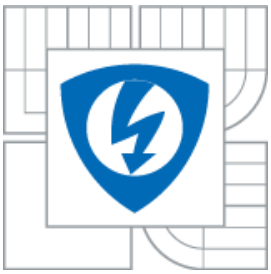
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ
ÚSTAV MIKROELEKTRONIKY

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

DEPARTMENT OF MICROELECTRONICS

PROGRAM PRO TESTOVÁNÍ RYCHLOSTI ŠIFROVACÍCH ALGORITMŮ

PROGRAM FOR TESTING OF SPEED OF ENCRYPTION ALGORITHMS

DIPLOMOVÁ PRÁCE
DIPLOMA THESIS

AUTOR PRÁCE — Bc. LUKÁŠ VOLF
AUTHOR

VEDOUCÍ PRÁCE — ING. JAROSLAV KADLEC , Ph.D.
SUPERVISOR

BRNO, 2010

Zadání

Vytvořte program pro testování rychlosti šifrovacích algoritmů. Vytvořený program bude testovat specifické vlastnosti jednotlivých šifrovacích protokolů a výsledky testování bude ukládat pro pozdější zpracování a statisticky vyhodnocovat do uživatelsky přehledných reportů. Program bude vytvořen v programovacím jazyku C#. Výsledná diplomová práce bude obsahovat přiložené CD s Vámi vytvořeným testovacím programem, výsledky Vašich testování šifrovacích algoritmů a zdrojovými kódy.

Licenční smlouva
poskytovaná k výkonu práva užít školní dílo

uzavřená mezi smluvními stranami:

1. Pan/paní

Jméno a příjmení: Lukáš Volf

Bytem: Polička, T. Novákové 397, 572 01

Narozen/a (datum a místo): 11.5.1982, Polička

(dále jen „autor“)

a

2. Vysoké učení technické v Brně

Fakulta elektrotechniky a komunikačních technologií

se sídlem Údolní 244/53, 602 00 Brno

jejímž jménem jedná na základě písemného pověření děkanem fakulty:

Prof. Ing. Vladislav Musil, CSc.

(dále jen „nabyvatel“)

Čl. 1
Specifikace školního díla

1. Předmětem této smlouvy je vysokoškolská kvalifikační práce (VŠKP):
   - □ disertační práce
   - □ diplomová práce
   - □ bakalářská práce
   - □ jiná práce, jejíž druh je specifikován jako .....

   (dále jen VŠKP nebo dílo)

Název VŠKP: Program pro testování rychlosti šifrovacích algoritmů

Vedoucí/ školitel VŠKP: Ing. Jaroslav Kadlec, Ph.D.

Ústav: Ústav mikroelektroniky

Datum obhajoby VŠKP: _____

VŠKP odevzdal autor nabyvateli v:

- □ tištěné formě - počet exemplářů ……..
- □ elektronické formě - počet exemplářů ……..

2. Autor prohlašuje, že vytvořil samostatnou vlastní tvůrčí činností dílo shora popsané a specifikované. Autor dále prohlašuje, že při zpracovávání díla se sám nedostal do rozporu s autorským zákonem a předpisy souvisejícími a že je dílo dílem původním.
3. Dílo je chráněno jako dílo dle autorského zákona v platném znění.
4. Autor potvrzuje, že listinná a elektronická verze díla je identická.

Článek 2
Udělení licenčního oprávnění

1. Autor touto smlouvou poskytuje nabyvateli oprávnění (licenci) k výkonu práva uvedené dílo nevýdělečně užít, archivovat a zpřístupnit ke studijním, výukovým a výzkumným účelům včetně pořizovaní výpisů, opisů a rozmnoženin.
2. Licence je poskytována celosvětově, pro celou dobu trvání autorských a majetkových práv k dílu.
3. Autor souhlasí se zveřejněním díla v databázi přístupné v mezinárodní síti
   - ☒ ihned po uzavření této smlouvy
   - □ 1 rok po uzavření této smlouvy
   - □ 3 roky po uzavření této smlouvy
   - □ 5 let po uzavření této smlouvy
   - □ 10 let po uzavření této smlouvy

   (z důvodu utajení v něm obsažených informací)
4. Nevýdělečné zveřejňování díla nabyvatelem v souladu s ustanovením § 47b zákona č. 111/ 1998 Sb., v platném znění, nevyžaduje licenci a nabyvatel je k němu povinen a oprávněn ze zákona.

Článek 3
Závěrečná ustanovení

1. Smlouva je sepsána ve třech vyhotoveních s platností originálu, přičemž po jednom vyhotovení obdrží autor a nabyvatel, další vyhotovení je vloženo do VŠKP.
2. Vztahy mezi smluvními stranami vzniklé a neupravené touto smlouvou se řídí autorským zákonem, občanským zákoníkem, vysokoškolským zákonem, zákonem o archivnictví, v platném znění a popř. dalšími právními předpisy.
3. Licenční smlouva byla uzavřena na základě svobodné a pravé vůle smluvních stran, s plným porozuměním jejímu textu i důsledkům, nikoliv v tísni a za nápadně nevýhodných podmínek.
4. Licenční smlouva nabývá platnosti a účinnosti dnem jejího podpisu oběma smluvními stranami.

V Brně dne: 27.5. 2010

…….. ……

Nabyvatel Autor

Abstrakt:

Program pro testování rychlosti šifrovacích algoritmů je projekt, jehož cílem je vytvoření programu, jenž bude testovat specifické vlastnosti jednotlivých šifrovacích protokolů a výsledky testování bude ukládat pro pozdější statistické zpracování. Program bude vytvořen v programovacím jazyku C#. Ve statistickém zpracování budou výsledky vlastností jednotlivých algoritmů zpracovány a zhodnoceny v přehledných reportech.

Abstract:

Program for testing of speed of encryption algorithms is project aimed to build a program, which will be able to test specific properties of each single encryption algorithm protocol and results of testing will be saved for later statistical analysis. Program is going to be build in C# program language. In statistical analysis part all data and results from testing of each single algorithm will be processed and reviewed in transparent reports.

Klíčová slova:

Šifrování, algoritmy, měřící program

Keywords:

Encryption, algorithms, measuring program

Bibliografická citace díla:

VOLF, L. *Program pro testování rychlosti šifrovacích algoritmů – diplomová práce*. Brno, 2010. 75 s. Vedoucí diplomové práce Ing. Jaroslav Kadlec, Ph.D. FEKT VUT v Brně

Prohlášení:

Prohlašuji, že jsem tuto vysokoškolskou kvalifikační práci vypracoval samostatně pod vedením vedoucího diplomové práce, s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury. 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 jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., včetně možných trestněprávních důsledků vyplývajících z ustanovení § 152 trestního zákona č. 140/1961 Sb.

V Brně dne 27. května 2010

.....

podpis autora

Poděkování:

Děkuji vedoucímu semestrálního projektu Ing. Jaroslavu Kadlecovi, Ph.D. za účinnou metodickou, pedagogickou a odbornou pomoc a další cenné rady při zpracování mého semestrálního projektu.

Obsah

Seznam obrázků

Úvod

To, aby zůstala informace na své cestě od zdroje k příjemci utajená před nevítanými příjemci, je problém, se kterým se setkáváme takříkajíc odnepaměti.

Již ve starém Řecku se používaly hole různého průměru, na které se namotával pruh tkaniny. Na ten byla posléze podélně napsána tajná zpráva. Adresát pak musel vědět, jaký průměr hole použít, aby mohl látku znovu namotat a text si přečíst. Dalším významným milníkem byl např. vynález šifrovací mřížky. Stejně jako ve spoustě dalších oblastí lidské činnosti nalezla i kryptografie významné uplatnění ve vojenství.

Její bouřlivý rozvoj můžeme sledovat především v první polovině minulého století v době obou světových válek. Především druhá světová válka přinesla nebývalý zájem o kryptografii i o kryptoanalýzu. O tento obor se najednou začaly zajímat hlavy států a generální štáby armád. Kryptoanalýza se stala tichou, neviditelnou a účinnou zbraní a kvalitní kryptografie se naopak stala obranným štítem. V této době byl přínos kryptoanalytiků tak nepřehlédnutelný a široký, že zasáhl na všech frontách a ovlivnil všechny hlavní válčící strany. Po válce si proto všechny hlavní mocnosti začaly budovat mohutná kryptografická a luštící centra a začalo se více dbát na rozvoj teorie. V rámci tohoto poválečného dění Claude E. Shannon nejprve v roce 1948 publikoval práci A Mathematical Theory of Communication, která je pokládána za základ teorie informace, a rok poté práci Communication Theory of Secrecy Systems, která je pokládána za základ moderní kryptologie. [1]

Poslední a pravděpodobně zatím největší impulsy pro rozvoj kryptologie přišly s nástupem moderní výpočetní techniky a výkonných počítačů v sedmdesátých letech. Nové technologické možnosti přinesly nové koncepty, a právě proto se dnes můžeme se šifrovacími technologiemi setkat v domácích počítačích, mobilních telefonech, internetu a bankovnictví.
Samotný vědní obor, který nazýváme kryptologie, můžeme rozdělit na kryptografii (vědní disciplína studující problém, jak skrýt obsah informace do tvaru, ze kterého tato informace není jednoduše zjistitelná) a kryptoanalýzu (zabývá se rozbíjením existujících šifrovacích algoritmů). Základním prostředkem ke zmíněnému utajení zpráv je šifrování. Šifra (nebo také šifrovací algoritmus) je matematická metoda, pomocí které se za „účasti“ šifrovacího klíče převede čitelný text do nečitelné podoby (šifrovaného textu).

1 Šifrovací systémy

Tato kapitola má za úkol seznámit čtenáře se základními pojmy z oblasti šifrování a se základy šifrovacích systémů, mezi něž patří symetrické, asymetrické, hashovací a hybridní šifrovací systémy.

1.1 Základní pojmy

- **Kryptologie** je věda, která se zabývá šifrováním ze všech úhlů pohledu. Jejími hlavními disciplínami jsou kryptografie a kryptoanalýza. [2]
- **Kryptografie** (z řeckého kryptós – „skrytý“ a gráphein „psát“) neboli šifrování je nauka o metodách utajování smyslu zpráv převodem do podoby, která je čitelná jen se speciální znalostí. Někdy je pojem obecněji používán pro vědu o čemkoli spojeném se šiframi jako alternativa k pojmu kryptologie. [2]
- **Kryptoanalýza** (z řeckého kryptós – „skrytý“ a analýein – „uvolnit“ či „rozvázat“) je věda zabývající se metodami získávání obsahu šifrovaných informací bez přístupu k tajným informacím, které jsou za normálních okolností potřeba. Tzn. především získání tajného klíče. V netechnickém kontextu je používán tento termín obecně pro prolamování kódu. Je vlastně opakem kryptografie, která šifry vytváří. [2]
- **Steganografie** neboli ukrývání zprávy jako takové. Sem patří různé neviditelné inkousty, vyrývání zprávy do dřevěné tabulky, která se zalije voskem apod. V moderní době lze tajné texty ukrývat například do souborů s hudbou či s obrázky namísto náhodného šumu. [2]
- **Šifra nebo šifrování** – kryptografický algoritmus, který převádí čitelnou zprávu neboli prostý text na její nečitelnou podobu neboli šifrový text. [2]
- **Klíč** je tajná informace, bez níž nelze šifrový text přečíst. [2]
- **Délka klíče** - ovlivňuje časovou náročnost při útoku hrubou silou, což je kryptoanalytická metoda, kdy postupně zkoušíme všechny možné hodnoty, kterých klíč může nabývat. [3]
- **Bloková šifra** je text, který je před vstupem do šifrovacího procesu rozdělen do stejně dlouhých částí (bloků) pro účely snazší aplikace šifrovacího algoritmu.
- **Symetrická šifra** je taková, která pro šifrování a dešifrování používá vždy stejný klíč.
- **Asymetrická šifra** používá pro šifrování veřejný klíč a pro dešifrování naopak klíč soukromý.

- **Hashovací funkce** je způsob, jak z celého textu vytvořit krátký řetězec – otisk (hash), který s velmi velkou pravděpodobností jednoznačně identifikuje původní text. [2]
- **Certifikáty a elektronický podpis** jsou softwarové prostředky, které umožní šifrování textu. [2]

1.2 Hlavní typy šifrovacích systémů

K tomu, aby bylo možno zašifrovaný text převést zpět do otevřené (čitelné) podoby, musíme logicky kromě konkrétního použitého algoritmu znát také šifrovací klíč. Největším problémem tedy zpravidla není jak zprávu zašifrovat, ale jak adresátovi bezpečně předat potřebný klíč. I z tohoto důvodu byly postupem času vyvinuty dva druhy šifrovacích algoritmů: symetrické a asymetrické. Třetí možností jejich kombinace je hybridní šifrování.

1.2.1 Symetrické šifrovací algoritmy

Symetrická šifra, někdy též nazývaná konvenční, je takový šifrovací algoritmus, který používá k šifrování i dešifrování jediný klíč. Stejný klíč musí mít k dispozici všichni, kteří se šifrovanými daty pracují. Logicky tedy vyplývá potřeba zajistit jeho bezpečné předání určeným osobám. Ve chvíli, kdy dojde k jeho prozrazení byť jen jedinou zúčastněnou osobou, jsou všechny jím zašifrované informace prozrazeny. Podstatnou výhodou symetrických šifer je jejich nízká výpočetní náročnost a jsou řádově 1000x rychlejší oproti asymetrickým šifrovacím algoritmům. Síla šifer se poměřuje délkou klíče udávanou v bitech (např 5-bitový klíč představuje 2^5 tedy 32 různých kombinací). Za bezpečné klíče se dnes považují algoritmy o délce klíče nad 128 bitů.

Obrázek 1 - Grafické znázornění symetrického šifrování

Nejpoužívanější symetrické šifrovací algoritmy v současnosti

- AES
- DES
- 3DES

- IDEA
- RCx
- Blowfish
- Twofish
- Serpent

1.2.2 Asymetrické šifrovací algoritmy

Asymetrický šifrovací algoritmus je oproti symetrickému poněkud složitější. Namísto jediného klíče zde totiž používáme tzv. klíčový pár (dva klíče) - veřejný a soukromý.

Veřejný klíč je určen k volnému šíření a je distribuován všem osobám, se kterými komunikujeme. Naproti tomu soukromý klíč musí zůstat přísně utajen u jeho vlastníka.. Cokoli zašifrujeme jedním klíčem, lze dešifrovat pouze druhým klíčem a naopak. Pouze jediný klíč pro šifrování a dešifrování tak použít nelze. Velkou výhodou tohoto přístupu oproti symetrickému šifrování je, že jeden z těchto klíčů můžeme dát k dispozici veřejnosti - veřejný klíč, ke kterému má přístup každý. Kdokoli nám chce zaslat zprávu nečitelnou pro třetí osobu, použije k jejímu zašifrování tento veřejný klíč. Ani on sám, ani žádný jiný vlastník našeho veřejného klíče není schopen takovouto zprávu dešifrovat. Rozšifrovat jí může pouze držitel druhého, privátního klíče.

Důvodem této vlastnosti asymetrických algoritmů jsou použité matematické funkce, jejichž reverzní výpočet je prakticky neproveditelný. Asymetrické šifrovací algoritmy jsou v porovnání se symetrickými obecně výrazně pomalejší, především v šifrování delších zpráv. Při šifrování se používají klíče až o velikosti 2048 bitů.

Matematicky tedy asymetrická šifra postupuje následujícím způsobem:

Šifrování

$$c = f(m, e). \qquad (2.1)$$

Dešifrování

$$m = g(c, d). \qquad (2.2)$$

V principu se mohou šifrovací a dešifrovací funkce lišit, zpravidla jsou však matematicky přinejmenším velmi podobné. [4]

Obrázek 2 - Grafické znázornění asymetrického šifrování

Nejpoužívanější asymetrické šifrovací algoritmy v současnosti

- Diffie-Hellman
- RSA
- DSA
- El-Gamal
- ECDSA

1.2.3 Hybridní šifrovací algoritmy

V praxi se asymetrické šifrování v čisté podobě téměř nevyužívá. Namísto toho se využívá jednoduchého způsobu kombinace symetrického a asymetrického šifrování, jakožto kompromisu mezi rychlostí symetrických a bezpečností asymetrických šifer.

Odesilatel použije klíč, kterým symetricky zašifruje zprávu. Tento klíč zašifruje veřejným klíčem osoby, které zprávu zasílá a pošle ho spolu se zprávou adresátovi. Adresát dostane asymetricky zašifrovaný klíč a symetricky zašifrovanou zprávu. Klíč dešifruje svým privátním klíčem a použije ho k dešifrování textu. Tímto způsobem odpadá problém s distribucí klíče při symetrickém šifrování a zároveň se celý proces zrychlí.

Tímto způsobem funguje drtivá většina softwaru používajícího asymetrickou kryptografii.

Pokud chceme používat hybridní šifrování mezi n účastníky, potřebujeme stejně jako u asymetrického šifrování 2n klíčů. Hybridní šifrování je teoreticky náchylnější na odhalení, ale pokud je útočník schopen dešifrovat klíč relace, tak je schopen číst pouze jedinou zprávu zašifrovanou pomocí tohoto klíče. Pokud by chtěl útočník přečíst i jinou zprávu, tak musí opět získat její klíč relace.

1.2.4 HASH algoritmus

Pomocí hashovací funkce můžeme vytvořit „otisk“ dokumentu. Obecně řečeno se jedná o matematickou funkci, kterou lze v jednom (přímém) směru spočítat velice snadno, zatímco v opačném směru (inverzním zobrazení) se výpočty dají provést jen s velkými obtížemi.

Vstupem hashovací funkce může být libovolná informace, na výstupu obdržíme její otisk, který je ve srovnání s vlastním dokumentem velmi malý a má pevnou délku. Výsledkem HASH funkce je zpravidla 128 nebo 160 bitů dlouhá sekvence. Pokud by se v původních datech změnil třeba jen jeden jediný bit, jejich HASH se výrazně změní.

Nejpoužívanější hashovací algoritmy v současnosti

- MD5
- SHA

2 Symetrické šifrovací algoritmy

Kapitola se zabývá nejznámějšími symetrickými šifrovacími algoritmy, jako jsou AES, DES, TripleDES, IDEA a Blowfish, a popisuje jejich základní funkční stavbu a techniku šifrování.

2.1 AES (Advanced Encryption Standard)

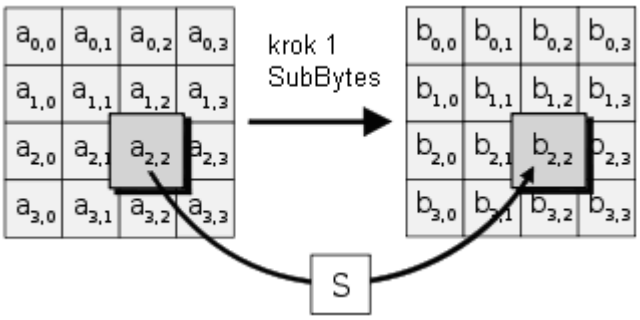
Šifrovací algoritmus byl vyvinut a předložen do veřejné soutěže NIST o federální šifrovací AES pod jménem „Rijndael“. V praxi tedy názvy "Rijdael" a "AES" odkazují na totéž.

Šifra využívá symetrického klíče (tj. stejný klíč je použit pro šifrování i dešifrování) a délka klíče může být 128, 192 nebo 256 bitů. Metoda je aplikována na data s pevně danou délkou - v tomto konkrétním případě 128 bitů. Pokud jsou šifrovaná data delší, zpracovávají se po jednotlivých blocích. Pokud jsou data kratší (typicky v případě posledního bloku se zbytkem dat), je potřeba je doplnit ("padding" - existuje pro něj několik algoritmů, od primitivního doplnění nulami po složitější schémata) na odpovídající délku. Šifra se vyznačuje vysokou rychlostí šifrování. V současné době není veřejně znám žádný případ plného prolomení této metody ochrany dat.

Nejjednodušším algoritmem používaným v AES je CBC - Cipher Block Chaining. Ten funguje tak, že před zašifrováním se odpovídající blok otevřeného textu XORuje předcházejícím blokem zašifrovaného textu. To znamená, že jednotlivé bloky jsou na sobě závislé, abyste dešifrovali konkrétní blok, musíte dešifrovat i všechny předchozí. V případě prvního bloku se data XORují náhodně vygenerovanými hodnotami v nultém bloku („inicializační vektor“). Tento blok se použije k dešifrování prvního bloku a pak zahodí. [2]

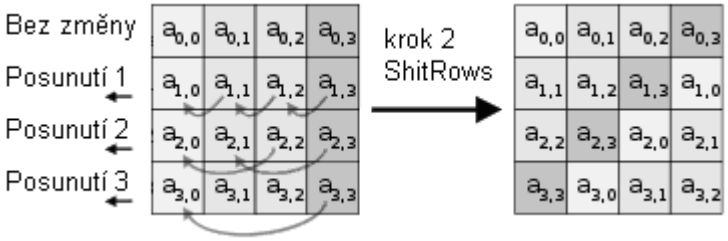
Šifrování probíhá ve čtyřech krocích:

- **SubBytes** - jednoduchá substituce, kde každý Byt je nahrazen jiným podle předem daného klíče, the Rijndael-S-Box (8bitů). [2]



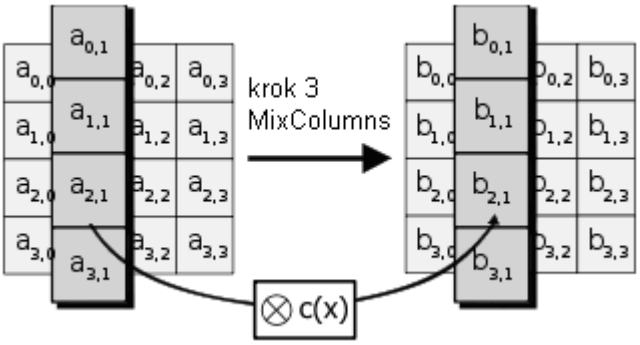
Obrázek 3 - AES šifrování - 1. krok – SubBytes [5]

- **ShiftRows** — v tomto kroku se jednotlivé byty přehází podle obrázku [2]



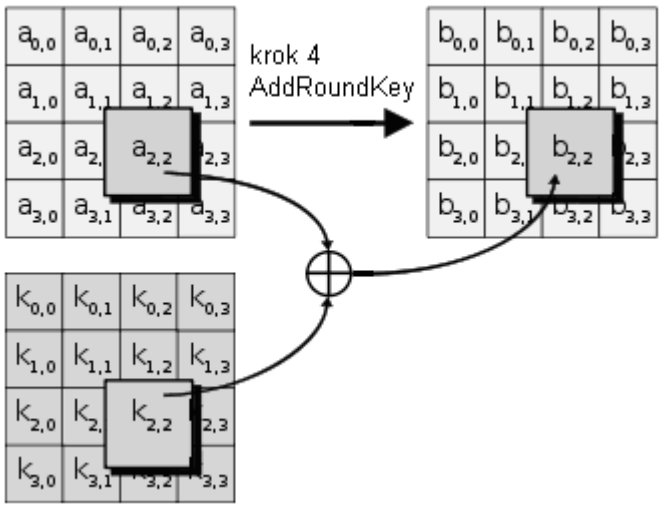
Obrázek 4 - AES šifrování - 2. krok – ShiftRows [5]

- **MixColumns** — při této operaci dochází k proházení sloupců a zároveň je každý sloupec vynásoben stejným polynomem *c(x)* [2]



Obrázek 5 - AES šifrování - 3. krok – MixColumns [5]

- **AddRoundKey** — každý byt je zkombinovaný se subklíčem (subklíč získáme z původního klíče pomocí Rijndaelovy tabulky). Každý byt subklíče zkombinujeme s příslušným bytem naší zprávy a dostaneme výslednou šifru [2]



Obrázek 6 - AES šifrování - 4. krok – AddRoundKey [5]

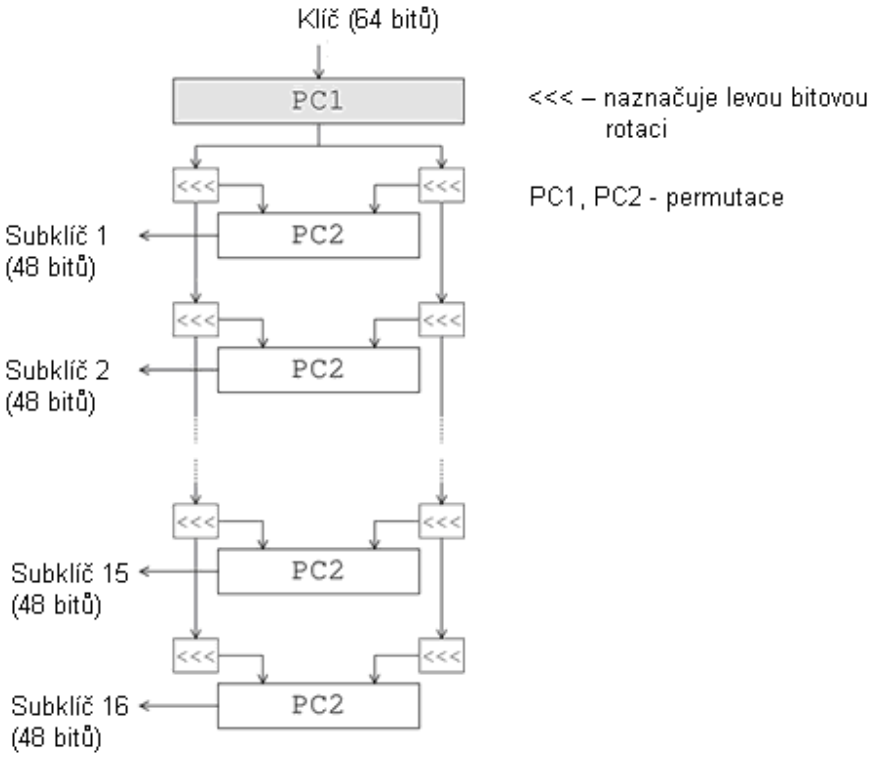
2.2 DES (Data Encryption Standard)

DES je symetrická šifra vyvinutá v 70. letech. V roce 1977 byla zvolena za standard pro šifrování dat v civilních státních organizacích v USA a následně se rozšířila i do soukromého sektoru.

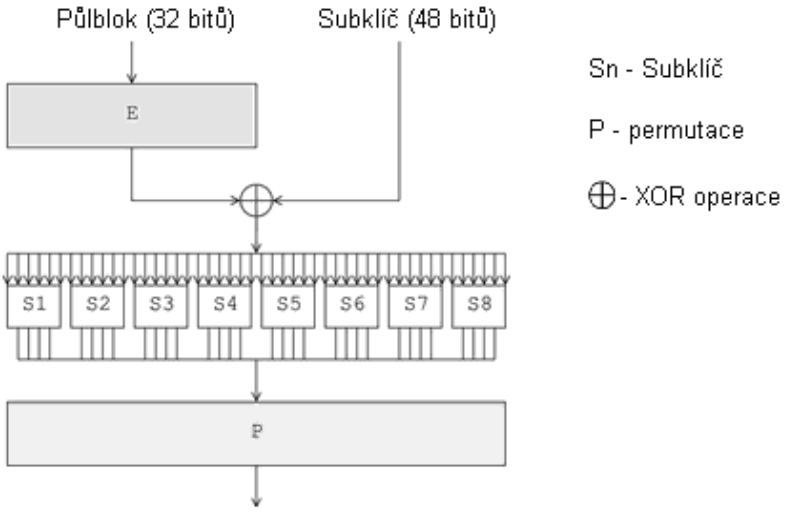
V současnosti je tato šifra považována za nespolehlivou, protože používá klíč pouze o délce 56 bitů. Navíc obsahuje algoritmus slabiny, které dále snižují bezpečnost šifry. Díky tomu je možné šifru prolomit útokem hrubou silou („Brutal Force Attack“) za méně než 24 hodin, ale i přesto se ještě někde používá. DES má dnes spíše historický význam, ale stal se vzorem a inspirací pro spoustu dnes používaných algoritmů.

DES používá klíč dlouhý 64 bitů, ale jen 56 bitů je opravdu využito. Zbývajících 8 je klamných. [2] Celá bezpečnost šifry je založena na síle klíče, proto je nutné výběru správného klíče věnovat čas a pozornost. Při výběru se doporučuje vyloučit tzv. slabé klíče.

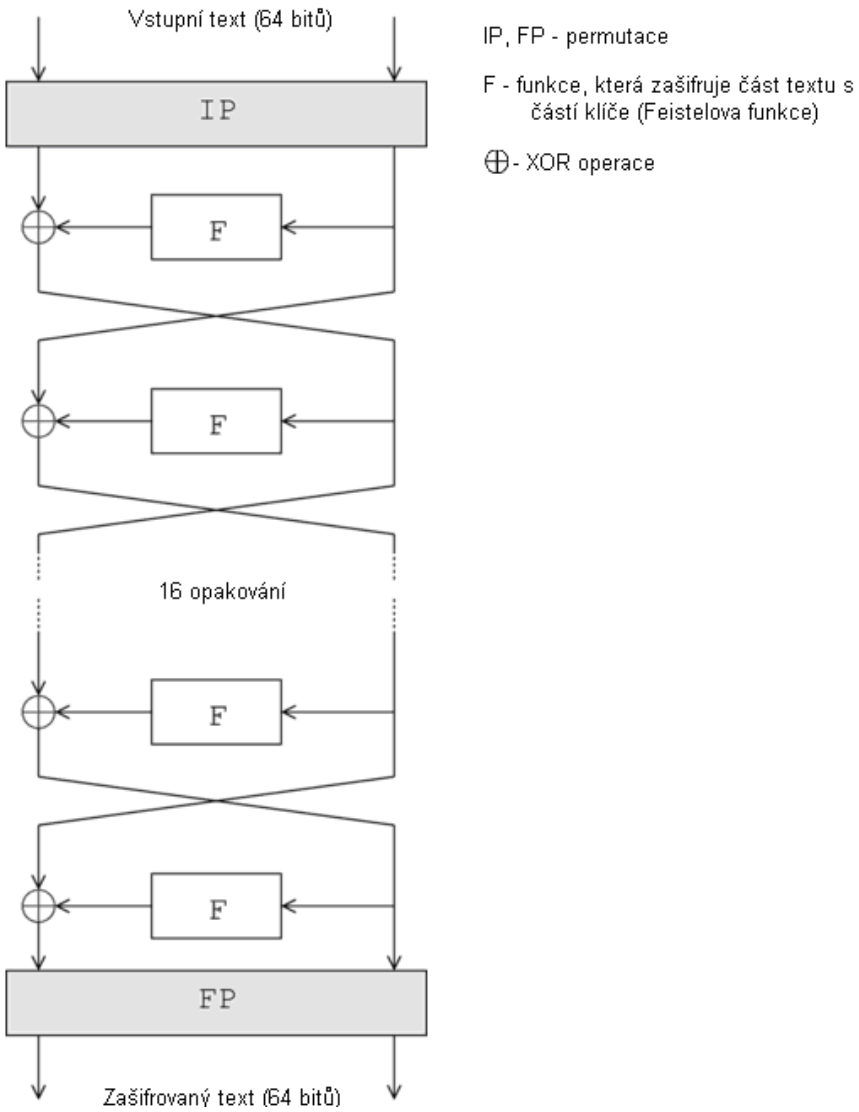
Při šifrování se stále opakují dvě operace a to substituce a permutace, které se opakují v každém cyklu. Těch má algoritmus DES 16 z nichž každý se skládá z jednoduchých aritmetických operací. Pro vlastní operace se 64 bitové bloky rozdělují na dva 32 bitové části. Tyto části se znovu spojí až po skončení posledního cyklu. Nakonec je celý blok podroben transformaci.



