Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Dôsledky zmien v kryptografických algoritmoch

Bakalárska práca

2017
Lenka Stúpalová

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Dôsledky zmien v kryptografických algoritmoch

Bakalárska práca

Študijný program:	Informatika
Študijný odbor:	2508 Informatika
Školiace pracovisko:	Katedra informatiky
Školiteľ:	Mgr. Peter Košinár

Bratislava, 2017
Lenka Stúpalová

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Lenka Stúpalová
Študijný program: informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor: informatika
Typ záverečnej práce: bakalárska
Jazyk záverečnej práce: slovenský
Sekundárny jazyk: anglický

Názov: Dôsledky zmien v kryptografických algoritmoch
Effects of changes in cryptographic algorithms

Cieľ: Korektná implementácia kryptografických objektov (šifry, hašovacie funkcie, generátory náhodných čísel, ...) je vo všeobecnosti náročná a náchylná na chyby. Dôsledky takýchto zmien môžu niekedy byť z hľadiska bezpečnosti zanedbateľné, no inokedy môže kvalitu výstupu významne ovplyvniť (napríklad generátor náhodných čísel po čase skonverguje na konštantný výstup). Občas sú dokonca zmeny vnesené úmyselne, s cieľom znemožniť aplikovanie známych informácií pri analýze (napríklad predpočítané rainbow tabuľky pre hašovaciu funkciu). Cieľom bakalárskej práce je vytvoriť aplikáciu, ktorá umožní prehľadným spôsobom demonštrovať vplyv rôznych druhov takýchto zmien (či už úmyselných, alebo omylov) na výsledné správanie vybraných známych kryptografických algoritmov, a tiež nájsť konkrétne príklady, ktoré môžu byť užitočné z metodického hľadiska -- ako ukážky toho, že prečo si treba pri ich implementácii dávať extra pozor na detaily.

Vedúci: Mgr. Peter Košinár
Katedra: FMFI.KI - Katedra informatiky
Vedúci katedry: prof. RNDr. Martin Škoviera, PhD.

Dátum zadania: 30.10.2016

Dátum schválenia: 31.10.2016 doc. RNDr. Daniel Olejár, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

Poďakovanie: Chcela by som poďakovať svojmu školiteľovi, Mgr. Petrovi Košinárovi, za jeho pomoc, usmerňovanie, odborné rady a trpezlivosť. Tento to nemal ľahké, nakoľko nie je zamestancom školy a ako externý školiteľ si musel nájsť čas aj popri svojom časovo náročnom zamestnaní. Ďalej by som chcela poďakovať aj svojmu oponentovi prvej verzie bakalárky, doc. RNDr. Martin Stanek, PhD., za jeho priateľský prístup, záujem o zdokonalenie práce a pomoc pri vykonávaní opráv.

Abstrakt

V tejto bakalárskej práci sa venujeme vplyvom zmien v kryptografických algoritmoch na ich výsledné vlastnosti. V práci sme postupne aplikovali rôzne zmeny na vybrané známe kryptografické algoritmy a generátory náhodných čísel. Tými boli z algoritmov: MD5, SHA-1, RC4, Murmur hash, FNV-1a a Pearson hash. Z generátorov náhodných čísel, sme sa venovali algoritmom LCG a Mersenne Twister. Následne sme sa pozreli na výsledné vlastnosti týchto zmenených algoritmov v porovnaní s vlastnosťami originálnych algoritmov. Cieľom bakalárskej práce bolo poukázať na to, že aj malé chyby v implementácií algoritmu môžu mať za následok veľké zmeny vo výsledných vlastnostiach. Prácou chceme upozorniť programátorov, aby si dávali pri implementácií týchto algoritmov pozor a nezanedbávali maličkosti. Chceme varovať aj tých, ktorí chcú do algoritmu vniesť chybu úmyselne, aby si svoje konanie radšej dobre rozmysleli.

Kľúčové slová: kryptografické algoritmy, zmeny v algoritmoch, prejavy zmien v algoritmoch, hashovacie funkcie, PRNG

Abstract

In this bachelor thesis we deal with changes in cryptographic algorithms on their resulting properties. At work, we have gradually applied various changes to selected known cryptographic algorithms and random number generators. These were algorithms: MD5, SHA-1, RC4, Murmur hash, FNV-1a and Pearson hash. From random number generators, we've devoted ourselves to an algorithm LCG and Mersenne Twister. Subsequently, we looked at the resulting properties of these altered algorithms compared to the properties of the original algorithms. The aim of the bachelor thesis was to point out that even minor errors in the implementation of the algorithm can result in great changes in the resulting properties. We want to remind programmers to be careful about implementing these algorithms and not neglecting the little ones. We also want to warn those who want to make the mistake intentionally to make their own decisions better.

