování spojení (handshake) a zabezpečování zpráv (tím je myšleno šifrování apod.). Z pohledu hierarchie se jedná o stejné dělení. Nižší vrstva (v angličtině TLS Record Protocol) a vyšší vrstva (v angličtině TLS Handshake Protocol). Toto stejné dělení používá specifikace protokolu TLS.

Společně s handshake protokolem na vyšší vrstvě existují ještě funkční bloky pro veškeré zpracovávání chybových stavů a expanze klíčů.

2.2.1 Nižší vrstva

zprostředkovává přenos dat mezi komunikačním médiem (zde je jako médium myšlen protokol nižší vrstvy ISO/OSI modelu, a to konkrétně vrstva L4 a protokol TCP, který se již zpracovává data, jako kterékoli jiné, již bez účasti TLS) a vyšší vrstvou. Zároveň, pokud je to vyžadováno, se stará o zabezpečení přenášených dat jak po stránce utajení, tak i autentizace. Tyto dvě funkce (utajení a autentizace zprávy) řeší, od verze protokolu TLS 1.3, pouze šifrovací funkce s režimem podporující AEAD (AES i CHACHA20 tento režim v TLS 1.3 podporují). Nižší vrstva potřebuje, kromě informací o použitých kryptografických systémech, také kryptografické klíče. Těchto klíčů je několik, a liší se dle místa, kde jsou používány. Existují klíče zvlášť pro handshake a zvlášť pro uživatelská data. A ani pro oba směry komunikace není použito stejného klíče

(přenos uživatelských dat směrem ze serveru na klienta používá jiného klíče než tentýž přenos dat směrem z klienta na server). Generování těchto klíčů je popsáno v kapitolách věnující se expanzi klíče.

Nižší vrstva kromě vlastního přenosu dat a jejich zabezpečení dokáže ve stavu, kdy jsou ještě data v čitelné podobě (např. zprávy o chybách nebo patřící handshake protokolu) vsunout navíc výplň, která se pak společně s daty zašifruje. Výsledkem jsou pak data v nečitelné podobě, jejichž délka neodpovídá skutečným přenášeným datům. Tímto způsobem lze ztížit jejich následnou kryptoanalýzu, neboť lze variabilní délkou výplň zajistit, aby výsledná zašifrovaná data měla stejnou délku.

Nebo lze provádět multiplexaci přenášených zpráv – je možné poslat zprávy handshake protokolu v jednom bloku. Tuto funkcionalitu lze používat za jistých podmínek, které, společně se strukturou nižší vrstvy, budou popsány v implementační části této práce.

2.2.2 Vyšší vrstva

Obsahuje logiku řízení celého protokolu TLS. Celé řízení je rozděleno mezi několik bloků, kde tři hlavní bloky jsou:

Handshake protokol

má za úkol vyjednat parametry a následně sestavit spojení pomocí předem definovaných zpráv. Sestavení spojení – handshake – probíhá podle postupu stanoveného v RFC, a při použití pouze povinných položek sestává ze čtyř zpráv. Jádrem algoritmu je stavový automat.

Při úspěšném provedení handshaku jsou ustanoveny šifry a hashovací funkce pro utajení zprávy a zajištění její integrity, každá z komunikujících stran má své hlavní klíče potřebné pro chod TLS, ze kterých se odvozují provozní klíče pro nižší vrstvu. Dále je minimálně ověřena identita serveru a volitelně i identita klienta. Zabezpečená komunikace může být zahájena. Algoritmus je popsán v kapitole 2.3.

2.2.3 Chybové stavy

RFC definuje seznam chybových zpráv (okolo 25) a situace, při kterých se musí použít. Kromě názvu chyby nese struktura zprávy i údaj o vážnosti chyby – upozornění (warning) nebo kritická chyba (fatal). Dle definice zasílají obě strany chybové zprávy ihned při vzniknutí chybového stavu. Pokud nastane kritická chyba, musí obě komunikující strany okamžitě ukončit spojení.

2.2.4 Expanze klíče

Nedílnou součástí TLS je taktéž část zabývající se expanzí klíče. Jedná se o proces, kdy se z hlavního krátkého klíče odvodí klíč několikanásobné délky. Například kryptosystém AES obsahuje vlastní algoritmus pro expanzi klíče využívající své vlastní transformace dat, které běžně používá v procesu šifrování/dešifrování, a následně provádí exkluzivní součet s jednotlivými výsledky těchto operací, které se provádějí dle určitých pravidel a tvoří tak nové klíče. Tímto způsobem dokáže AES vytvořit ze 128bitového klíče sadu klíčů o celkové délce 1408 bitů.

TLS využívá pro expanzi vlastní upravený algoritmus, který v základu využívá funkci HKDF. Expanzní algoritmus „Derive-Secret“ zastřešuje expanzní blok HKDF funkce, kde definuje své řetězce vstupující na místo volitelného řetězce v HKDF. Jedním z vlastních vstupů je i hash již vyměněných zpráv v rámci sestavování spojení (ClientHello, ServerHello, Finished).

Taktéž se využívá i blok extrakce z HKDF, kde jako vstup se používá PSK klíč a (EC)DHE klíč. Pokud jeden z klíčů není k dispozici, jako náhrada je použit řetězec nul o délce hashe hashovací funkci, jenž je použita v HKDF (stejná jako byla vyjednána při sestavování spojení). Celý postup pro expanzi klíče je popsán v RFC 8446 na straně 93 spolu s názvy odvozených klíčů, kde do bloku extrakce vstupují hodnoty shora jako sůl (obecně klíč HMAC funkce) a hodnoty zleva jsou vstupy (vstupní data HMAC funkce).

HKDF (HMAC-based Extract-and-Expand Key Derivation Function) je funkce pro expanzi klíče, který byl publikován organizací IETF v RFC 5869 [38]. Funkce se skládá ze dvou částí – extrakce a expanze.

Extrakce – jedná se o pouhou HMAC hashovací funkci, která zajišťuje dostatečně silný (kryptograficky) klíč (pokud je použit klíč ve formě „obyčejného“ hesla) pro použití v bloku expanze. Pokud je zajištěn dostatečně silný klíče, lze extrakci vynechat a použít tento klíč jako vstup bloku expanze.

Expanze – využívá se opět HMAC hashovací funkce. Při tomto úkonu je generován klíč o libovolné délce – z výstupu se vezme požadovaná délka a zbytek se zahodí. Základem expanze je řetězení hashů z HMAC funkce.

2.3 Sestavení spojení

Jak již bylo dříve zmíněno, k započetí zabezpečené komunikace je zapotřebí nejdříve sestavit spojení (provést handshake) a ustanovit šifry a jejich klíče. K tomu slouží v základu 4 zprávy:

- ClientHello
- ServerHello
- EncryptedExtensions
- Finished

Princip sestavení spojení je následující. Klient zašle jako první serveru Hello zprávu, ve které sděluje, jakou kryptografickou skupinu (cipher suite) podporuje. Další parametry, jež jsou vyžadovány ve struktuře zprávy ClientHello. TLS 1.3 zasílá v proměnné hodnotu odpovídající označení TLS verze 1.2. V předchozích verzí TLS tato hodnota představovala nejvyšší podporovanou verzi. TLS 1.3 musí pro odlišení od předchozí verze zasílat v rozšiřujících položkách (Extensions), konkrétně položka „supportted_versions“. Na základě kontroly existence této položky může server jednoznačně identifikovat verzi protokolu. Tato vlastnost platí i v opačném případě, tedy kdy klient kontroluje ServerHello zaslanou serverem.

Server příchozí zprávu příjme a zpracuje. Pokud ani jednu z uvedených kryptografických skupin, nebo parametrů pro výměnu klíče server nepodporuje, zašle zpět chybovou hlášku, která vyvolá ukončení spojení. V některých případech může zaslat klientu zprávu HelloRetryRequest, pokud si server vybere skupinu, na kterou mu ale

klient nezaslal parametry, jež jsou pro použití tohoto algoritmu vyžadovány. V takovém případě klient znovu odpovídá ClientHello s doplněnými údaji. Pokud předchozí ClientHello obsahovala již uživatelská data, musí je v této vynechat – v odpovědi na HelloRetryRequest nejsou uživatelská data přípustná. Pokud server zaslal společně s HelloRetryRequest i cookies, musí ji klient zahrnout ve své odpovědi. Jinak dojde k chybovému stavu a spojení se ukončí.

Za normálních okolností zasílá server zpět zprávu ServerHello, která je podobná jako ClientHello, ale liší se v detailech. Například neobsahuje seznam podporovaných kryptografických skupin, ale pouze jednu, jenž byla serverem vybrána ze seznamu skupin zaslaných klientem.

V tuto chvíli si obě strany odvodí šifrovací klíče a následná komunikace probíhá už v zašifrované podobě. Další zprávu zasílá server a to „EncryptedExtensions“ a „CertificateRequest“. První zpráva obsahuje doplňující parametry (typu Extensions – viz tabulka na straně 37 RFC 8446 [51]), které se aktivně neúčastní kryptografických úkonů. Tato zpráva musí být zaslána vždy, a vždy hned po zaslání ServerHello. Stejně tak musí být hned za touto zprávou zaslána zpráva „CertificateRequest“, která je volitelná a zasílá se v případě, kdy server provádí autentizaci klientské strany pomocí certifikátu.

Server, pokud ověřuje identitu pomocí certifikátů, zasílá ihned po zprávě „CertificateRequest“ svůj certifikát a následně podepsanou speciální hodnotu ve zprávě „CertificateVerify“. Speciální hodnota se tvoří pomocí hashe již proběhlé komunikace sestavování spojení – handshaku. To znamená, že všechny zprávy od začátku handshaku – tj. ClientHello – se zřetězí a jsou vstupem hashovací funkce. Výsledkem je kontrolní hash zpráv, jež byly doposud poslány v rámci jednoho sestavování spojení, kterým lze zaručit jejich integritu. Podepsáním tohoto kontrolního hashe může komunikující strana dokázat, že vlastní privátní klíč certifikátu, který zaslal v předchozí zprávě. Tuto zprávu společně se svým certifikátem musí zaslat obě komunikující strany, pokud provádí autentifikaci certifikátem.

Poslední zprávou při sestavování spojení je zpráva „Finished“, která je povinná a zasílá jak server, tak klient a obsahuje hodnotu HMAC, kde vstupní hodnotou je kontrolní hash celého handshaku – řetězec proběhlé komunikace, jak bylo popsáno v předchozím odstavci. Klíč použit pro výpočet hodnoty HMAC je odvozen z hlavního šifrovacího klíče pomocí expanzního bloku funkce HKDF. Obě komunikující strany musí ověřit přijatý obsah této zprávy, pokud by se nepodařilo ověřit hodnotu HMAC vůči celému handshaku (všechny doposud zaslané zprávy), musí být spojení přerušené za pomoci chybových hlášení. Jinak je sestavování spojení dokončeno a může začít zasílání aplikačních dat přes nově vzniklý zabezpečené komunikační kanál.

Zprávy nesmí přijít mimo pořadí (např. ClientHello nesmí přijít na server uprostřed sestavování spojení, jinak dojde k chybovému stavu a spojení musí být ukončeno). Dále nesmí žádná z komunikujících stran zaslat odpovědi na dotazy, na které nebyly tázány.

2.4 Ukončení spojení

Ukončit spojení lze pouze pomocí chybových hlášek. Jak již bylo několikrát v textu řečeno, při přijmu chybové hlášky typu chyba (error/fatal alert) kteroukoli z komunikujících stran, musí tato strana spojení okamžitě přerušit a smazat veškeré kryptografické klíče.

Další možností pro ukončení je pomocí chybové hlášky druhého možného typu „close_notify“, jenž slouží výhradně k „nenásilnému“ ukončení spojení. Ukončují se oba směry nezávisle na době, to znamená, že například server zašle close_notify, tím na své straně uzavře odchozí směr (už dále data neodesílá). Klient přijme zprávu o ukončení a uzavře svůj příchozí směr (jakékoli data po obdržení close_notify musí být ignorována). Klient ale stále může posílat serveru data, má-li k odeslání, tento směr zatím ukončen nebyl. Pokud klient nemá již, co by poslal protistraně, zasílá též serveru close_notify, uzavírá svůj odchozí směr a server po přijetí této zprávy uzavírá svůj příchozí směr a ignoruje data obdržené po ukončení spojení. Spojení je nyní ukončené.