Obrázek 7 - Metodika tvorby subklíčů v DES algoritmu [6]



Obrázek 8 - Feistelova funkce používaná v DES algoritmu [6]



Obrázek 9 - Celková struktura algoritmu DES [6]

2.3 TripleDES (3DES)

Algoritmus 3DES (někdy též 3TDES) je trojnásobnou aplikací šifry DES. Nejčastěji používaná varianta 3DES pracuje s klíčem o celkové délce 168 bitů (3 x 56 bitů), můžeme se však také setkat s variantou o délce klíče 192 bitů (3 x 64 bitů).

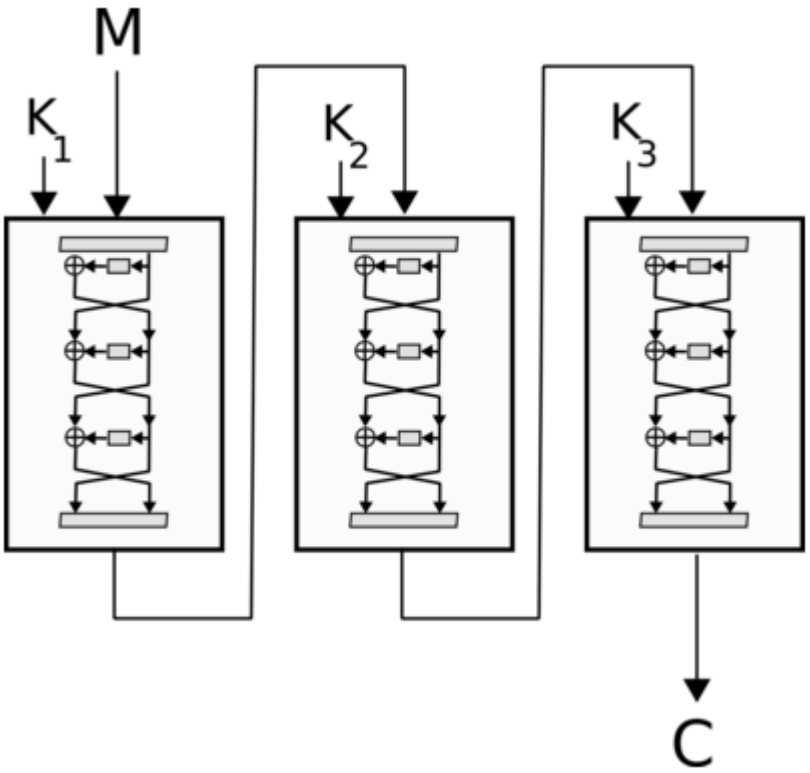
V době kdy se ukázalo, že 56- bitový klíč šifry DES není dost bezpečný, aby chránil před hrubou sílou útoku, byl aplikován jednoduchý způsob jak zvýšit velikost klíče bez potřeby přejít na nový algoritmus. Použití tří implementací šifry DES je vhodné pro předcházení středně velkým útokům, pro které již nebylo dvojité, 112 bitové (2 x 64 bitů) DES (2TDES)

šifrování bezpečné. Všimněte si, že DES není skupina; kdyby byla, výstup šifry 3DES by byl rovnocenný k jednotlivé DES operaci a tím pádem by nebyl bezpečný. [7]

Nejjednodušší varianta 3DES operuje tímto způsobem:

DES(kk3;DES(kk2;DES(kk1;MM))), kde M je blok dat, který má být zašifrován a kk1, kk2, a kk3 jsou klíče šifry DES. Tato varianta je známá jako EEE, protože všechny tři DES operace jsou šifrování. C je výstupní zašifrovaný blok dat. Postupně se tedy informace zašifruje pomocí prvního klíče, dále následuje další zašifrování pomocí druhého klíče a nakonec zašifrování třetím klíčem. Dešifrování probíhá stejným způsobem, je tedy inverzní operací.

3DES je oproti novějším algoritmům daleko pomalejší, a proto se postupně přestává používat.



Obrázek 10 – Schéma šifrovacího algoritmu TripleDES (varianta EEE) [7]

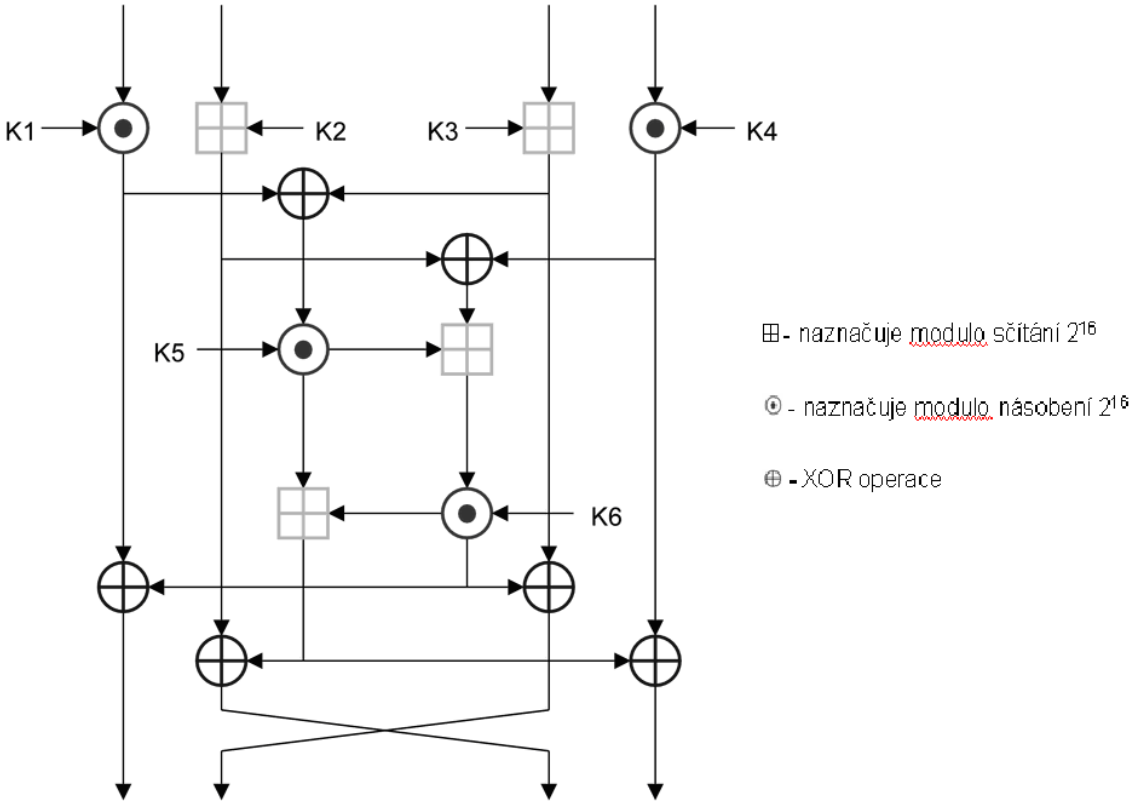
2.4 IDEA (International Data Encryption Algorithm)

IDEA je bloková šifra, kterou navrhli Xuejia Lai a James L. Massey ze Švýcarského národního technologického institutu (ETHZ) v Zürichu. Tento algoritmus měl nahradit DES. IDEA je vylepšena, aby odolala moderním kryptoanalytickým útokům. Je založena na

kombinování různých matematických operací. IDEA pracuje s bloky o velikosti 64 bitů, které se dělí na subbloky o velikosti 16 bitů. Používá klíč o velikosti 128 bitů. Skládá se z řady osmi identických transformací a vstupní transformace (poloviční průchod). Šifrování i dešifrování probíhá podobně. IDEA střídá operace z různých skupin, které jsou v jistém smyslu algebraicky neslučitelné. Tyto tři základní operace pracují s 16 bitovými řetězci jsou:

- Bitová nonekvivalence
- sčítání modulo 2^{16}
- násobení modulo $2^{16} + 1$

Systém algoritmu je řešen tak, že výstup z operace jednoho typu není nikdy použit ke vstupu operace stejného typu.



Obrázek 11 - Metodika šifrování v IDEA algoritmu

Každý 64 bitový blok je rozdělen na 4 subbloky X1, X2, X3, X4. V každém cyklu probíhá několik operací, nakonec se zamění 2. a 3. subblok. Během šifrování je použito 52 subklíčů (6 pro každý cyklus K1, K2, K3, K4, K5, K6 a 4 pro závěrečnou transformaci), které získáme rozdělení hlavního 128 bitového klíce na 8 16-bitových subklíčů. V každém cyklu je

použito 6 subklíčů. Po ukončení cyklu je hlavní klíč otočen o 25 míst a pak znova rozdělen na 8 subklíčů. [8]

Popis jedné iterace algoritmu:

1. X1 vynásob s K1,
2. X2 sečti s K2,
3. X3 sečti s K3,
4. X4 vynásob s K4,
5. výsledek kroku 1 xoruj s výsledkem kroku 3,
6. výsledek kroku 2 xoruj s výsledkem kroku 4,
7. výsledek kroku 5 vynásob s K5,
8. výsledek kroku 6 sečti s výsledkem kroku 7,
9. výsledek kroku 8 vynásob s K6,
10. výsledek kroku 1 sečti s výsledkem kroku 9,
11. výsledek kroku 1 xoruj s výsledkem kroku 9,
12. výsledek kroku 3 xoruj s výsledkem kroku 9,
13. výsledek kroku 2 xoruj s výsledkem kroku 10,
14. výsledek kroku 4 xoruj s výsledkem kroku 10.

Výstupem tohoto cyklu jsou pak výsledky kroku 11, 12, 13 a 14. Subbloky 2 a 3 se mezi sebou zamění, čili vstupem pro další cyklus jsou výsledky kroku 11, 13, 12 a 14. Tento postup se pak ještě 7krát opakuje. Po posledním opakování se provede konečná transformace:

1. X1 vynásob s K1,
2. X2 sečti s K2,
3. X3 sečti s K3,
4. X4 vynásob s K4.

Nakonec se výsledné subbloky spojí v jeden 64 bitový zašifrovaný text. Dešifrování se provádí stejným způsobem, pouze použití klíce je odlišné. [9] [8]

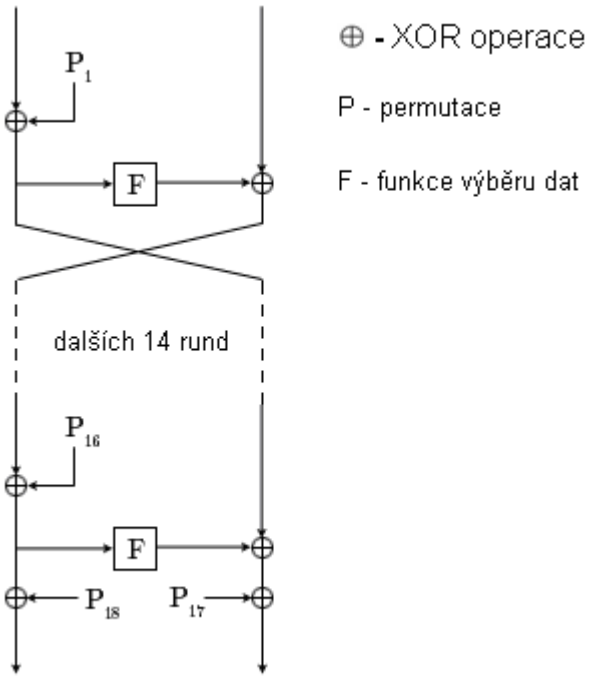
2.5 BLOWFISH

Šifra Blowfish byla zveřejněna roku 1994 a je to nelicencovaná alternativa k algoritmu DES. Jedná se o symetrickou blokovou šifru s velikostí bloku 64 bitů a délkou klíče nejvýše 448 bitů.

Algoritmus je tvořen dvěma částmi: část expanze klíče a část šifrování dat. Expanze klíče převádí klíč s libovolnou délkou (nejvýše však 56 bytů) na několik polí podklíčů. Šifrování dat je prováděno po blocích 64 bitů, v šestnácti rundách. Každá runda provádí permutaci závislou na klíči a substituci závislou jak na kódovaných datech tak i klíči. Všechny operace

použité v algoritmu jsou XOR a sčítání 32 bitových slov. Navíc jsou v každé rundě prováděny čtyři operace výběru dat z pole vypočteného indexu. [10]

Blowfish používá velký počet podklíčů, které musí být vypočteny ze zadaného klíče ještě před samotným šifrováním, resp. dešifrováním dat. Podklíče jsou uloženy celkem v pěti polích. První pole, označované jako P-pole nebo P-box, má celkem 18 položek (každá má velikost 32 bitů), dále označovaných P1, P2,…, P18. Zbývající pole jsou označována jako S-pole nebo S-boxy. Každý S-box má 256 položek (32 bitů). Pokud budeme pracovat s S-boxem i (i = 1, 2, 3, 4), pak jednotlivé položky budeme označovat Si,0, Si,1,…, Si,255. [10]



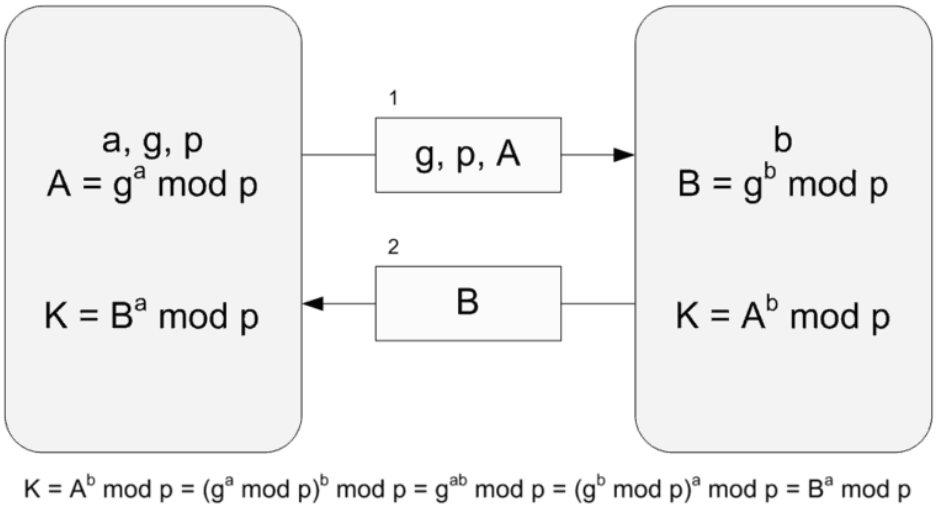
Obrázek 12 - Metodika šifrování v algoritmu Blowfish

3 Asymetrické šifrovací algoritmy

Kapitola se zabývá nejznámějšími asymetrickými šifrovacími algoritmy, jako jsou Diffie-Hellman a RSA, a popisuje jejich základní funkční stavbu a techniku šifrování.

3.1 Diffie-Hellman (D-H)

Diffie-Hellman (D-H) výměna klíčů je kryptografický protokol, který umožňuje navázat bezpečné spojení. Pro bezpečné spojení je potřeba si vyměnit klíč k symetrické šifře přes ještě nezabezpečený kanál. Právě tento protokol to umožňuje aniž by byl klíč jednoduše poslán v otevřené formě. Obsah klíče Diffie-Hellman vyměněný dvěma stranami může být založen na 768, 1 024 nebo 2 048 bitech obsahu klíče, které se nazývají skupiny Diffie-Hellman č. 1, 2 nebo 2048. Síla skupiny algoritmu Diffie-Hellman odpovídá síle klíče vypočteného z výměny klíčů pomocí algoritmu Diffie-Hellman. Silnější skupiny Diffie-Hellman společně s delšími klíči zvyšují výpočetní obtížnost odhalení tajného klíče. Algoritmus Diffie-Hellman je starší než kryptografické algoritmy RSA a nabízí lepší výkon.



Obrázek 13 - Diffie-Hellman šifrovací algoritmus [11]

Principielně se problém opírá o složitost výpočtu diskrétního logaritmu [12]

- Jeden z účastníků komunikace zveřejní číslo α a m.
- Každý i-tý účastník si zvolí číslo k_{priv}^{i}, které bude sloužit jako jeho soukromý klíč a spočte svůj veřejný klíč jako

$$k_{pub}^{i} = \alpha^{(k_{priv}^{i})} mod \quad m \ , \qquad (4.1)$$

ten zveřejní.

- Pokud spolu chtějí komunikovat účastníci i, j, spočtou si obě strany klíč komunikace podle vzorce:

$$K_{i,j} = \alpha^{(k_{priv}^{i} \cdot k_{priv}^{j})} mod \quad m = (k_{pub}^{j})^{(k_{priv}^{i})} mod \quad m = (k_{pub}^{i})^{(k_{priv}^{j})} mod \quad m \ . \qquad (4.2)$$

3.2 RSA (Rivest, Shamir, Aleman)

RSA (název odvozen z iniciálů autorů Ron Rivest, Adi Shamir a LenAdleman) je šifra s veřejným klíčem, popsaná v roce 1977. Princip tohoto systému je jednoduchý.

Bezpečnost RSA je postavena na předpokladu, že rozložit číslo na součin prvočísel (faktorizace) je velmi obtížná úloha. Z čísla $n = pq$ je tedy v rozumném čase prakticky nemožné zjistit činitele p a q. Vynásobíme-li dvě prvočísla, výsledkem je pak součin, který použijeme za veřejný klíč. Bez znalosti obou prvočísel je téměř nemožné provést rozklad na původní prvočísla. Bezpečnost RSA spočívá v tom, že není znám rychlý algoritmus na faktorizaci, což je rozklad čísla na dvě. Z čísla $n = pq$ je tedy v rozumném čase prakticky nemožné zjistit činitele p a q.

Osoba A chce komunikovat s osobou B prostřednictvím otevřeného (nezabezpečeného) kanálu a osoba B by chtěla osobě A poslat soukromou zprávu.

Nejprve si osoba A bude muset vyrobit pár veřejného a soukromého klíče:

1. Zvolí dvě různá velká náhodná prvočísla p a q.
2. Spočítá jejich součin $n = pq$. *(4.3)*
3. Spočítá hodnotu Eulerovy funkce $\varphi(n) = (p - 1)(q - 1)$. *(4.4)*
4. Zvolí celé číslo e menší než $\varphi(n)$, které je s $\varphi(n)$ nesoudělné.
5. Nalezne číslo d tak, aby platilo $de \equiv 1 \ (mod \ \varphi(n))$. *(4.5)*
6. Jestli e je prvočíslo tak $d = (1+r*\varphi(n))/e$, kde $r = [(e-1)\varphi(n)\hat{}(e-2)]$. *(4.6)*

Veřejným klíčem je dvojice (n, e), přičemž n se označuje jako modul, e jako (šifrovací, příp. „veřejný“ exponent) exponent. Soukromým klíčem je dvojice (n, d), kde d se označuje jako dešifrovací či soukromý. Veřejný klíč poté osoba A pošle (uveřejní) osobě B . Soukromý klíč naopak uchová v tajnosti. [13]

Zašifrování zprávy

Osoba B chce nyní osobě A zaslat zprávu M. Tuto zprávu převede nějakým dohodnutým postupem na číslo m ($m < n$).

Šifrovým textem odpovídajícím této zprávě pak je číslo

$$c = m^e \bmod n. \quad (4.7)$$

Tento šifrový text poté zašle nezabezpečeným kanálem osobě A.

Dešifrování zprávy

Osoba A od osoby B získá šifrový text *c*. Původní zprávu m získá následujícím výpočtem:

$$m = c^d \bmod n. \quad (4.8)$$

Fakt, že tímto výpočtem získáme původní zprávu, je důsledkem následující rovnosti:

$$cd = (m^e)^d = m^{ed} \ (\bmod n). \quad (4.9)$$

A jelikož

$$e^d = 1 \ (\bmod p - 1) \quad (4.10)$$

a

$$e^d = 1 \ (\bmod q - 1), \quad (4.11)$$

díky malé Fermatově větě platí, že

$$m^{ed} = m \ (\bmod p) \quad (4.12)$$

a zároveň

$$m^{ed} = m \ (\bmod q). \quad (4.13)$$

Jelikož *p* a *q* jsou různá prvočísla, pomocí čínské věty o zbytcích je dáno

$$m^{ed} = m \ (\bmod pq). \quad (4.14)$$

Tudíž

$$c^d = m \bmod n. \quad (4.15)$$

Příklad

V tomto příkladu jsou pro jednoduchost použita extrémně malá čísla, v praxi se používají o mnoho řádů větší. [13]

p = 61 (první prvočíslo)

q = 53 (druhé prvočíslo)

n = pq = 3233 (modul, veřejný)

e = 17 (veřejný, šifrovací exponent)

d = 2753 (soukromý, dešifrovací exponent)

Pro zašifrování zprávy 123 probíhá výpočet:

šifruj(123) = 12317 mod 3233 = 855

Pro dešifrování pak:

dešifruj(855) = 8552753 mod 3233 = 123

4 Hashovací algoritmy

Kapitola se zabývá nejznámějšími hashovacími algoritmy, mezi něž patří algoritmus MD5 a SHA, a dále popisuje jejich základní funkční stavbu a techniku šifrování.

4.1 MD5 (Message-Digest Algorithm)

Algoritmus MD5 zpracuje vstupní informaci o proměnné délce do výstupního bloku (hashe) o velikosti 128 bitů. Vstup zprávy je doplněn takovým způsobem, aby celková délka vstupu byla dělitelná 512 (v bitech). Pokud jsou bloky kratší, je potřeba je doplnit (padding). Doplnění probíhá následujícím způsobem. Zpráva je doplněna na konci nejprve jedním bitem rovným jedné. Pak je doplňována nulami tak, aby vznikl soubor o délce, která je o 64 bitů kratší než násobek 512. Zbylých 64 bitů je vyplněno číslem, které charakterizuje délku původní zprávy. Výsledek má tedy délku, která je násobkem 512. Výstupní hash je tedy rozložen do jednotlivých 512 bitových bloků (16 x 32 bitů). [14]

Hlavní MD5 algoritmus pracuje s bloky o velikosti 128 bitů (stavový blok), rozdělenými na čtyři části, označenými A, B, C a D o délce 32 bitů. V počátku algoritmu jsou hodnoty těchto slov (word) rovné definované pevné iniciální hodnotě. Algoritmus zpracovává vždy 512 bitový blok vstupu, výsledkem je nový stavový blok.

Zpracování 512 bitového bloku zprávy sestává ze čtyř základních cyklů (rund). Každá runda je složena z 16 základních operací založených na nelineární funkci F, modulárním součtu a levé rotaci. Celkem mohou nastat čtyři případy funkce F a v každé rundě se použije právě jedna.

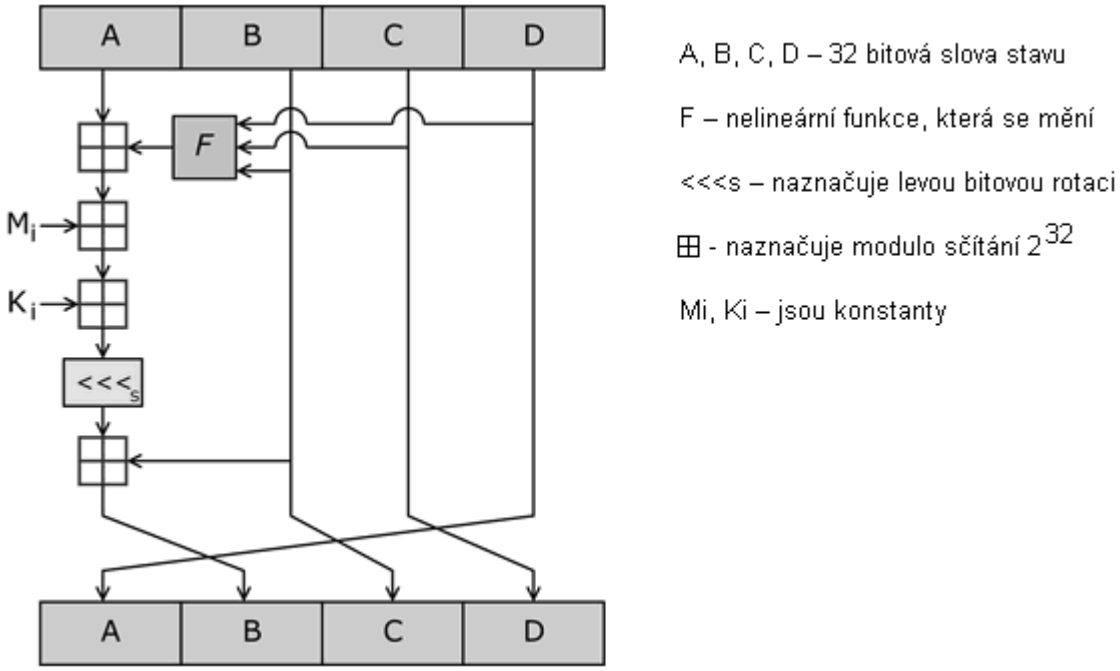
$$F(X,Y,Z) = (X \wedge Y) \vee (\neg X \wedge Z) \quad (5.1)$$

$$G(X,Y,Z) = (X \wedge Z) \vee (Y \wedge \neg Z) \quad (5.2)$$

$$H(X,Y,Z) = X \oplus Y \oplus Z \quad (5.3)$$

$$I(X,Y,Z) = Y \oplus (X \vee \neg Z) \quad (5.4)$$

$\oplus, \wedge, \vee, \neg$ označuje XOR, AND, OR a NOT operace. [14]



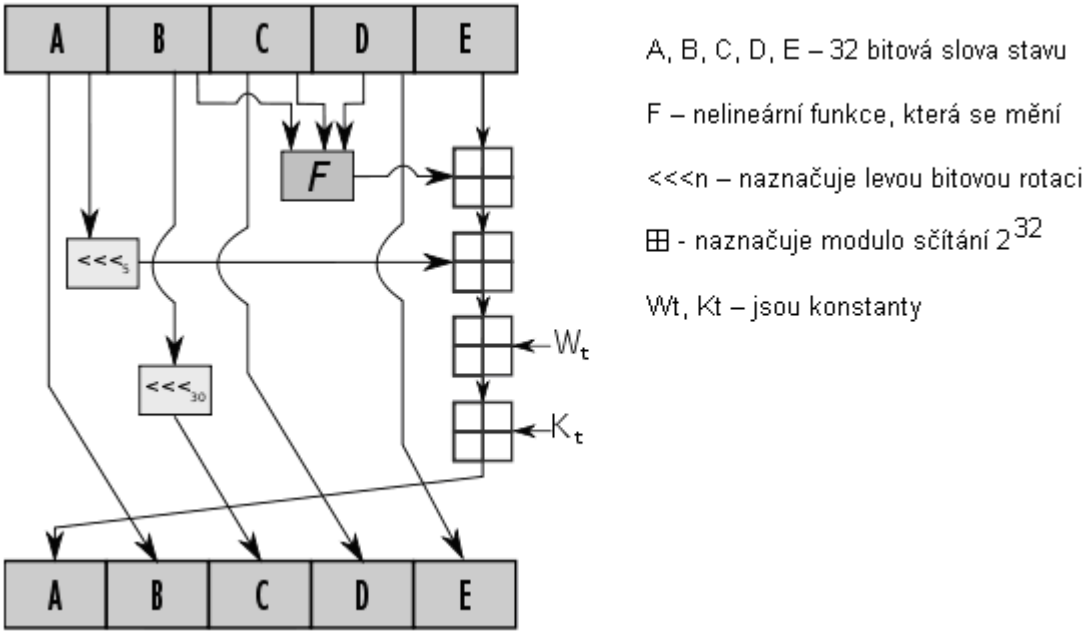
Obrázek 14 - Jedna iterace při šifrování pomocí algoritmu MD5 [14]

4.2 SHA (Secure Hash Algorithm)

SHA je rozšířená hashovaní funkce, která vytváří ze vstupních dat výstup (hash - otisk) fixní délky. Hash je též označován jako kontrolní součet. SHA je rodina pěti algoritmů: SHA-1, SHA-224, SHA-256, SHA-384 a SHA-512. Poslední čtyři varianty se souhrnně uvádějí jako SHA-2. SHA-1 vytvoří obraz zprávy dlouhý 160 bitů a ostatní algoritmy vytvářejí otisk o délce čísla následujícího za jejich názvem v bitech. SHA se používá u několika různých protokolů a aplikací, včetně TLS a SSL, SSH a IPsec, ale i pro kontrolu integrity souborů nebo ukládání hesel. Je považována za nástupce hashovací funkce MD5.