Keywords: cryptographic algorithms, changes in algorithms, effects of changes in algorithm, hash functions, PRNG

Obsah

Zoznam obrázkov

Úvod

Každý z nás už určite niekedy potreboval predať alebo uchovať informáciu a súčasne si byť istý, že sa ju nikto druhý nedozvie. Problémom je, keď na to potrebujeme použiť nejaký komunikačný kanál, ktorý je verejne dostupný. Tu nám pomôže šifrovanie. Útočník, resp. neželaný čitateľ takto získa len údaje ako zašifrovaný text, ale nie obsahovú informáciu. Avšak mnohé šifrovacie algoritmy sú odhalené a ich kódy zverejnené. Alebo dokonca už sú predpočítané rôzne zašifrované hodnoty - napr. heslo „admin“ bude mať výstupnú hodnotu po aplikovaní MD5 vždy 21232f297a57a5a743894a0e4a801fc3 a takýchto slov je predpočítaných ako budú vyzerať už veľmi veľa. Podobne je to aj s generátormi náhodných čísel, ktoré sú počítané deterministickými algoritmami, a preto sa dajú ľahko predpočítať podľa inicializačnej hodnoty - seedu. Preto by človeka mohlo napadnúť „trochu pozmeniť“ algoritmus, napr. len prepísať nejakú neškodnú konštantu, veď sa nemôže nič stať. A nadobudne mylný pocit väčšieho bezpečia. Ľahko sa mu totiž môže stať, že „jeho“ hašovacia funkcia bude produkovať často kolízie, „jeho“ zašifrovaný text bude obsahovať korelácie a „jeho“ generátor náhodných čísiel začne po chvíli generovať samé 0 či bude obsahovať vzory a opakovať sa.

Ďalším druhom zmien v algoritmoch môžu byť neúmyselné chyby. Tieto nie sú príliš neobvyklé, o čom vypovedajú aj skúsenosti reverzných inžinierov. Človek ju do algoritmu môže vniesť úplne nevedome ak si napr. chce prepísať kód z internetu do programu, spraví chybu pri kopírovaní, niečo omylom zmaže alebo pridá. Ešte nebezpečnejšie je vlastné napísanie implementácie algoritmu napr. podľa inštrukcií z Wikipedie či iných zdrojov, kde tieto algoritmy nemusia byť uvedené správne alebo na nich chýbajú uvedené určité dôležité detaily či človek jednoducho niečo preskočí a vynechá.

Cieľom tejto práce je demonštrovať vplyvy takýchto zmien v niektorých vybraných a známych šifrovacích algoritmoch, hašovacích funkcií a generátorov pseudonáhodných čísel. Pre každý algoritmus zvlášť vhodne navrhneme a zostrojíme viacero rôznych druhov zmien a poukážeme na výsledné vlastnosti zašifrovaného textu. Následne pojednáme o tom, prečo takýto druh zmeny ovplyvnil, resp. neovplyvnil „silu“ zašifrovania. Touto prácou sa teda snažíme vystríhať programátorov od nie príliš uvážených zmien v algoritmoch. Taktiež by práca mohla byť pomocným nástrojom pri odhalení napr. spôsobu šifrovania disku nejakým zlým ransomwarom, ak útočník vniesol zmenu do algoritmu, ktorá ovplyvní vlastnosti výstupného textu a my spozorujeme koreláciu,

nakoľko vlastnosti výsledného textu môžu napovedať o druhu zmeny, ktorá bola do algoritmu zavedená.

Práca je viac implementačná, resp. experimentálna, takže na začiatku si len stručne popíšeme základy kryptológie ako takej a prejdeme ku skúmaniu a experimentovaniu s algoritmami. Vždy sa stručne oboznámime s vybranou funkciou a popíšeme si jej algoritmus. Následne budeme našimi vytvorenými softvérmi postupne implementovať chyby, spúšťať takéto algoritmy na rôznych vstupoch a sledovať zmeny správania na vybraných vlastnostiach. Tie zaujímavejšie si popíšeme s priloženými vyprodukovanými grafmi ich vlastností.

Pozn.: Názov práce znie „Dôsledky zmien v kryptografických algoritmoch“, v práci sa však venujeme okrem algoritmov používaných priamo na kryptografické účely aj nekryptografickým hašovacím funkciám a generátorom pseudonáhodných čísel. Tieto s nimi súvisia, môžu sa používať súčasne s kryptografickými v jednom systéme a môžu byť tiež slabinou pri útokoch. Napríklad aj kryptograficky silná šifrovacia funkcia môže používať nekryptografický alebo dokonca poškodený generátor náhodných čísel na vygenerovanie svojho kľúča.

Kapitola 1

Kryptológia

V tejto kapitole uvedieme čitateľa do problematiky a priblížime si niektoré základné kryptologické a kryptografické pojmy, ktoré budeme neskôr v práci potrebovať. Informácie v tejto kapitole boli čerpané z nasledujúcich zdrojov: [22], [21], [18], [16], [6]. Z nich boli vybrané len definície a pojmy relevantné pre túto prácu, t.j. také, ktoré budeme ďalej v texte potrebovať. Pozn.: Definície sú zjednodušené pre ľahšie pochopenie aj neodbornému čitateľovi.

Kryptológia je náuka o tom, ako utajiť obsah správ. Delí sa na kryptografiu, kryptoanalýzu a steganografiu.

Kryptografia - z gr. „tajné písanie“, skúma a navrhuje šifrovacie systémy. Avšak je to viac ako len šifrovanie, okrem toho, že sa snaží spraviť správy nečitateľnými, snaží sa aj zabezpečiť dôvernosť údajov, integritu, autentickosť, nepoprenie autorstva, nepoprenie doručenia a podobne.

Kryptoanalýza - skúma spôsoby získavanie obsahu zašifrovanej správy a iné možnosti útokov (cez uvedené bezpečnostné atribúty). Je vlastne opakom kryptografie.

Steganografia - skúma spôsoby utajenia, že vôbec nejaká správa existuje. Napr. použitie neviditeľného atramentu alebo v oblasti informatiky - ukrytie textovej správy do bitmapy (obrázka).

Nás však v tejto práci zisťovanie pôvodného obsahu správy ani ukrytie existencie správy nezaujíma, preto sa ďalej budeme venovať kryptografii.

1.1 Základné definície

Šifrovanie - Utajenie správy, prevedenie z jej čitateľnej podoby do nezrozumiteľnej.