3 IMPLEMENTACE TLS A JEHO ČÁSTÍ

Implementaci protokolu lze obecně rozdělit do dvou rovin podle toho, co pak následně vykonává implementovaný algoritmus – hardwarová nebo softwarová implementace. Hardwarová implementace spočívá v použití fyzických komponent, které vykonávají požadovanou funkci bez účasti počítačového programu. Jsou to vesměs různé integrované obvody, ASIC (Application Specific Integrated Circuit), CPLD (Complex Programmable Logic Device) nebo FPGA (Field Programmable Gate Array) obvody. Jejich návrh a vytvoření je složitější, ale v konečném řešení nebo při zpracování dat bývají v určitých případech rychlejší než při softwarové implementaci za použití klasických procesorů.

Pro hardwarovou implementaci existují zařízení označována jako akcelerátory (v angličtině jsou také pojmy SSL/TLS offloading[1] nebo load-balancing). Tato řešení pracují v režimech, kdy nejčastěji akcelerují výpočty prováděné v asymetrickém šifrování (RSA, DSS a DSS), které je z celého procesu použití TLS nejpomalejším článkem. Fyzicky se může jednat např. o rozšiřující kartu do počítače, nebo v případě offloadingu je to fyzicky oddělené zařízení (podobně jako firewall) od počítače nebo serveru, které zpracovává veškerou SSL/TLS komunikaci a s koncovým počítačem, případně serverem, komunikuje v čitelné podobě. Tato zařízení také disponují úložištěm kryptografických klíčů a certifikátů, což ztěžuje jejich odcizení v situaci, kdy by byl počítač napaden a klíče by byly uloženy v konkrétním softwaru. Tato zařízení distribuuje např. firma Fortinet [33] jako zařízení Fortigate, společnost Kemp [34] apod.

Společnost Atmel [24] vyvinula knihovny Atmel Hardware-TLS, které spolupracují s knihovnami WolfSSL a OpenSSL za využití svého ko-procesoru Atmel CryptoAuthentication ATECC508A. Tento čip implementuje TLS autentizaci – jedná se implementaci asymetrické šifry. A také obsahuje úložiště pro kryptografické klíče a certifikáty. Tímto způsobem dokáže ko-procesor ulevit procesoru daného zařízení. Je to jedna z možností, jak levně zvýšit objem komunikace. Symetrickou šifru počítá procesor, který je několikanásobně rychlejší než při počítání asymetrické šifry. Navíc existuje pro některé procesory například od společnosti Intel [35] sada instrukcí AES-NI (Intel AES-New Instructions), jež se stará o zpracování AES. Nárůst propustnosti se tak u této šifry několikanásobně znásobil. Hardwarová implementace AES proto nemusí být nutná, pokud není kladen důraz na velmi vysokou objem přenášených dat.

Implementaci na platformě FPGA nabízí částečně společnost Helion [36], jenž nabízí jednotlivé části, jenž tvořící TLS, nikoli celé TLS. Jedná se o RSA autentizaci a výměnu klíčů, DH výměnu klíčů, šifry AES-CBC, 3DES-CBC a ARC4, a hashovací funkce MD5, SHA-1 a SHA-256. Pro TLS verzi 1.3 bohužel nedostačující.

Softwarové řešení TLS implementují knihovny jako jsou OpenSSL, WolfSSL, GNUSSL, JSSE (java), NSS, matrixSSL a jiné. Výše zmíněné knihovny podporují verzi TLS 1.3.

[1] Pod pojmem offloading se rozumí proces, při kterém dochází přesunu výpočtu na jiný procesor. V tomto případě je to z CPU hostujícího počítače na jiný processor/zařízení (FPGA, ASIC,…). Je to forma load-balancingu, což je rozložení zátěže na více zařízení/procesorů

3.1 FPGA

FPGA (Fiel Programmable Gate Array) neboli programovatelné pole hradel, je číslicový obvod spadající pod programovatelné obvody PLD (Programmable Logic Device) a je jednou z 3 hlavních skupin těchto obvodů spolu s SPLD (Simple Programmable Logic Device) a CPLD (Complex Programmable Logic Device).

První programovatelné obvody začaly vznikat v 70. letech minulého století. Jejich základem byly 2 pole – pole AND hradel následované polem OR hradel. Jejich programování se provádělo již při výrobě, pomocí masky. Následovaly obvody, které měly programovatelnou část AND hradel a pevnou část OR hradel. V této době vznikl první návrhový software pro programování těchto obvodů – PALASM (PAL Asembler). [4]

Tyto obvody byly programovatelné jen jednou. S příchodem pamětí EPROM a EEPROM došlo k další změně, jenž vedla k zařazení těchto pamětí do obvodů PLD a následně šlo tyto obvody přeprogramovat (vymazat elektronicky nebo pomocí ultrafialového světla). V 80. letech vznikl první vyšší programovací jazyk pro popis číslicových systémů – HDL (Hardware Description Language) – jazyk ABEL (Advanced Boolean Expression Language).

Vznikly architektury CPLD a FPGA. CPLD využívá pole AND a OR hradel a pro naprogramování pak paměti typu EEPROM nebo FLASH. FPGA využívají pro svou funkci paměti typu SRAM, pomocí které generují logickou funkci. Označuje se pojmem LUT tabulka (Look-up table) a funguje na principu, kdy v paměti je uložena hodnota – výstup funkce, a k této hodnotě se přistupuje pomocí adresy, což může být, pokud se jedná o 8bitovou adresu, 8 adresních vodičů, které pak reprezentují vstup (8 vstupních vodičů, kdy kombinace jejich logických stavů 0 a 1 tvoří adresu v paměti. Na této adrese je pak uložena hodnota, která reprezentuje opět výstupní vodiče). Tato metoda má výhodu od standartní hradlové logiky v tom, že veškeré logické funkce mají stejné konstantní zpoždění při průchodu signálu, na rozdíl od hradel (víc hradel, více zpoždění).

Jedním z rozdílů ve fungování těchto PLD obvodů je ve způsobu inicializace. CPLD s EEPROM pamětí běží hned po přivedení napětí, kdežto obvody FPGA s SRAM pamětí musí nejdříve, po startu, nahrát konfiguraci z vnější paměti (např. FLASH).

Návrh řešení na FPGA prochází několika kroky. Prvním z nich je napsání kódu popisující číslicový systém. K použití se nabízí jazyky HDL, jako je VHDL (VHSIC HDL – Very High Speed Integrated Circuit Hardare Description Language) nebo Verilog. Existuje i verze jazyka C pro PLD. Následuje simulace a případné generování RTL schématu, což je zkratka pro Register-Transfer Logic – jednoduše řečeno je to proces, kdy se napsaný kód HDL jazyka převede na schéma, za použití logických prvků (hradla, multiplexy, klopné obvody atd.). Dalším krokem je pak syntéza, při které se návrhový software pokusí převést napsaný kód/RTL schéma do podoby, kdy je logika návrhu tvořena ne obecnými logickými prvky (AND, OR apod), ale fyzickými komponenty, které se nacházejí fyzicky na čipu. Následuje implementace na konkrétní čip, kdy se výstup syntézy překládá na seznam propojených komponent a návrhový software se pokouší co možná nejlépe umístit celé řešení na čip, aby dosáhl co možná nejvyšší efektivity využití čipu. Posledním krokem je pak generování bitstreamu, což je výsledný firmware pro FPGA. [17]

Tato práce se v následujících kapitolách bude věnovat implementaci částí protokolu TLS pro použití na síťových kartách využívající technologii FPGA. Jedná se o síťové karty s velmi vysokou propustností, jejichž součástí je FPGA čip, a které dokážou podporovat ethernetový standard až do rychlosti 400 Gbit/s. Disponují QSFP rozhraním pro připojení optických vláken, jsou osazeny velkou a rychlou vyrovnávací pamětí (např. GDDR6). Zpravidla bývají připojené k počítači přes vnitřní rozhraní PCI Express x16 3. generace. Aplikační řešení je rozděleno do dvou částí, kdy část aplikace běží přímo na FPGA čipu (VHDL, Verilog) a komunikuje s druhou, obslužnou částí, která běží jako software v operačním systému počítače.

Takováto síťová karta je k dostání u společnosti BittWare [18], která nabízí síťovou kartu S7t-VG6 s 7nm FPGA čipem Achronix, rozhraním pro 400 Gbit ethernet, 8 GB GDDR6 s propustností až 512 GB/s.

Nebo karta od společnosti Silicom [37], fb2CGhh@KU15P, která podporuje 2x 100Gbit ethernet, obsahuje FPGA čip Xilinx Kintex UltraScale+, s 8 GB pamětí DDR4.

Na českém trhu pak společnost Netcope Technologies [19], která nabízí své NFB (Netcope FPGA Board) karty s čipy od společností Xilinx a Intel. Například karta NFB-200G2QL osazená čipem FPGA Xilinx Virtex UltraScale+, dále rozhraní podporující ethernet až do rychlosti 100 Gbit, s pamětí 3x 288Mb QDRIIIe SRAM.

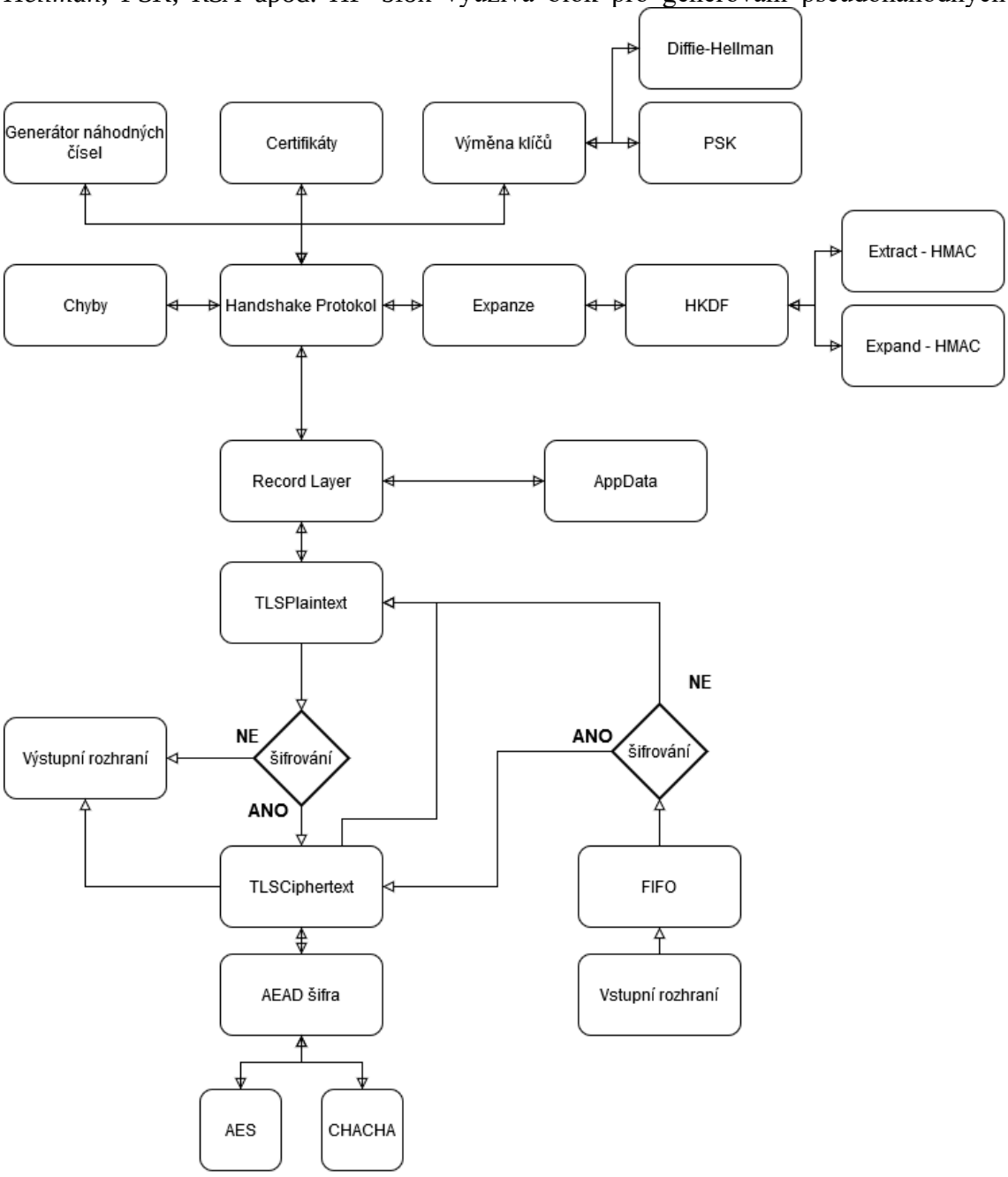
3.2 TLS

V následující podkapitole bude popsán návrh základních komponent protokolu TLS 1.3. Z časové náročnosti implementace veškerých funkcionalit protokolu nebudou implementovány veškeré funkce. A některé komponenty budou implementovány v základních verzích, nebudou obsahovat veškeré náležitosti, jenž protokol TLS 1.3 vyžaduje. Avšak návrh je koncipován tak, aby bylo možné s těmito komponentami pracovat co nejvíc modulově, a tudíž je dle potřeby později nahradit novějšími verzemi, aniž by se narušila funkčnost ostatních částí implementace. Základní návrh protokolu je na obrázku č. 6 níže.