SHA-1 (stejně jako SHA-0) vytváří 160 bitový obraz zprávy s maximální délkou $2^{64} - 1$ bitů. Je založený na principech, které používal R. L. Rivest v návrhu MD4 a MD5 algoritmů.

Hashovací algoritmus SHA-1 rozdělí vstupní zprávu na bloky o délce 512 bitů, poslední blok zprávy doplňuje a zarovnává, včetně přidání údaje o délce zprávy, na nějž je vyhrazeno posledních 64 bitů.



Obrázek 15 - Jedna iterace při šifrování pomocí algoritmu SHA-1 [14]

5 Analýza výkonnosti algoritmů

V této kapitole se seznámíme s teorií spojenou s algoritmy a vysvětlíme si, pomocí jakých kritérií a v jakých případech se dají algoritmy poměřovat a zda se dá uvažovat o vyjádření jejich výkonnosti. Dále se podíváme na možnosti zpracování algoritmů za pomoci výpočetní techniky a na systémy reálného času, kde je možné vyjádřit čas nutný pro zpracování algoritmu.

5.1 Algoritmus

Algoritmem je označován přesný návod nebo postup, kterým lze vyřešit určitý problém. Je to též předpis, který splňuje všechny následující podmínky: [15]

- jednoznačně udává postup při řešení určitého problému
- prokazatelně končí po provedení konečného počtu kroků
- prokazatelně vede k požadovanému výsledku

Pojem realizace algoritmu chápeme jako řešení úlohy podle daného algoritmu. Chceme-li danou úlohu řešit, musíme vědět, co máme řešit, tzn. úloha musí být specifikovaná. Musíme vědět [15]

- co je dáno, tj. jaké jsou vstupní údaje (informace)
- co je výstup řešení, tj. jaké jsou výstupní údaje

5.2 Algoritmická úloha a její specifikace

Informace, ze kterých řešení vychází, nazýváme vstupní údaje a údaje, které jsou produktem řešení, jsou výstupní údaje. Vstupní údaje zadáváme v explicitním tvaru (konkrétní hodnota) a výstupní údaje jsou zadány implicitně (skrytě pomocí pravidel pro jejich odvození). [15]

Podmínku, kterou musí splnit výstupní údaje, nazveme výstupní podmínka. Ale i vstupní hodnoty nemusí být libovolné, ale vyhovují určité podmínce, kterou nazveme vstupní podmínka.

Vstupní a výstupní podmínka charakterizuje úlohu, tj. specifikuje to, co má být řešeno. Nejedná se o konkrétní úlohu, ale o celou třídu úloh, jejichž vstupní údaje splňují vstupní podmínku a výstupní údaje výstupní podmínku. Najdeme-li algoritmus, vyhovující této vstupní a výstupní podmínce, říkáme, že úloha je algoritmická. Vstupní a výstupní údaje budeme označovat symbolicky. [15]

5.3 Základní vlastnosti algoritmu a jejich ověření

Po specifikaci toho co budeme řešit, je třeba přistoupit k tomu, jak to budeme řešit, tzn. popsat postup řešení. Procesor vykoná určitý počet akcí – operací, přičemž každá akce má předepsané podmínky realizace, účinek a dobu trvání. Akce, které je procesor schopen vykonávat, jsou obvykle jednoduché a transformace vstupních údajů na výstupní se realizuje jejich kompozicí (postupně). Postup řešení předpisuje následnost provádění jednotlivých akcí.

Obecná pravidla určující postupnou transformaci vstupních údajů na výstupní nazýváme algoritmus.

Algoritmus se skládá z elementárních, dále nedělitelných, částí, kterým se nazývají kroky. Provedení jednoho kroku se nazývá iterace. Algoritmus s konečným počtem kroků se nazývá konečný. Naopak nekonečný algoritmus se skládá z nekonečného počtu kroků (např. výpočet prvočísel).

Základní vlastnosti algoritmu:

- hromadnost
- determinovatelnost
- rezultativnost

Hromadnost

Hromadností rozumíme skutečnost, že algoritmus je použitelný nejen pro jednu konkrétní úlohu (s konkrétními vstupními údaji), ale pro jeho libovolné n-tice vstupních údajů plňujících vstupní podmínku. [15]

Determinovatelnost

V každém kroku algoritmu musí být jednoznačně určeno, co se má provést. Realizace algoritmu nesmí být podmíněna jinými podmínkami než těmi, které jsou v něm uvedeny. [15]

Rezultativnost

Proces, který je předepsaný algoritmem musí být vždy konečný, tedy končí po určitém počtu provedených kroků. [15]

Algoritmus je použitelný pro libovolné celé kladné číslo, a má vlastnost determinovanosti. Hromadnost a determinovanost se ověří lehce, složitější je to u rezultativnosti, která je spojena s další vlastností, tj. správnost (ta se určuje vzhledem k vstupní a výstupní podmínce).

Algoritmus je správný tehdy, když pro všechny vstupní údaje splňující vstupní podmínku se proces předepsaný algoritmem zastaví (konečnost algoritmu) a výstupní údaje splňují výstupní podmínku. Aby byl algoritmus správný, je důležitá správnost specifikace úlohy. [15]

5.4 Složitost algoritmů a jejich porovnávání

Obecně lze jakoukoli úlohu řešit několika možnými způsoby. Je proto nutné stanovit kritéria pro rozhodování o tom, který z jednotlivých algoritmů je „lepší“ a který „horší“.

Obvykle algoritmy posuzujeme podle vlastností výsledného programu. Zajímají nás především – rychlost výpočtu a velikost potřebné operační paměti. Těmto vlastnostem říkáme časová a paměťová složitost algoritmu (někdy také efektivita algoritmu). [15]

Hodnocení algoritmů provádíme většinou podle dvou kritérií, která většinou v praxi stojí proti sobě – složitost časová a paměťová. [15]

Mnohdy máme k dispozici dva postupy řešení, z nichž jeden je pomalejší a vystačí s velmi malou pracovní pamětí, zatímco druhý je rychlejší, ale má vyšší paměťové nároky. Nejrychlejší algoritmus nebývá optimální z hlediska paměťové složitosti a naopak algoritmus s nejmenšími paměťovými nároky zase nebývá nejrychlejší. [15]

V současné době se v programování na první místo řadí efektivita časová, nevede-li tato volba k příliš extrémním nárokům na paměť. Doba výpočtu a paměťové nároky závisí na vstupních datech. Délka výpočtu obvykle není ovlivněna jejich konkrétními hodnotami, ale velikostí vstupních dat (počet čísel nebo znaků, které algoritmus zpracovává). V méně častých případech závisí časové a paměťové nároky algoritmu i na hodnotách vstupních dat. Jinou skupinu tvoří algoritmy, které generují rozsáhlé výstupy, jejichž velikost přímo závisí na vstupním údaji a je přitom určující pro dobu trvání výpočtu.

Máme-li v praxi porovnat dva algoritmy podle jejich časové složitosti a zvolit lepší z nich, bude záležet na tom, zda je v zadání vymezeno, pro jak velká vstupní data bude program používán. Pokud toto vymezení nemáme, je podstatné, který algoritmus je rychlejší pro velké hodnoty N (N je velikost vstupních dat). Hovoříme o asymptotické časové složitosti. (např. algoritmy polynomiální a exponenciální). Čas a prostor potřebný pro výpočet algoritmu závisí na velikosti vstupních dat. Proto má složitost nejčastěji podobu funkce velikosti dat, udávané počtem položek N. U asymptotické časové složitosti se toto N blíží nekonečnu.

Porovnání má podobu tří různých složitostí: [16]

- O Omikron – vyjadřuje horní hranici chování
- Ω Omega – vyjadřuje dolní hranici chování
- Θ Theta – vyjadřuje třídu chování

V praxi nejvyužívanější je složitost Omikron. Vyjadřuje horní hranici časového chování algoritmu a tedy dobu, do které algoritmus určitě skončí. To však platí od určité hranice konstanty N_o. [16]

Složitost Omega je definovaná podobně jako Omikron. Pouze s tím rozdílem, že nevyjadřuje horní, ale dolní hranici časového chování algoritmu. Od určitého N_o výše můžeme o algoritmu říci, že neskončí dříve než je tato doba dolní hranice. [16]

Složitost Theta vyjadřuje třídu časového chování algoritmu. Tzn. Určuje meze časového chování, mezi kterými se od určitého N_o pohybuje doba provádění algoritmu.

Rozdělení třídy Theta: [16]

- *Theta(1)* – označujeme algoritmy s *konstantní* časovou složitostí
- *Theta(n)* – označujeme algoritmy s *lineární* časovou složitostí
- *Theta(log(n))* – označujeme algoritmy s *logaritmickou* časovou složitostí
- *Theta(n*log(n))* – označujeme algoritmy s *linearitmickou* časovou složitostí
- *Theta* (n^2) – označujeme algoritmy s *kvadratickou* časovou složitostí
- *Theta*(n^3) – označujeme algoritmy s *kubickou* časovou složitostí
- *Theta*(k^n) – označujeme algoritmy s *exponenciální* časovou složitostí

5.5 Paměťová a časová složitost algoritmů

Časovou složitostí algoritmu rozumíme závislost jeho časových nároků na velikosti řešeného problému nebo na konkrétních vstupních datech. Je tedy určena počtem elementárních operací (kroků algoritmu), které budou provedeny při výpočtu programu s danými vstupními daty. Je to funkce, která každé hodnotě N udávající velikost konkrétního řešeného problému přiřazuje počet operací vykonaných při výpočtu podle daného algoritmu. [15]

Paměťová složitost algoritmu je definována jako závislost paměťových nároků algoritmu na velikosti řešeného problému nebo vstupních datech. Je to opět funkce, která každé velikosti vstupních dat N přiřazuje počet paměťových míst potřebných pro uskutečnění výpočtu.

Někdy nelze určit jedinou funkci časové složitosti, protože výsledek závisí na náhodném faktoru, např. na uspořádání vstupních dat. Rozlišujeme časovou složitost v nejhorším případě a časovou složitost v průměrném případě. [15]

5.6 Výpočetní složitost algoritmu

Jedním z obzvlášť důležitých problémů, s nimiž se během analýzy algoritmů setkáváme, je výběr správné míry výpočetní náročnosti. Na první pohled při rychlosti současné výpočetní techniky by se zdálo, že čas výpočtu nebude limitujícím faktorem a případné navýšení výpočetního času při zvýšení rozsahu vstupních dat bude buď zanedbatelné, nebo přijatelných mezích.

Budeme se tedy zabývat dobou výpočtu určitého algoritmu na počítači. Pod pojmem algoritmus budeme rozumět program napsaný v určitém programovacím jazyce. Každý může namítnout, že doba výpočtu závisí na typu počítače (jeho výpočetní rychlosti). Proběhne-li například nějaký program na určitém počítači za deset vteřin, není to pro nás žádná reprezentativní informace. Abychom tento výpočetní čas nemuseli uvádět pro každý typ počítače, budeme měřit výpočetní čas počtem elementárních příkazů, přičemž budeme abstrahovat od rozdílu času jejich trvání na počítači.

Další okolnost, která ovlivňuje dobu výpočtu je rozsah vstupních dat. Proto budeme tento výpočetní čas určovat v závislosti na rozsahu vstupních dat s tím, že u konkrétního algoritmu musíme definovat, co rozumíme rozsahem vstupních dat. U programu pro třídění n položek bude toto číslo n zřejmě rozsahem vstupních dat.

Další námitkou při určování výpočetního času je číselný obsah vstupních dat. Je zřejmé, že při některé metodě pro třídění dat bude výpočetní čas pro soubor dat, který je již setříděn podstatně menší než výpočetní čas pro soubor dat značně nesetříděného souboru (například setříděného opačným způsobem).

Zde připustíme, že v trvání algoritmu mohou být značné rozdíly vlivem číselných hodnot vstupních dat (při stejném rozsahu vstupních dat). Proto výpočetní čas definujeme jako počet operací, které jsou provedeny při daním rozsahu vstupních dat a při nejnepříznivějších hodnotách vstupních dat. Tuto hodnotu označujeme jako výpočetní složitost algoritmu, označíme ji *f(n)*, kde *n* je rozsah vstupních dat. Samozřejmě je problém najít ta nejnepříznivější vstupní data pro konkrétní algoritmus.

Podíváme-li se na následující tabulku, která obsahuje krátké srovnání prováděcích dob algoritmů za těchto následujících předpokladů:

- nechť doba průběhu jedné elementární operace je 1 mikrosekunda

- nechť doba trvání algoritmu A je úměrná dané vybrané matematické funkci, např. pro vstupní data o rozměru *x* a funkci *n!* je doba trvání programu úměrná *x!*

Tabulka 1 - Porovnání časové náročnosti algoritmů

A	10	20	30	40	50	60
n	10 μs	20 μs	30 μs	40 μs	50 μs	60 μs
n^2	100 μs	400 μs	900 μs	1600 μs	2500 μs	3600 μs
n^3	0,001s	0,008 s	0,027s	0,064 s	0,125 s	0,215 s
2^n	0,001s	1s	17,9 min	12,7 dne	35,7 let	366 stol.
3^n	0,59 s	58 min	6,5 roku	3855 stol.	200 x 100^6 stol	1,3 x 10^{13} stol.
n!	3,6 s	768 stol.	8,4 x 100^{16} stol.	2,6 x 10^{32} stol.	9,6 x 10^{48} stol.	2,6 x 10^{66} stol.

Z tabulky se dají vyvodit následující závěry. Algoritmy se složitostí *n*, n^2 a n^3 při zvětšování rozsahu vstupních dat z 10 na 60 nevykazuje takové navýšení doby výpočtu, které by bylo v praxi neúnosné a nerealizovatelné. Na druhé straně algoritmus se složitostí 2^n, 3^n a *n!* je nepoužitelný pro větší rozsah dat, neboť doba výpočtu při určitých hodnotách vstupních dat může být nepřijatelná. První skupina algoritmů se označuje jako rychlé, polynomiální algoritmy, druhá skupina algoritmy nepolynomiální, exponenciální.

Příkladem algoritmu se složitostí *n* je algoritmus hledání daného čísla v množině o velikosti *n* porovnávání jednotlivých čísel této množiny s daným číslem. Složitost n^2 mají třídící algoritmy, n^3 algoritmus inverze matic, 2^n algoritmus hledání nejkratší cyklické trasy mezi *n* městy.

Ukazuje se, že podstatné u každého algoritmu je to, jaký je jeho růst výpočetní složitosti pro n rostoucí do nekonečna.

Pokud pro dvě funkce *f(n), g(n)* jedné proměnné *n* platí:

$$f(n) \leq c.g(n), \qquad (6.1)$$

pro všechna přirozená čísla *n* od hodnoty n_0 a určité kladné číslo *c*, pak budeme označovat, že

$$f(n)=O(g(n)), \text{ příp. } f=O(g). \qquad (6.2)$$

Symbolem O je tzv. velké O a označuje, že funkce *f(n)* roste řádově ne rychleji než funkce *g(n)* .

Potom například všechny algoritmy, jež mají výpočetní složitost ve tvaru *c.n* jsou *O(n)*, tedy s řádově nejvýše lineárním růstem výpočetního času.

Obdobně definujeme symbol Ω. Pokud platí pro dvě funkce *f(n), g(n)* jedné proměnné *n* platí:

$$f(n) \geq c.g(n), \qquad (6.3)$$

pro všechna přirozená čísla $n \geq n_0$ a určité kladné číslo *c*, pak budeme označovat, že

$$f(n) = \Omega(g(n)), \text{ příp. } f = \Omega(g). \qquad (6.4)$$

Symbolem Ω označuje, že funkce *f(n)* roste řádově ne pomaleji než funkce *g(n)* .

Pak lze definovat polynomiální algoritmus, jež můžeme nazývat časovou složitostí a právě s ní se můžeme nejčastěji setkat u katalogových charakteristik daných algoritmů. Funkce se obvykle označuje písmenem *O*.

$$f(n)=O(n^k), \qquad (6.5)$$

pro *k* přirozené číslo.

Naopak algoritmus je nepolynomiální, jestliže neexistuje k přirozené číslo tak, že výpočetní složitost tohoto algoritmu je

$$f(n)=O(n^k). \qquad (6.6)$$

Dále budeme se snažit klasifikovat každý algoritmus vzhledem k jeho výpočetní složitosti a to tak, že algoritmus s výpočetní složitostí *f(n)* je efektivnější než algoritmus s výpočetní složitostí *g(n)*, jestliže

$$f(n)=O(g(n)). \qquad (6.7)$$

Platí následující hierarchie výpočetních složitostí:

$$1, \log_2 n, ..., \sqrt[3]{n}, \sqrt{n}, n, n\log_2 n, n\sqrt{n}, n^2, n^3, ..., 2^n, n!, n^n.$$

Pro symbol O dále platí:

a) je-li $f(n)=O(g(n))$ a $h(n)=O(g(n))$, pak $f(n)+h(n)=O(g(n))$,
b) je-li $f(n)=O(g(n))$ a $q(n)=O(r(n))$, pak $f(n).q(n)=O(g(n).r(n))$,
c) je-li $f(n)=O(g(n))$ a $g(n)=O(h(n))$, pak $f(n)=O(h(n))$,
d) pro libovolnou funkci $f(n)$ a $g(n)$ platí $O(f(n))+O(g(n))=O(max\{f\}n],g\}n]\})$,
e) pro libovolnou funkci $f(n)$ a $g(n)$ platí $O(f(n)).O(g(n))=O(f(n).g(n))$.

5.7 Parametry ovlivňující časovou výkonnost algoritmu

Základní kritéria výběru správného algoritmu závisí na souvislostech, v jakých se bude používat. Jedním z obzvlášť důležitých problémů, s nimiž se během analýzy algoritmů setkáváme, je výběr správné míry výpočetní náročnosti. Musí být natolik reprezentativní, aby se na efektivitě stejného algoritmu dokázal dohodnout uživatel malého PC i velké pracovní stanice nebo zabezpečovací systém, či automatizační systém využívající šifrování. Při hodnocení výpočetní náročnosti však také musíme zohlednit další velké množství parametrů majících vliv na výkon algoritmu. Je nutné zodpovědět následující otázky:

- na jakém PC (konfigurace) byl algoritmus prováděn
- na jaké frekvenci byly taktovány hodiny procesoru
- zatížení systému, v němž byl proces prováděn a jakou měl prioritu
- jakým programovacím jazykem (C, C++, C# atd.) byl napsán a jaký byl použit kompilátor při sestavování programu, případně zda šlo o „debug“ či „release“ verzi programu
- zda byly zapnuty volby optimalizace kódu v konkrétním kompilátoru
- samotná efektivita napsaného algoritmu (použité knihovny)

5.8 Systém reálného času

Obecně existuje několik definic operačních systémů reálného času (RTOS), které jsou v některých případech dosti protikladné. I přes tuto určitou nejednotnost je možné RTOS definovat následujícími způsoby: „Systém reálného času je takový, jehož správnost výpočtu nezáleží pouze na logické správnosti výpočtu, nýbrž záleží také na čase, ve kterém byl výsledek vypočten. Pokud časové podmínky systému nejsou dodrženy, říká se, že systém selhal“ (Donald Gillies). K této jednoduché definici je možné navíc dodat: „Jelikož časová podmínka v operačním systému je natolik bazální, je nutné, aby byla vždy splněna. Zajištění časování vyžaduje, aby systém byl predikovatelný.“ [17]

Požadavky na RTOS, které jsou ve své podstatě skryty v předchozích definicích, lze shrnout do následujících bodů: [17]

- RTOS musí být víceúlohový a preemtivní
- Musí podporovat prioritní systém procesů a vláken
- Musí existovat dědičnost priorit
- Operační systém musí podporovat predikovatelné synchronizační mechanismy
- Chování operačního systému musí být dostatečně známo

Na základě předchozích požadavků musí výrobce operačního systému poskytnout několik základních metrik, které jsou primární při rozhodování o vhodnosti operačního systému pro řešení požadované úlohy. Jmenovitě to jsou: [17]

- Latence přerušení, nebo-li doba, které uplyne mezi událostí přerušení a spuštěním obsluhy
- Délky trvání jednotlivých systémových volání
- Maximální doba, po kterou operační systém a ovladače zakazují (maskují) přerušení

Základním rozdílem mezi obecným operačním systémem a RT operačním systémem je požadavek na „deterministické“ chování v čase u RT operačních systémů. Formálně „deterministické“ chování v čase znamená, že služby operačního systému spotřebují pouze známé a požadované množství času. Teoreticky lze čas těchto služeb vyjádřit pomocí matematických vztahů. Tyto vztahy musí být přísně algebraické a nesmí obsahovat žádné náhodné složky. Náhodné prvky v čase služeb mohou způsobit náhodná zpoždění v aplikačním software a mohou způsobit, že aplikace náhodně nestihne stanovené RT časové termíny – a tento scénář je jasně nepřijatelný pro RT vestavěné systémy. [18]

5.8.1 Prioritní systém a multitasking

V závislosti na verzi operačního systému máme k dispozici určitý počet úrovní přerušení. Jak již bylo řečeno, operační systém je preemptivní a prioritní systém je právě oním klíčem, který určuje, jak jsou postupně přidělována časová kvanta jednotlivým běžícím vláknům. Velikosti časového kvanta, po které je vláknu dedikován procesor, jsou také v různých verzích operačního systému různě řízena. Intenzita, se kterou jsou potom jednotlivá vlákna vykonávána, pak nezáleží pouze na prioritním schématu, ale i na délce nastavených časových kvant. [17]

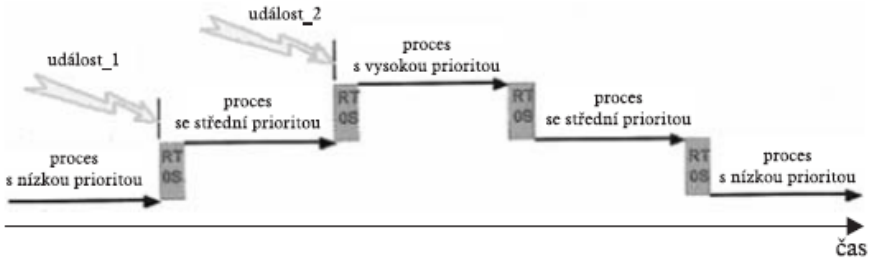
Přiřazení vláken do jednotlivých prioritních skupin definuje, jak jsou časová kvanta kernelem přidělována. Základní pravidlo stanovuje, že jsou dříve obsluhována vlákna s vyšší

prioritou. Pokud však v systému existuje více vláken se stejně vysokou prioritou, pak jsou jim časová kvanta cyklicky přidělována. Aktuálně běžící vlákno tak může být přerušeno pouze kernelem při vypršení kvanta, vláknem s vyšší prioritou a samozřejmě obsluhou hardwarového přerušení. [17]

Jedinou výjimkou z pravidla jsou vlákna s prioritou 0, která je také označována jako THREAD_PRIORITY_TIME_CRITICAL. Běh vlákna s takovou prioritou není nikdy kernelem přerušen a vlákno je vykonáváno až do svého konce nebo do chvíle, kdy přijde do stavu čekání na systémový zdroj. Zde je velmi nutné, aby vlákno nebylo vykonáváno příliš dlouho, poněvadž může dojít k degradaci odezvy operačního systému o vnější podněty. [17]

5.8.2 Inverze priorit

Inverzí priorit rozumíme situaci, která nastane, pokud vlákno s vyšší prioritou požaduje přístup k systémovým zdrojům, které v danou chvíli právě exkluzivně drží vlákno s nižší prioritou. V tomto případě dojde k preempci do vlákna s vyšší prioritou, které však nemůže běžet díky zablokovanému systémovému zdroji. To je velice nepříjemná situace, zejména v RTOS. Jediným řešením je umožnit vláknu, které systémový zdroj drží, co nejrychleji doběhnout a umožnit tak i jiným vláknům pokračovat v jejich činnosti. K vyřešení této situace Windows používá systém inverze priorit, který umožní vláknu s nižší prioritou zdědit prioritu kritického vlákna, rychle vykonat potřebné operace až do chvíle uvolnění požadovaného systémového zdroje a dále pak nechat pokračovat v práci kritické vlákno. [17]



Obrázek 16 - Příklad prioritního preemptivního plánování na časové ose [18]

5.8.3 Synchronizační mechanismy

Jedním z požadavků na RTOS je existence dobrých synchronizačních mechanismů, kterými se zabezpečuje korektní souslednou paralelně běžících aplikací a dále pak se chrání přístup ke sdíleným systémovým či jiným zdrojům. [17]

Stejně jako jiné operační systémy, i Windows disponuje řadou systémových objektů, které umožňují realizovat synchronizaci běžících vláken (události, kritické sekce, mutexy, semafory atd.). Způsob, jakým jsou požadavky na synchronizaci zpracovávány, je silně

ovlivněn potřebou inverze priorit. Aby bylo inverzi vůbec možné realizovat, musí být jednotlivé požadavku na synchronizační objekty ukládány do FIFO front, podle priority vlákna, které požadavek vyvolalo. Při řešení problematiky inverze priorit pak plánovač kernelu reorganizuje požadavky v jednotlivých frontách. [17]

Pro synchronizaci je také nutná existence časovačů generujících události s nastavenou periodou. Ve schopnostech časovačů se jednotlivé verze operačního systému Windows dosti výrazně liší. Vždy jde však pouze o typ aplikace a časových nároků, které klademe na operační systém a úlohy na něm běžící. [17]

Potřeba přesně měřit čas může být také uspokojena pomocí interních výkonných čítačů, které mají mnohem vyšší rozlišení než je tomu u časovače generujícího zprávy. Jejich dostupnost je však podmíněna konkrétním typem hardware a podporou v OAL. Pokud tato podpora existuje, pak je možné měřit čas s přesností na jednotky mikrosekund. [17]

5.8.4 Zpracování přerušení

Primární úlohou real-time aplikací je včasná odezva na externí události, která je velmi striktně omezena časovým intervalem. Pomalá nebo dokonce chybějící odezva (nezpracování události) je považována za závažnou chybu operačního systému. Jedním z nejdůležitějších požadavků na RTOS je tedy co nejrychlejší odezva na externí událost. Ta je ve výpočetních systémech manifestována generováním přerušení. [17]

Aby se proces zpracování přerušení co nejvíce urychlil, je ve Windows rozdělen do dvou částí. První část zpracování přerušení je provedena obslužnou procedurou přerušení (Interrupt Service Routine – ISR) a druhá část potom pomocí vláknem služby přerušení (Interrupt Service Thread – IST). [17]

Úkolem obslužné procedury přerušení je co nejrychlejší obsloužení požadavku přerušení tak, aby žádající zdroj nemusel dlouho čekat. Nejčastěji jde pouze o načtení nebo vyslání požadovaných dat. Další případné zpracování či analýza dat musí být provedena později. V souvislosti se zpracováním přerušení se vždy objeví otázka možnosti vnořených přerušení. Vnořeným přerušením se rozumí možnost přerušení obsluhy aktuálního přerušení jiným, nejčastěji se stejnou nebo vyšší prioritou. Zde je nutné uvést, že priorita přerušení nemá nic společného s prioritou vláken nebo procesů. Priorita přerušení může být dána jednak hardwarovým nebo softwarovým řešením. I přes existenci vnořených přerušení je vysoce žádoucí, aby obslužné procedury byly co nejkratší a nejrychlejší. [17]

5.8.5 Latence přerušení

Latencí přerušení se rozumí časová prodleva, která uběhne mezi časem, kdy vyvolané externí přerušení se dostane k procesoru a momentem, kdy je spuštěna obslužná funkce

přerušení. Při popisu RTOS, ale zejména při návrhu RT aplikace je nutné přesně znát hodnoty této časové prodlevy. [17]

Druhá část zpracování dat přerušení se provádí pomocí IST. IST je systémové vlákno, které je většinu času v čekacím stavu. Jakmile doběhne obsluha procedury přerušení, je na základě její návratové hodnoty obnoven běh patřičného IST vlákna. To pak provádí dodatečné zpracování potřebných dat, která byla načtena, nebo mají být příště vyslána. Libovolné IST vlákno však může být přerušeno ISR procedurou, aby byla zaručena co nepromptnější odezva.

6 Testování rychlosti šifrovacích algoritmů

Předposlední kapitola je věnována samotnému testování rychlosti šifrovacích algoritmů pomocí vytvořeného programu. V první části jsou nastíněny metody, pomocí kterých se dá k měření přistupovat. Dále se tato kapitola zabývá samotným programem, tedy jeho vytvořením, popisem a návodem, jak obsažené funkce programu správně využívat. Poslední část kapitoly je věnována zhodnocení naměřených výsledků.

6.1 Obecný úvod do testování

Předmětem našeho testování bude změřit časovou náročnost šifrování a dešifrování dat. Délka dat bude proměnná a naším úkolem bude opakovaně měřit časovou závislost vzhledem k proměnné délce použitého klíče a objemu šifrovaných dat pro vybrané šifrovací algoritmy. Měření bude nutné provádět s co největší možnou rozlišovací schopností. Při testování by také bylo vhodné zohlednit (zaznamenat) nezbytné systémové prostředky, jako spotřebu paměti, výkon procesoru apod. (viz kap. 6.7).

Aby měly jednotlivá měření vypovídající hodnotu, bude nutné provádět opakované testování ve větších statistických celcích (jednotkách), abychom mohli pomocí metod pro zpracování statistických dat vyloučit, případně alokovat některé odchylky. Dále bude možné z naměřených hodnot počítat průměrné hodnoty, které by měly poskytnout určité vypovídací hodnoty. Všechna naměřená data a výchozí podmínky pro testování budou zaznamenána a bude na ně brán potaz při budoucím vyhodnocování.

Veškeré měření by mělo být prováděno pomocí programu vytvořeného v jazyce C#. Jednotlivé šifrovací algoritmy nejsou předmětem programování, a proto budou v programu použity již existující funkce z knihoven .NET, které se nacházejí na webových stránkách .NET Framework Developer Center společnosti Microsoft.[1]

Programem naměřené časové hodnoty sice nebudou mít přímou vypovídací hodnotu (viz kap. Algoritmy), avšak i tak by z nich mělo být možné vyhodnotit a porovnat náročnost jednotlivých algoritmů a potvrdit tak obecně známá fakta o jejich náročnosti.