Dešifrovanie - Zistenie obsahu utajenej správy, prevedenie správy naspäť do čitateľnej, pôvodnej.

Kľúč - je istá informácia, ktorá určuje *zmenu* správy z čitateľnej na nezrozumiteľnú, resp. naopak pri dešifrovaní.

Kerckhoffsov princíp - hovorí o tom, že „sila“ šifry by nemala byť postavená na utajení jej algoritmu, ale na bezpečnosti kľúča, resp. že šifra by mala ostať bezpečná aj po tom, ako sa dozvieme jej algoritmus. Ten sa môže dostať na verejnosť pomocou reverzného inžinierstva ako aj infiltráciou indivíduua do firmy, ukradnutím a zverejnením. Tento princíp však porušuje niekoľko algoritmov. Toto sa ale pre ne môže stať osudným, ako už naznačila história.

Lavínový efekt - Avalanche effect - žiaduca vlastnosť typicky pre blokové šifry (viď 1.1.4) a hašovacie funkcie (viď 1.1.6). Malá zmena textu na vstupe spôsobí obrovské zmeny vo výstupe.

Efektívne vypočítateľné = vypočítateľné v rozumnom/dosiahnuteľnom čase, t.j. vypočítateľné v polynomiálnom čase.

1.1.1 Symetrické šifrovanie

Symetrické šifrovanie je spôsob šifrovania, ktorý používa taký istý kľúč pre zašifrovanie ako aj následne pre dešifrovanie. Tento spôsob šifrovania je výpočtovo oveľa rýchlejší ako asymetrické šifrovanie aj preto, že (zvyčajne) môže používať kratšie kľúče. V dnešnej dobe stačí kľúč dĺžky 128 bitov. Už takýto dlhý kľúč útočník nie je schopný efektívne vypočítať, pretože potrebuje obvykle vyskúšať všetky možné kľúče a tých je 2^{128}. Samozrejme toto šifrovanie má zmysel len ak predpokladáme, že kľúč na dešifrovanie nepozná nikto cudzí a vlastní ho len príjemca. Preto si musíme dávať pozor na bezpečné prenesenie a uschovanie kľúča. V súčasnosti sa symetrické šifrovanie využíva na bežné šifrovanie a dešifrovanie diskov, nakoľko nespôsobuje žiadne pocítiteľné spomalenie.
Symetrické šifry môžeme ďalej deliť na blokové a prúdové.

1.1.2 Asymetrické šifrovanie

Pri asymetrickom šifrovaní treba na dešifrovanie iný kľúč ako pre zašifrovanie. Existujú teda dva kľúče, jeden verejný a jeden súkromný, ktoré sú zvolené tak, aby po aplikácii oboch na nejakú správu ostala správa v konečnom dôsledku bez zmeny a zároveň nebolo možné efektívne vypočítať z verejného kľúča súkromný. Verejný kľúč sa používa na šifrovanie a súkromný na dešifrovanie.
Táto metóda je výpočtovo náročnejšia, nakoľko je treba oveľa dlhšie kľúče, aby sa súkromný kľúč nedal efektívne vypočítať z verejného. Keby sme chceli rovnako „náročnú zistiteľnosť“ kľúča (teda bezpečnosť) ako pri 128 bitovom kľúči pri symetrickom šifrovaní, potrebovali by sme (pri implementáciách asymetrických šifier založených na faktorizácií integerov, ako napr. RSA šifra) kľúč veľkosti 3072 bitov [3]. Avšak nemusíme zabezpečovať bezpečné prenesenie kľúčov, pretože súkromný kľúč máme len my a znalosť verejného kľúča aj neželanými osobami by nám, pri správnej implementácii, nemala nijako uškodiť.

1.1.3 Hybridné šifrovanie

Pretože symetrické a asymetrické šifrovanie majú navzájom tieto svoje výhody a nevýhody opačné, používajú sa obe súčasne ako hybridné šifrovanie. Toto využíva výhody symetrického šifrovania - výpočtovú rýchlosť a asymetrického šifrovania - bezpečnosť prenosu kľúčov.
Najskôr sa vytvorí kľúč symetrickej šifry, ktorou sa zašifruje správa, následne sa tento kľúč prenesie asymetrickým šifrovaním. Kľúč si teda vie prečítať len príjemca a len on si vie správu dešifrovať.

1.1.4 Blokové šifry

Bloková šifra šifruje správu po blokoch, teda robí operácie nad fixným počtom bitov. Ak je počet bitov väčší ako určený, tak správu rozdelí na časti. (Časti sa môžu, alebo nemusia vzájomne ovplyvňovať.) Ak je naopak správa kratšia ako určený počet bitov, správa sa doplní, nakoľko niektoré blokové algoritmy nevedia robiť s kratšími reťazcami. Toto vyplnenie sa nazýva aj **padding**.
Blokové šifrovanie je iteratívna šifra, a teda sa pri ňom používa napr. niekoľkonásobná jednoduchá transformácia.

1.1.5 Prúdové šifry

Prúdové šifrovanie je metóda, v ktorej sú postupne s bitmi na vstupe kombinované bity vygenerované pseudonáhodným bitovým generátorom. Na kombináciu sa používa operácia XOR.

Blokové šifry sú síce zvyčajne trochu pomalšie ako prúdové, ale dá sa na nich ľahšie simulovať, a preto sú prúdové šifry menej používané ako blokové. Ďalším problémom prúdových šifier je, že si treba dávať pozor na použitie kľúča, aby sa kľúč nepoužil viac-krát a vždy ho inicializovať znova.