Jak již bylo zmíněno, TLS se skládá ze dvou hlavních částí:

- Vyšší vrstva (nejdůležitější Handshake protokol) – stará se o následující funkce:
    - Autentizace
        - Asymetrické systémy – RSA, ECDSA, EdDSA
        - Symetrické systémy – PSK
    - Vyjednávání kryptografických parametrů a režimů
        - Vyjednání klíče
            - (EC)DHE
            - Pouze PSK
            - PSK s (EC)DHE
    - Ustanovuje materiál potřebný pro vytváření kryptografických klíčů
- Nižší vrstva – stará se o zabezpečení veškerých přenášených dat (šifrování)
    - Pro zabezpečení se používá pouze blokových šifer typu AEAD – provádí se zároveň zabezpečení integrity dat

Blok *Handshake protokol* (dále jen blok *HP*) by se dal popsat jako stavový automat, který má za úkol sestavovat zprávy pomocí různých struktur, jež jsou předem definované a stačí jen vhodně zvolit a naplnit daty. Logika, jenž v těchto věcech rozhoduje, tu nebude popsána, neboť je komplikovaná a obsahuje mnoho stavů a jevů, které je potřeba kontrolovat. Dále je tento blok napojen na komponentu *Výměna klíčů*, která se stará a generování a výměnu materiálu pro dohodnutí klíčů. Děje se tak pomocí bloků *Diffie-Hellman*, *PSK*, *RSA* apod. *HP* blok využívá blok pro generování pseudonáhodných



Obrázek 6: Schéma návrhu TLS

hodnot a také je tu blok *Certifikáty*, jenž by se měl starat o jejich manipulaci. *HP* také řídí bloky *Expanze* a *Chyby*. S blokem *Expanze* se kromě řízení vyměňuje i materiál pro generování provozních klíčů. Posledním propojením bloku *HP* je pak spoj na řídící

logiku nižší vrstvy – *Record layer*.

Blok *Chyby* je databází veškerých chybových zpráv, které vytváří a odesílá na žádost a poskytnutých informacích blokem *HP*.

Blok *Expanze* se stará o veškeré klíče, jež se při komunikaci používají. Generuje klíče podle schématu, které je v RFC na straně 93. Toto generování je závislé na poskytnutém materiálu poskytnuté blokem *HP*. Provádí se blokem *HKDF*, jehož definice je v RFC 5869 [38] Kryptografickým jádrem tohoto bloku je algoritmus *HMAC*. *HKDF* také provádí aktualizaci klíčů – neprovádí se nová výměna, nýbrž se zasílá speciální zpráva bloku *HP*, který na základě této žádosti informuje *HKDF* o provedení aktualizaci zvýšením čítačů a vygenerování nových klíčů za použití těchto hodnot. Tyto klíče jsou pak distribuovány do různých bloků, viz. schéma.

Blok *Record layer* (dále jen RL) pak obsahuje veškerou logiku pro řízení veškerých funkčních bloků nižší vrstvy. Tím je myšleno předávání dat mezi vyšší vrstvou a komunikačním médiem, jejich segmentace a řízení zabezpečení. Blok *RL* příjme data z vyšší vrstvy (*HP*, *Chyby, AppData)* a uloží si je ve své struktuře *TLSPlaintext*. Tato struktura obsahuje identifikátor obsahu dat (Handshake, chybový stav, aplikační data), samotná data a výplň. Použití výplně je volitelné a spočívá hlavně ve ztížení kryptoanalýzy těchto dat útočníkem. Provádí se se zde segmentace a fragmentace dat. Lze dokonce posílat i více zpráv typu „handshake“ najednou, ale chybové zprávy lze posílat pouze po jedné. V tomto kroku lze poslat data přímo na komunikační médium. Tak se děje, pokud je spojení v rané fázi – nejsou ještě ustanoveny klíče ani šifry, například zpráva ClientHello. Anebo, pokud je již ustanoveno šifrování a všechny jeho náležitosti, předávají se data struktuře *TLSCiphertext*. Tato struktura provádí zabezpečení dat pomocí AEAD šifrovacích kryptosystémů. Těmi jsou ve verzi TLS 1.3 pouze AES-GCM a CHACHA20. Struktura vhodně segmentuje celou strukturu *TLSPlaintext* do bloků, dle potřeb šifrovacích algoritmů, a postupně si skládá výsledné zašifrované bloky. Nakonec AEAD šifra vyprodukuje i TAG, což je obdoba HMAC. Následně je možné celou strukturu odeslat na komunikační médium.

Obdobně pracuje proces příjmu dat. Data z komunikačního média jsou ukládány do paměti (FIFO) a následně jsou z této paměti vyzvedávány blokem *RL*. Ten taktéž obsahuje stavový automat, ze kterého je možné určit, v jakém stavu se celé TLS nachází. Pokud je to stav, kdy není ustanoveno šifrování, data z FIFO jsou naplněny do struktury *TLSPlaintext*, v opačném případě do struktury *TLSCiphertext*. Data se dešifrují a AEAD šifra buďto vrátí dešifrovaná data nebo chybu. Data se následně předají do struktury *TLSPlaintext*. Odtud pomocí rozhodovací logiky *RL* a typu dat putují buďto do bloku *HP*, *Chyby* nebo do přímo do aplikace. V případě *HP* a *Chyby* skončí výsledná informace právě v bloku *HP*, který provede příslušnou reakci v závislosti na přijaté informaci a vnitřním stavu bloku. Tím může být vyhodnocení, zda nebyla porušena posloupnost zasílaných zpráv. Například zpráva ClientHello nesmí přijít uprostřed provádění handshaku. V takovém případě musí dojít k okamžitém ukončení.

V následujícím textu jsou popsány některé funkční bloky, jejich implementace a výsledky.

3.3 RSA

Kryptosystém RSA, pojmenovaný po svých autorech Rivest, Schamir a Adleman, je nejrozšířenější systém z řad asymetrických šifer, který využívá pro svou funkci veřejný (a i soukromý) klíč. Byl publikován autory v roce 1978 a používá se dodnes. Jeho podstata spočívá v problému faktorizace velkých čísel a problému se spočtením odmocniny vyšších řádů [1]. To znamená, že se spoléhá na obtížnost vypočítat prvočíselný rozklad velmi vysokého čísla. Bez tohoto rozkladu nelze spočítat další parametry, které vedou k odhalení soukromého klíče z klíče veřejného. Využívá se zde modulové aritmetika a základní rovnicí celého RSA je:

$$c = m^e \bmod n \tag{3.1}$$

kde *c* značí zašifrovanou zprávu *m*, *e* je šifrovací klíč a *n* je modulo. Pro dešifrování je použita stejná rovnice 3.1, kdy dojde k záměně *c* za *m* a v exponentu je místo šifrovacího klíče *e* použit dešifrovací klíč *d*.

Pro jednotlivá čísla existují pravidla, které musí splňovat, aby byl celý systém použitelný:

- Modulo *n* je definováno jako násobek dvou náhodně zvolených prvočísel *p* a *q*. Obě prvočísla jsou musí zůstat utajené. Číslo *n* už je veřejně známé (přenáší se mezi účastníky komunikace
- *r*, jež musí splňovat rovnici $x^r \bmod n = 1$ (3.2). Pokud se budou voleny hodnoty *p*, *q* a *n* podle předchozího odstavce, je *r* výsledkem funkce *φ(n)*. Jedná se o Eulerovu funkci, jenž se využívá pro výpočet počtu všech čísel, která jsou nesoudělná s číslem *n*. Počítá se prvočíselným rozkladem čísla *n*. Platí, že hodnotu Eulerovy funkce pro *n* lze spočítat jako součin hodnot Eulerovy funkce jednotlivých činitelů, jenž tvoří prvočíselný rozklad. Pro prvočíslo *p* pak platí $\varphi(p) = p - 1$ (3.3). Z toho vyplývá, že hodnotu *r* pro číslo *n* lze vypočítat následovně

$$r = \varphi(n) = \varphi(p \cdot q) = \varphi(p) \cdot \varphi(q) = (p-1) \cdot (q-1) \tag{3.4}$$

- Exponenty *e* a *d* se pak volí, aby splňovali rovnici $e \cdot d \bmod r = 1$ (3.5). *e* se volí jako prvočíslo a *d* se následně dopočítává.
- Číslo *m*, které reprezentuje zprávu v čitelné podobě, musí být menší než číslo *n*.

Čísla *n* a *e* pak tvoří veřejný klíč a jsou volně přístupná, kdežto čísla *p*, *q*, *r* a *d* tvoří soukromý klíč a musí být utajená. Šifrování může být zahájeno podle rovnice 3.1.

Pro zaručení bezpečnosti systému (problém faktorizace čísel) se musí volit dostatečně velká čísla. Tato hranice se neustále posunuje s neustálým navyšováním výpočetního výkonu počítačů. Pro přehled resistence kryptosystému v závislosti na délce

používaných klíčů (v bitech) slouží následující tabulka č. 1.

Tabulka 1: Odhad délky klíče v závislosti na zvyšování výpočetního výkonu

Časové období	Délka klíče
2016 - 2030	2048
2030 - 2040	3072
2040 - 2050	7680
> 2050	15360

Hodnoty jsou převzaty ze zdroje [9], který se odkazuje mimo jiné i na dokumenty vydané NIST, a to [10].

Z této tabulky i z dokumentu jasně vyplývá, že klíče o délce 512 bitů a 1024bitů jsou již v dnešní době nedostatečné. S nárůstem délky klíče sice narůstá odolnost kryptosystému vůči jeho prolomení, avšak toto navyšování přináší i výpočetní náročnost celého kryptosystému a to znamená, že jednotlivé operace (šifrování/dešifrování) se zpomalují. Z tohoto důvodu je lepší volit délku klíčů přiměřeně k požadavku na rezistenci celého systému (Pokud se jedná např. o certifikát s roční platností, není třeba volit délku 15360 bitů).

3.3.1 Implementace RSA ve VHDL

Pro implementaci byl použit již existující kód z veřejně dostupných zdrojů na serveru www.github.com. Jedná se o autora vystupujícího na serveru pod jménem scarter93 a projekt nese název RSA-Encryption. Ze zdrojových souborů se jedná o autory Stephen Carter, Luis Galett a Jacob Barnett. Ke zveřejněnému kódu není připojená dokumentace, tudíž bylo nutné analyzovat celý kód a ověřit validitu vyprodukovaných výsledků. Analýza kódu společně s úpravami jsou popsány v následujícím textu.

Je nutno předem zmínit, i přestože je proces šifrování/dešifrování definován relativně jednoduchou rovnicí, je reálná implementace obtížná hlavně z důvodu velké délky klíčů. Pro srovnání, v jazycích (jako je např. C, C++, atd) je délka, běžně používaného, celého čísla typu int 32 bitů, to je rozsah přibližně 4 miliardy. Nebo, pro desetinná čísla, je použit rozsah 64 bitů. V celých číslech je to přibližně 18 trilionů. Vynásobením dvou *n*-bitových čísel dostaneme číslo s délkou *2n*. Umocněním dvou n-bitových čísel pak dostaneme délku n^2. Délka aktuálně používaných klíčů v RSA je nyní 2048 bitů. To znamená, že po umocnění by byla délka lehce přes 4 miliony bitů. Následná redukce zkrátí délku opět na původní velikost, ale mezikrok by stále zůstával. Provedení výpočtu takovýchto rozměrů je nereálné a taky zbytečné, když se skončí zpět na původní délce 2048 bitů. Pro tento výpočet lze použít algoritmus „umocni a vynásob“ z anglického výrazu „square and multiply“. Tento algoritmus dokáže efektivně převést operaci mocnění velkým číslem na posloupnost operací mocnění 2 a jednoduchým násobením. A protože operace modulo (redukce) lze použít na jednotlivé činitele, lze jednotlivé výsledky průběžně redukovat (defacto po každé operaci násobení nebo mocnění), a tudíž mezivýsledek tolik nenarůstá na velikosti bitové délky.