Dále bude možné takto naměřená data porovnat s časy naměřenými na hardwarových prostředcích k šifrování přímo určených a učinit tak obecné závěry nejen k náročnosti algoritmů, ale také stanovit určitou přesnost takto naměřených hodnot.

---

[1] odkaz: http://msdn.microsoft.com/en-us/library/system.security.cryptography.aspx

6.2 Metodika testování

Každého ihned napadne, že dnes určitě existuje nejedna možnost, jak výše uvedené testy za pomocí programovacího jazyka C# realizovat. Nám se v současné době nabízejí tři schůdné metody, avšak každá z nich má své výhody a nevýhody, které spočívají především v přesnosti naměřených výsledků.

6.2.1 Metoda DateTime

Třída: DateTime
Jmenný prostor: System

Třída DateTime slouží k vyjádření aktuálního času s maximálním rozlišením 100 nanosekund. Třída DateTime funguje tak, že načítá počet tiků uplynulých od 1.ledna roku 0, 0:0:0 hodin, přičemž doba jednoho tiku trvá 100 nanosekund. Třída obsahuje velké množství metod a vlastností, pomocí kterých můžeme aktuální čas vyjádřit.

Pomocí této třídy lze změřit uplynulý čas následujícím způsobem. Pro odstartování měření uložíme aktuální čas (vytvořením nové instance) a pro ukončení měření uplynulého času opět uložíme nový aktuální čas. Výsledný čas získáme odečtením první uložené hodnoty od druhé. Tím získáme uplynulý čas měření. Reálně je však pomocí této metody možné vyjádřit čas řádově pouze v mikrosekundách. Měření touto metodou je v podstatě podobné jako u třídy Stopwatch, avšak zde je měření méně přesné a hodí se tedy pro krátké časové intervaly.

6.2.2 Metoda Stopwatch

Třída: Stopwatch
Jmenný prostor: System.Diagnostics

Třída Stopwatch slouží k měření uplynulého času s co největším možným časovým rozlišením. Třída může měřit uplynulý čas buď v jednom intervalu, nebo celkový čas v několika nezávislých intervalech. Měření času se provádí pomocí dvou metod Start a Stop. Metoda Start spustí měření uplynulého času a metoda Stop naopak měření zastaví. Celkový uplynulý čas poté můžeme získat pomocí vlastností Elapsed (Elapsed, ElapsedMilliseconds, ElapsedTicks), jež určují v jakém výstupním formátu je čas předán.

Pro přesná velmi přesná měření použijeme čas naměřený ve vlastnosti ElapsedTicks. V této vlastnosti je uložen počet tiků, jež proběhly během měření. Jeden tik trvá na každém hardwarovém sestavení různě dlouho a je určen frekvencí. Pro převedení na časovou hodnotu

je nejdříve nutné vypočítat, kolik tiků proběhne na konkrétním sestavení za vteřinu, což provedeme pomocí vlastnosti Frequency. Pokud podělíme číslo 109 frekvencí, získáme časový údaj odpovídající délce jednoho tiku v ns. To je tedy nejvyšší rozlišovací schopnost časovače.

Frekvence časovače (timer Stopwatch) ukazuje jeho přesnost a rozlišovací schopnost. Velikost frekvence záleží na rozlišovací schopnosti časovacího mechanizmu, jež časovač řídí. Pokud instalovaný hardware a operační systém podporují vysoké rozlišení hodnoty counteru, pak frekvence odpovídá frekvenci counteru. Jinak je hodnota frekvence odvozena ze systémového času. Frekvence časovače je závislá na instalovaném hardware, a proto se na každém PC mění.

Pokud podělíme číslo 109 frekvencí, získáme časový údaj odpovídající délce jednoho tiku v nanosekundách (údaj s přesností typu v něm je hodnota uložena). To je tedy nejvyšší rozlišovací schopnost časovače. Naměřený uplynulý čas v ticích pak tedy snadno převedeme na čas v základních jednotkách vynásobením počtu tiků s délkou jednoho tiku.

Takto naměřený čas pak má maximální rozlišovací schopnost, která se pohybuje v řádu jednotek nanosekund. Tato metoda se tedy hodí především pro měření krátkých časových intervalů s maximálním rozlišením.

Výše uvedené metody mají jednu velkou nevýhodu. Při měření času provádění výpočtů je možné, že dojde k dočasnému přerušení výpočetního procesu (vlákna) v důsledku jiného spuštěného procesu, avšak měření času se provádí i nadále. Záleží však také na jejich prioritě. Naměřené časy pomocí těchto metod je tedy nutné brát s určitou rezervou a nebo je možné tyto chyby částečně vyloučit pomocí velkého počtu měření a výslednou dobu poté zprůměrňovat.

6.2.3 Metoda měření vykonaných instrukcí

Tato metoda je pro měření uplynulého času nejpřesnější a spočívá v samotném měření použitých instrukcí při prováděných výpočtech. Její nespornou výhodou oproti předchozím dvěma metodám je její přesnost, protože zde není nutné zohledňovat přerušování výpočetního procesu. Jednotlivé instrukce mají pevnou dobu časového trvání, a proto je pouze nutné vyčíslit počet použitých instrukcí a dále zjistit počet instrukcí za vteřinu, jež je schopno jednotlivé hardwarové sestavení (PC) při dané taktovací frekvenci vykonat. Takto získané údaje podělíme a získáme tak přesné vyjádření času při prováděných výpočtech.

Tato metoda se hodí pro nejpřesnější měření časových intervalů, avšak její aplikace na měření je velmi složitá.

6.3 Program pro testování rychlosti šifrovacích algoritmů

Sestavením rešerše šifrovacích algoritmů, popisem analýzy výkonnosti algoritmů a nastíněním metod pro testování rychlosti algoritmů se diplomová práce přesunula do své druhé, praktické části, ve které je již bylo za úkol vytvořit testovací program.
Následující kapitola je tedy věnována programu určenému pro testování rychlosti šifrovacích algoritmů. V kapitole je zmapován hrubý postup vývoje a tvorby programu a v další části se nachází popis programu a jeho používání.

6.3.1 Tvorba programu

V první fázi jsem se přibližně půl roku seznamoval se syntaxí programovacího jazyka C# a s prostředím Visual Studio 2007 od společnosti Microsoft, ve kterém měl být program realizován.

Program jsem vytvářel jako aplikaci typu WinForm, kde se dá velmi rychle docílit požadovaného vzhledu a funkcionality budoucího programu pomocí vytvořených komponent (Form, Button, TextBox, apod.).

Před samotným vytvořením programu jsem nevytvářel vývojové diagramy, protože mi budoucí struktura programu nepřipadala natolik složitá.

První verzi programu jsem psal přímo do hlavní třídy formuláře a pouze jsem testoval a ověřoval funkcionality daných tříd určených pro šifrování. Kód tak byl velmi neuspořádaný, nepřehledný a nesplňoval tak nepsaná pravidla pro tvorbu „čtivého“ kódu. Navíc bylo obtížné a zdlouhavé do této verze vkládat další algoritmy a nešlo provádět několikanásobné měření. Po vytvoření této první verze byla již velmi zřetelná struktura, kterou by se měla tvorba programu ubírat.

V následujících verzích programu jsem si již navrhnul vlastní třídy, které spojovaly implementaci tříd pro jednotlivé šifrovací algoritmy (viz Přílohy – Výpis tříd). Ke každé z nich jsem také vytvořil třídu, která mohla provádět několikanásobnou implementaci opakujících se úkonů. Vytvořil jsem tedy samostatné třídy pro symetrické a asymetrické šifrování a třídy pro tvorbu hashe. Každá z těchto tříd měla vlastní metody a vlastnosti, pomocí kterých bylo možné provádět měření času. Pro měření času je v programu implementována třída Stopwatch, která se mi z hlediska implementace a přesnosti jevila jako nejvhodnější. Každá z uvedených tříd je dále doplněna třídou pro vícenásobné opakování algoritmu, která danou třídu implementuje. Pomocí první třídy se tedy provádějí úkony daného šifrovacího algoritmu včetně měření času. Implementací v doplňkové třídě můžeme daný proces opakovat a naměřené hodnoty pak předat do vlastností, pomocí kterých můžeme

hodnoty předat k dalšímu zpracování (např. pro grafické zobrazení nebo statistické vyhodnocování).

Po vytvoření funkční kostry programu s vlastními třídami implementujícími šifrovací algoritmy přišlo na řadu oddělení ovládací a šifrovací části programu do samostatných procesů, aby bylo možné při spuštěných výpočtech program kontrolovat. Oddělil jsem tedy obě popsané části programu do samostatných vláken.

V další části tvorby programu jsem se věnoval tvorbě grafického zobrazení naměřených hodnot. Pro zobrazování hodnot jsem použil knihovnu ZedGraph, která se pro tyto účely zdála jako vhodná alternativa a nabízela nepřeberné množství možností. Implementace knihovny nebyla po nastudování jejích metod a vlastností příliš složitá.

Dalším úkolem, který jsem v programu řešil, byl způsob exportu (uložení) naměřených dat. Pro výstupní soubor padla volba na formát XML. Pro export hodnot do formátu XML jsem přímo použil knihovnu od společnosti Microsoft.

Po vytvoření funkčního programu bylo nutné upravit program po grafické stránce, přičemž jsem všechny prvky sjednotil a program rozdělil na část pro vstupní data a na část ovládací spolu se zobrazením naměřených hodnot (viz kapitola 6.3.3 Popis programu).

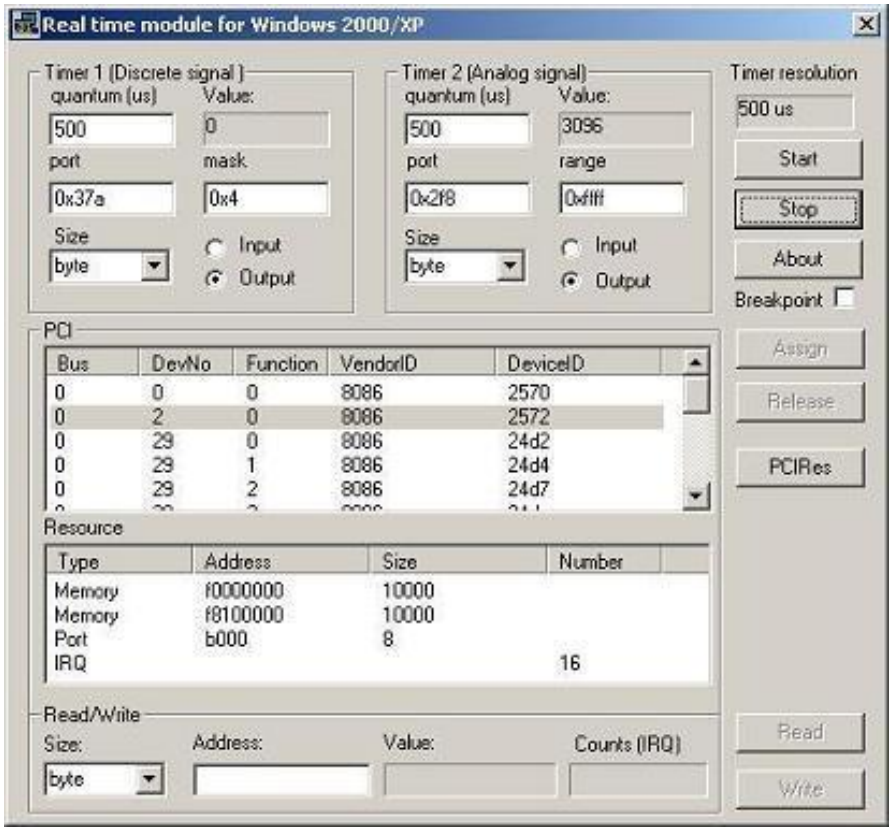
6.3.2 Měření rychlosti šifrovacích algoritmů a jeho nedostatky

Po vytvoření funkčního programu se všemi potřebnými funkcemi přišlo na řadu samotné měření rychlosti šifrovacích algoritmů.

Po několika testovacích měřeních bylo z většiny grafů zřejmé, že při měření časů dochází k určité chybě. Při souvislém opakovaném měření času šifrování/dešifrování se v naměřených hodnotách v pravidelných intervalech objevují velmi vysoké hodnoty časů (špičky), které zkreslují výsledky měření (v krajních případech dosahují až 30 %). Při měření času pomocí odděleného vlákna se tato chyba sice trochu eliminuje, ale ve výsledku nemá na měření příliš velký vliv.

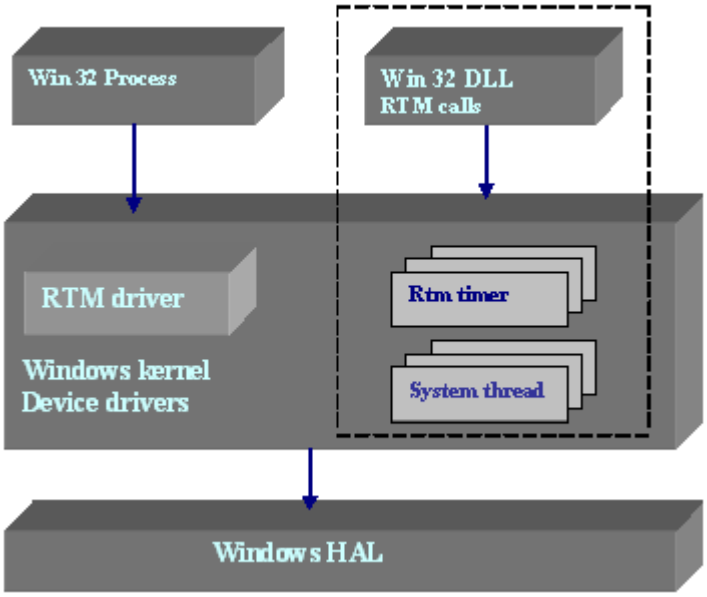
V měření jsem tedy dále pokračoval tím, že jsem se snažil analyzovat a odstranit zdroj těchto špiček. Dále jsem zjistil, že se zvětšujícím se objemem šifrovaných dat se četnost těchto špiček zvyšuje. Nejpravděpodobnějším vysvětlením vzniku špiček je obsluha zpracování přerušení operačního systému při přepínání procesů (multitaskingu). Tato obsluha přiděluje jednotlivým vláknům běžících v operačním systému určitý definovaný čas, při kterém mají přístup k systémovým prostředkům (viz kapitola 5.8).

Po selekci zdroje chyb jsem se jej snažil eliminovat. Nejprve přišly na řadu testy s programem **Real time module for Windows** [2] (freeware). Program obsluhuje nastavení ovladače, který je schopen řídit časovač přerušení, avšak jeho spolupráce je možná pouze se systémy Windows 2000 a XP. Po nastavení příslušných hodnot časů pro časovač stačí uvést nastavení do chodu pomocí tlačítka Start (pomocí tlačítka Stop naopak zastavit). Při takto spuštěném programu jsem následně sledoval, jak se promítne jeho nastavení do samotného měření. Po několika měřeních jsem zjistil, že program bohužel měření vůbec neovlivnil a ve výsledných grafech se špičky objevovaly se stále stejnou četností a intenzitou.



Obrázek 17 - Program pro ovládání časovače přerušení [19]

[2] odkaz: http://www.codeproject.com/KB/system/RealTimeModule.aspx?msg=2820096



Obrázek 18 - Schéma ovládání modulu časovače pomocí programu [19]

Po tomto neúspěchu jsem narazil na další dvě možnosti. První spočívá v metodice synchronizace vláken a druhá, méně pravděpodobná ve vynucení úklidu systémových prostředků pomocí Garbage collectoru. Následkem toho jsem provedl drobné úpravy zdrojového kódu. Nejprve jsem se zaměřil na spouštění garbage collectoru, jehož vynucení jsem nastavil vždy po provedení bloku kódu, které obsluhuje měření. Po testování jeho vlivu na naměřené hodnoty jsem bohužel opět neregistroval žádné výrazné změny oproti dosavadnímu měření.

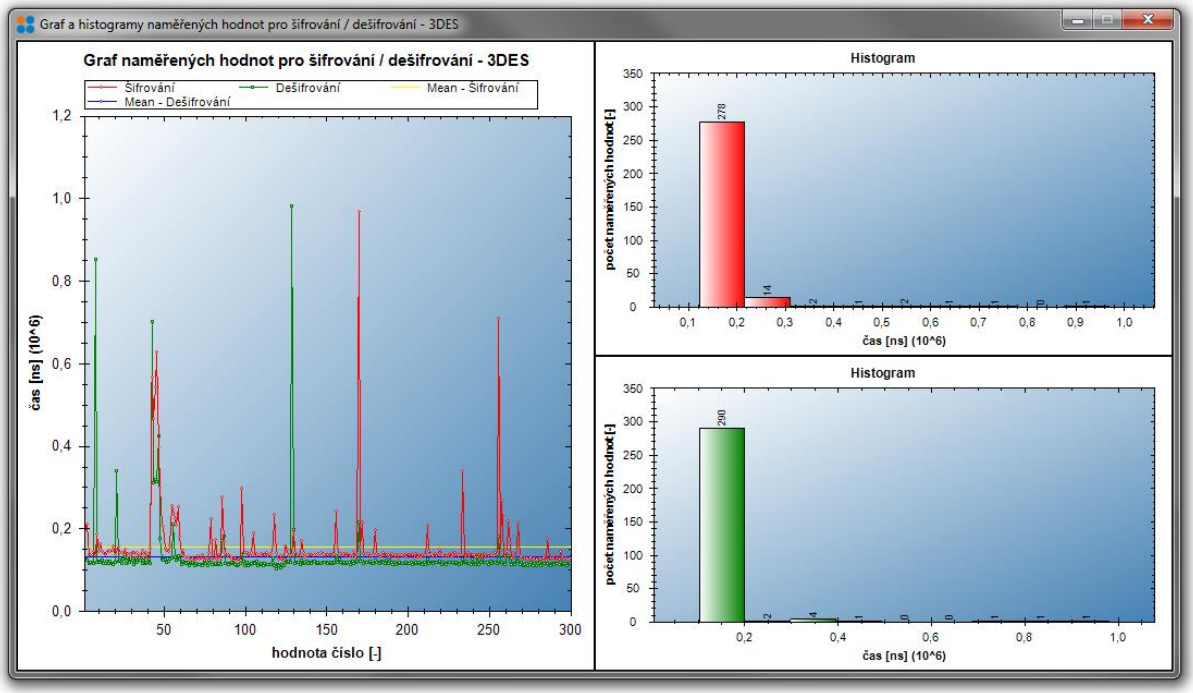
Dále jsem se tedy soustředil na metodiku synchronizace vláken. V prvé řadě jsem otestoval vliv metody Interrupt, která po zavolání na zablokované vlákno násilně ukončí jeho zablokování, které je způsobeno přepínáním procesů. Po otestování programu jsem zjistil, že výskyt špiček v naměřených hodnotách se mírně snížil, avšak vynuceným přerušením docházelo k desynchronizaci vláken a tím i jejich pádu (program např. provedl pouze určité procento měření a poté se vlákno zhroutilo a nebylo tak schopné předat výsledky). Toto slibné řešení jsem tedy musel zavrhnout.

Na konec jsem se chybu snažil vyřešit pomocí třídy Mutex a Monitor, které jsou přímo určeny k řízení synchronizace jednotlivých vláken a tím zajišťují kontrolu přístupu k systémovým zdrojům. Při použití třídy Mutex se použití jejích metod nepatrně projevilo pouze při spouštění ladící verze.

V testovacím programu jsem tedy následně ponechal všechny inovace, které mohly mít případný vliv na měření. Po následném porovnání naměřených hodnot před a po aplikaci výše uvedených změn v programu mohu konstatovat, že celkový výskyt špiček se mírně snížil a

výsledné hodnoty jednotlivých měření se nyní nerozcházejí tak často, avšak problém rozhodně eliminován nebyl.

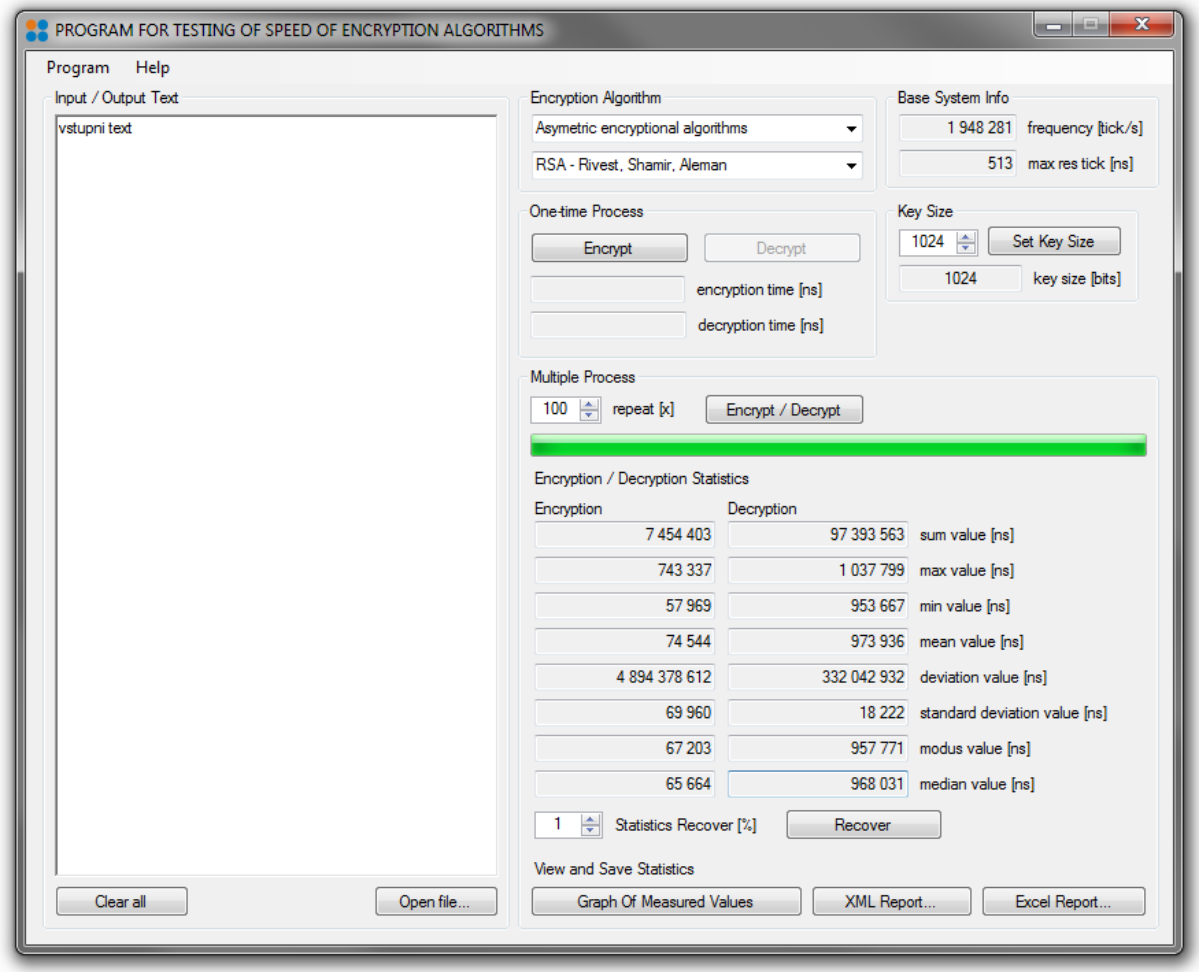
Jediným východiskem pro reprodukovatelnost měření bylo aplikovat na naměřené hodnoty statistické metody a vyřadit tak problémové hodnoty ze statistického celku. Proto jsem do programu zahrnul i funkci, která po nastavení procenta chybovosti nahradí případné odchylky průměrnými hodnotami. Již při nastavení 2-3 % chybovosti měřených hodnot se průměrné hodnoty naměřených časů velmi ustalují a reprodukovatelnost procesu je tak velmi vysoká.



Obrázek 19 - Graf naměřených hodnot obsahující špičky

6.3.3 Popis programu

Program pro testování rychlosti šifrovacích algoritmů se skládá z jednoho základního okna, které můžeme rozdělit na dvě funkční části, část pro vstup dat určených k šifrování a na část ovládací, která také zobrazuje naměřené výsledky.



Obrázek 20 - Výchozí okno programu

Levá část programu obsahuje velké okno - *Input / Output Text*, do kterého může uživatel zadat vstupní data, které budou použita při šifrování. Při samotném šifrování a dešifrování toto okno zobrazuje aktuální stav dat. Dále pod oknem pro vstupní data nalezneme dvě tlačítka. První, levé tlačítko, slouží k vymazání dat v okně a pomocí tlačítka na pravé straně můžeme data načíst z libovolného souboru s textovými daty.

Pravá část programu slouží k ovládání a zobrazení naměřených hodnot.

V horní pravé části nalezneme sekci - *Base System Info*, se základními informacemi o frekvenci časovače a maximálním rozlišení časovače daného počítače, na kterém je program spuštěn.

V prostřední části nahoře se nacházejí ovládací položky - *Encryption Algorithm*, ve kterých si můžeme zvolit typ šifrovacích algoritmů, které pro daný úkon použijeme.

Po volbě typu algoritmů si následně můžeme zvolit konkrétní typ šifrovacího algoritmu z dané skupiny. V nabídce jsou:

- Asymetrické šifrovací algoritmy
    - RSA
- Hashovací algoritmy
    - MD5
    - SHA1
    - SHA256
- Symetrické šifrovací algoritmy
    - AES
    - DES
    - 3DES

Pokud si při volbě typu šifrovacího algoritmu zvolíme asymetrický typ, zobrazí se dále pod sekcí *Base System Info* další sekce – *Key Size*, ve které si můžeme nastavit velikost klíče pro asymetrický šifrovací algoritmus. Pomocí nastavovacího políčka zvolíme hodnotu klíče a nastavíme pomocí tlačítka *Set Key Size*. Čas pro nastavení klíče může být velmi dlouhý a proto nás program v případě větší hodnoty sám upozorní, průběh nastavení také můžeme sledovat v políčku *key size* (pokud se klíč nastavuje, zobrazuje se hodnota processing) a při nastaveném klíči se přímo zobrazí jeho hodnota.

Dále se pod volbou algoritmů nachází část pro proces šifrování s jedním opakováním – *One-time Process*. Ten obsahuje tlačítka – *Encrypt* a *Decrypt*, která zajistí zašifrování a dešifrování dat ve vstupním okně. Dešifrování je možné pouze po zašifrování dat a naopak. V textových polích pod těmito tlačítky se nám poté zobrazí naměřené hodnoty daného šifrovacího procesu.

Celá pravá dolní část je vyhrazena pro část s několikanásobným opakováním – *Multiple Process*. V jeho horní části se nachází ukazatel počtu opakování (jeho maximální hodnota je 1000 opakování) a vedle něj tlačítko – *Encrypt / Decrypt*, které spustí vybraný šifrovací proces. Ihned pod těmito prvky je umístěn ukazatel průběhu zpracování.

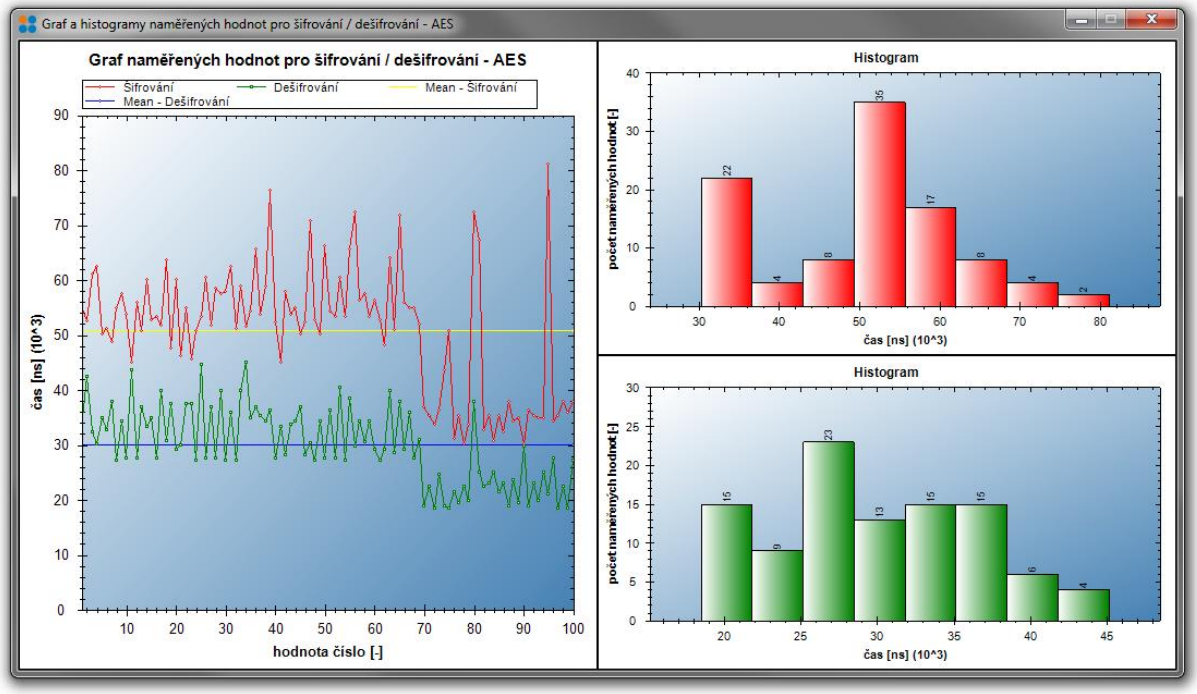
Dále následují textová pole, která zobrazují statistické veličiny v procesu šifrování.

- *sum value* - obsahuje celkový čas strávený opakovaným šifrováním (případně dešifrováním).
- *max value* – zobrazuje maximální hodnotu naměřeného času stráveného na jednu šifrovací operaci
- *min value* - zobrazuje minimální hodnotu naměřeného času stráveného na jednu šifrovací operaci.
- *mean value* - zobrazuje průměrnou hodnotu ze statistického celku naměřených hodnot.
- *deviation value* - zobrazuje rozptyl hodnot statistického celku.