1.1.6 Hašovacie funkcie

Hašovacie funkcie sú funkcie, ktoré vytvoria zo zadaného vstupu reťazec pevnej dĺžky, tzv. odtlačok, pričom nezáleží na tom či je vstup kratší alebo niekoľkonásobne dlhší ako dĺžka odtlačku. Odtlačok sa nazýva haš a zvyčajne mával dĺžku 128 či 160 bitov. Avšak dĺžka hašu sa postupne navyšuje a v novo navrhnutých súčasných algoritmoch môže byť výstup dlhý aj viac bitov, napr. až 256. Na rozdiel od šifrovania tu prichádza k redukcii informácií, preto nejde pôvodnú správu naspäť zrekonštruovať. Hašovanie je teda len jednosmerné. Výpočet hašu je v niektorých funkciách či implementáciách veľmi rýchly, čo je pre ne dobrá vlastnosť. V iných hašovacích funkciách môže byť výpočet hašu pomalý, čo môže byť tiež dobrá vlastnosť, ktorá takto napr. spomalí útočníkom brute-force útok (útok hrubou silou, skúšanie všetkých možností). Preto sa hašovacie funkcie hojne používajú napr. na: overenie integrity dát ako kontrolný súčet, detekciu chýb - toto je potrebné nielen na odhalenie zmeny textu útočníkom, ale aj chýb zlyhania techniky či vplyvu prostredia, ktoré nemajú nulovú šancu nastania pri zápise či prenose dát, indexovanie - kde haš reprezentuje akoby vlastnosti dát, bezpečné uloženie hesiel v databázach, rýchle vyhľadávanie.

Tieto funkcie sú deterministické a nepoužívajú žiadne kľúče, preto je haš ku každému reťazcu deterministicky určený. Teda pre rovnaký vstup vyjde vždy rovnaký výstup. (Napr. MD5 haš k heslu admin bude vždy 21232f297a57a5a743894a0e4a801fc3.) Každá hašovacia funkcia by mala spĺňať tieto vlastnosti:

- Odolnosť vzoru (Preimage resistance/Odolnosť výstupu). Útočník by si nijako nemal vedieť vytvoriť z výstupného hašu taký reťazec, ktorý by mal rovnaký haš.
- Odolnosť 2. vzoru (Second preimage resistance/Odolnosť vstupu). Útočník by nemal vedieť vytvoriť zo vstupu a jeho hašu iný reťazec, ktorý by mal rovnaký haš.
- Odolnosť voči kolíziám. Útočník nevie vytvoriť dva rôzne vstupy s rovnakým výstupom. Kolízie sú nežiaduce, ale nedá sa im vyhnúť úplne - existujú vždy, a preto by mali byť ťažko nájditeľné.

Dôsledkom týchto 3 vlastností je aj skutočnosť, že funkcia má lavínový efekt (spomínaný v začiatkoch tejto kapitoly). Tieto vlastnosti môžeme zhrnúť ako nezistiteľnosť fungovania algoritmu.

Mnohé hašovacie algoritmy však už boli odhalené a prelomené alebo je ich bezpečnosť nedostatočná. Číňania Wangová, Feng, Lai a Yu už prelomili niekoľko hašovacích funkcií a varujú [26]: „Besides hash functions we break, there are some other hash functions not having ideal security. “

Okrem hašovacích funkcií, ktoré majú kryptografické vlastnosti, poznáme aj také, ktoré nemajú tieto vlastnosti. Tieto sa používajú napr. len na overenie integrity či na vytváranie indexov do tabuliek. Takéto hašovacie funkcie nemusia byť odolné proti hľadaniu vzoru. Požadovanou vlastnosťou však stále ostáva odolonosť voči kolíziám a lavínový efekt. Tu však odolnosť voči kolíziám nesúvisí s útočníkmi, ale so schopnosťou nevytvárať kolízie pre náhodné dáta.

Kapitola 2

Vlastnosti a implementácia

V tejto kapitole si povieme niečo o tom, ako vizualizovať niektoré vlastnosti algoritmov. Taktiež si povieme o štandardizovaných testoch. Následne poukážeme na to, ktoré vlastnosti skúmame my a ktoré sme implementovali, prečo a ako fungujú. Na konci kapitoly pojednáme o tom, aké druhy zmien môžu v algoritmoch nastávať a ktoré sme vnášali do vybraných algoritmov my.

2.1 Testy náhodnosti

Testy náhodnosti (angl. Randomness tests) sú používané na analýzu distribúcie dát, resp. na zistenie či dáta neobsahujú vzory. Takýchto testov je nespočetne veľa a neustále sa tvoria nové. My si popíšeme niektoré z nich, a to konkrétne NIST testy a Diehard testy.

2.1.1 NIST testy

NIST je národná inštitúcia štandardov a technológií USA. ktorá ustanovuje a štandardizuje miery a spôsoby merania takmer vo všetkých oblastiach života. Bola založená už v roku 1901 a dnes sa spolieha nespočetné množstvo produktov a služieb na jej štandardy. Okrem iného pojednáva aj o kryptografických algoritmoch, ktoré by mali byť štandardne používané. [13] My sa pozrieme na jej štatistické testy pre generátory náhodných čísel.

Aj vedci z NIST-u tvrdia [19], že existuje nekonečne veľa možných štatistických testov (pre testovanie generátorov náhodných čísel), z ktorých **každý testuje prítomnosť alebo absenciu určitého „vzoru“.** Ak je nejaký z týchto vzorov detegovaný, indikuje to, že sekvencia nie je náhodná. Nakoľko existuje takáto veľká množina rôznych testov posudzovania či je postupnosť náhodná alebo nie, neexistuje konečná množina vybraných testov, ktorá by bola považovaná za kompletnú.