Algoritmus „square and multiply“ pracuje následovně. Exponent se převede do binárního čísla a postupuje se od nejvýznamnějšího bitu MSB, tedy zleva doprava. První 1 – opíše se mocněné číslo. Následně se provádí dokola, až po konec exponentu, tento

postup: při výskytu 0 se celek umocní 2, při výskytu 1 se celek umocní, a navíc vynásobí mocněným číslem. [39]

Pokud bude na mocnění 2 nahlíženo jako na násobení, byla získána posloupnost činitelů kratší, než kdyby se nahlíželo na původní mocnění jako na násobení. Nyní stačí jen násobit a redukovat (modulo). Bohužel i přes toto zjednodušení je násobení a redukce stále nákladná na zdroje. Naštěstí i pro tyto 2 operace existuje řešení. Tuto operaci bude v textu nazývána modulové násobení a řešením je Montgomeryho algoritmus. Tento algoritmus navrhl a publikoval v roce 1985 Peter L. Montgomery [13]. Jedná se o metodu násobení 2 celých čísel, které jsou následně zmodulovány číslem *m*, aniž by bylo použito dělení číslem *m*. Tento algoritmus je efektivní a jednoduchý pro hardwarovou implementaci.

Montgomery definuje pro svou reprezentaci *n*-residua, která jsou dána rovnicí 3.6:

$$A = a \cdot R \ (mod\ m) \tag{3.6}$$

Obdobně i pro druhé číslo *b*. Montgomeryho algoritmus počítá součin dán rovnicí 3.7:

$$U = A \cdot B \cdot R - 1 \ (mod\ m) \tag{3.7}$$

Kde R^{-1} je multiplikativní inverze k R podle rovnice 3.8

$$R - 1 \cdot R = 1 \ (mod\ m) \tag{3.8}$$

K výpočtu je ještě potřeba čísla *n'*, které odpovídá rovnici 3.9

$$R \cdot R - 1 - n \cdot n' = 1 \tag{3.9}$$

Následně lze pomocí Montgomeryho algoritmu, který je uveden na straně 3 [40] jako funkce *MonPro(A,B)*, spočítat součin *U* podle rovnice 3.7. Malou úpravou tohoto algoritmu, konkrétně vynásobením součinu *U* číslem 1 stejnou funkcí *MonPro*, lze vypočítat obyčejný součin. Za podmínky, že číslo *m* musí být liché. Zmíněná úprava (*ModMul(a, b, n)*) je na téže straně. Nicméně se v této variantě výpočet neobejde bez výpočtu hodnot R^{-1} a *n'*, což může být stále překážka. Naštěstí existuje další varianta úpravy algoritmu, jenž pracuje na bitové úrovni. Zmíněný modifikovaný algoritmus je uveden na straně 6 [40] a počítá s hodnotou $R = 2^k$, tím pádem $R^{-1} = 2^{-k}$. V binární soustavě se jedná jen o bitový posuv vlevo a vpravo. Dále musí být splněny podmínky, že *n* je liché a hodnoty *a, b* a *n* jsou menší než hodnota *R*. A protože se postupuje po bitech, stačí znát z hodnoty *n'* jen nejméně důležitý bit (LSB), který je roven 1, pokud je *n'* liché. A *n'* je liché, neboť *n* je liché. Tímto se výpočet podstatně zjednodušil. Pro použití v RSA stačí použít variantu *ModMul* ze strany 3 [40] a jako funkci *MonPro* použít algoritmus ze strany 6 [40].

K implementaci RSA bylo využito práce autorů práce [14], ve které autoři popisují taktéž Montgomeryho algoritmus a prezentují několik variant tohoto algoritmů, které se liší ve zpracování operandů. Například varianta pro zkrácení doby průběhu algoritmu tak, že při jednom kroku se načtou 2 bity apod.

Na straně 3 [14] pak autoři definují pseudokód přímo pro použití v RSA (včetně), který odpovídá kódu, jenž vytvořil autor VHDL kódu RSA [8] (místo proměnné *Z* používá autor proměnnou *R*, jejich funkce je totožná).

Jelikož v práci [8] nebyla obsažena dokumentace, nebylo možné určit, jakým způsobem je volena konstanta *K*. Proto bylo potřeba provést nové úpravy, které spočívaly v úpravě vstupních operandů (funkce *Montgomery_format* a konstanta *K*). Výsledný algoritmus vypadá následovně:

```
K = 2^k
Z = Montgomery_format(1, K, m)
P = Montgomery_format(Plaintext, K, m)
for (int i = 0; i < k; i++)
      if (exp[i] == 1)
            Z = Montgomery(Z, P, m)
      P = Montgomery(P, P, m)
Z = Montgomery(1, Z, m)
C = Z
```

Číslo *k* reprezentuje bitovou délku operandů. Tímto je splněna podmínka dříve zmíněného algoritmu *MonPro* z 6. strany [40]. Pokud jsou použity například 32bitové čísla, je jejich maximální hodnota 2^{32}-1, kdežto *K*(v algoritmu *MonPro* se jedná o *r*) je rovno 2^{32}. *exp* je exponent, čili šifrovací/dešifrovací klíč. *Plaintext* je neupravená zpráva v čitelné podobě, *C* je výsledným produktem celého výpočtu RSA, tudíž kryptogram zprávy *Plaintext*.

Výše zmíněný pseudokód je variantů pro načítání bitů zprava doleva, vyžaduje při tom, aby běžely paralelně vedle sebe 2 instance násobičky. Autoři [14] definují i variantu pro načítání bitů opačným směrem. Tato varianta používá jen jednu instanci násobičky, avšak pro svou činnost vyžaduje 2krát delší čas pro výpočet. Takže volba je zde rychlost versus zabrané zdroje.

Struktura výsledného kódu je členěna na 3 komponenty. První, nadřazenou, komponentou je samotný algoritmus „square and multiply“, společně s dalším řízením celého výpočtu. Tato komponenta má na svém rozhraní několik jednobitových řídících signálů a pak signály, které tvoří datový vstup a výstup celé komponenty. Délka těchto signálů je dána parametrem. Upraveno bylo rozhraní této top komponenty a doplnění o chybějící signály modula, obou klíčů potřebných pro funkci RSA. Jádrem této komponenty je stavový automat, jehož stavy odpovídají jednotlivým krokům algoritmu „square and multiply“.

Druhou komponentou je montgomery_format.vhd. Tato komponenta byla doplněna a nahrazuje původní funkci pro výpočet residua operandu, jenž autor [8] počítal pomocí původního Montgomeryho algoritmu *MonPro*. Tento nový výpočet odpovídá výše zmíněné teorii, konkrétně ta část, jenž se zmiňuje o *n*-residuech – rovnice 3.6.

Poslední komponentou je montgomery_multiplier.vhd, což je přímo implementace funkce *MonPro* na straně 6 [40]. Tato komponenta je v nadřazené top komponentě instanciována 2krát.

Rozhraní jednotlivých komponent je v následujících tabulkách č. 2, 3 a 4.

Jádro RSA - modulové mocnění (modular_exponentiation.vhd)			
Název	Směr	Bitová délka	Popis
N	in	32*1	vstupní zpráva (čitelná podoba nebo kryptogram)
enc_key	in	32*1	šifrovací (veřejný) klíč
dec_key	in	32*1	dešifrovací (soukromý) klíč
M	in	32*1	modulo
enc_dec	in	1	volba šifrování/dešifrování (klíčů)
clk	in	1	hodinový signál
reset	in	1	reset
data_rdy	out	1	signalizace dostupnosti výsledku C
C	out	32*1	výsledná zpráva
*1 - bitová délka je upravována parametrem WIDTH_IN na konkrétní hodnotu, v tomto případě byla použita délka 32 bitů			

Tabulka 2: Rozhraní modulu RSA

Montgomery - Montgomeryho reprezentace operandů (montgomery_format.vhd)			
Název	Směr	Bitová délka	Popis
A	in	32*1	vstupní operand A
r	in	32*1	Montgomeryho konstanta
n	in	32*1	modulo
clk	in	1	hodinový signál
start	in	1	signalizace - start
data_rdy	out	1	signalizace - konec
a_out	out	32*1	výsledek
*1 - bitová délka je upravována parametrem WIDTH_IN na konkrétní hodnotu, v tomto případě byla použita délka 32 bitů			

Tabulka 3: Rozhraní modulu pro výpočet residua

Montgomery - modulové násobení (montgomery_multiplier.vhd)			
Název	Směr	Bitová délka	Popis
A	in	32*1	vstupní operand A
B	in	32*1	vstupní operand B
N	in	32*1	modulo
latch	in	1	signalizace - start
data_read	out	1	signalizace - výpočet dokončen
clk	in	1	hodinový signál
reset	in	1	reset
M	out	32*1	výsledek
*1 - bitová délka je upravována parametrem WIDTH_IN na konkrétní hodnotu, v tomto případě byla použita délka 32 bitů			

Tabulka 4: Rozhraní modulu pro modulové násobení

3.3.2 Testování

Pro testování byly zvoleny hodnoty splňující základní kritéria, jež byly zmíněné dříve v textu. Zvolené hodnoty jsou v tabulce č. 5.

Testovací hodnoty		
Název	Hodnota	Popis
N	50	zpráva v čitelné podobě
M	143	modulo
enc_key	17	šifrovací klíč
dec_key	113	dešifrovací klíč
C	85	kryptogram zprávy N
p	11	prvočíslo tvořící číslo M
q	13	prvočíslo tvořící číslo M
r	120	výsledek Eulerovy funkce pro M
WIDTH_IN	32	bitová šířka operandů

Tabulka 5: Hodnoty pro testování

Hodnota šifrovacího klíče byla volena náhodně. Hodnota dešifrovacího klíče byla dopočítána, aby splňovala podmínku $e \cdot d \bmod r = 1$, pomocí jednoduchého kódu psaného v jazyce C.

```
unsigned long e = 17;
unsigned long r = 120;
unsigned long d = 0;

for (unsigned long i = e+1; i<r; i++)
```

```
{
	if ((e*i % r) == 1)
	{
		d = i;
		printf_s("\nVysledek: %d ", d);
		//getchar();
		break;
	}
	printf_s("\n %d ", i);
}
```

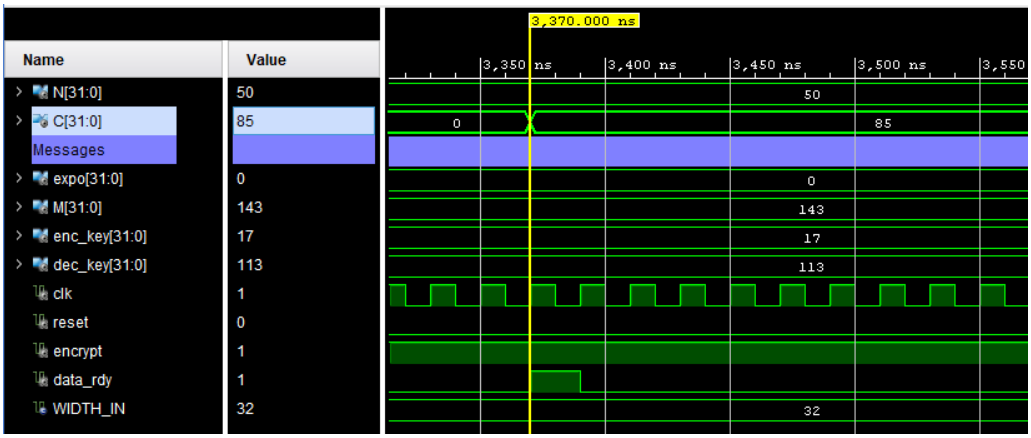
Tímto primitivní způsobem lze dohledat hodnotu dešifrovacího klíče. Iterace se provádí, dokud se nedosáhne r. Díky provádění operace mod r by šlo pokračovat v hledání dalších dešifrovacích klíčů podle vztahu $d = d + n \cdot r$, kde n je přirozené číslo, ale ani klíče nesmí být větší, než je hodnota M, tudíž to nemá smysl.