- *deviation value* - zobrazuje směrodatnou odchylku statistického celku.
- *modus value* – zobrazuje hodnotu modusu ze statistického celku
- *median value* – zobrazuje hodnotu mediánu ze statistického celku.

Pod políčky zobrazujícími statistické veličiny nalezneme tlačítko *Recover* spolu s ukazatelem procent *Statistics Recover*. Tato políčka slouží k eliminaci hodnot zkreslujících statistické veličiny (nežádoucí špičky). Pomocí ukazatele procent nastavíme, kolik procent hodnot chceme ve statistickém celku nahradit průměrnými hodnotami a pomocí tlačítka *Recover* úpravu provedeme.

V poslední, dolní části oblasti *Multiple Process* se nachází tři tlačítka. Tlačítko *Graph of measured times* slouží ke grafickému zobrazení naměřených hodnot. Po stisku tlačítka vyskočí nové okno obsahující graf a histogram naměřených hodnot. Tlačítko *XML Report*… slouží k uložení aktuálně naměřených hodnot ve formátu XML, jež mohou sloužit k dalšímu zpracování. Tlačítko *Excel Report*... slouží pro export statistického vyhodnocení měření do excelovské tabulky.



Obrázek 21 - Okno programu se zobrazeným grafem naměřených hodnot

V horní části programu se nachází menu, které obsahuje položky Program – Close – zavře program a položku Help – About program – zobrazí informace o programu.

6.3.4 Používání programu

Měření rychlosti šifrovacích algoritmů pomocí vytvořeného programu provedeme po jeho spuštění následujícím způsobem.

Po spuštění programu se nám objeví základní okno s výše uvedenými položkami (obr. 17). Do okna *Input / Output* text zadáme text, který chceme šifrovat. V pravé části v sekci *Encryption Algorithm* následně zvolíme typ šifrovacího algoritmu.

Poté již můžeme provést samotné šifrování, buď přímo pomocí tlačítek tlačítek *Encrypt* a *Decrypt*, nebo několikanásobným opakováním, kdy nastavíme v okénku *repeat* počet opakování a proces spustíme pomocí tlačítka *Encrypt* / *Decrypt*. Při přímém šifrování se nám výsledný čas zobrazí v oknech přímo pod tlačítky a výsledek operace také můžeme vidět v okně *Input / Output*. Při vícenásobném opakování se nám po provedení procesu zobrazí souhrn naměřených výsledků v oknech sekce *Encryption / Decryption Statistics*.

Naměřené hodnoty jsou většinou zkresleny kvůli uvedeným problémům se špičkami. Na jejich částečné odstranění a upravení hodnot slouží tlačítko *Recover* a okénko *Statistics Recover* (nastavení procenta), které upraví nastavené procento hodnot na průměrnou hodnotu.

Naměřené výsledky si následně můžeme zobrazit pomocí grafů v novém okně pomocí tlačítka *Graph of measured times* nebo exportovat do XML souboru pomocí tlačítka *XML report*… Pokud budeme provádět několik měření vícenásobného procesu, můžeme si graf naměřených hodnot nechat otevřený, provést další měření a opět hodnoty zobrazit (otevře se další nové okno). Takto si můžeme otevřít naměřené hodnoty ve více oknech a případně je i porovnávat.

6.4 Naměřené hodnoty a jejich porovnání

Měření jsem prováděl na dvou operačních systémech - Windows XP a Windows 7 s nainstalovaným .NET Frameworkem verze 3.5, pro tři základní velikosti vstupních textů – krátký, střední a dlouhý (viz Přílohy).

Aby bylo měření reprodukovatelné, dodržoval jsem při měření pro vstupní texty jednotlivých velikostí na všech počítačových sestavách následující postup:

- Nejprve jsem spustil program (případně nastavil velikost klíče RSA),
- nastavil jsem příslušné hodnoty opakování (1000x) a vložil vstupní text,
- poté jsem spustil proces šifrování (dešifrování) pro zvolený algoritmus,
- následně jsem upravil naměřené hodnoty pomocí *Statistics Recover* o 3 % (dané procento nejvyšších hodnot - špiček se upraví na průměrnou hodnotu ze statistického celku) z důvodu vzniklých odchylek při měření,

- hodnoty jsem vykopíroval a uložil do tabulky,
- pro další velikost vstupního textu jsem program restartoval.

Při měření algoritmu RSA jsem používal následující velikosti klíčů: 1024, 5000 a 12000 bitů, aby bylo možné vstupní text (viz Přílohy - krátký text – 68 znaků, střední text – 529 znaků a dlouhý text – 1383 znaků) pomocí klíče zašifrovat. Se zvětšující se délkou klíče se také velmi zvyšovala doba potřebná pro změření hodnot, která se v krajním případě pohybovala v řádu několika hodin. Na některých počítačových sestavách se bohužel nezdařilo měření algoritmu RSA při použití většího klíče, protože z naměřených hodnot nebylo možné provést statistické vyhodnocení z důvodu jejich přílišné velikosti, které způsobilo přetečení bufferu nastaveného typu proměnné při jeho zpracování. Případné měření nebylo možné provést z důvodu velké časové náročnosti, která se pohybovala v řádu desítek hodin.

Pro měření pomocí zbývajících šesti algoritmů nebylo nutné nastavovat žádné speciální parametry a pro všechny velikosti vstupních textů se tedy měření provádělo vždy stejným způsobem. Veškerá měření pro tyto algoritmy proběhlo bez problémů.

Měření jsem prováděl celkem na devíti počítačových sestavách, přičemž na osmi z nich v operačním systému Windows XP a na třech v operačním systému Windows 7.

Počítačové sestavy byly velmi rozmanitého výpočetního charakteru od klasických stolních PC, přes notebooky až po netbook.

Tabulka 2 - Počítačové sestavy použité pro měření rychlosti šifrovacích algoritmů

Počítačová sestava	CPU		RAM	OS
	frekvence [MHz]	typ (označení)	kapacita [MB]	
PC 1 - Athlon	1840	Athlon 2200+	768	Windows XP
PC 2 - Sempron	1920	Sempron 2800+	736	Windows XP
Notebook - Asus F5GL	2000	Dual Core T3200	1750	Windows XP
Notebook - Acer BL50	1600	Celeron M	248	Windows XP
Notebook - Asus K50IJ	2000	Core 2 Duo T5870	3000	Windows XP / 7
Notebook - Asus A6J	1660	Centrino Duo T2300	512	Windows XP
Notebook - MSI GX623	2200	Core 2 Duo T6600	4000	Windows XP / 7
Netbook - Asus Eee PC	1600	Atom N270	1000	Windows XP
Notebook - Compaq	1900	Athlon Dual-Core QL-60	3000	Windows 7

6.4.1 Zhodnocení naměřených výsledků z grafů

Z prvního povrchního náhledu na výsledné grafy lze obecně konstatovat, že se zvětšující se délkou vstupního textu pro šifrování se úměrně zvětšoval i čas potřebný pro zašifrování (resp. vytvoření hashe) a to platí pro oba 32-bitové operační systémy, na 64-bitovém OS Windows 7 byly naměřené časy přibližně poloviční.

Pokud dále budeme zkoumat jednotlivé typy šifrovacích algoritmů, lze z grafů vyvodit následující závěry.

Asymetrický šifrovací algoritmus RSA ukázal největší exponenciální závislost změny délky vstupního textu (resp. klíče) na nárůst časové náročnosti výpočtu. Při zvětšení textu o desetinásobek byl potřebný čas na zašifrování stonásobný. Pro dešifrování zašifrované zprávy vzhledem k délce vstupního textu lze při pohledu na časovou náročnost konstatovat stejný závěr. Rozdíl mezi dobou šifrování a dešifrování měl také stejnou závislost, přičemž dešifrování krátkého textu oproti jeho šifrování trvalo desetkrát déle a pro dlouhý text byla tato doba stonásobná (viz Přílohy - Graf 1.1, 2.1). Pokud budeme hodnotit asymetrický šifrovací algoritmus z hlediska 32-bitového operačního systému, lze říci, že OS Windows XP byl přibližně stejně rychlý jako OS Windows 7. Naopak 64-bitový OS Windows 7 byl při šifrování algoritmem RSA až pětkrát rychlejší než 32-bitový OS Windows XP (viz Přílohy - Graf 3.1).

Asymetrické šifrovací algoritmy se tedy hodí spíše k šifrování klíčů či přístupových hesel, než pro šifrování plynulého textu či komunikace, protože jsou velmi pomalé, ale zároveň bezpečné.

Naměřené časy:

- Šifrování – 100 μs – 20 ms
- Dešifrování – 1 ms – 2 s

Hashovací algoritmy ukázaly přibližně lineární závislost doby vytvoření hashe na změně délky vstupního textu. S narůstající délkou textu se úměrně zvyšuje doba pro vytvoření hashe asi v poměru 10:1. Z hlediska rychlosti vytvoření hashe jsou algoritmy MD5 a SHA1 téměř identické. Algoritmus SHA256 je v porovnání s předchozími algoritmy rychlejší pro krátké texty, ale při delších textech je naopak pomalejší (viz Přílohy - Grafy 1.2 – 1.4 a Grafy 2.2 – 2.4).

Při porovnání naměřených časů z hlediska operačních systému můžeme učinit podobné závěry jako u algoritmu asymetrického (viz Přílohy - Grafy 3.2 – 3.4).

Hashovací algoritmy jsou velmi rychlé, proto se úspěšně používají při šifrování datové komunikace, na druhou stranu nejsou tolik bezpečné.

Naměřené časy:

- Vytvoření hashe – 10 μs – 100 μs

Symetrické šifrovací algoritmy ukázaly také přibližně lineární závislost doby zašifrování podobně jako hashovací algoritmy. Pokud zhodnotíme dešifrování zprávy, můžeme zde sledovat jeden zajímavý fakt. Při dešifrování zprávy se celkový čas vzhledem k délce textu nemění, dešifrování krátké a dlouhé zprávy trvá přibližně stejnou dobu. Nejrychlejším symetrickým algoritmem pro šifrování a dešifrování je AES, algoritmy DES a 3DES jsou přibližně třikrát pomalejší (viz Přílohy - Grafy 1.5 – 1.7 a Grafy 2.5 – 2.7).

Při porovnání naměřených časů z hlediska operačních systému můžeme učinit stejné závěry jako u předchozích algoritmů. Zajímavé zjištění učiníme, pokud porovnáme naměřené časy v 32-bitovém a 64-bitovém OS. Pro všechny předchozí algoritmy platilo, že OS Windows 7 (64-bitový) byl vždy několikanásobně rychlejší, v případě algoritmů DES a 3DES byly naměřené časy velmi podobné (viz Přílohy - Grafy 3.6 – 3.7).

Naměřené časy:

- Šifrování – 30 μs – 250 μs
- Dešifrování - 20 μs – 100 μs

Z grafů závislosti střední hodnoty času šifrování a dešifrování na výpočetním zařízení můžeme učinit stejné závěry jako v předchozích odstavcích. Asymetrický šifrovací algoritmus je nejpomalejší, řádově 10x – 1000x při šifrování a řádově 50x – 5000x při dešifrování, ve srovnání se symetrickými a hashovacími algoritmy. Hashovací algoritmy jsou naopak nejrychlejší (viz Přílohy - Grafy 1.8 – 1.10 a Grafy 2.8 – 2.10).

Pokud porovnáme výpočetní rychlost počítačových sestav zhledem k jednotlivým šifrovacím algoritmům (detailnější zkoumání grafů, viz Přílohy CD - Grafy 1.8 – 1.10 a Grafy 2.8 – 2.10), můžeme říci, že podle očekávání byly nejnovější počítačové sestavy rychlejší oproti starším modelům (v tomto segmentu vévodily především modely s dvoujádrovým procesorem), ale rozdíl nebyl příliš markantní.

Velkým překvapením však byl starší jednojádrový procesor Sempron, který dosáhnul při výpočtech nejlepších výsledků. Nejpomalejší počítačovou sestavou byl netbook s jednojádrovým procesorem Atom N270, jehož výsledky byly v průměru dvakrát pomalejší.

Zajímavých výsledků také dosáhnul 64-bitový OS oproti 32-bitovému na dvoujádrovém 64-bitovém procesoru, kde byl ve většině algoritmů minimálně dvakrát rychlejší.

7 Závěr

V první části diplomové práce, v rámci semestrálního projektu 1, jsem vytvořil rešerši jednotlivých šifrovacích algoritmů a osvětlil některé základní pojmy, které s touto tématikou úzce souvisí.

V druhé části diplomové práce, v rámci semestrálního projektu 2, jsem popisoval měření výpočetní náročnosti algoritmů a možnosti jejich porovnávání a také jsem stanovil metody jejich měření v tomto projektu. Dále jsem v práci popisoval operační systémy reálného času, které s problematikou měření také úzce souvisejí.

Následně jsem se začal postupně seznamovat s programovacím jazykem C# a jeho syntaxí, protože cílem diplomové práce bylo vytvořit program, který bude vybrané šifrovací algoritmy testovat.

Po získání potřebného množství znalostí s oblasti programování jsem začal vytvářet samotný program. Postupně jsem jej zdokonaloval a rozšiřoval, ale také prováděl i velké množství oprav, než jsem program odladil do korektní finální podoby. Program je schopen provádět měření sedmi algoritmů – RSA, MD5, SHA1, SHA256, AES, DES a 3DES, přičemž provede statistické vyhodnocení a výsledky je následně možné zobrazit v přehledných grafech.

Poslední část diplomové práce je věnována měření algoritmů, jejich zpracování a vyhodnocení výsledků měření. Praktickým měřením jsem si ověřil, že asymetrické šifrovací algoritmy jsou řádově 100x – 1000x pomalejší oproti hashovacím a symetrickým šifrovacím algoritmům, což je ovlivněno především bitovou délkou šifrovacího klíče, ale také složitostí algoritmů, z čehož také vyplývá jejich bezpečnost.

Cílem diplomové práce bylo vytvořit program pro testování rychlosti šifrovacích algoritmů, což se ve výsledku podařilo, avšak při měření hodnot časů vzniká problém se špičkovými hodnotami, které je nutné dodatečně statisticky upravovat. Chyba je nejspíše způsobena operačním systémem Windows, který není schopen plně splnit požadavky systému reálného času. I přes tyto drobné problémy však bylo možné ověřit teoretické poznatky o rychlosti šifrovacích algoritmů.

8 Seznam odborné literatury

[1] WROBLEWSKI, PIOTR**.** *Algoritmy – Datové struktury a programovací techniky.* Brno : ComputerPress Brno, 2004. str. 351.

[2] Symetrická kryptografie. [Online] https://is.mendelu.cz/eknihovna/opory/zobraz_cast.pl?cast=7026.

[3] ŘÍHA, PAVEL**.** Šifrování – Úvod do problematiky. *Root.cz.* [Online] 17. 11 1999. [cit. 1999-11-17]. http://www.root.cz/clanky/sifrovani-uvod-do-problematiky.

[4] SCHNEIER, BRUCE**.** Schneier on Security. *Bruce Schneier.* [Online] 18. 2 2005. http://www.schneier.com/blog/archives/2005/02/cryptanalysis_o.html.

[5] Advanced Encryption Standard. *Wikipedia.* [Online] http://en.wikipedia.org/wiki/Advanced_Encryption_Standard.

[6] Data Encryption Standard. *Wikipedia.* [Online] http://en.wikipedia.org/wiki/Data_Encryption_Standard.

[7] TripleDES. *Wikipedie.* [Online] http://cs.wikipedia.org/wiki/TripleDES.

[8] KLÍMA, VLASTIMIL**.** Moderní Kryptografie. [Online] 11. 4 1997. http://crypto-world.info/klima/mffuk/Symetricka_kryptografie_I_2007.doc.

[9] ŠEVČÍK, DAVID. *Technologie datové bezpecnosti vnitrních sítí.* Zlín : Univerzita Tomáše Bati ve Zlíně, 2007. str. 67.

[10] AES Advanced Encryption Standard. *Clippers.* [Online] 2008. [cit. 2006-09-18]. http://www.clipperz.com/learn_more/crypto_foundations/aes_advanced_encryption_standard.

[11] Diffie-Hellman. *Wikipedia.* [Online] http://en.wikipedia.org/wiki/Diffie%E2%80%93Hellman_key_exchange.

[12] GOLLMANN, DIETER**.** *Computer Security Second Edition.* West Sussex : John Wiley & Sons, Ltd., 2006.

[13] Kryptografie. *Kryptografie.* [Online] http://www.kryptografie.wz.cz/uk.htm.

[14] PINKAVA, JAROSLAV**.** Hashovací funkce v roce 2004. *Crypto-world.* [Online] 10 2004. http://crypto-world.info/pinkava/clanky/hash_2004.pdf.

[15] KOSTOLÁNYOVÁ, KATEŘINA**.** *Algoritmizace a řešení problémů.* Ostrava : Ostravská univerzita v Ostravě - Pedagogická fakulta, 2002. str. 108.

[16] JÁGR, PETR**.** *Hledání nejkratších cest grafem.* Brno : VUT Brno, 2007.

[17] KAČMÁŘ, DALIBOR**.** Zpracování informací v reálném čase v prostředí. *Fakulta strojní.* [Online] 1999. http://www.fs.vsb.cz/akce/2000/asr2000/Sbornik/papers/kacmar.pdf.

[18] ČERNOHORSKÝ, JINDŘICH A SROVNAL, VILÉM**.** Základní koncepce. *AT&P journal*. 2005, 2005.

[19] Real Time Module for Windows Vista, XP/2000. *The Code Project.* [Online] http://www.codeproject.com/KB/system/RealTimeModule.aspx?msg=2820096.

[20] PETERKA, JIŘÍ**.** Cryptography. *eArchiv*. [Online] 22. 7 1997. [cit. 1999-07-22]. http://www.earchiv.cz/axxxk160/a707k162.php3.

Přílohy

Krátký text (počet znaků s mezerami: 68)

Mikroelektronika je jedním z nejperspektivnějších technických oborů.

Střední text (počet znaků s mezerami: 529)

Mikroelektronika je jedním z nejperspektivnějších technických oborů. Zabývá se návrhem a výrobou polovodičových součástek na bázi submikronových technologií od nejjednodušších diod, přes tranzistor až po integrované obvody (IO) s velmi vysokou hustotou integrace reprezentované paměťmi DRAM nebo 64-bitovými mikroprocesory. Tyto integrované obvody a součástky naleznete v každém dnešním elektronickém výrobku. Bez oboru mikroelektronika by nebylo dnešních počítačů, televizorů, mobilů, prostě žádné moderní elektronické zařízení.

Dlouhý text (počet znaků s mezerami: 1383)

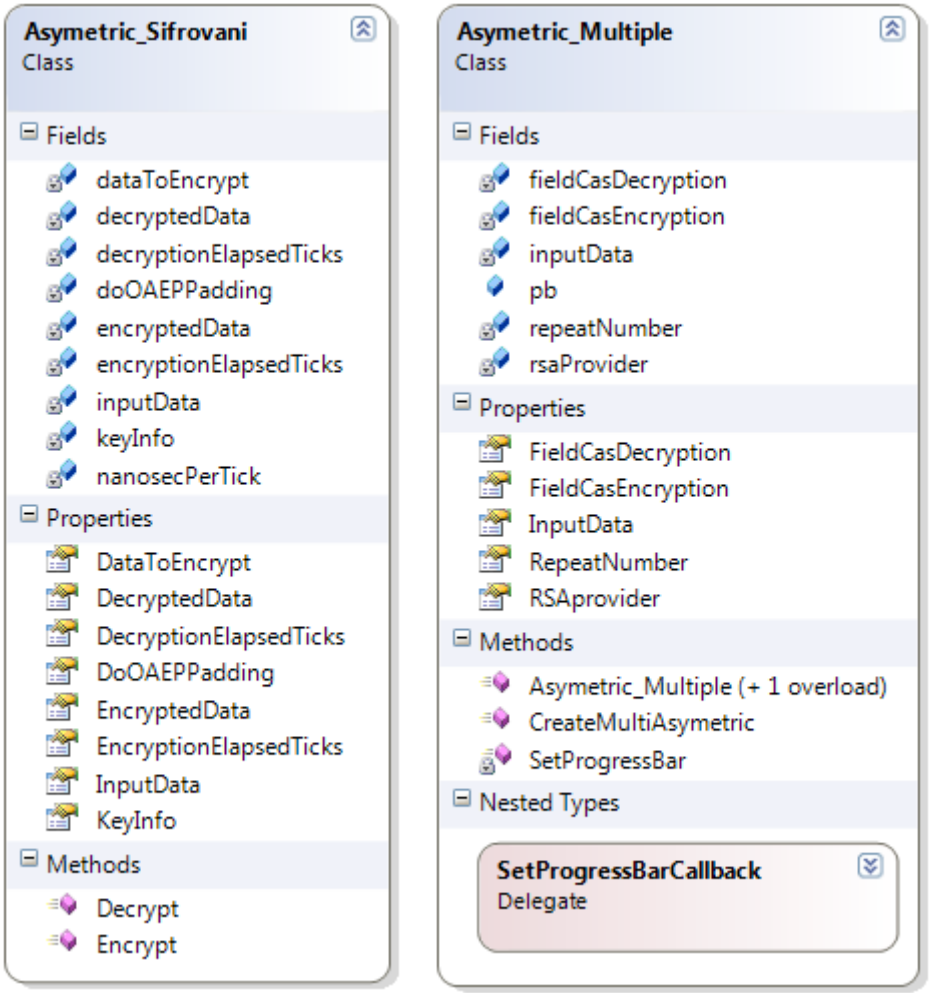
Mikroelektronika je jedním z nejperspektivnějších technických oborů. Zabývá se návrhem a výrobou polovodičových součástek na bázi submikronových technologií od nejjednodušších diod, přes tranzistor až po integrované obvody (IO) s velmi vysokou hustotou integrace reprezentované paměťmi DRAM nebo 64-bitovými mikroprocesory. Tyto integrované obvody a součástky naleznete v každém dnešním elektronickém výrobku. Bez oboru mikroelektronika by nebylo dnešních počítačů, televizorů, mobilů, prostě žádné moderní elektronické zařízení.

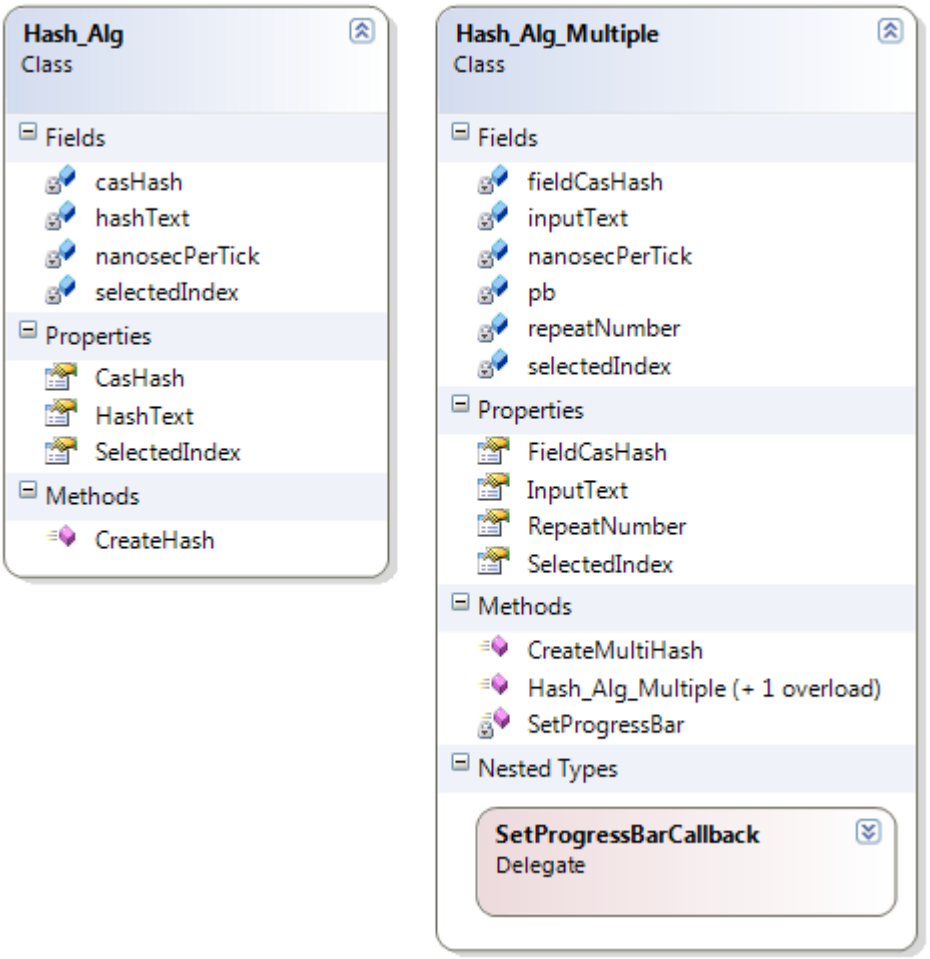
A přesně tato problematika, tedy návrh, vývoj a testování těchto obvodů nás zajímají. Jak je optimalně navrhnout dle požadavků zákazníka, jak je vyrobit, a také jak je použít. Je to v podstatě velmi široký obor, ve kterém se uplatní obvodáři, technologové, fyzici a chemikové. Samozřejmě, že jeden člověk nemůže obsáhnout všechny tyto profese, proto se každý specializuje podle vlastních schopností a uvážení.

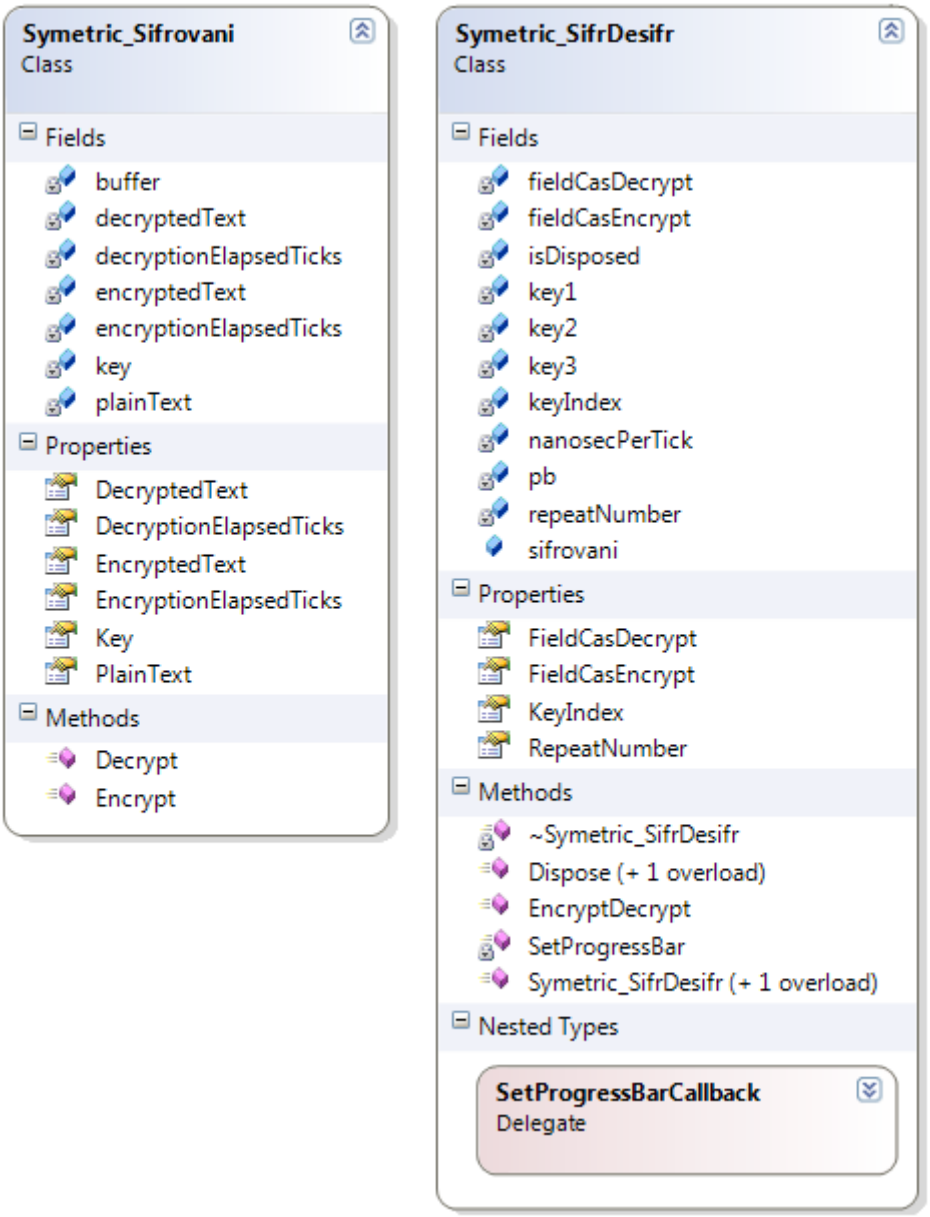
Mikroelektronické technologie se užívají při výrobě celé řady součástek. Předně jsou to polovodičové součástky jako tranzistory, tyristory atd., monolotické integrované obvody, dále hybridní integrované obvody, které umožňují použít více různých technologických postupů na jeden substrát (základní desku), také optoelektronické součástky. Používají se též při výrobě CD disků, hologramů, při nanášení speciálních povrchů na určité materiály atd.