Pri testovaní môžu nastať dva zlé prípady:

- Test povie, že dáta nie sú náhodné aj ak sú. Bežné hodnoty nastania tohto prípadu v kryptografii sa pohybujú okolo 0.01. (To znamená, že bude nesprávne zamietnutý 1 dobrý generátor postupností zo 100 testovaných generátorov.)

- Test povie, že dáta sú náhodné aj ak nie sú. Tento jav je pomerne bežný, pretože je mnoho možných typov „nenáhodnosti“.

Hlavným cieľom týchto štandardizovaných NIST testov je znížiť pravdepodobnosť nastania druhej neželanej chyby. NIST preto navrhla 16 testov a odporúča, aby bol frekvenčný test prevedený ako prvý. Ak tento test neuspeje, potom je veľmi pravdepodobné, že neuspejú ani ostatné. Preto si podrobnejšie popíšeme len tento.

Frekvenčný test. Tento test je zameraný na pomer núl a jednotiek v celej postupnosti. Účelom je určiť či má postupnosť približne rovnaký počet bitov 0 a 1, čo je očakávané v skutočne náhodnej sekvencii. Evaluácia tohoto testu:

1. Určíme hodnotu S ako súčet všetkých bitov postupnosti, pričom bit 1 má hodnotu 1 a bit 0 má hodnotu -1. Ak je počet núl a jednotiek rovnaký, výsledok bude 0.

2. Vypočítame $s_o = |S|/\sqrt{n}$, kde n je počet bitov na vstupe.

3. Vypočítame $P = erfc(\frac{s_o}{\sqrt{2}})$, kde

$$erfc(z) = \frac{2}{\sqrt{\pi}} \int_z^\infty e^{-u^2} du.$$

Ak je vypočítaná hodnota $P < 0.01$, postupnosť je považovaná za nenáhodnú. V druhom prípade týmto testom prešla. Odporúčané minimálne n, teda počet vstupných bitov, pre tento test je 100 bitov. Samozrejme sa netestuje len jedna takáto postupnosť, ale sleduje sa aj distribúcia P hodnôt.

Ďalšie testy sú obdobné, sú tu ešte napríklad: frekvenčný test na blokoch, test behov (beh je neprerušená postupnosť rovnakých bitov), test najdlhšieho behu jednotiek v bloku, test binárnej matice, test neprekrývajúcich sa šablón, sériový test a iné. Záujemcovia sa môžu o nich viac dočítať v použitom diele pre túto podkapitolu - [19].

2.1.2 DIEHARD testy

George Marsaglia navrhol v roku 1995 tieto štatistické testy kvality, ktoré sú pravdepodobne najznámejším softvérovým balíkom na testovanie generátorov náhodných čísel. Nevýhodou týchto testov však bolo, že boli značne neflexibilné a mali obmedzené vstupné reťazce, napr. fixné dĺžky vzoriek. Testy Diehard obsahujú 15 testov, niektoré z nich sú totožné s už spomínanými testami štandardizovanými NIST-om. [11]

2.2 Postup a legenda implementácie

Pre potreby práce sme vytvorili sadu softvérov. Pre kryprografické a hašovacie funkcie je to najskôr generátor vstupných reťazcov - ten vytvorí ľubovoľný počet ľubovoľne dlhých (prípadne náhodne dlhých) reťazcov. Reťazce sú uložené v kódovaní ASCII v hexadecimálnej podobe. Takéto reťazce sú generované náhodne.

Potom je tu voľba generovania **opakujúcich sa reťazcov**, zložených len z opakovania sa jedného písmena, zvolili sme písmeno „a“. V týchto reťazcoch má teda postupne prvý reťazec dĺžku 1 - „a“, druhý dĺžku 2 - „aa“, až posledný dĺžku 1025.

V našom generátore vstupných reťazcov sa nachádza ešte ďalšia možnosť, a tou je generovanie **reťazcov** zložených v hexadecimálnom tvare zo samých núl, teda v ASCII reprezentovaných ako znak **null**. Takýto vstup je podobne ako opakujúce sa jedno písmeno postupne dĺžky 1 až 1025.

Ako ďalšiu aplikáciu sme vytvorili nástroj, ktorý spracuje vygenerované reťazce a použije naň používateľom vybraný kryptografický či hašovací algoritmus. V ponuke sú okrem originálnych algoritmov aj algoritmy s rôznymi vnesenými chybami. Chyby boli postupne vnesené priamo do správnej implementácie algoritmu. Chyby sú volené tak, aby boli výsledky „zaujímavé“. Vykonané zmeny často spôsobovali, že výstup bol konštantný, obsahoval v hexadecimálnom tvare len znaky „F“ alebo „0“. Takéto chyby sme nechceli príliš rozoberať a popisovať, pretože programátor si ich na výstupe ľahko všimne. Preto sa v práci snažíme prezentovať hlavne zaujímavejšie chyby, a to také, kde sa výstup môže zdať na prvý pohľad náhodný. Softvér je implementovaný v jazyku Java/JavaFX. Výber jazyka nijako neovplyvnil výber chýb. Implementovali sme rôznorodé zmeny. Okrem zmien, ktoré môžu nastať v Jave (napr. zabudnutie uzátvorkovania) aj zmeny, ktoré v Jave nemôžu nastať (napr. zabudnutie inicializácie premennej), ale aj zmeny, ktoré môžu nastať rovnako vo všetkých jazykoch (napr. iné hodnoty konštánt).