Ověření vypočítaných hodnot modulu RSA bylo provedeno pomocí webové aplikace Wolfram Alpha, dostupné [7]. Jedná se výpočetní engine, který se snaží zpracovávat jakýkoliv dotaz přijatý volně v textu a odpovědět na něj maximálním způsobem (výpočet, různé definice). Nevrací odkazy na možné řešení. Wolfram Alpha je vyvíjen britskou společností Wolfram Research, publikován roku 2009.

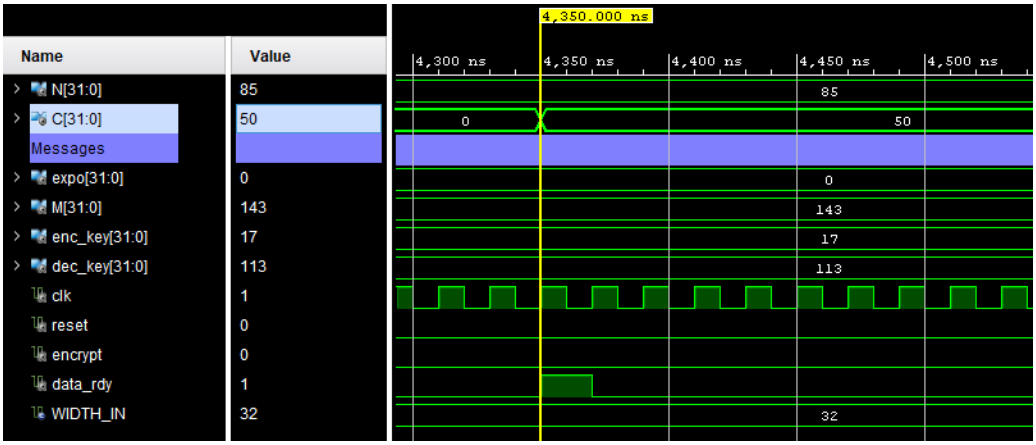
Test je prováděn v simulačním prostředí vývojového studia Vivado (obrázky č. 7 a 8) od společnosti Xilinx. Výsledky testu jsou v následujících dvou obrázcích. Výpočet šifrování trval 165 taktů, dešifrování zabralo 214 taktů. Ale ani tyto hodnoty nelze brát jak absolutní hodnoty, neboť doba trvání výpočtu závisí na velikosti klíčů (myšleno, ačkoliv se jedná o 32bitové čísla, je rozdíl mocnit číslem 5 a 500 000).

Zabraných zdrojů FPGA čipu je následovné:

- 1261 LUT tabulek
- 1054 registrů



Obrázek 7: RSA simulace - proces šifrování



Obrázek 8: RSA simulace – proces dešifrování

3.4 Diffie-Hellman

Jedná se o algoritmus používající se pro výměnu kryptografického klíče přes veřejný Internet. Algoritmus publikovali v roce 1976 autoři Whitfield Diffie a Martin Hellman. Algoritmus, sérií výpočtů, dokáže předat informace protistraně, pomocí kterých si obě komunikující strany odvodí stejný klíč. Aniž by byly předem oběma účastníkům distribuovány jakékoli privátní informace. A aniž by z předávaných informací mezi obě účastníky mohla třetí strana odvodit tentýž klíč.

Navíc dokáže algoritmus zajistit tzv. dopředné utajení (angl. Forward secrecy), což je označení pro algoritmy, jenž mají na starosti výměnu kryptografického klíče, které garantují, že i když by došlo k prozrazení hlavního privátního klíče jedné z komunikujících stran, není možné z něj odvodit klíč, na kterém se dohodly obě strany, protože k odvození výsledného dílčího klíče oběma stranami je zapotřebí, kromě běžných statických veřejných parametrů, i obou veřejných parametrů, jenž generují obě strany. Případně byla by zachycena komunikace včetně výměny všech veřejných parametrů, došlo by pak ke kompromitování jen malé části uživatelských dat. Samozřejmě za předpokladu, že pro každé nové spojení se generují nové dílčí klíče.

Algoritmus využívá pro svou funkci matematického problému diskrétního logaritmu, což v kostce znamená (bez podrobného vysvětlování a matematických definicí): Mějme matematickou operaci modulo a k ní hodnotu modula, prvočíslo 17. K tomuto číslu mějme grupu s prvky 0 – 16. Nyní mějme generátor grupy, například číslo 3. Vlastnost generátoru je ta, že jeho postupným mocněním, podle rovnice:

$$3^x \ mod\ 17 \qquad (3.10)$$

Získáme všechny prvky grupy, tedy čísla v rozsahu 1-16.

Zjistit výsledek rovnice například $3^4 \ mod\ 17 = 13$ je jednoduché. Ale opačně, zjistit z výsledku 13, jakého bylo použito exponentu už je obtížné. Zvlášť pro velmi velká čísla. Tento problém se nazývá problém diskrétního logaritmu.

Průběh algoritmu je následující:

1. Zvolí se hodnota modula *p*, která je prvočíslem. K tomuto prvočíslu je i grupa. Je potřeba najít hodnotu generátoru g pro tuto grupu. Jinou možností je zvolit hodnoty z již předem vygenerovaných grup. Tyto grupy jsou zmíněné například v RFC 7919 [41], kde jsou definovány hodnoty *p* a *g*. Tyto hodnoty jsou veřejné.
2. Každá z komunikujících stran si zvolí náhodné číslo. Mějme stranu A, která si volí číslo a, a stranu B, která volí analogicky číslo b. Tyto čísla jsou soukromá, musí být po celou dobu komunikace udržovány v tajnosti.
3. Provede se výpočet hodnot *A* podle rovnice: $A = g^a \ mod \ p$ (3.11), a analogicky i pro hodnoty *B*.
4. Hodnoty A a B si obě strany mezi sebou vymění => tyto hodnoty jsou taktéž veřejné.
5. Vypočítá se hodnota $SA = B^a \ mod \ p$ (3.12) a hodnota $SB = A^b \ mod \ p$ (3.11). A když se dosadí za hodnoty *A*, *B* výrazy z kroku 3, dostaneme následující rovnice

$$SA = B^a = g^{b^a} = g^{ba} \qquad (3.12)$$
$$SB = A^b = g^{a^b} = g^{ab} \qquad (3.13)$$

   Hodnoty *SA* a *SB* jsou dle výše uvedených rovnic 3.12 a 3.13 totožné, a jsou výsledkem procesu – kryptografický klíč mají k dispozici obě komunikující strany. A kdokoli další, jenž by zachytil celou tuto výměnu parametrů, není schopen odvodit výsledný klíč, protože vždy mu bude scházet jedna z hodnot a či b.[42]

3.4.1 Implementace DH ve VHDL

Při implementaci tohoto protokolu, bylo využito podobnosti s RSA. Ačkoliv jsou DH a RSA dva kryptografické systém, kde má každý z nich jinou úlohu a každý je postaven na rozdílném matematickém problému (DH využívá problém diskrétního logaritmu, kdežto RSA využívá problému faktorizace velkých čísel), matematická funkce v jádrech obou systémů je stejná. Myšlena je funkce modulárního mocnění (podle rovnice 3.1).

$A = g^a \ mod \ p$ (3.11) $\qquad$ $c = m^e \ mod \ n$ (3.1)

Z tohoto důvodu, při implementaci, je DH pouze nadstavbou jádra RSA. Tudíž nebude výpočet znovu popisován. Implementovaná nadstavba je defacto pouze řídící logika, která vhodně předává operandy komponentě pro modulové mocnění. Rozhraní modulu DH je popsáno v následující tabulce č. 6.

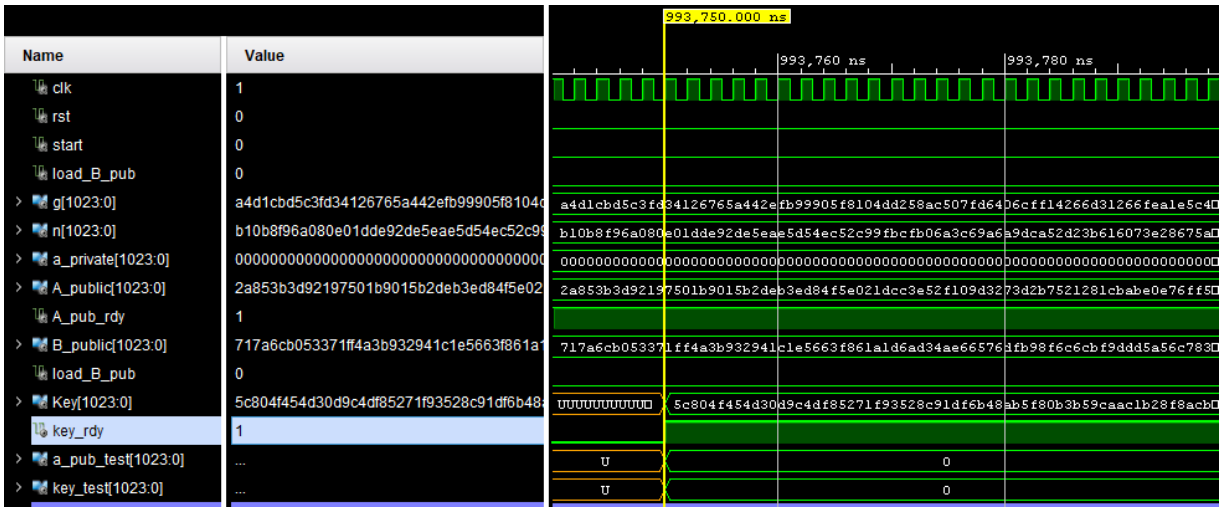
Diffie-Hellman (DH.vhd)			
Název	Směr	Bitová délka	Popis
clk	in	1	hodinový signál
rst	in	1	signalizace - reset
g	in	128*1	vstupní parametr - generator
n	in	128*1	vstupni parametr - modulo
a_private	in	128*1	vstupni parametr - soukromá hodnota *a*
A_public	out	128*1	výstupní parametr - veřejná hodnota *A*
B_public	in	128*1	vstupní parametr - veřejná hodnota protistrany
Key	out	128*1	výstupní parametr - vyjednaný klíč
start	in	1	signalizace - start
A_pub_rdy	out	1	signalizace - hodnota *a*
load_B_pub	in	1	signalizace - načtení hodnoty *B*
Key_rdy	out	1	signalizace - konec
*1 - bitová délka je upravována parametrem n_size na konkrétní hodnotu, v tomto případě byla použita délka 128 bitů			

Tabulka 6: Rozhraní TOP komponenty Diffie-Hellman

Popis funkce: Před použití potřeba provést reset. Pro start přivést signál *start* log. 1, souběžně s náběžnou hranou si komponenta kopíruje hodnoty *g*, *n* a *a_private* do svých vnitřních registrů. Následně provádí výpočet hodnoty *A*. Dokončení výpočtu je signalizováno signálem *A_pub_rdy* a současně je hodnota *A* přivedena na výstup *A_public*. Nyní je potřeba dodat komponentě hodnotu *B* protistrany (což odpovídá signálu *A_public*, pokud by protistrana použila tutéž komponentu DH) a zároveň přivést signál *load_B_pub* do stavu log. 1. Vnitřní registr pro hodnotu *B* se resetuje jen při resetu, tudíž je možné tuto hodnotu nahrát kdykoliv. Komponenta, jakmile spočítá hodnotu *A*, kontroluje následně, zda byla nahraná hodnota *B*. Pokud ano, pokračuje s výpočtem klíče. Výsledek je signalizován signálem *Key_rdy* a výsledek dostupný na signálu *Key*.

3.4.2 Testování

Pro testování byly použity hodnoty z RFC 5114 [43], strana 14, příloha A.1. A DH grupa ze strany 3, 2.1. Pro ověření signály *a_pub_test* a *key_test* obsahují exkluzivní součet s hodnotami z výše zmíněného RFC a signály *A_public* a *Key*. Tudíž v případě správného fungování komponenty musí nabývat pouze hodnoty 0. Je možné vidět na obrázku č. 9 níže.



Obrázek 9: Simulace komponenty DH

Výpočet trval 496 819 taktů. A stejně jako v případě RSA, ani tato hodnota není konečné číslo, záleží zvolených číslech.