Výpis tříd

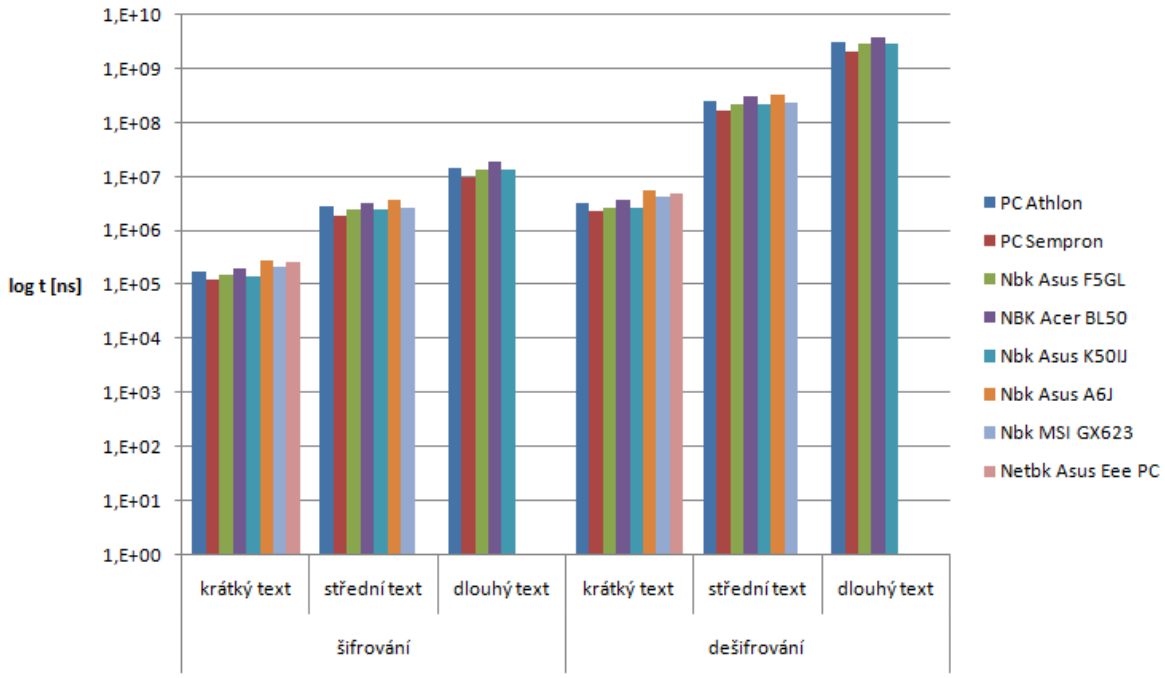
- Asymetric_Sifrovani, Asymetric_Multiple,
- Hash_Alg, Hash_Alg_Multiple,
- Symetric_Sifrovani, Symetric_SifrDesifr



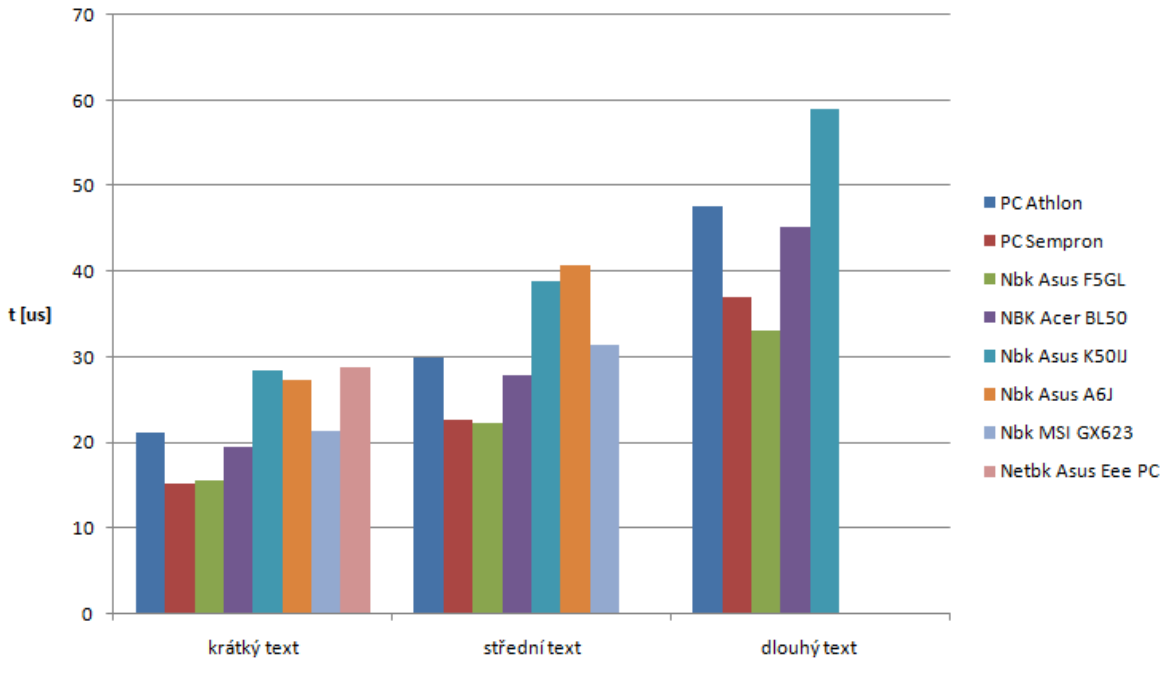

Hash_Alg
Class
Fields
casHash
hashText
nanosecPerTick
selectedIndex
Properties
CasHash
HashText
SelectedIndex
Methods
CreateHash
Hash_Alg_Multiple
Class
Fields
fieldCasHash
inputText
nanosecPerTick
pb
repeatNumber
selectedIndex
Properties
FieldCasHash
InputText
RepeatNumber
SelectedIndex
Methods
CreateMultiHash
Hash_Alg_Multiple (+ 1 overload)
SetProgressBar
Nested Types
SetProgressBarCallback
Delegate

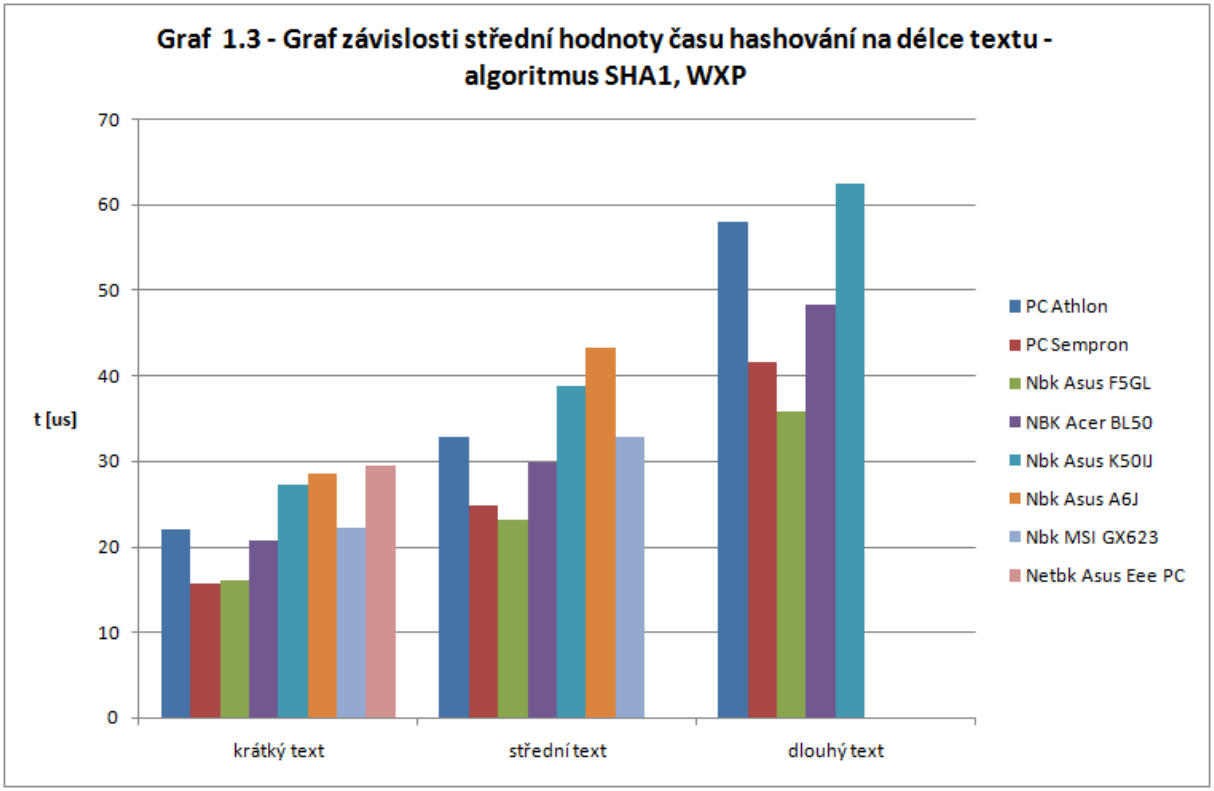

Symetric_Sifrovani
Class
Fields
buffer
decryptedText
decryptionElapsedTicks
encryptedText
encryptionElapsedTicks
key
plainText
Properties
DecryptedText
DecryptionElapsedTicks
EncryptedText
EncryptionElapsedTicks
Key
PlainText
Methods
Decrypt
Encrypt
Symetric_SifrDesifr
Class
Fields
fieldCasDecrypt
fieldCasEncrypt
isDisposed
key1
key2
key3
keyIndex
nanosecPerTick
pb
repeatNumber
sifrovani
Properties
FieldCasDecrypt
FieldCasEncrypt
KeyIndex
RepeatNumber
Methods
~Symetric_SifrDesifr
Dispose (+ 1 overload)
EncryptDecrypt
SetProgressBar
Symetric_SifrDesifr (+ 1 overload)
Nested Types
SetProgressBarCallback
Delegate

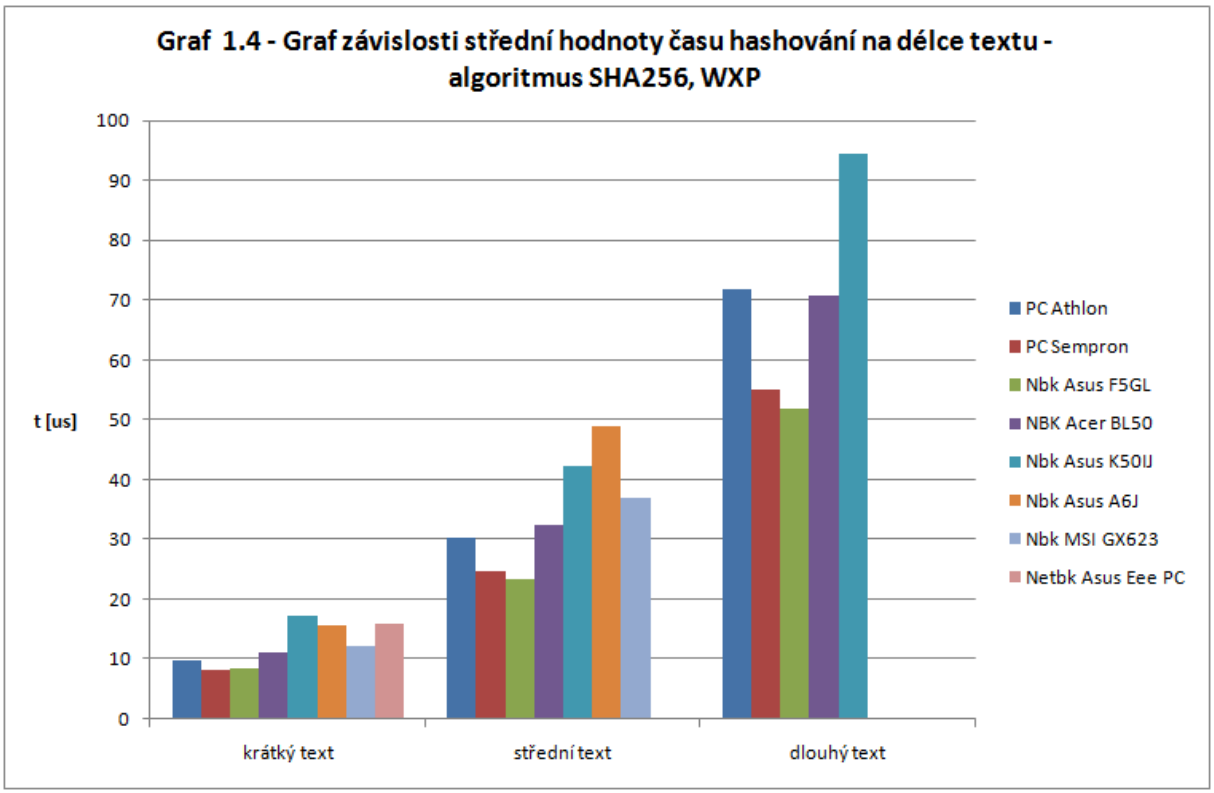
Graf 1.1 - Graf závislosti střední hodnoty času šifrování a dešifrování na proměnné délce textu - algoritmus RSA, WXP

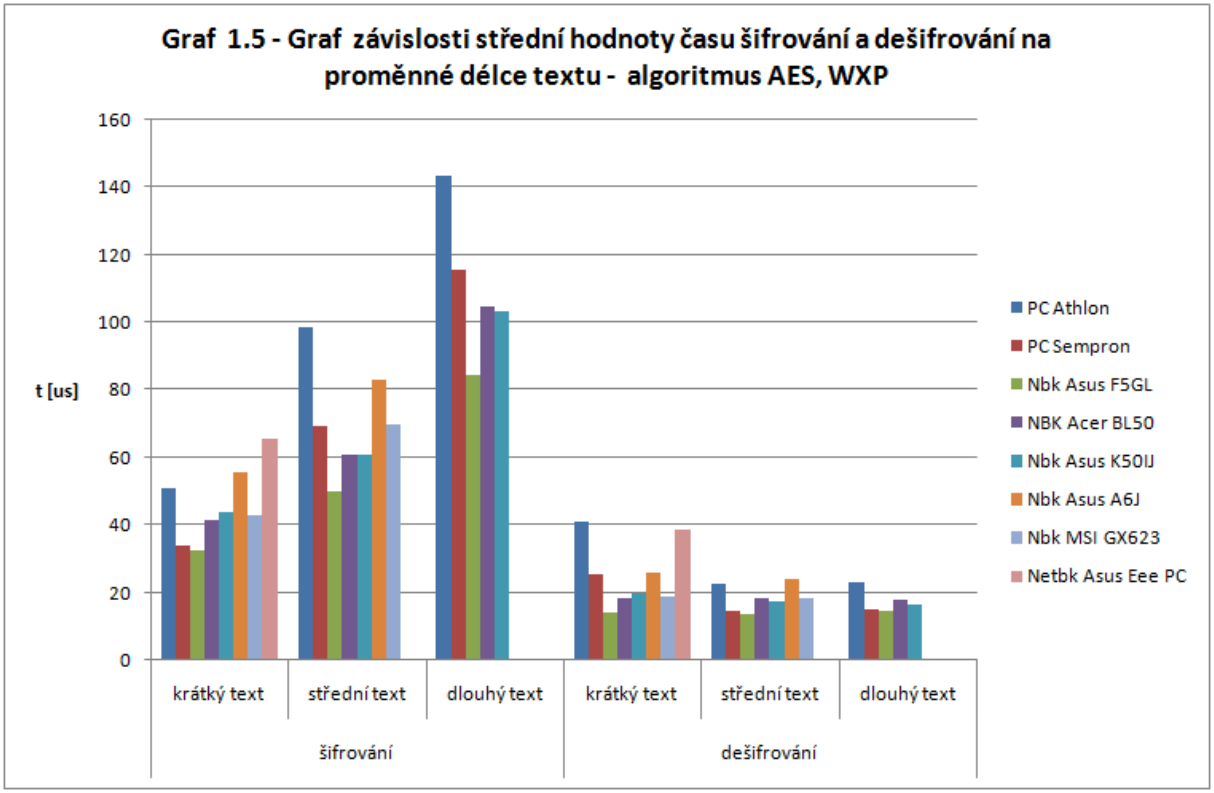


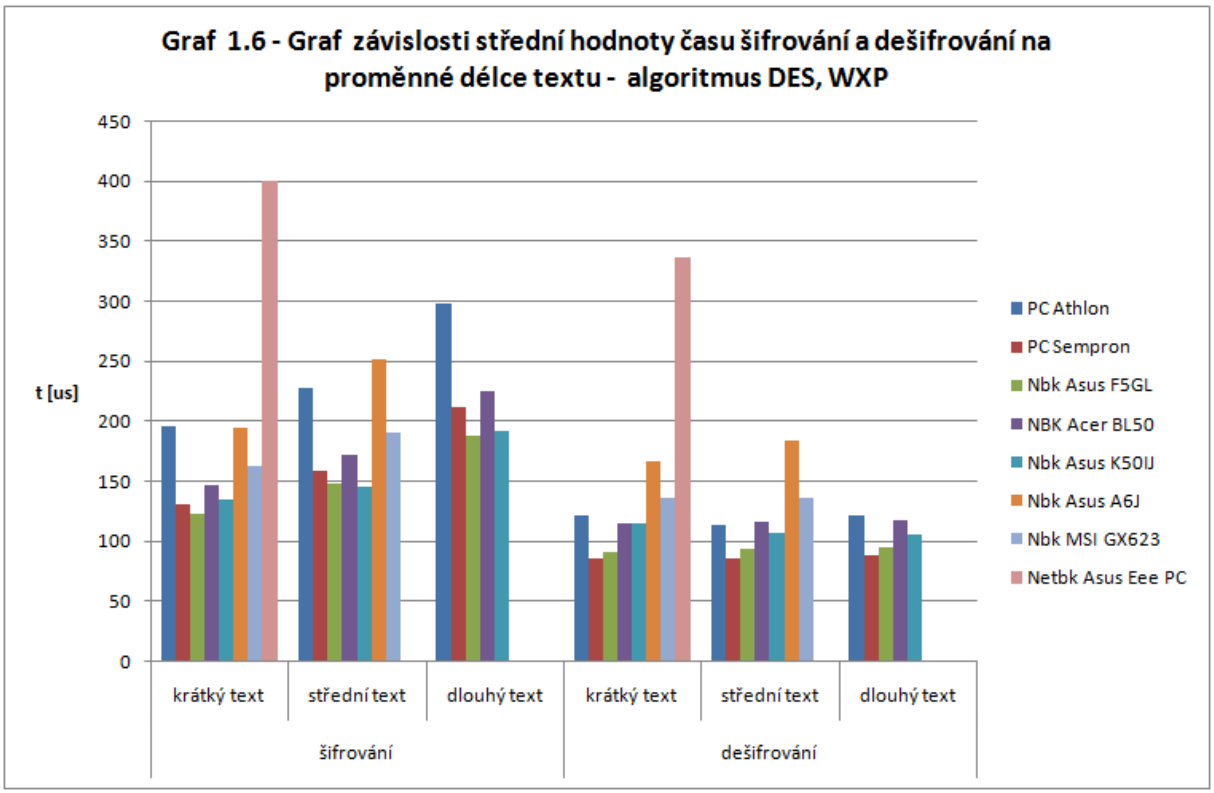
Graf 1.2 - Graf závislosti střední hodnoty času hashování na délce textu - algoritmus MD5, WXP

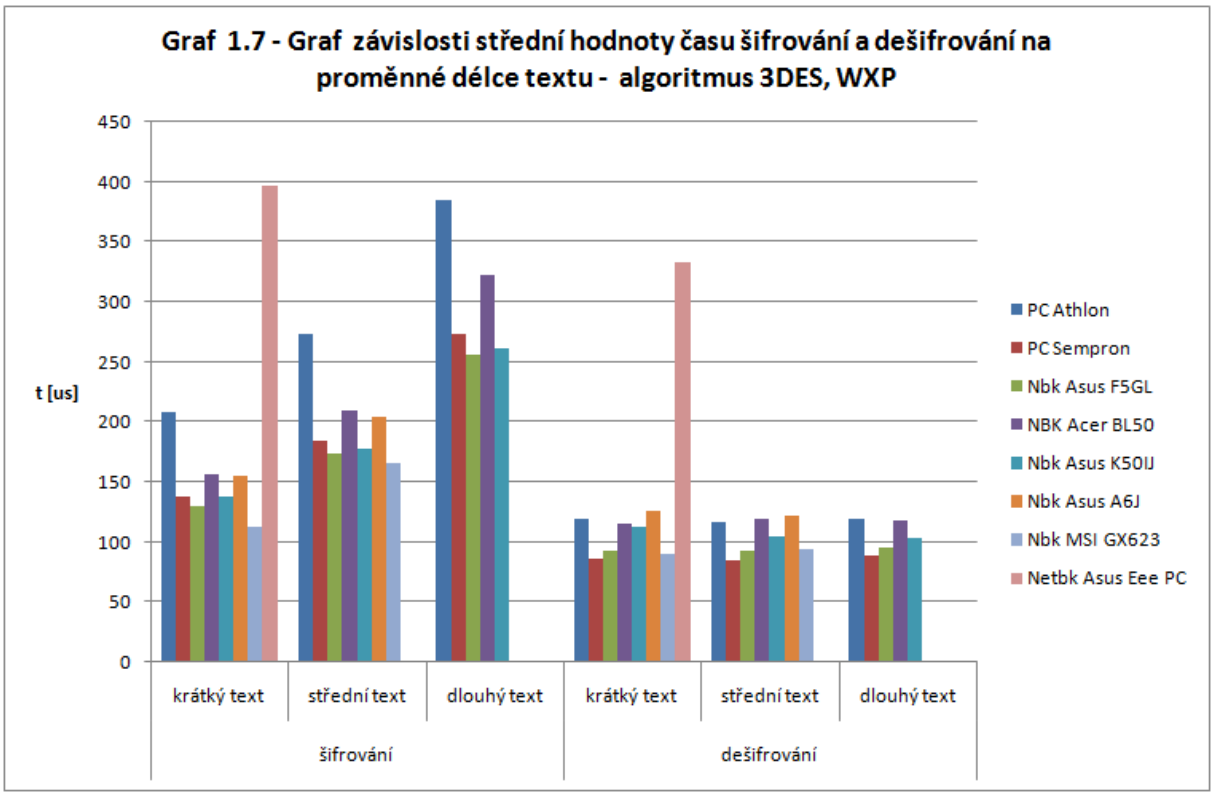


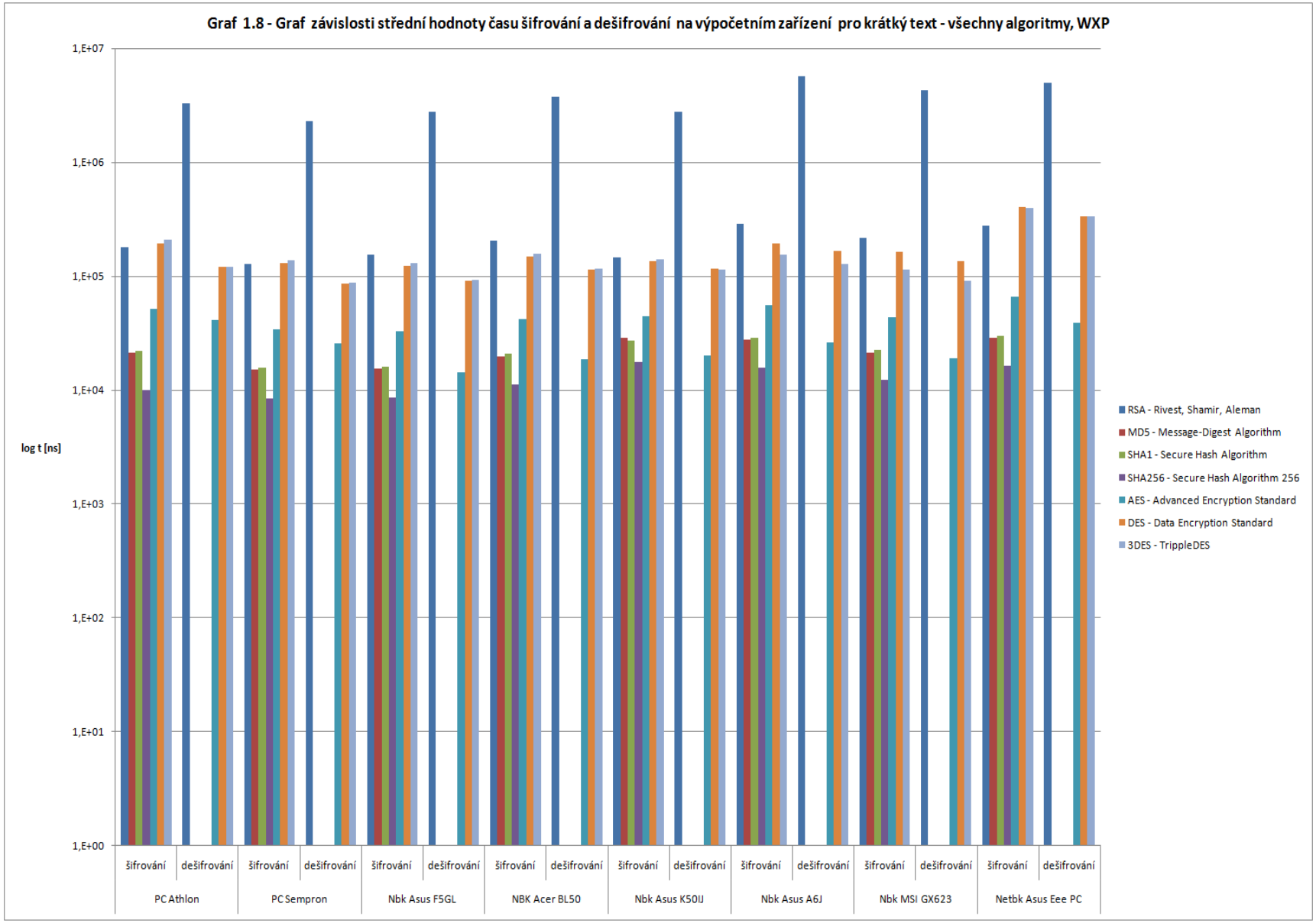

Graf 1.3 - Graf závislosti střední hodnoty času hashování na délce textu - algoritmus SHA1, WXP
t [us]
70
60
50
40
30
20
10
0
krátký text
střední text
dlouhý text
PC Athlon
PC Sempron
Nbk Asus F5GL
NBK Acer BL50
Nbk Asus K50IJ
Nbk Asus A6J
Nbk MSI GX623
Netbk Asus Eee PC

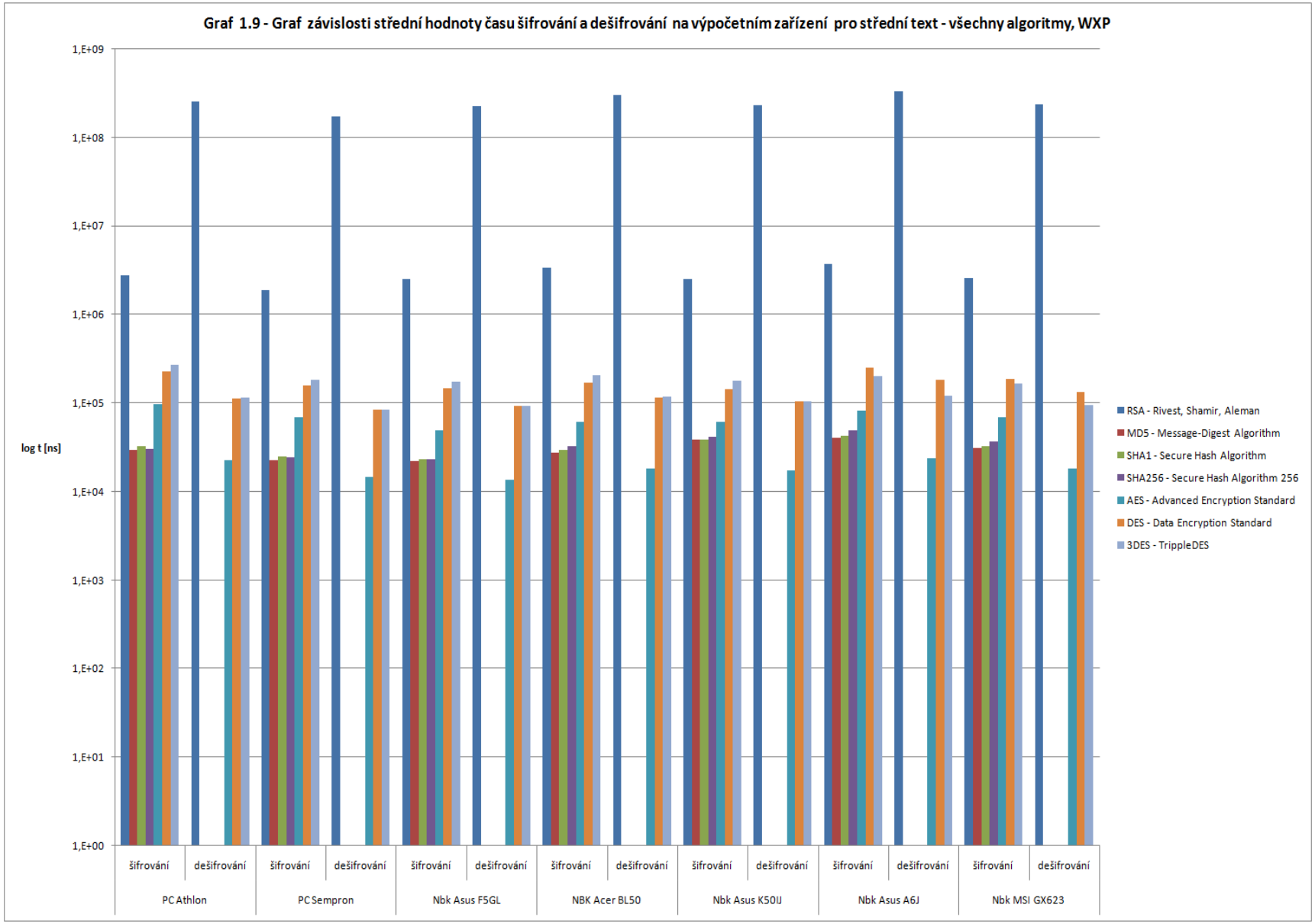

Graf 1.4 - Graf závislosti střední hodnoty času hashování na délce textu - algoritmus SHA256, WXP
t [us]
100
90
80
70
60
50
40
30
20
10
0
krátký text
střední text
dlouhý text
PC Athlon
PC Sempron
Nbk Asus F5GL
NBK Acer BL50
Nbk Asus K50IJ
Nbk Asus A6J
Nbk MSI GX623
Netbk Asus Eee PC