Podobne sa správa aj generátor pseudonáhodných čísiel, ktorý vygeneruje ľubovoľne dlhú postupnosť pomocou neupravovaných alebo upravených algoritmov podľa výberu z ponuky.

Ďalším softvérom je softvér, ktorý zobrazuje vybrané vlastnosti výstupnej postupnosti či už pre generátory náhodných čísel alebo kryptografické a hašovacie funkcie. Jednou z vlastností je zobrazenie korelácie vstupných a výstupných reťazcov pomocou heatmapy. Táto je popísaná ďalej 2.2.1. Ďalšou vlastnosťou je distribúcia znakov na zvolenej pozícií. Táto zobrazí, pre danú pozíciu znaku výstupu, s akou pravdepodobnosťou sa tu vyskytli ktoré hexadecimálne znaky.

Pre PRNG je to napr. zobrazenie v čase, ktoré na vodorovnej osi zobrazuje postupne narastajúci čas/poradie vygenerovaného čísla a na zvislej osi zobrazuje hodnotu tohto čísla. Ďalej je to distribúcia čísiel pomocou zvoleného modula, ktorá poukazuje na náhodné rozloženie čísel.

Vybrané implementované vlastnosti ani zďaleka neodhaľujú všetky možné nedostatky, ktoré mohli vzniknúť vnesením zmeny do algoritmu, avšak sú postačujúce na demonštráciu zmien vo vlastnostiach a pre naplnenie cieľov zadania práce.

Všetky časti softvéru sú jednoducho rozšíriteľné. Pri generátore sa dá jednoducho upraviť znak opakujúcej sa konštanty, ako aj maximálna dĺžka, po ktorú sa znak opakuje; v hašovacej a šifrovacej časti sa dajú ľahko implementovať ďalšie iné algoritmy či prirobiť zmeny, ktoré chceme vniesť do algortimov. Podobne je to aj s generátormi čísel. Softvéry na zobrazovanie vlastností sa dajú tiež ľahko rozšíriť o ďalšie vlastnosti.

Ďalšie kapitoly práce

V ďalších kapitolách práce postupne uvedieme algoritmy, popíšeme si vnesené zmeny a poukážeme pomocou obrázkov na výsledné vlastnosti. Ku každému algoritmu si vždy v prvej časti popíšeme všeobecné informácie o algoritme, ako napr. čo je zač, aké má vlastnosti či popíšeme už vykonané štúdie. V kapitolách „Informácie o algoritme“ je popis alebo pseudokód algoritmu. Tento nemá slúžiť ako návod na vlastné implementovanie daného algoritmu, a preto nie je uvedený príliš podrobne. Popis algoritmu uvádzame pre pochopenie fungovania algoritmu ako takého, aby sme v následnej časti „výsledky“ mohli predpokladať znalosť fungovania algoritmu, ako aj pre lepšie pochopenie a predstaviteľnosť popisov vnesených zmien. V prípade potreby vlastnej implementácie odporúčame čitateľovi inšpirovať sa radšej zdrojovými kódmi týchto algoritmov, ktoré sú uvedené v prílohe na CD.

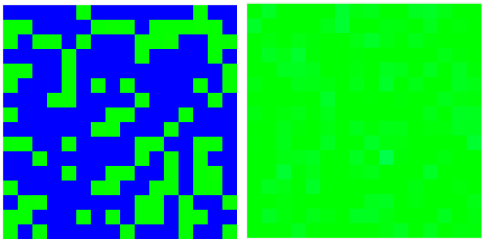
2.2.1 Heatmapa

Heatmapa, slovensky „teplotná mapa“, je grafické zobrazenie hodnôt matice. Zobrazenie je 2D a je vo forme mapy alebo akéhosi diagramu a hodnoty sú reprezentované farbami. Heatmapa sa využíva vo veľa rôznych oblastiach a je veľmi nápomocná, pretože poskytuje vizuálne zhrnutie informácií a umožňuje čitateľovi pochopiť aj komplexnejšie dáta. Dáta prezentované číslami by boli oveľa ťažšie pre pochopenie a neprehľadné. Mapa zobrazuje vzťahy medzi hodnotami a pomocou farieb indikuje ich hodnoty (aktivitu), väčšinou sa používajú tmavšie farby na označenie nízkej aktivity a jasnejšie na označenie vysokej aktivity.

Implementácia heatmapy

V našej práci využívame výhody heatmapy. Používame ju na zobrazenie korelácie medzi vstupnými a výstupnými bitmi (teda x-tým vstupným znakom a y-tym výstupným). V práci je ďalej používané označenie ako „x-tý vstupný znak“. Znak tu znamená znak z rozmedzia 0-F, teda 4 bity reťazca. V heatmape sa ďalej nachádzajú zvislo vstupné

znaky 0-F (na zvolenej jednej pozícii vo vstupnom reťazci), na vodorovnej osi je to zase 16 výstupných znakov 0-F (pre vybranú pozíciu výstupných reťazcov) (obr. 2.1). Čiže napr. pre jeden vstupný a jeden výstupný reťazec existuje *(dĺžka vstupu) x (dĺžka výstupu)* rôznych heatmáp. Samozrejme heatmapy nerobíme pre každý jeden samostatný vstup a výstup, ale zobrazujeme v nich hneď niekoľko stotisícov (resp. zvolený počet) takýchto dvojíc súčasne.



Obr. 2.1: Vľavo - ukážka heatmapy. Vpravo - ideálna heatmapa.