3.5 SHA

Secure Hash Algorithm, jak z názvu vyplývá, jedná se o algoritmus pro vypočítání hashe. Jak již bylo dříve zmíněno, hash je hodnota o pevně dané velikosti. Délka vstupu nemá vliv na délku hashe. Hashovací funkce se považuje za bezpečnou, je-li jednosměrná, tudíž z hashe nelze (přijatelně jednoduchým způsobem) určit původní vstupní řetězec. A měla by též splňovat podmínku, že pro dva různé vstupní řetězce neexistuje stejný hash.

Standard SHA publikoval ve svém dokumentu FIPS 180 americký národní úřad standardů a technologií NIST v roce 1993. SHA bylo určeno hlavně pro použití s autentizací zpráv (DSA podpisy). Algoritmus je schopen zpracovávat zprávy po blocích o délce 512 bitů. Zpráva však nesmí přesáhnout velikost 2^64 bitů. Výstupem je hash o délce 160 bitů. V roce 1995 byl publikován standard SHA-1, který revizí toho z roku 1993 (SHA-0). SHA-1 přidává některé nové operace, jež zlepšují bezpečnost celého algoritmu. Tento algoritmus taktéž zpracovává 512bitové bloky dat a tvoří 160bitový hash.

V roce 2002 byly publikovány nové algoritmy, označovány někdy jako SHA-2. Oficiálně jsou to však SHA-256, SHA-384 a SHA-512. V roce 2008 se přidal další algoritmus, SHA-224. Tyto algoritmy pak generují hashe o délkách 160, 224, 256, 384 a 512 bitů. Pro první 3 je velikost vstupního bloku dat 512bitů, pro SHA-384 a SHA-512 je to 1024 bitů. Pro výpočet hashe se používá sled logických funkcí. Pro SHA-1 je to 80 funkcí, které jsou rozděleny na 4 a mění se každých 20 iterací. Pro zbylé algoritmy je to jen 6 funkcí, které ale už nejsou řazeny sekvenčně, na rozdíl od SHA-1. Společně s logickými funkcemi jsou definovány i konstanty, jež je při výpočtu třeba. Ty jsou společné pro hashe SHA-224 a SHA-256, a pak stejné pro SHA-384 a SHA-512. Stejně tak i postup výpočtu pro tyto 2 skupiny. Rozdíl mezi SHA-224 a SHA-256, a stejně tak i

SHA-384 a SHA-512, je ten, že na začátku jsou použity jiné inicializační hodnoty a výsledek (256bitový hash, případně 512bitový hash) je ořezán na délku 224 bitů, případně 384 bitů.

V roce 2015 přibyly navíc variace SHA-512/224 a SHA-512/256. Tyto 2 nové variace pracují obdobně jako SHA-224 a SHA-384. Je použit algoritmus SHA-512 s rozdílnými inicializačními hodnotami, pro každou variaci, a výstup je následně ořezán na požadovanou délku. Postupy výpočtů společně s funkcemi i konstantami jsou definovány v dokumentu FIPS-180-4 [44].

Co se týče zarovnání vstupní zprávy, je definován jeden postup, jenž využívají všechny algoritmy výše definované. Zarovnání na blokovou délku se provádí tak, že za poslední bit vstupního řetězce, řekněme že délky 40bitů, se připojí bit s hodnotou ‚1'. Velikost vstupního řetězce (bez přidané hodnoty ‚1') je reprezentována 64bitovou hodnotou, která je umístněná na konci posledního bloku. A prostor mezi vstupním řetězcem a bity hodnoty reprezentující velikost je vyplněn nulami. Pro velikost bloku 1024 bitů je tato hodnota, udávající velikost, reprezentována 128 bity.

V roce 2015 taktéž vznikla další generace, SHA-3. Tento algoritmus se od předchozích generací liší. Je založena na algoritmu Keccak, který má odlišnou vnitřní strukturu. SHA-3 není zatím součástí TLS, tudíž nebude dále zmiňována ani popisována. Informace lze najít v dokumentu FIPS 202 [45].

3.5.1 Implementace SHA-256 ve VHDL

Pro implementaci bylo použito již existující řešení, opět z webu www.github.com. Autorem je uživatel Kubes [46], ze zdrojového kódu je to Danny Savory. Přiloženou dokumentací k tomuto dílu je pouze oficiální dokument FIPS 180-4. Proto byla provedena analýza zdrojové kódu k určení chování aplikace, včetně jejího rozhraní, které je popsáno v následující tabulce.

Ovládání aplikace není složité. Rozhraní komponenty je v tabulce č. 7 níže. Generická proměnná *RESET_VALUE* nastavuje aktivní hodnotu, při které se provádí reset (v mém kódu je použito aktivní hodnoty ‚1'). Pro spuštění aplikace je potřeba nastavit *n_blocks* na celkový počet bloků, do kterých je rozdělena celková zpráva. Toto rozdělení musí podléhat dokumentu FIPS 180, konkrétně sekci „message padding". Dále se naplní proměnná *msg_block_in* a celý proces se spustí aktivací signálu *data_ready*. Data v proměnné *msg_block_in* musí zůstat platné i celý následující hodinový takt po aktivaci signálu *data_ready*. Pokud se zpráva sestává pouze z jednoho bloku, po dokončení výpočtu je signalizováno signálem *finished*. Pokud je použito více bloků, tak signalizace pro naplnění dalším blokem zprávy je pomocí signálu *next_data*. Po aktivaci tohoto signálu se naplní proměnná *msg_block_in* dalšími daty je současně aktivován signál *data_ready*. Pro vstupní proměnnou platí stejná pravidla jako při prvotní spuštěním aplikace. Musí být data validní i následující hodinový takt.

SHA-256 (sha_256_core.vhd)			
Název	Směr	Bitová délka	Popis
clk	in	1	hodinový signál
rst	in	1	signalizace - reset
data_ready	in	1	signalizace - start/načtení dalčího bloku dat
n_blocks	in	-	signalizace - celkový počet bloků dat
msg_block_in	in	512	vstupní data
finished	out	1	signalizace - konec
next_data	out	1	signalizace - výzva dalšího bloku dat
data_out	out	256	výstupní data

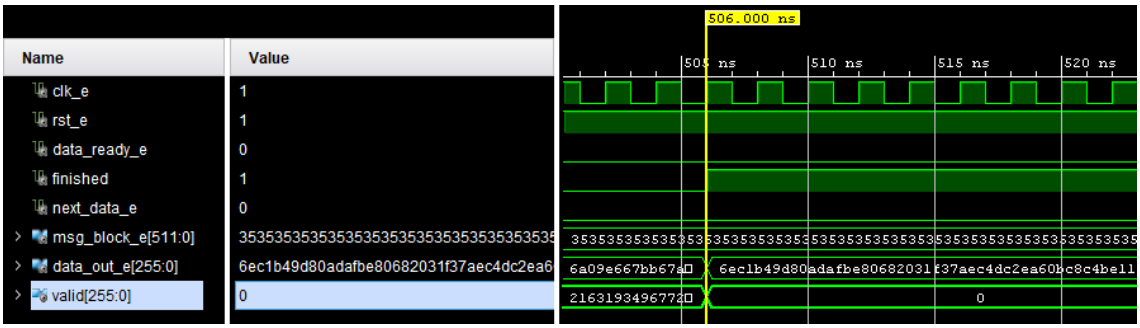
Tabulka 7: Rozhraní komponenty SHA-256

3.5.2 Testování

Pro ověření byla použita online aplikace na webu [47]. Pro test byly použity následující data: řetězec obsahující 55krát hodnotu „5“. Ověření je provedeno pomocí signálu *valid* (obrázek č. 10), jehož hodnota je dána sexorováním výstupu z komponenty a výsledku z online aplikace, viz obrázek č. 10 níže. Výpočet trval 201 taktů. Zabraných zdrojů je:

- 5939 LUT tabulek
- 3222 Registrů
- 256 F7 multiplexů
- 128 F8 multiplexů

Aplikace je schopna běžet na frekvenci 20MHz.



Obrázek 10: Simulace komponenty SHA-256

3.6 AES-GCM

AES (Advanced Encryption System) je kryptografický systém, jenž publikoval taktéž NIST – dokument FIPS 197 [48] z roku 2001. Jedná se o standard používaný v amerických úřadech. Jádrem AES je bloková šifra Rijndael podporující vstupní data o velikosti bloku 128 bitů a délky klíčů o variantách 128, 196 a 256 bitů. Tento kryptosystém jsem popisoval ve své bakalářské práci [49], a proto následující popis bude stručný.

Bloková šifra Rijndael šifruje/dešifruje vstupní data, rozdělená do bajtové mřížky o rozměrech 4x4B, v jednotlivých průchodech/iteracích tzv. kolech (rounds). Počet kol je dán velikostí klíče na 10, 12 a 14 kol. Na úrovní jednoho kola se jedná o celkem 4 transformační funkce – *SubBytes*, *ShiftRows*, *MixColumns* a *AddRoundKey*. *SubBytes* po bajtech provádí substituci s tabulkou *Sbox*, která má předem definovaný obsah dle matematického vztahu, takže je možné substituční hodnoty i vypočítat. *ShiftRows* provádí rotaci bajtů vlevo v jednotlivých řádcích mřížky 4x4B, podle daných pravidel. *MixColumns* provádí výpočty s jednotlivými bajty v rámci sloupce mřížky. Poslední funkce *AddRoundKey* xoruje data s odvozeným klíčem. Odvozený klíč se získává z hlavního klíče pomocí expanzní funkce, která je součástí AES. Tímto je zajištěno, že v každém kole je použit jiný klíč. A protože každé kolo musí mít svůj vlastní klíč, expanze generuje 10, 12 nebo 14 128bitových klíčů v závislosti na délce hlavního klíče. Dešifrování se provádí použitím inverzních transformačních funkcí a odvozených klíčů v obráceném pořadí než při šifrování. Každá transformační funkce má svou inverzní funkci. A protože v TLS 1.3 je použito této blokové šifry pouze v režimech odvozených ze základního režim CTR, není dešifrování vůbec zapotřebí. Režim GCM byl popsán v 1. kapitole.

3.6.1 Implementace AES-GCM ve VHDL

Pro implementaci části AES bylo již existující řešení, jehož autorem je Ing. David Smékal. Autorovo řešení pracuje s délkou klíče 256 bitů, ale je možná úprava i pro podporu kratších délek. Tato implementace dokáže používá pipelining, což znamená, že algoritmus je dělen na více paralelních větvích a dokáže tak zpracovávat větší množství dat. Rozhraní této komponenty je v tabulce č. 8.

AES-256 (Encryption.vhd)			
Název	Směr	Bitová délka	Popis
RX	in	128	vstupní data
KEY	in	1920	klíč po expanzi
TX	out	128	výstupní data
VLD	in	1	signalizace platných vstupních dat
RDY	out	1	signalizace - konec
CLK	in	1	hodinový signál
RST	in	1	reset

Tabulka 8: Rozhraní AES

GCM část nebyla implementována.

3.6.2 Testování

Pro testování byly zvoleny testovací vektory z dokumentu [48]. Výsledek simulace je na obrázku č. 11.

Obrázek 11: Simulace AES

Po syntéze této komponenty byly zabrány tyto zdroje:

- 10129 LUT
- 898 Registrů
- 3584 F7 MUX
- 1792 F8 MUX

3.7 HMAC-SHA256

Jak již bylo v úvodu zmíněno, algoritmus HMAC slouží k výpočtu kontrolní hodnoty zprávy, jenž je použita k zaručení integrity zprávy. Používá se při tom hashovací funkce, v této variantě je to hashovací funkce SHA-256. HMAC je definován v RFC 2104 [50] z roku 1997.

Definice výpočtu algoritmu je relativně jednoduchá. Jedná se o dvojité hashování zvolenou hashovací funkcí – v tomto případě se bude jednat o SHA-256. Kromě vstupních dat a klíče je použito výplně, což je statická hodnota. Taktéž je tu přítomna hodnota velikosti bloku dat. Obecně může být různá, ale opět při použití SHA-256 je to konkrétní hodnota 512 bitů.