Graf 1.5 - Graf závislosti střední hodnoty času šifrování a dešifrování na proměnné délce textu - algoritmus AES, WXP
160
140
120
100
80
60
40
20
0
t [us]
krátký text
střední text
dlouhý text
krátký text
střední text
dlouhý text
šifrování
dešifrování
PC Athlon
PC Sempron
Nbk Asus F5GL
NBK Acer BL50
Nbk Asus K50IJ
Nbk Asus A6J
Nbk MSI GX623
Netbk Asus Eee PC

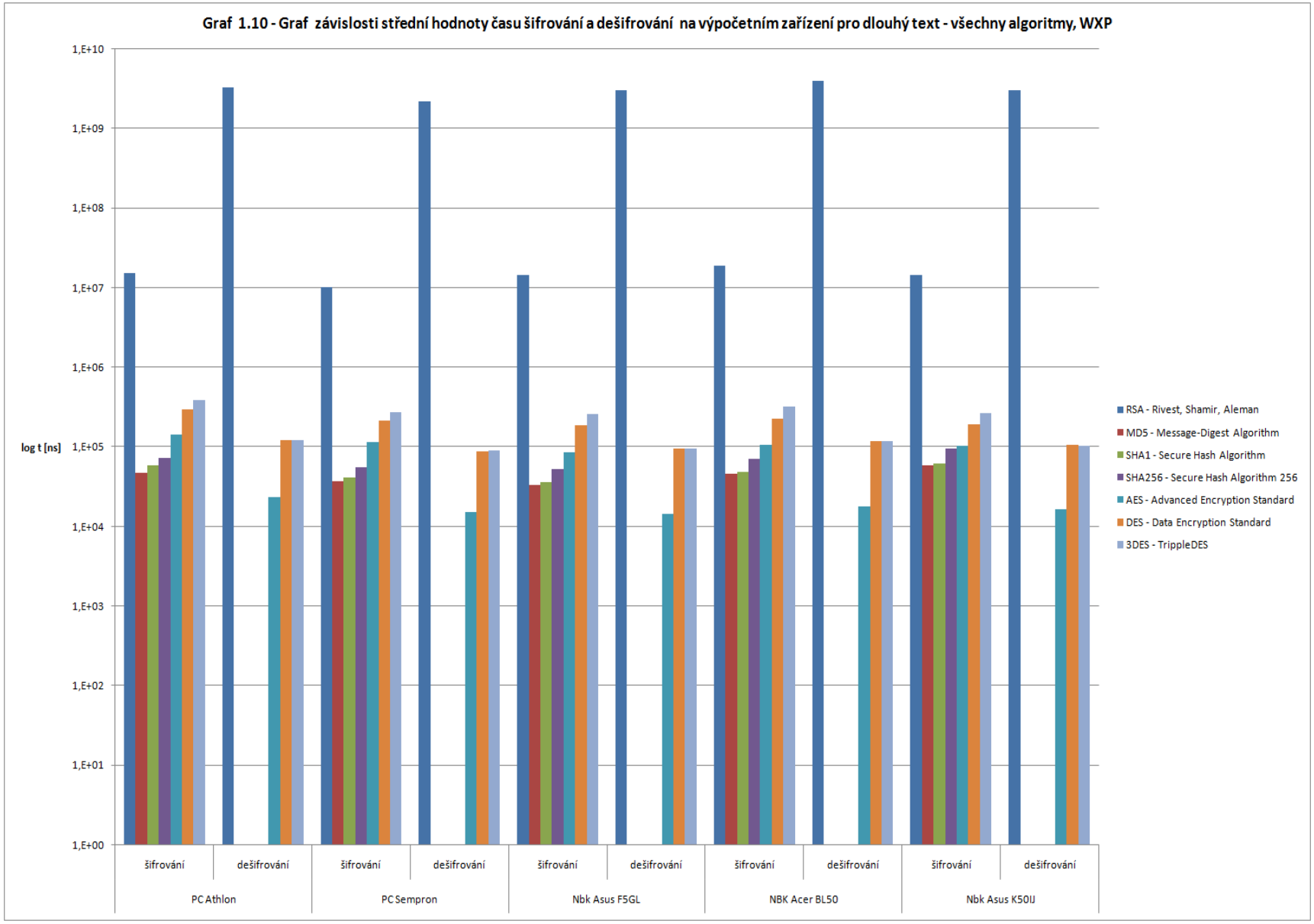

Graf 1.6 - Graf závislosti střední hodnoty času šifrování a dešifrování na proměnné délce textu - algoritmus DES, WXP
450
400
350
300
250
200
150
100
50
0
t [us]
krátký text
střední text
dlouhý text
krátký text
střední text
dlouhý text
šifrování
dešifrování
PC Athlon
PC Sempron
Nbk Asus F5GL
NBK Acer BL50
Nbk Asus K50IJ
Nbk Asus A6J
Nbk MSI GX623
Netbk Asus Eee PC


Graf 1.7 - Graf závislosti střední hodnoty času šifrování a dešifrování na proměnné délce textu - algoritmus 3DES, WXP
t [us]
450
400
350
300
250
200
150
100
50
0
krátký text
střední text
dlouhý text
krátký text
střední text
dlouhý text
šifrování
dešifrování
PC Athlon
PC Sempron
Nbk Asus F5GL
NBK Acer BL50
Nbk Asus K50IJ
Nbk Asus A6J
Nbk MSI GX623
Netbk Asus Eee PC

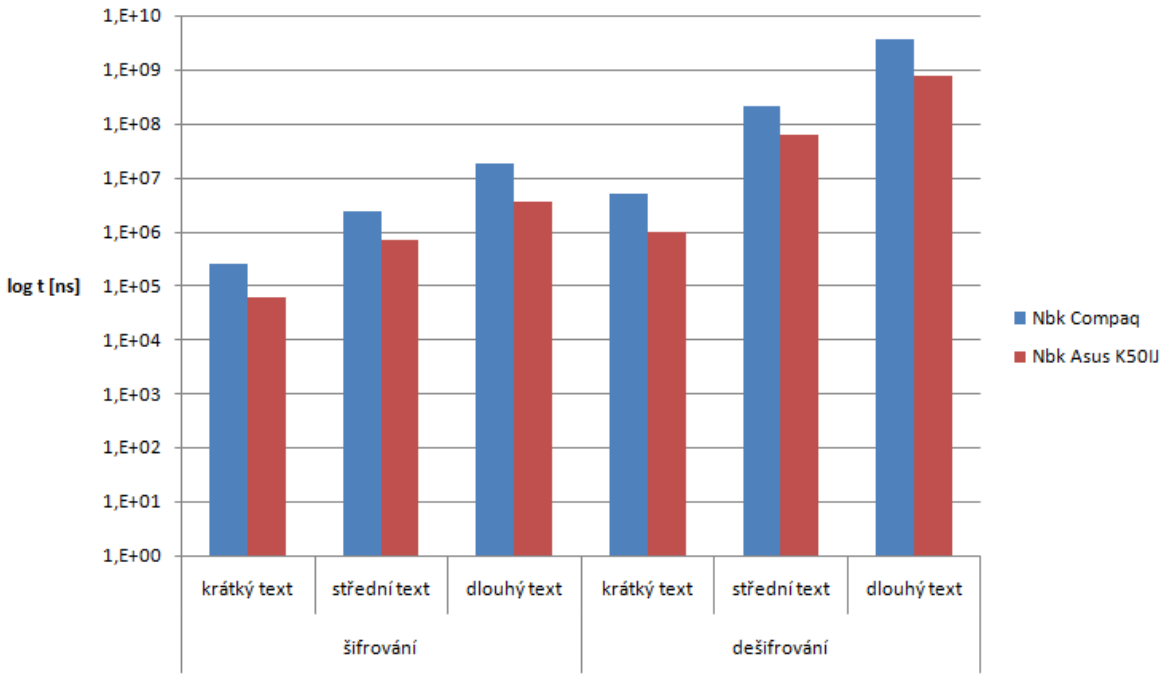

Graf 1.8 - Graf závislosti střední hodnoty času šifrování a dešifrování na výpočetním zařízení pro krátký text - všechny algoritmy, WXP
log t [ns]
1,E+07
1,E+06
1,E+05
1,E+04
1,E+03
1,E+02
1,E+01
1,E+00
šifrování
dešifrování
PC Athlon
PC Sempron
Nbk Asus F5GL
NBK Acer BL50
Nbk Asus K50IJ
Nbk Asus A6J
Nbk MSI GX623
Netbk Asus Eee PC
RSA - Rivest, Shamir, Aleman
MD5 - Message-Digest Algorithm
SHA1 - Secure Hash Algorithm
SHA256 - Secure Hash Algorithm 256
AES - Advanced Encryption Standard
DES - Data Encryption Standard
3DES - TrippleDES


Graf 1.9 - Graf závislosti střední hodnoty času šifrování a dešifrování na výpočetním zařízení pro střední text - všechny algoritmy, WXP
log t [ns]
1,E+09
1,E+08
1,E+07
1,E+06
1,E+05
1,E+04
1,E+03
1,E+02
1,E+01
1,E+00
šifrování
dešifrování
PC Athlon
PC Sempron
Nbk Asus F5GL
NBK Acer BL50
Nbk Asus K50IJ
Nbk Asus A6J
Nbk MSI GX623
RSA - Rivest, Shamir, Aleman
MD5 - Message-Digest Algorithm
SHA1 - Secure Hash Algorithm
SHA256 - Secure Hash Algorithm 256
AES - Advanced Encryption Standard
DES - Data Encryption Standard
3DES - TrippleDES

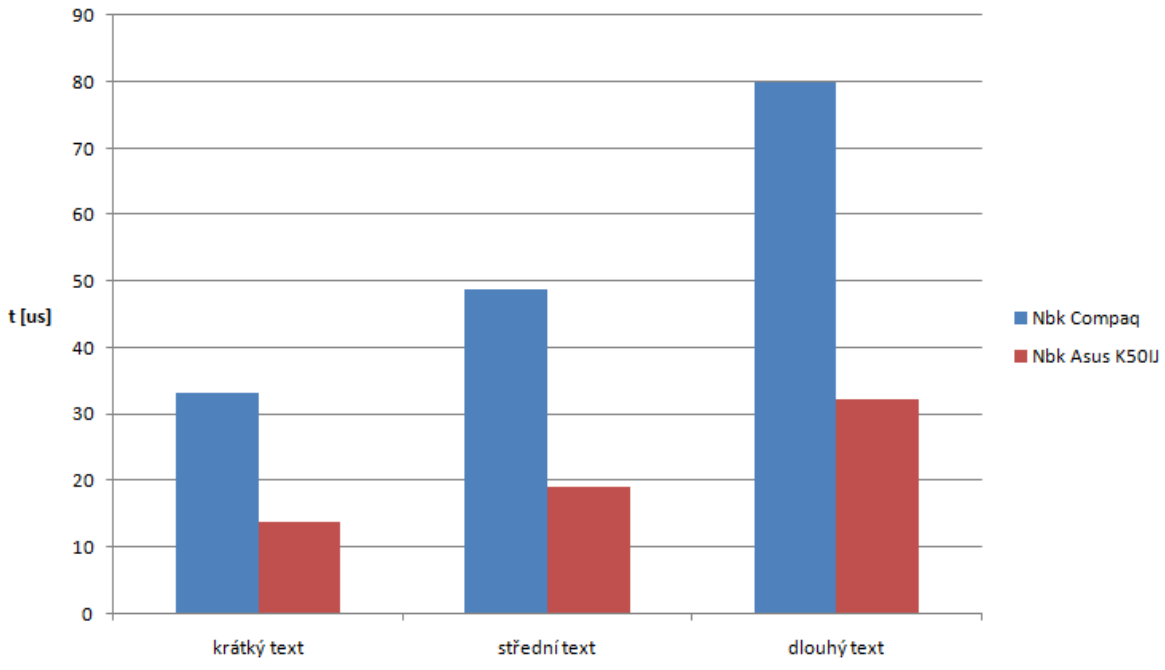
Graf 1.10 - Graf závislosti střední hodnoty času šifrování a dešifrování na výpočetním zařízení pro dlouhý text - všechny algoritmy, WXP

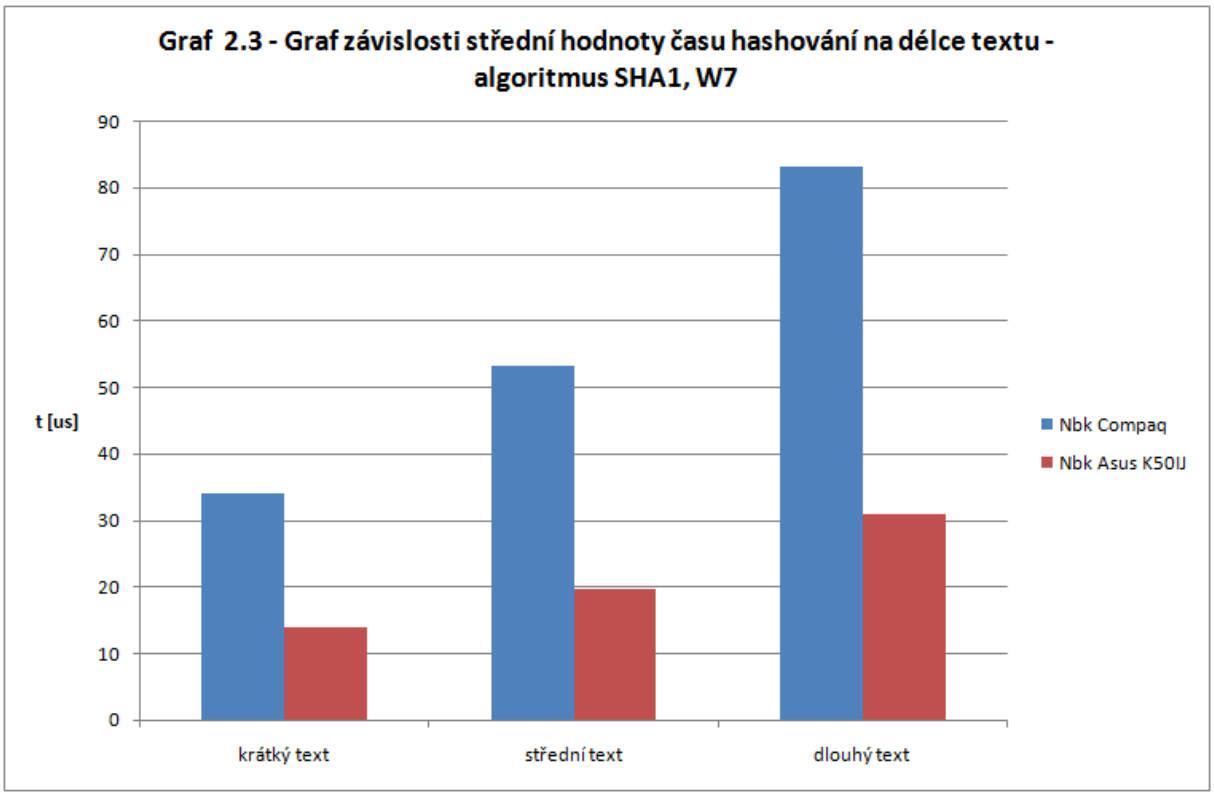


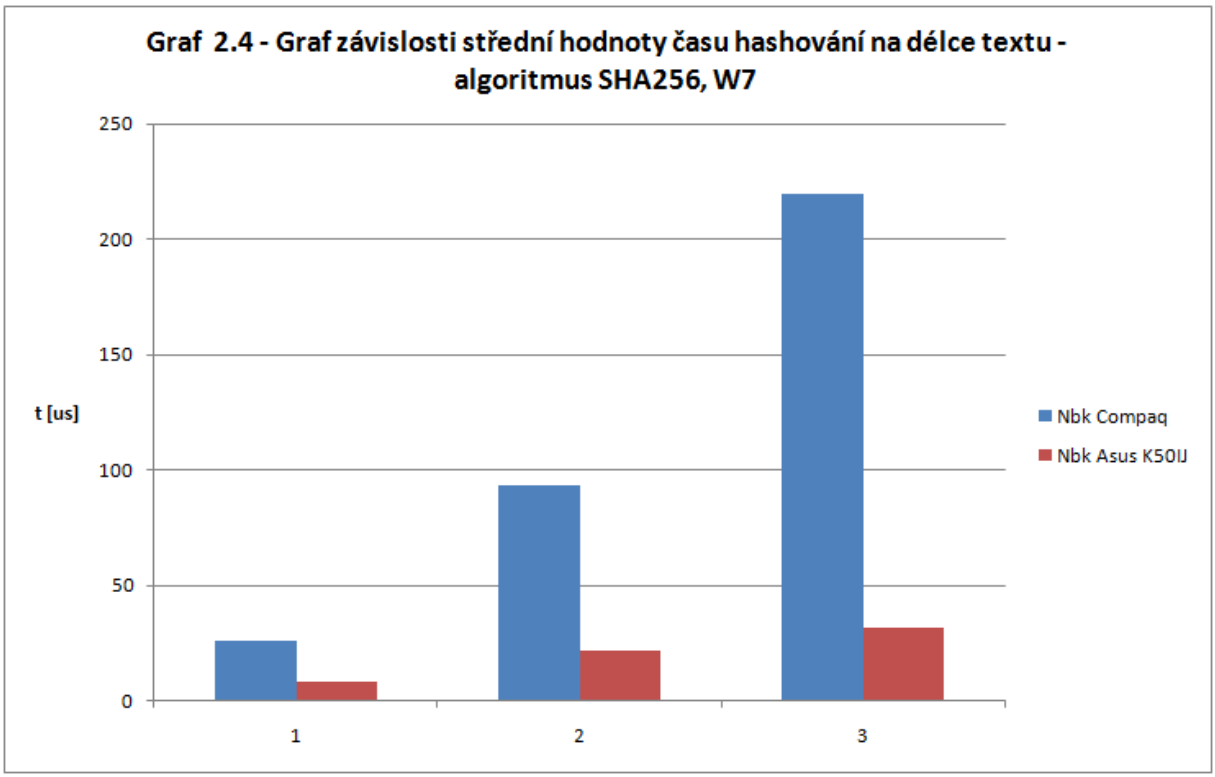
Graf 2.1 - Graf závislosti střední hodnoty času šifrování a dešifrování na proměnné délce textu - algoritmus RSA, W7

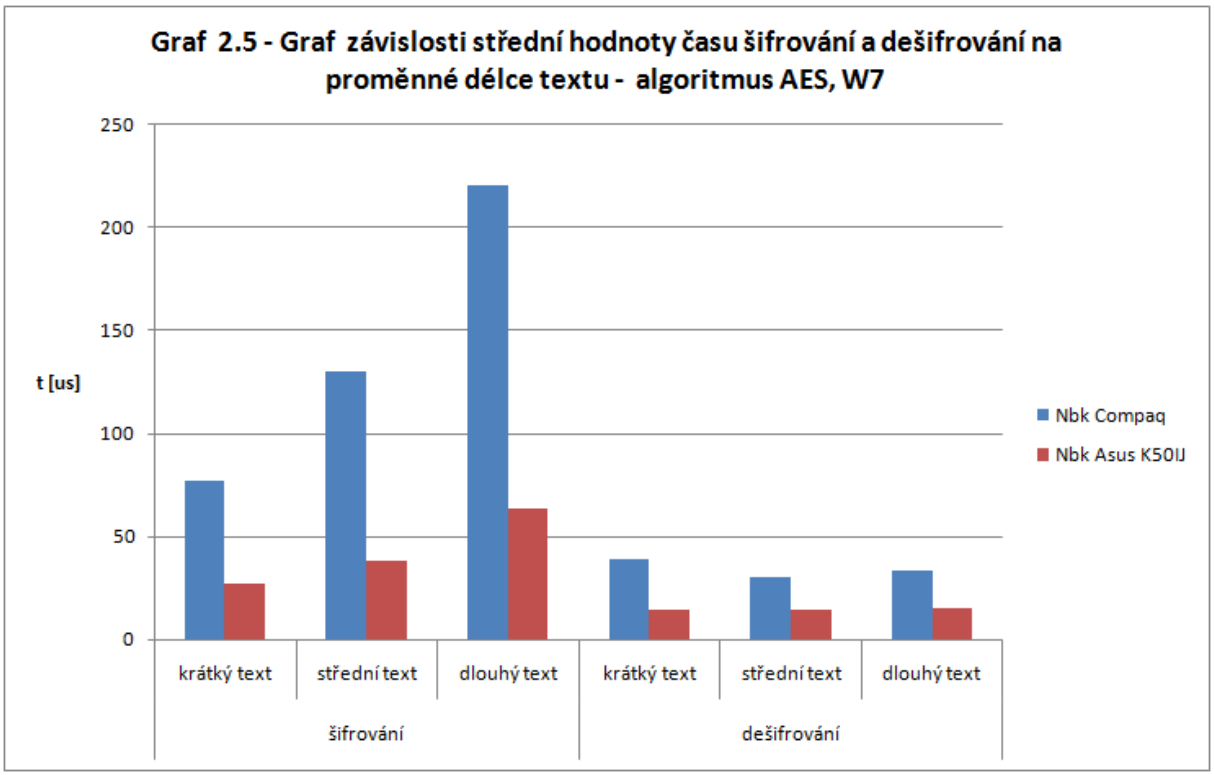


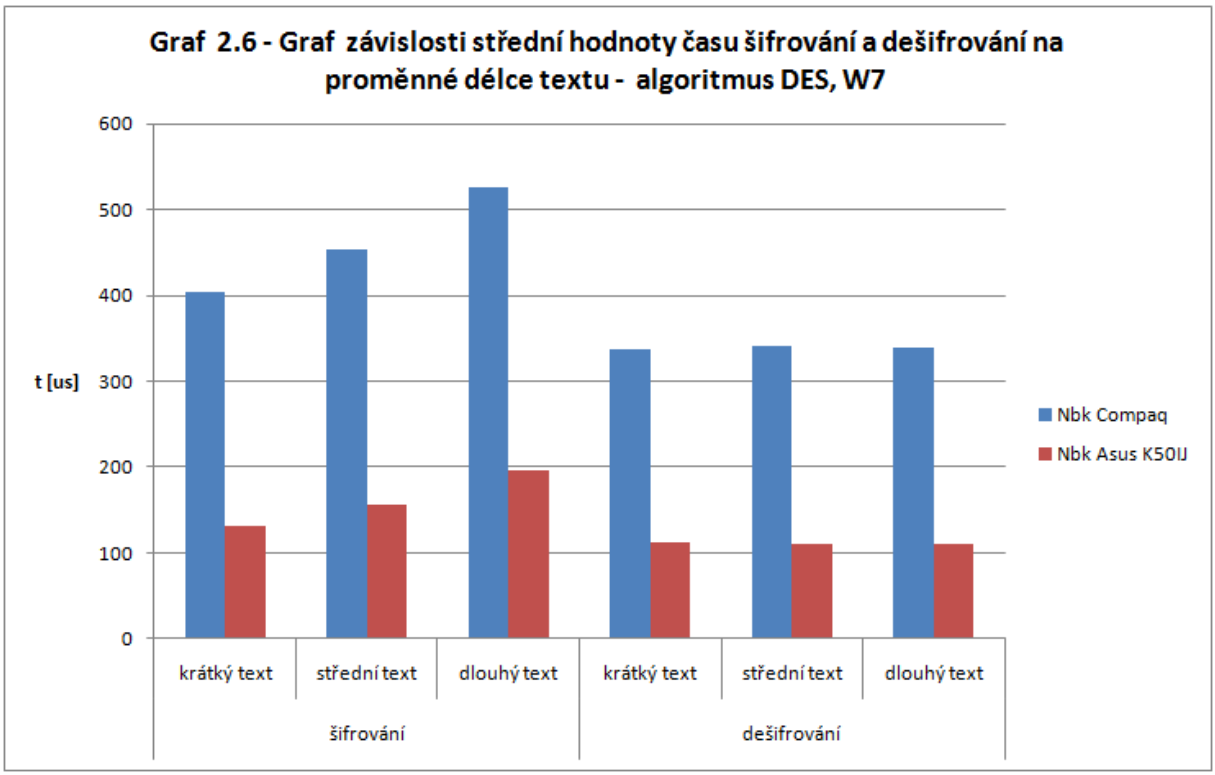
Graf 2.2 - Graf závislosti střední hodnoty času hashování na délce textu - algoritmus MD5, W7

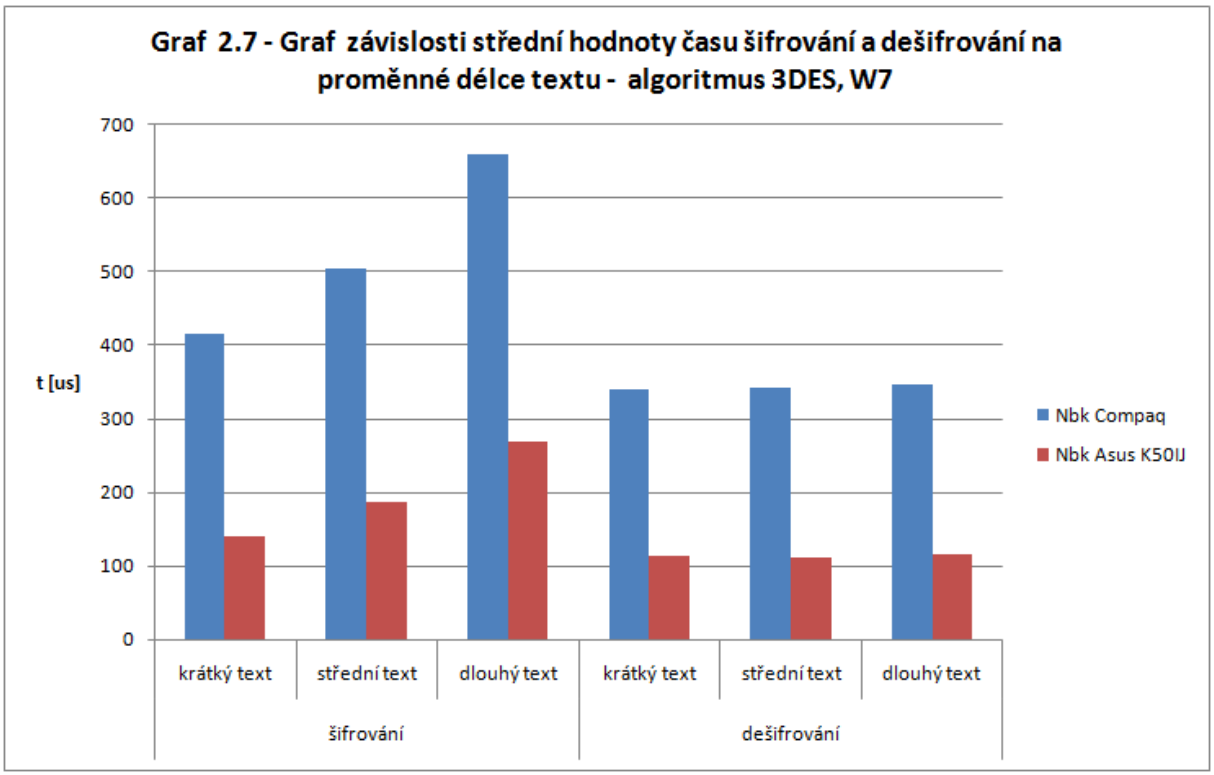


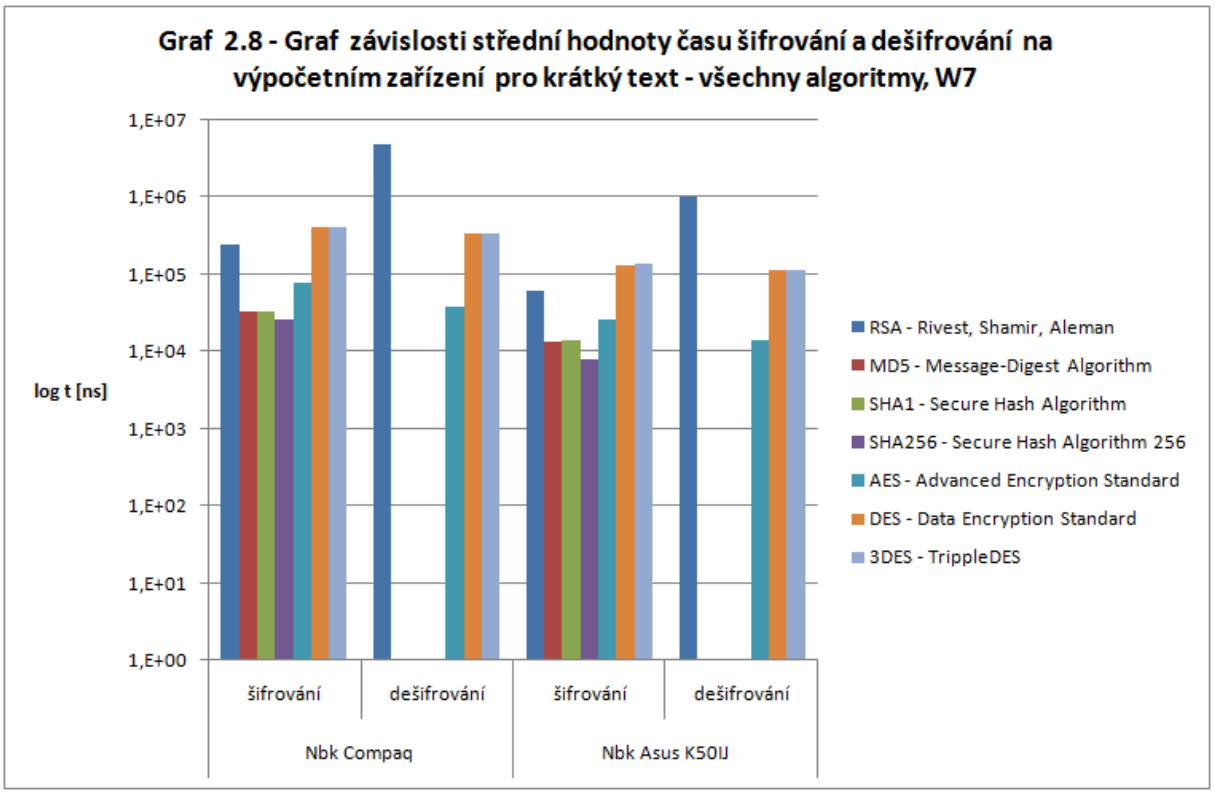

Graf 2.3 - Graf závislosti střední hodnoty času hashování na délce textu - algoritmus SHA1, W7
90
80
70
60
50
40
30
20
10
0
t [us]
krátký text
střední text
dlouhý text
Nbk Compaq
Nbk Asus K50IJ