Farebná škála je zvolená od modrej cez zelenú až po červenú (obr. 2.2), pričom zelená reprezentuje priemernú hodnotu tabuľky pre jedno políčko. Úplne modré políčko značí hodnotu 0, červené značí maximálnu hodnotu. Ak je celý jeden riadok modrý, znamená to, že tento znak na zvolenej pozícií sa na vstupe nevyskytol. Ideálna heatmapa by mala mať všetky polia zelenej farby (obr. 2.1 vpravo).

Obr. 2.2: Farebná škála heatmapy.

2.3 Príklady zmien v algoritmoch

V tejto podkapitole si povieme o zmenách, ktoré sme vnášali do algoritmov, prečo a ako by mohli vzniknúť.

Zmena do algoritmu môže byť vnesená úmyselne alebo neúmyselne. Úmyslené zmeny budeme voliť pre každý algoritmus samostatne. Tieto sú vnášané do algoritmov pravdepodobne kvôli vyhnutiu sa rôznym útokom na ne. Taktiež môžu vznikať snahou vytvoriť „lepší“ algoritmus. Do tejto kategórie môžeme zaradiť zároveň aj všetky neúmyselné zmeny, pretože tie mohli byť vnesené taktiež úmyselne ako aj neúmyselne.

Neúmyselné zmeny, teda chyby, môžu vznikať z rôznych dôvodov, napr. kvôli ponáhľaniu sa a nepozornosti, preklepu, zlému zkopírovaniu, čerpaniu z nedôveryhodného zdroja či nepochopením pseudokódu, ktorý programátor používa ako predlohu, podľa ktorej kód implementuje. Neúmyselné chyby môžu ďalej vzniknúť kompilátorom alebo

vlastnosťami jazyka, v ktorom programujeme. Rôzne kompilátory majú rôzne defaultné správanie a mali by sme sa najskôr oboznámiť s ich špecifikáciou. My sme ďalej v práci implementovali tieto chyby:

1. Neinicializovanie konštanty.

V niektorých programovacích jazykoch sa človeku ľahko môže stať, že na začiatku kódu vytvorí premennú, ktorú potom ďalej bude používať, no medzitým ju zabudne inicializovať a zabudne jej prideliť hodnotu. Toto sa v jazyku, v ktorom sme chyby implementovali, teda v Jave, stať nemôže. Tá nespustí program s neurčenými premennými. Preto sme túto chybu simulovali tak, že sme premennej priradili hodnotu 0.

2. Iné hodnoty konštánt.

Takáto chyba vyzerá skôr akoby bola vnesená úmyselne, ale pri veľkých a viacciferných číslach sa môže stať, že človek z čísla nejakú cifru vypustí či napíše niektorú cifru inú.

3. Zlé nainicializovanie poľa alebo tabuľky.

Podobne ako pri bode 1. či v 2. bode. Tabuľky používajú viaceré algoritmy, ktorým sa budeme venovať. Táto sa dá podobne ako konštanty, zabudnúť inicializovať úplne, alebo nainicializovať nesprávnymi hodnotami.

4. Pokazený for-cyklus.

For-cyklus sa dá pokaziť v rôznych jazykoch rôznymi spôsobmi. V jazykoch používajúcich bodkočiarku na ukončenie každého príkazu sa dá preklepnúť bodkočiarka hneď za zavolaním for-cyklu. Takýto preklep je v kóde vcelku nenápadný a ťažko sa hľadá voľným okom.

```
for (int i=0;i<n;i++);
{
     ...
}
```

Ďalším spôsobom pokazenia for-cyklu môže byť zabudnutie začiatočných zátvoriek za for-cyklom. Toto sa môže ľahko stávať programátorom, ktorý prechádzajú z programovacích jazykov, ktoré nepoužívajú takéto zátvorky na také programovacie jazyky, v ktorých sú potrebné. V tomto prípade sa vykoná vo for-cykle žiadaný počet krát len prvý príkaz, nachádzajúci sa hneď za for-cyklom, ktorý často ešte nič relevantné nevykonáva, len napr. priraďuje hodnoty premenným.

```
for (int i=0;i<n;i++)
    1.príkaz
    2.príkaz
```

V iných programovacích jazykoch, ktoré nepoužívajú bodkočiarky ani zátvorky, ako napr. v Pythone, sa dá pokaziť for-cyklus zlým odsadením. Toto sa opäť môže stať ľuďom, ktorý prechádzajú z iných programovacích jazykov, kde odsadenie slúžilo len pre lepšiu orientáciu v kóde.

5. Nekorektná implementácia bežných operácií.

Do tejto kategórie sme zaradili napr. nesprávne naimplementovanú operáciu výmeny obsahu dvoch premenných. V náročnejšej verzii - výmena obsahu dvoch políčok poľa. Takúto základnú vec by mal vedieť spraviť každý programátor, no každý z nás bol raz začínajúci a neskúsený programátor, ktorý musel prísť najskôr na základné veci. Taktiež programátori, ktorý sa s výmenou obsahu dvoch premenných ešte nestretli, môžu ľahko zle pochopiť pseudokód pre túto operáciu, ktorý mohol vyzerať (alebo pre niektoré jazyky aj vyzerá) takto:

```
a,b = b,a.
```

Takúto vec mohol pochopiť jednoducho a bez premýšlania prepísať ako

```
a = b;
b = a;
```

Výmenu obsahu dvoch premenných sa podaktorí môžu snažiť aj nejako zefektívniť alebo implementovať bez použitia tretej, pomocnej, premennej či šetriť miestom a napísať príkaz do jedného riadku napr. pomocou operácie XOR.