Na začátku, pokud je klíč kratší než délka bloku dat, je klíč doplněn nulami na délku bloku. Pokud je klíč delší, provede se hash tohoto klíče a následně se doplní opět nulami, pokud je to třeba. Následně se klíč xoruje s vnitřní výplní, která je tvořena opakující se hodnotou 0x36 a má délku jako blok dat. K tomuto výsledku se přidají vstupní data a provede se hash tohoto celku. V dalším kole se klíč, který je již zarovnaný podle předchozích pravidel, xoruje s vnější výplní, která má stejné vlastnosti jako vnitřní výplň, ale je použita hodnota 0x5c. K tomuto výsledku se připojí vzniklý hash z předchozího kola a provede se opět hash tohoto celku. Výsledný hash je pak finálním výsledkem celého algoritmu HMAC.

3.7.1 Implementace HMAC-SHA256 ve VHDL

Pro implementaci HMAC-SHA256 byla použita stejná implementace hashovací funkce SHA-256, jenž byla zmíněná v kapitole 3.4. Avšak byla provedena nepatrná změna v této implementaci, a to ve stavovém automatu. Jedná se o akci, jenž se provádí při dokončení hashování. V původním verzi zůstal stavový automat ve stavu „dokončeno“ a pro nové spuštění bylo potřeba provést reset. Toto chování bylo změněno a nyní přechází automat do stavu reset automaticky. Nároky na vstupní klíč byly změněny a nyní je vyžadováno, aby klíč vstupoval do komponenty už zarovnán na 512 bitů, jak bylo zmíněno dříve v textu.

Princip fungování celé komponenty HMAC je podobný SHA. Rozhraní komponenty se nachází v tabulce č. 9 níže. Zarovnání a předávání dat na vstupu je delegované přímo na komponentu SHA, a proto platí pro vstupní data stejné pravidla. Jako první vstupuje do SHA upravený klíč, následně je provedena delegace. Vstupní data je nutno zaslat pouze na výzvu signálem *next_data* a následně předání vstupních dat je signalizováno pomocí *data_ready*. Celkový počet bloků dat je také nutno předem znát a uvést. Navíc přibyla nutnost signalizovat poslední blok vstupních dat signálem *last_msg_block*. Od tohoto posledního předání dat probíhá výpočet HMAC samostatně. Jeho dokončení je signalizováno signálem *hmac_rdy*. Pro opětovné spuštění je potřeba reset.

HMAC-SHA256 (hmac_sha256.vhd)			
Název	Směr	Bitová délka	Popis
clk	in	1	hodinový signál
rst	in	1	signalizace - reset
start	in	1	signalizace - start
key	in	512*1	vstupni parametr - klíč
last_msg_block	in	1	signalizace - poslední blok dat
msg_block_in	in	512	vstupní data
hmac	out	256	výstupní HMAC hash
hmac_rdy	out	1	signalizace - konec
next_data	out	1	signalizace - výzva pro další blok dat
data_ready	in	1	signalizace - načtení dalšího bloku dak
n_blocks	in	-	signalizace - celkový počet bloků dat

*1 - bitová délka je upravována parametrem n_size na konkrétní hodnotu, v tomto případě byla použita délka 512 bitů

Tabulka 9: Rozhraní komponenty HMAC

Zabraných zdrojů HMAC komponenty je následovný:

- 7252 LUT tabulka
- 5314 registrů
- 256 F7 multiplexů
- 128 F8 multiplexů

Běžet pak dokáže na frekvenci 20 MHz a výpočet zabere nejmíň 812 taktů. Simulace

komponenty je vyobrazena na obrázku č. 12 níže.

Obrázek 12: Simulace HMAC-SHA256

4 ZÁVĚR

Byla provedena částečná implementace částí protokolu. Implementovány a odsimulovány byly následující části:

- RSA – v základní podobě, statické klíče apod.
- DH – v základní podobě, statické klíče, nejsou použity grupy
- SHA-256
- AES-GCM – implementována šifrovací část AES, pro GCM stačí pouze tato část
- HMAC-SHA256 – vstupní klíč musí být před použitím upraven

U všech zmíněných komponent bylo provedeno ověření funkčnosti a validita výpočtu s pozitivním výsledkem. Přehled zabraných zdrojů po simulaci včetně časování je v následující tabulce č. 10. Nutno podotknout, že hodnoty jsou orientační ve smyslu, že kromě toho, že lze u komponent RSA a DH měnit bitovou délku operandů a tím měnit i časové nároky, čas výpočtu zabrané zdroje, doba výpočtu je závislá na zvolených hodnotách operandů, i když jejich bitová délka bude stále stejná.

Přehled zabraných zdrojů jednotlivých komponent					
Komponenta	LUT	REG	F7 MUX	F8 MUX	Poznámky
DH	53007	67767	0	0	pro 2048bitové operandy
RSA	66833	61680	1	0	pro 2048bitové operandy
SHA-256	5939	3222	256	128	
HMAC-SHA256	7252	5314	256	128	
AES128	10129	898	3584	1792	

Tabulka 10: Přehled zabraných zdrojů jednotlivých komponent

Ačkoli nebyla provedena kompletní implementace ani na simulační úrovni, tak jsem postupným implementování výše zmíněných komponent a zkoumáním této problematiky dospěl k závěru, že implementace celého TLS na platformě FPGA, a jí podobných, není vhodná. Navzdory tomu, že by bylo možné tuto implementaci dokončit aspoň v omezené míře. K tomuto závěru mě vedlo několik věcí.

Zabrané zdroje. Ačkoliv výkonnější FPGA čipy obsahují více zdrojů, tak implementace, byť nekompletní, zabírá mnoho zdrojů. Příkladem může být právě komponenta DH, která sdílí část RSA, takže se to týká i téhle komponenty. V reálném použití existují různé bitové délky operandů, tj. délky 512, 1024, 2048, 4096, 8192 a 16384. To znamená více zabraných zdrojů. Kvůli principu fungování technologie FPGA bylo provedeno několik variací syntéz k určení koeficientu růstu zabraných zdrojů, protože s rostoucí implementací dochází k větší náročnosti na provedení samotné implementace a na čipu tak zabírá víc zdrojů, neboť některých zdrojů je na dané ploše čipu omezený počet, a tak musí být použity zdroje z jiných částí čipu. Minimálně tím narůstá délka provedených spojů, a tudíž i časové nároky. Také počet možných spojů není nekonečný.

Syntézy byly prováděny na 2 různých čipech a syntézních strojích. Výsledky se přibližně shodovaly a pro odhad jsou dostačující. Z důvodu časové náročnosti

implementace byla provede syntéza pouze po hodnotu 4096 bitů, zbytek je extrapolace. Jako referenci budeme uvažovat FPGA čip Virtex UltraScale+ xcvu9p se zdroji 1 182 240 LUTs, 2 364 480 Regs (vývojové prostředí má v seznamu čip, který má dvojnásobné hodnoty). Výsledky jsou v tabulce č. 11. Navržené implementace nebyly odzkoušeny na reálném hardwaru.

DH - zabrané zdroje podle délky operandů - syntéza					
bitová délka	LUT	REG	F7 MUX	poměr LUT	poměr REG
16	774	742		0,07	0,03
32	1309	1383		0,11	0,06
64	2478	2666		0,21	0,11
128	4605	5231		0,39	0,22
256	8989	10372	512	0,76	0,44
512	16615	20624		1,41	0,87
1024	32865	41095		2,78	1,74
2048	53007	67767		4,48	2,87
4096	107707	135455	4096	9,11	5,73
8192	215414	270910		18,22	11,46
16384	430828	541820		36,44	22,91
MAX LUT	1182240				
MAX REG	2364480				

Tabulka 11: Zabrané zdroje DH

Podobné hodnoty dostaneme i pro komponentu RSA, byť jsou o něco menší. Pro zajímavost lze provést srovnání. Pokud by celá implementace TLS běžela jako aplikace ve firmwaru pro FPGA síťové karty, jenž byl použit v práci [49], tak samotný tento firmware zabíral přibližně 127 000 LUTs a 117 000 Regs.

Dalším důvodem pro nevhodnou platformu jsou instance jednotlivých komponent. V FPGA nelze měnit firmware za běhu. Sice je komponenta DH napsaná s podporou různých délek, avšak tato délka musí být zvolena na před samotnou implementací a generování firmwaru. Proto by musela být zvolena hned na začátku maximální podporovaná délka i za cenu, že v provozu bude použita kratší bitová délka.

Také musí být instanciován celý program. To znamená veškeré podporované komponenty najednou. I když není možné je použít zároveň. Například šifra CHACHA20 bude zabírat zdroje i když se bude aktivně používat AES. V tomto je hlavní rozdíl mezi platformou FPGA a procesory, jež jsou použity v klasických počítačích a serverech. Tyto procesory jsou tvořeny určitými prvky, jejichž zapojení se nijak nemění. Program pak zpracovávají sekvenčně, řádek po řádku. A také dokážou v programu libovolně skákat dopředu a dozadu. To je výrazná výhoda oproti FPGA, neboť tohoto se využívá v rozhodovací logice if-else. Procesor se v takovém případě přesune v programu na jiné místo, FPGA musí fyzicky vyhradit zdroje pro obě varianty, a navíc je taková implementace většinou složitá a musí se složitě hlídat různé signály a jejich ovládání, neboť vše probíhá paralelně. Z tohoto důvodu je taktéž nevhodné použít FPGA pro

komplikovanou řídící logiku, jakou má TLS, byť by bylo fyzicky možné ji implementovat.

S instancemi souvisí ještě jeden problém. I kdyby se podařilo implementovat TLS takovým způsobem, aby v čipu zabíralo zdroje v rozumné míře (nebyla provedena optimalizace), tak jedna instance celého TLS dokáže obsloužit pouze jedno spojení. Pro další simultánní spojení je potřeba fyzicky umístit další kopii TLS na čip FPGA vedle stávající implementace. Tudíž se dostaneme na podporu pouze několika málo paralelních spojení. Význam akcelerované FPGA síťové karty pak postrádá smysl. Částečným řešením těchto problémů by mohl být kompromis mezi FPGA a CPU, kdy by se řídící logika převedla do softwaru pro CPU. Software by pak přes vnitřní rozhraní dával povely aplikaci uvnitř FPGA. Tato aplikace by sloužila pouze jako kalkulačka pro náročné výpočetní operace. A software pro CPU by řídil stav TLS a dokázal by hravě obsloužit, oproti FPGA, libovolný počet spojení.

5 SEZNAM ZKRATEK

3DES-EDE – Triple Data Encryption Standard – Encryption Decryption Encryption

AEAD – Authenticated Encryption with Associated Data

AES – Advanced Encryption Standard

ASIC – Application-Specific Integrated Circuit

CBC – Cipher Block Chaining

CCM – Counter with CBC-MAC

CPLD – Complex Programmable Logic Device

CPU – Central Processing Unit

CTR - Counter

DES - Data Encryption Standard

DH – Diffie-Hellman

DoD – Department of Defense

DSA – Digital Signature Algorithm

DSS – Digital Signature Standard

DTLS – Datagram Transport Layer Security

ECB – Electronic Code Book

EEPROM – Electrically Erasable Programmable Read-Only Memory

EPROM - Erasable Programmable Read-Only Memory

FIPS – Federal Information Processing Standards

FPGA – Field Programmable Gate-Array

GCM – Galois Counter Mode

GDDR – Graphic Double Data Rate

HKDF – Hash-based Key Derivation Function

HMAC – Hash-based Message Authentication Code

IDEA – International Data Encryption Algorithm

IETF – Internet Engineering Task Force

IP – Internet Protocol

ISO/OSI – International Organisation for Standardization / Open Systems Interconnection

LSB – Least Significant bit

LUT – Look-Up Table

MAC – Message Authentication Code

MD5 – Message Diggest 5

NFB – Netcope FPGA Board

NIST – National Institude of Standards and Technology

NSA – National Security Agency

OFB – Output Feedback

PKI – Public Key Infrastructure

PLD – Programmable Logic Device

PRNG – Pseudorandom Number Generator

QSFP – Quad Small Form-factor Pluggable

RC2 – Rivest Cipher 2

RC4 – Rivest Cipher 4

RFC – Request for Comments

SHA – Secure Hash Algorithm

SPLD – Simple Programmable Logic Device

SRAM – Static Random Access Memory

SSL – Secure Socket Layer

TCP – Transmission Control Protocol

TLS – Transport Layer Security

UDP – User Datagram Protocol

VHDL – Very High Speed Integrated Circuits Hardware Description Language

6 LITERATURA

[1] FERGUSON, Niels a Bruce SCHNEIER. *Practical cryptography*. New York: Wiley, c2003. ISBN 0-471-22357-3.