Graf 2.4 - Graf závislosti střední hodnoty času hashování na délce textu - algoritmus SHA256, W7
250
200
150
100
50
0
t [us]
1
2
3
Nbk Compaq
Nbk Asus K50IJ

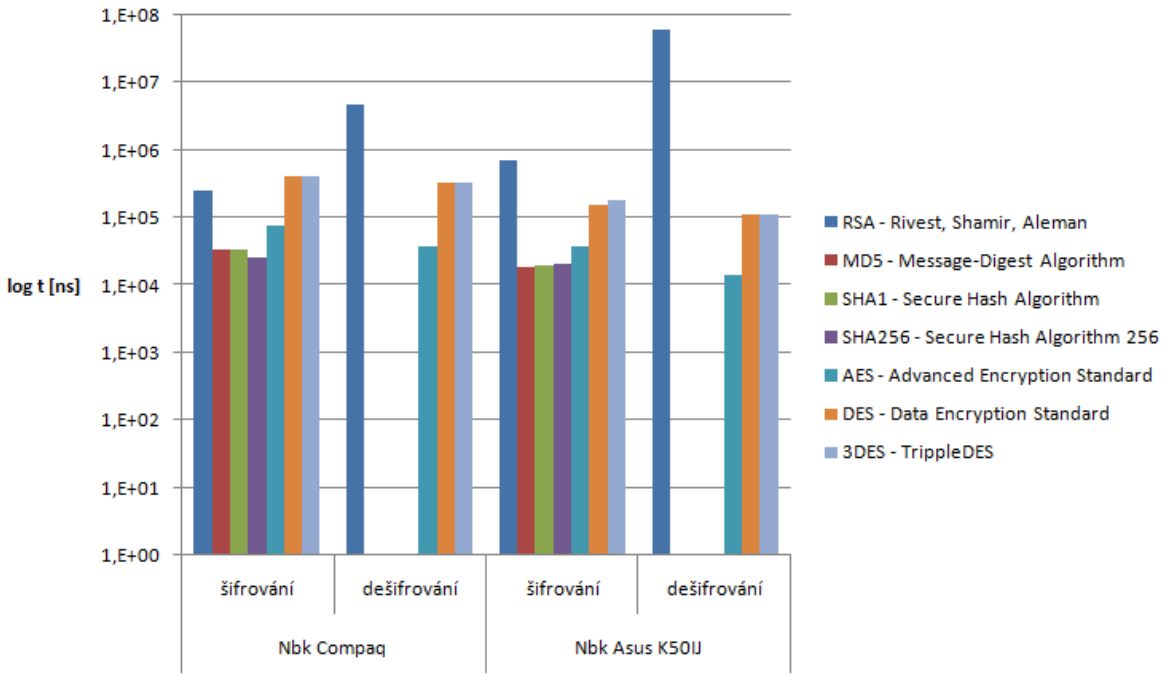

Graf 2.5 - Graf závislosti střední hodnoty času šifrování a dešifrování na proměnné délce textu - algoritmus AES, W7
250
200
150
100
50
0
t [us]
krátký text
střední text
dlouhý text
krátký text
střední text
dlouhý text
šifrování
dešifrování
Nbk Compaq
Nbk Asus K50IJ


Graf 2.6 - Graf závislosti střední hodnoty času šifrování a dešifrování na proměnné délce textu - algoritmus DES, W7
600
500
400
300
200
100
0
t [us]
krátký text
střední text
dlouhý text
krátký text
střední text
dlouhý text
šifrování
dešifrování
Nbk Compaq
Nbk Asus K50IJ

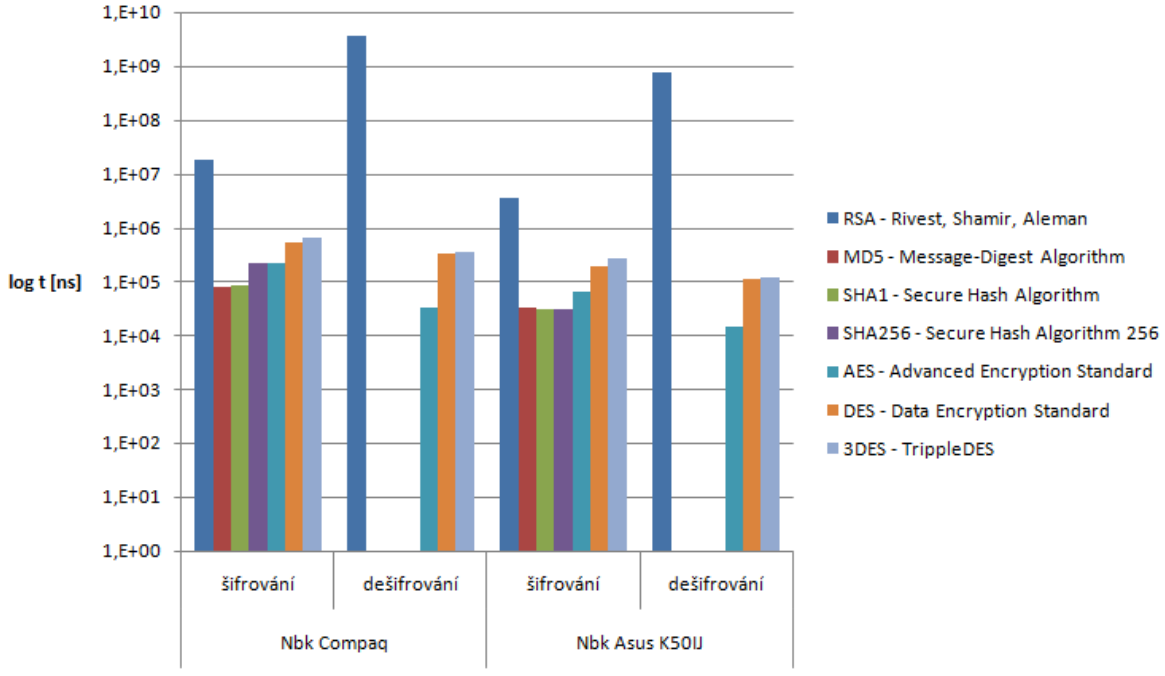

Graf 2.7 - Graf závislosti střední hodnoty času šifrování a dešifrování na proměnné délce textu - algoritmus 3DES, W7
t [us]
700
600
500
400
300
200
100
0
krátký text
střední text
dlouhý text
krátký text
střední text
dlouhý text
šifrování
dešifrování
Nbk Compaq
Nbk Asus K50IJ


Graf 2.8 - Graf závislosti střední hodnoty času šifrování a dešifrování na výpočetním zařízení pro krátký text - všechny algoritmy, W7
log t [ns]
1,E+07
1,E+06
1,E+05
1,E+04
1,E+03
1,E+02
1,E+01
1,E+00
šifrování
dešifrování
šifrování
dešifrování
Nbk Compaq
Nbk Asus K50IJ
RSA - Rivest, Shamir, Aleman
MD5 - Message-Digest Algorithm
SHA1 - Secure Hash Algorithm
SHA256 - Secure Hash Algorithm 256
AES - Advanced Encryption Standard
DES - Data Encryption Standard
3DES - TrippleDES

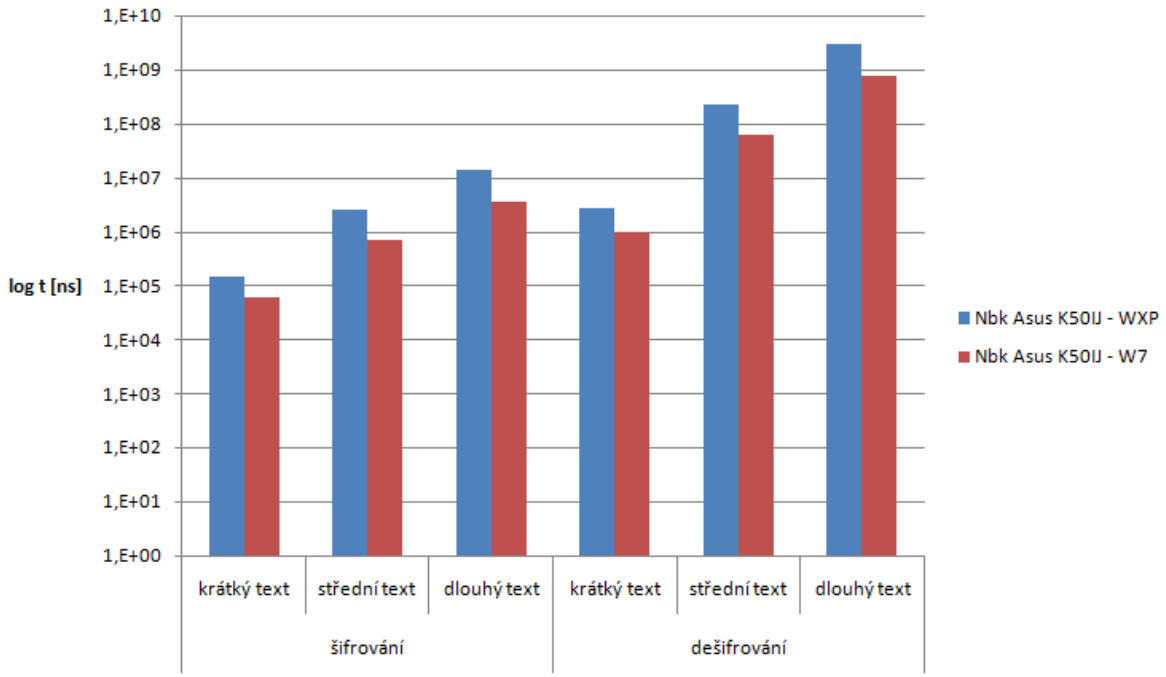
Graf 2.9 - Graf závislosti střední hodnoty času šifrování a dešifrování na výpočetním zařízení pro střední text - všechny algoritmy, W7



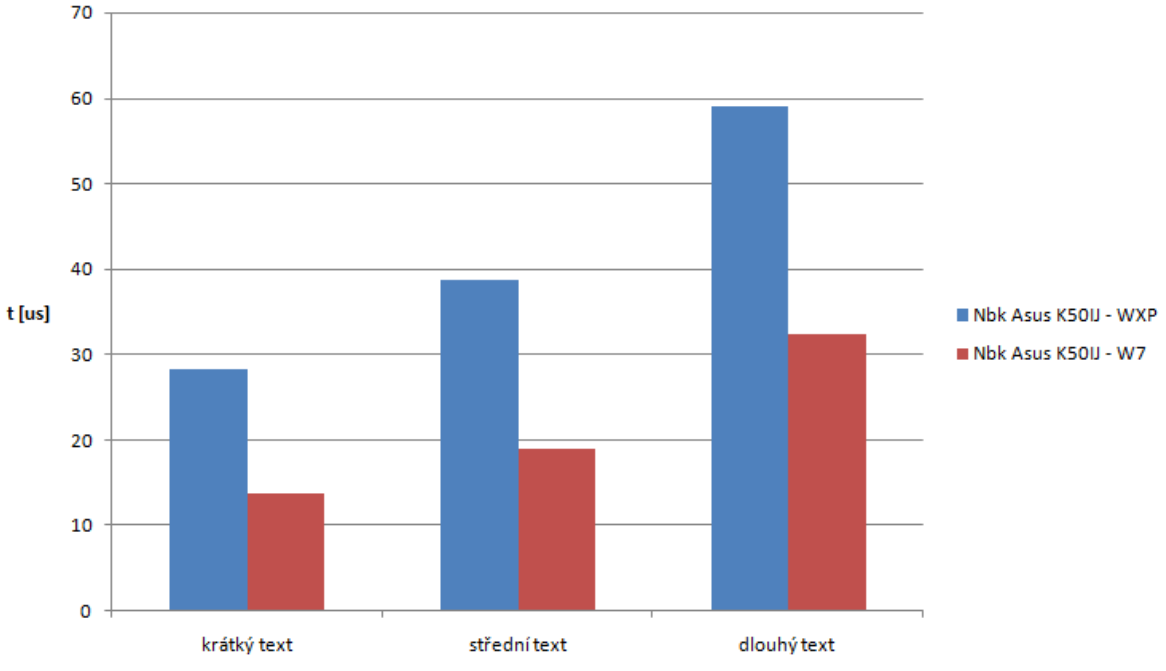
Graf 2.10 - Graf závislosti střední hodnoty času šifrování a dešifrování na výpočetním zařízení pro dlouhý text - všechny algoritmy, W7

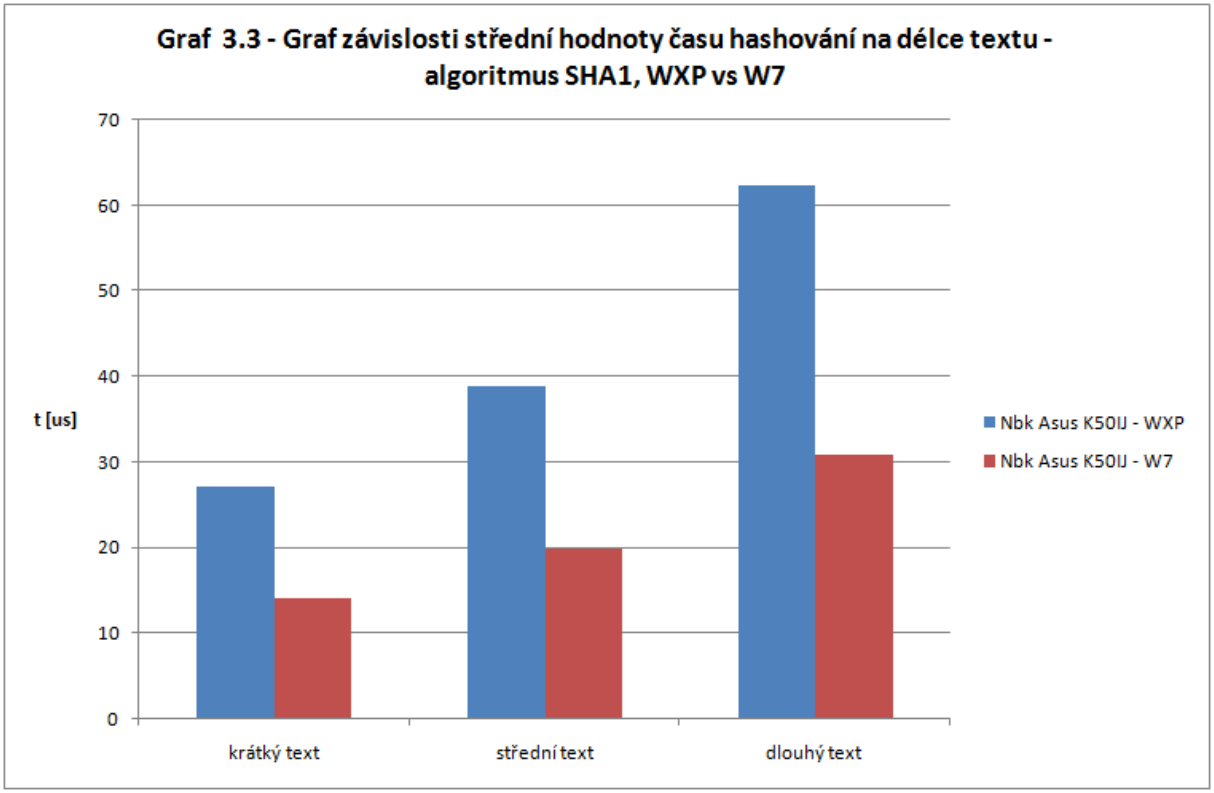


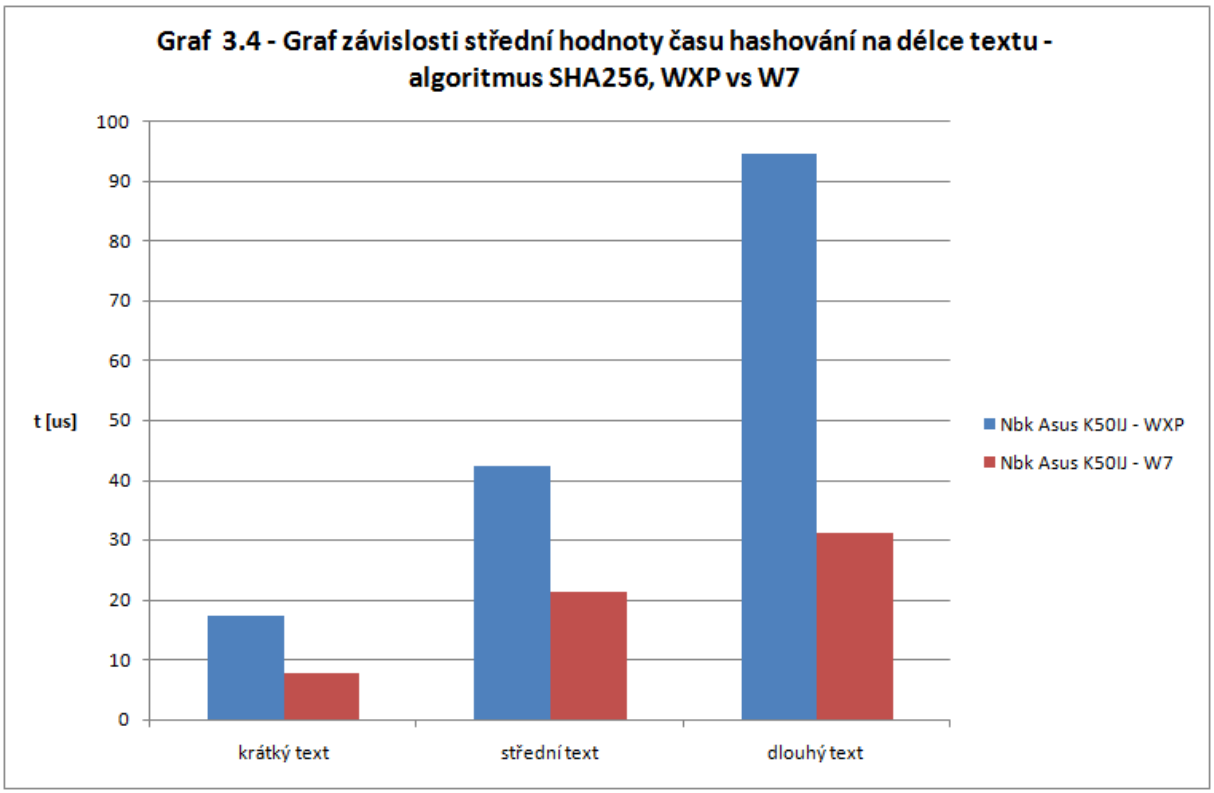
Graf 3.1 - Graf závislosti střední hodnoty času šifrování a dešifrování na proměnné délce textu - algoritmus RSA, WXP vs W7

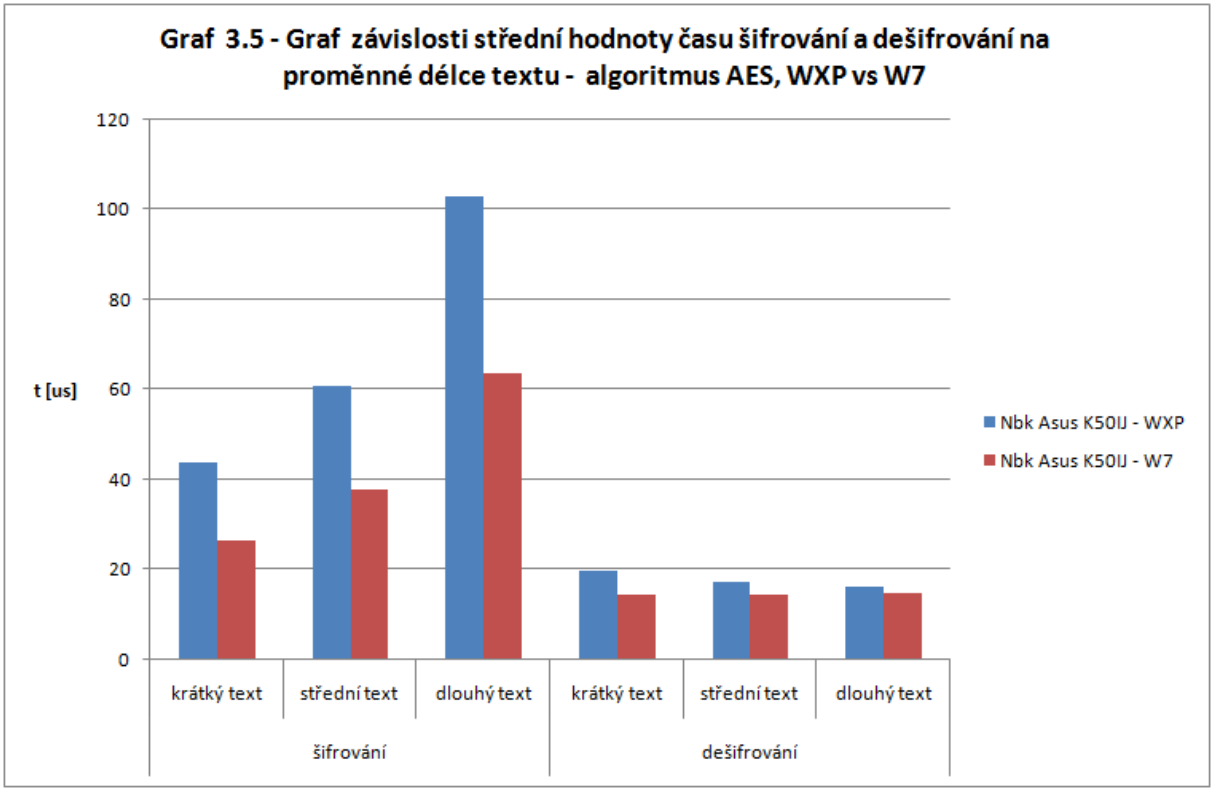


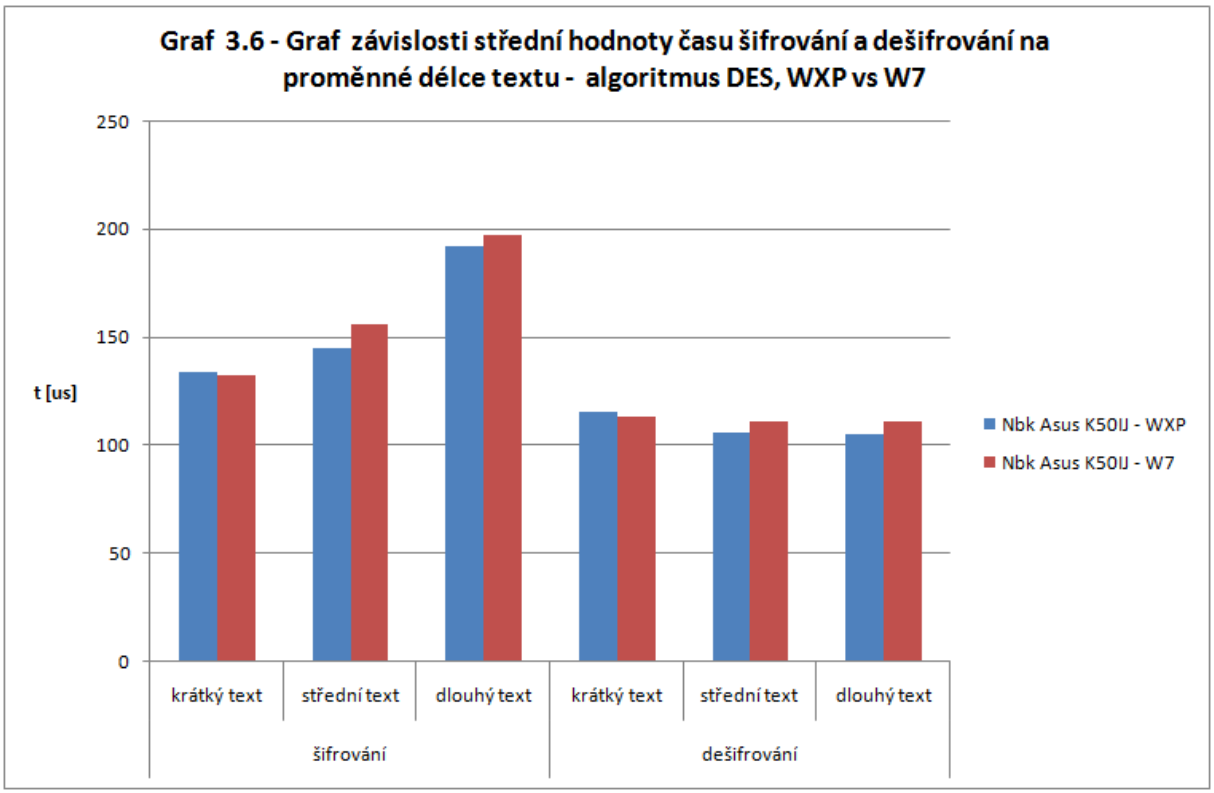
Graf 3.2 - Graf závislosti střední hodnoty času hashování na délce textu - algoritmus MD5, WXP vs W7

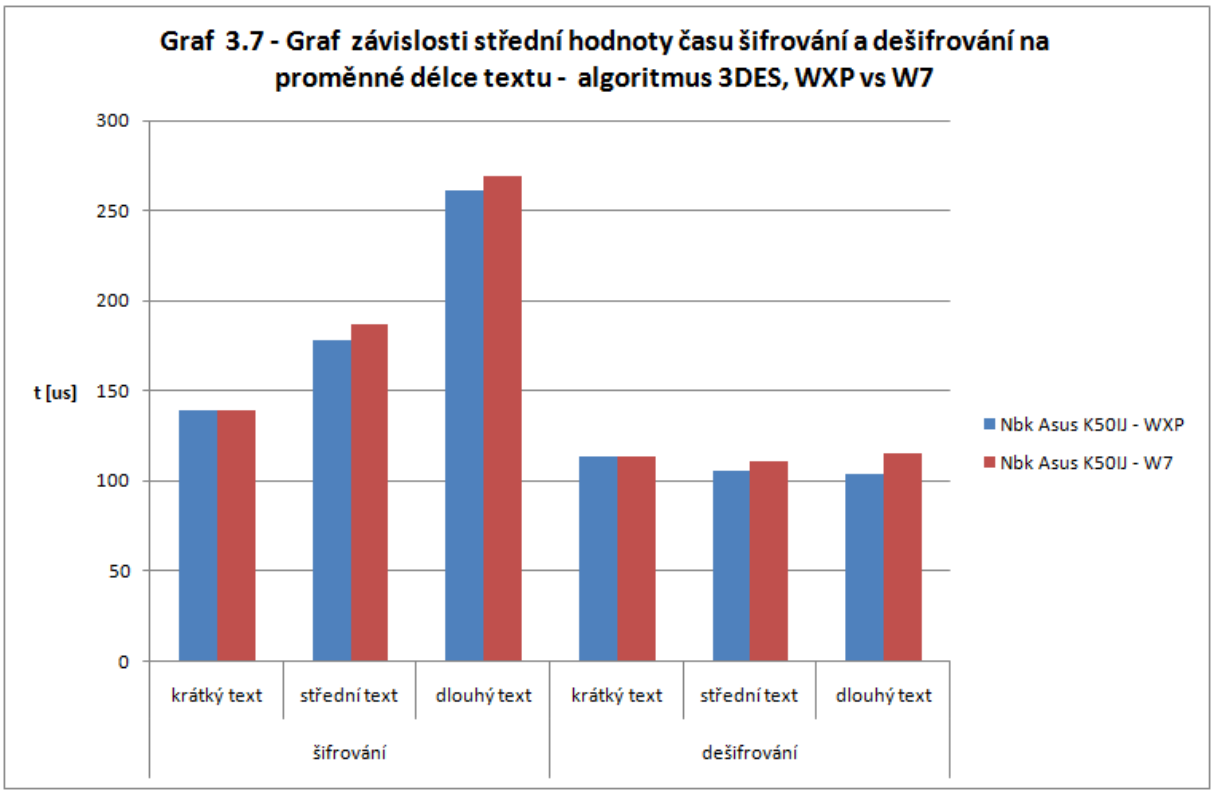



Graf 3.3 - Graf závislosti střední hodnoty času hashování na délce textu - algoritmus SHA1, WXP vs W7
70
60
50
40
30
20
10
0
t [us]
krátký text
střední text
dlouhý text
Nbk Asus K50IJ - WXP
Nbk Asus K50IJ - W7


Graf 3.4 - Graf závislosti střední hodnoty času hashování na délce textu - algoritmus SHA256, WXP vs W7
100
90
80
70
60
50
40
30
20
10
0
t [us]
krátký text
střední text
dlouhý text
Nbk Asus K50IJ - WXP
Nbk Asus K50IJ - W7


Graf 3.5 - Graf závislosti střední hodnoty času šifrování a dešifrování na proměnné délce textu - algoritmus AES, WXP vs W7
120
100
80
60
40
20
0
t [us]
krátký text
střední text
dlouhý text
krátký text
střední text
dlouhý text
šifrování
dešifrování
Nbk Asus K50IJ - WXP
Nbk Asus K50IJ - W7


Graf 3.6 - Graf závislosti střední hodnoty času šifrování a dešifrování na proměnné délce textu - algoritmus DES, WXP vs W7
250
200
150
100
50
0
t [us]
krátký text
střední text
dlouhý text
krátký text
střední text
dlouhý text
šifrování
dešifrování
Nbk Asus K50IJ - WXP
Nbk Asus K50IJ - W7


Graf 3.7 - Graf závislosti střední hodnoty času šifrování a dešifrování na proměnné délce textu - algoritmus 3DES, WXP vs W7
t [us]
300
250
200
150
100
50
0
krátký text
střední text
dlouhý text
krátký text
střední text
dlouhý text
šifrování
dešifrování
Nbk Asus K50IJ - WXP
Nbk Asus K50IJ - W7