[2] MAO, Wenbo. *Modern Cryptography: Theory and Practice*. 5th ed. Upper Saddle River: Prentice Hall, 2004. ISBN 0-13-066943-1.

[3] ASHENDEN, Peter J. *The student's guide to VHDL*. San Francisco: Morgan Kaufman Publishers, c1998. ISBN 1-55860-520-7.

[4] PINKER, Jiří a Martin POUPA. *Číslicové systémy a jazyk VHDL*. Praha: BEN - technická literatura, 2006. ISBN 80-7300-198-5.

[5] BOYD, Colin a Anish MATHURIA. *Protocols for authentication and key establishment*. New York: Springer, c2003. ISBN 3-540-43107-1.

[6] BURDA, Karel. *Aplikovaná kryptografie*. Brno: VUTIUM, 2013. ISBN 978-80-214-4612-0.

[7] *Wolfram Alpha* [online]. [cit. 2019-12-21]. Dostupné z: https://en.wikipedia.org/wiki/Wolfram_Alpha

[8] CARTER, Stephen, Luis GALLET a Jacob BARNETT. RSA-Encryption. *Github* [online]. 2016 [cit. 2019-12-21]. Dostupné z: https://github.com/scarter93/RSA-Encryption/blob/master/montgomery_multiplier_tb.vhd

[9] KeyLength. *KeyLength* [online]. [cit. 2019-12-21]. Dostupné z: https://www.keylength.com/

[10] BARKER, Elaine a Allen ROGINSKY. Transitioning the Use of Cryptographic Algorithms and Key Lengths. *KeyLength* [online]. [cit. 2019-12-21]. Dostupné z: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-131Ar2.pdf

[11] SAHU, Sushanta Kumar a Manoranjan PRADHAN. *FPGA Implementation of RSA Encryption System* [online]. 2011 [cit. 2019-12-21]. Dostupné z: https://pdfs.semanticscholar.org/2566/3e2c848665e069cebf829d445e1235068640.pdf

[12] ANAND, Ankit a Pushkar PRAVEEN. *Implementation of RSA Algorithm on FPGA* [online]. 2011 [cit. 2019-12-21]. Dostupné z: https://www.ijert.org/research/implementation-of-rsa-algorithm-on-fpga-IJERTV1IS5454.pdf

[13] MONTGOMERY, Peter L. *Modular Multiuplication Without Trial Division* [online]. 2007 [cit. 2019-12-21]. Dostupné z: https://web.itu.edu.tr/~orssi/dersler/cryptography/Montgomery.pdf

[14] ŠKOBIC, Velibor, Branko DOKIC a Željko IVANOVIC. *FPGA Implementation of Montgomery Modular Multiplier* [online]. 2014 [cit. 2019-12-21]. Dostupné z: http://jds.elfak.ni.ac.rs/ssss2014/proceedingsAndPublication/separated%20chapters/25%20FPGA%20Implementation%20of%20Montgomery%20Modular%20Multiplier.pdf

[15] FPGA síťové karty. *Liberouter* [online]. 2014 [cit. 2019-12-21]. Dostupné z: https://www.liberouter.org/technologies/cards/

[16] RESCORLA, E. The Transport Layer Security (TLS) Protocol Version 1.3. *IETF* [online]. [cit. 2019-12-21]. Dostupné z: https://www.liberouter.org/technologies/cards/

[17] ANDREI, Victor. *FPGA Design Flow* [online]. 2013 [cit. 2019-12-21]. Dostupné z: https://indico.desy.de/indico/event/7001/session/0/contribution/1/material/slides/0.pdf

[18] *BittWare* [online]. 2013 [cit. 2019-12-21]. Dostupné z: https://www.bittware.com/

[19] *NFB-200G2QL* [online]. [cit. 2019-12-21]. Dostupné z: https://www.netcope.com/getattachment/bb2b8efa-9925-438d-b895-897d7c1e4745/NFB-200G2QL-product-brief.aspx

[20] SSL/TLS offloading. *Fortinet* [online]. [cit. 2019-12-21]. Dostupné z: https://help.fortinet.com/fos50hlp/56/Content/FortiOS/fortigate-load-balancing/ldb-ssl-tls-offload.htm

[21] SSL/TLS HW implementace. *Helion* [online]. [cit. 2019-12-21]. Dostupné z: https://www.heliontech.com/ssl.htm

[22] SSL offloading. *Comodo* [online]. [cit. 2019-12-21]. Dostupné z: https://securebox.comodo.com/ssl-sniffing/ssl-offloading/

[23] *SSL offload* [online]. [cit. 2019-12-21]. Dostupné z: https://avinetworks.com/glossary/ssl-offload/

[24] Atmel Hardware-TLS. *Microchip* [online]. [cit. 2019-12-21]. Dostupné z: http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-45176-Hardening-Transport-Layer-Security-for-IoT_Flyer.pdf

[25] DWORKIN, Morris. Recommendation for Block Cipher Modes of Operation: The CCM Mode for Authentication and Confidentiality. *NIST* [online]. 2004 [cit. 2019-12-21]. Dostupné z: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38c.pdf

[26] DWORKIN, Morris. Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC. *NIST* [online]. 2007 [cit. 2019-12-21]. Dostupné z: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38d.pdf

[27] MCGREW, David A. a John VIEGA. *The Security and Performance of the Galois/Counter Mode (GCM) of Operation* [online]. 2007 [cit. 2019-12-21]. Dostupné z: https://eprint.iacr.org/2004/193.pdf

[28] FRUHWIRTH, Clemens. *New Methods in Hard Disk Encryption* [online]. 2005 [cit. 2019-12-21]. Dostupné z: https://clemens.endorphin.org/nmihde/nmihde-A4-os.pdf

[29] KELSEY, John, Bruce SCHNEIER a David WAGNER. *Key-Schedule Cryptoanalysis of IDEA, G-DES, GOST, SAFER and Triple-DES* [online]. [cit. 2019-12-21]. Dostupné z: https://www.schneier.com/academic/paperfiles/paper-key-schedule.pdf

[30] KRAWCZYK, H. *HMAC-based Extract-and-Expand Key Derivation Function (HKDF)* [online]. 2010 [cit. 2019-12-21]. Dostupné z: https://www.rfc-editor.org/rfc/pdfrfc/rfc5869.txt.pdf

[31] Escrowed Encryption Standard. *NIST* [online]. 1994 [cit. 2019-12-21]. Dostupné z: https://csrc.nist.gov/csrc/media/publications/fips/185/archive/1994-02-09/documents/fips185.pdf

[32] FREIER, A. a P. KARLTON. The Secure Layer (SSL) Protocol Version 3.0. *IETF* [online]. 2011 [cit. 2019-12-21]. Dostupné z: https://csrc.nist.gov/csrc/media/publications/fips/185/archive/1994-02-09/documents/fips185.pdf

[33] Fortinet. *Fortinet* [online]. [cit. 2020-05-01]. Dostupné z: https://www.fortinet.com/

[34] Kemp Technologies. *Kemp Technologies* [online]. [cit. 2020-05-01]. Dostupné z: https://kemptechnologies.com/

[35] Intel Corporation. *Intel Corporation* [online]. [cit. 2020-05-01]. Dostupné z: https://www.intel.com/content/www/us/en/homepage.html

[36] Helion Technology. *Helion Technology* [online]. [cit. 2020-05-01]. Dostupné z: https://www.heliontech.com/index.htm

[37] Silicom. *Silicom* [online]. [cit. 2020-05-01]. Dostupné z: https://www.silicom-usa.com/cats/fpga-based-cards/100-gigabit-fpga-cards/

[38] KRAWCZYK, H. a P. ERONEN. RFC 5869. *RFC 5869* [online]. 2010, Květen [cit. 2020-05-01]. Dostupné z: https://tools.ietf.org/html/rfc5869

[39] Square and Multiply. *Practical Networking* [online]. [cit. 2020-05-01]. Dostupné z: https://www.practicalnetworking.net/stand-alone/square-and-multiply/

[40] Montgomery Multiplication. *Colin and Margaret* [online]. [cit. 2020-05-01]. Dostupné z: https://colinandmargaret.co.uk/Research/Mont_Mult_2ndEd_v4.pdf

[41] GILLMOR, D. RFC 7919. *RFC 7919* [online]. Srpen [cit. 2020-05-01]. Dostupné z: https://tools.ietf.org/pdf/rfc7919.pdf

[42] *Diffie-Hellman* [online]. 2016 [cit. 2020-05-01]. Dostupné z: https://www.math.brown.edu/~jhs/MathCrypto/SampleSections.pdf

[43] LEPINSKI, M. a S. KENT. *RFC 5114* [online]. 2008, Leden [cit. 2020-05-01]. Dostupné z: https://tools.ietf.org/html/rfc5114

[44] *FIPS 180-4* [online]. 2015, Srpen [cit. 2020-05-01]. Dostupné z: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf

[45] *FIPS 202* [online]. 2015, Srpen [cit. 2020-05-01]. Dostupné z: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf

[46] KUBES a Danny SAVORY. *SHA-256* [online]. 2018, Květen 27 [cit. 2020-05-01]. Dostupné z: https://github.com/kubes/SHA-256

[47] *SHA-256 online* [online]. 2018 [cit. 2020-05-01]. Dostupné z: https://emn178.github.io/online-tools/sha256.html

[48] *FIPS 197* [online]. 2001, Listopad 26. [cit. 2020-05-01]. Dostupné z: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf

[49] ZUGÁREK, Adam. *Implementace šifrovacích algoritmů na platformě FPGA* [online]. 2017, Červen 19. [cit. 2020-05-01]. Dostupné z: https://www.vutbr.cz/studenti/zav-prace?zp_id=101867

[50] KRAWCZYK, H., M. BELLARE a R. CANETTI. *RFC 2104* [online]. 1997, Únor [cit. 2020-05-01]. Dostupné z: https://tools.ietf.org/html/rfc2104

[51] RESCORLA, E. *RFC 8446* [online]. 2018, Srpen [cit. 2020-05-01]. Dostupné z: https://tools.ietf.org/html/rfc8446

[52] LEPINSKI, M. a S. KENT. *RFC 5114* [online]. 2018, Leden [cit. 2020-05-01]. Dostupné z: https://tools.ietf.org/html/rfc5114

[53] ADRIAN, D., K. BHARGAVAN, Z. DURUMERIC a P. *Imperfect Forward Secrecy:How Diffie-Hellman Fails in Practice* [online]. [cit. 2020-05-01]. Dostupné z: https://weakdh.org/imperfect-forward-secrecy-ccs15.pdf

[54] *Enabling Perfect Forward Secrecy* [online]. [cit. 2020-05-01]. Dostupné z: https://www.digicert.com/kb/ssl-support/ssl-enabling-perfect-forward-secrecy.htm

[55] DIERKS, T. a E. *RFC 5246* [online]. [cit. 2020-05-01]. Dostupné z: https://tools.ietf.org/html/rfc5246

[56] DIERKS, T., C. ALLEN. *RFC 2246* [online]. Leden [cit. 2020-05-01]. Dostupné z:

https://tools.ietf.org/pdf/rfc2246.pdf

[57] SHEFFER, Y., R. HOLZ. *RFC 7525* [online]. 2015, Květen [cit. 2020-05-01]. Dostupné z: https://tools.ietf.org/html/rfc7525

[58] ERONEN, P., H. TSCHOFENIG. *RFC 4279* [online]. 2015, Prosince [cit. 2020-05-01]. Dostupné z: https://tools.ietf.org/pdf/rfc4279.pdf

[59] MCGREW, D. *RFC 5116* [online]. 2008, Leden [cit. 2020-05-01]. Dostupné z: https://tools.ietf.org/pdf/rfc5116.pdf

[60] KELLY, S. a S. FRANKEL. *RFC 4868* [online]. 2008, Květen [cit. 2020-05-01]. Dostupné z: https://tools.ietf.org/html/rfc4868

[61] CHEN, Lily. *Cryptographic Standards and Guidelines* [online]. 2007 [cit. 2020-05-01]. Dostupné z: https://csrc.nist.gov/projects/cryptographic-standards-and-guidelines/example-values

[62] DIERKS, T. a E. RESCORLA. *RFC 4346* [online]. 2006, Duben [cit. 2020-05-01]. Dostupné z: https://tools.ietf.org/pdf/rfc4346.pdf