6. Iné preklepy.

Do tejto kategórie zaradíme preklepy, ktoré sú špecifické pre daný algoritmus a implementáciu.

7. Nesprávne pochopenie pseudokódu.

Tetno prípad môže nastať, keď v pseudokóde sú použité rôzne špeciálne značky, ktorých význam nie je nikde vysvetlený alebo sa predpokladá, že ich každý ovláda. Niektoré značky sa však môžu používať aj v rôznych významoch. Takýmito znakmi môže byť napr. plus v krúžku $\oplus$ alebo plus v štvorčeku , ktoré môžu byť pochopené aj ako plus aj ako napr. operácia XOR.

Kapitola 3

Kryptografické algoritmy

If you think cryptography will solve your problem, then you don't understand cryptography ... and you don't understand your problem. –Bruce Schneier

V tejto kapitole popíšeme bližšie tie kryptografické algoritmy a hašovacie funkcie, ktoré sme skúmali. Najprv si povieme základné veci o algoritme, jeho použitie, výhody a silné stránky, resp. prečo sme si ho zvolili, potom uvedieme ukážky častí kódu, na ktoré sme sa zamerali, vysvetlíme aké druhy zmien sme zvolili a prečo, následne poukážeme na zaujímavé výsledné vlastnosti (ktoré sme získali použitím nášho vytvoreného softvéru) pomocou grafov či tabuliek.

Spojenie kryptografických algoritmov a hašovacích funkcií v tejto kapitole nastáva preto, lebo oboje sa môžu použiť na dosiahnutie rovnakých cieľov. Spolu ich môžeme rozdeliť na dva druhy:

- Tie, ktoré sú využívané skôr na overovanie integrity či indexovanie. Zmeny v týchto algoritmoch sú skôr neúmyselného charakteru.
- Tie, ktoré sú používané na skutočné šifrovanie údajov s úmyslom znemožnenia zistiteľnosti pôvodnej informácie. Zmeny v týchto algoritmoch môžu vznikať už aj úmyselne, s naivným cieľom vytvoriť „bezpečnejší“ algoritmus, nakoľko značné množstvo algoritmov už bolo prelomených (ako si ukážeme ďalej v texte).

3.1 Šifrovacie algoritmy

3.1.1 MD5

Všeobecné informácie

MD5 je hašovacia funkcia (viď 1.1.6). Pôvodne bola navrhnutá ako kryptografická haš funkcia, ale dnes už o nej vieme, že má veľa zraniteľností. MD5 bola prelomená v roku

2004 (už spomínanými) Číňanmi, ktorí dokážu so svojim algoritmom nájsť približne každú hodinu 1 kolíziu. (Pri MD4 to s týmto istým algoritmom stihnú už za 1 sekundu.) [26]. Odvtedy vznikli už aj mnohé vylepšenia týchto útokov. MD5 by sa preto nemala používať ako kryptografická, ale mala by sa využívať len napr. na kontrolu integrity. Nie je tomu tak a stále sa hojne používa na utajenie hesiel v databázach webových aplikáciách. [20]: „ The most frequent way passwords are stored are with a weak hashing algorithm, such as MD5 or SHA1. “ V apríli tohto roka prestala podporovať podpisovanie JAR súborov pomocou MD5 aj firma Oracle a takéto dokumenty označuje ako nepodpísané [17].

Táto funkcia má takú vlastnosť, že ak dva reťazce majú rovnaký haš, potom aj ak k týmto reťazcom pripojíme rovnaký ďalší reťazec, budú mať rovnaký haš. Teda ak md5(x) == md5(y), potom md5(x+q) == md5(y+q).

Rodina MD

Funkcia MD5 je odvodená z funkcie MD4. Z MD4 sú odvodené aj funkcie skupiny SHA a funkcie skupiny RIPEMD, preto si sú všetky tieto skupiny podobné a majú podobné vlastnosti.[10]

Informácie o algoritme [10]

MD5 vytvára 128 bitové odtlačky. Vstup spracováva po 512 bitových blokoch. Algoritmus MD5 je vcelku dosť dlhý a ako si môžeme všimnúť, používa veľa konštánt a logických operácií. Jeho chod môžeme zhrnúť do 4 krokov:

1. Doplní paddingom vstupnú správu na 512-bitové bloky tak, aby v poslednom bloku ostalo len 448 bitov. Vždy doplní len počet bitov z rozmedzia 1 až 512, teda ak mala správa dĺžku napr. 448 bitov, doplní 512 bitov, teda správu doplní na 960 bitov.

2. Do správy doplní zvyšných 64 bitov, reprezentujúcich dĺžku správy. Správa ako taká teda môže mať veľkosť len $2^{64} - 64$ bitov. Po tomto kroku teda máme n rovnakých úsekov dĺžky 512.

3. Inicializuje si 4 registre, a, b, c a d do ktorých si ukladá priebežné výsledky a následne výsledok. Premenné inicializuje na hodnoty: a = 0x67452301; b = 0xefcdab89; c = 0x98badcfe; d = 0x10325476;

4. Spracovanie po blokoch prebieha v štyroch kolách po 16 krokoch, v ktorých používa opakovane rôzne logické a aritmetické operácie. Tieto operácie sú zoskupené do štyroch funkcií, F, G, H a I nasledovne:

$F(a, b, c) = (a \wedge b) \vee (\neg a \wedge c)$,
$G(a, b, c) = (a \wedge c) \vee (b \wedge \neg c)$,
$H(a, b, c) = a \oplus b \oplus c$,
$I(a, b, c) = b \oplus (a \vee \neg c)$.

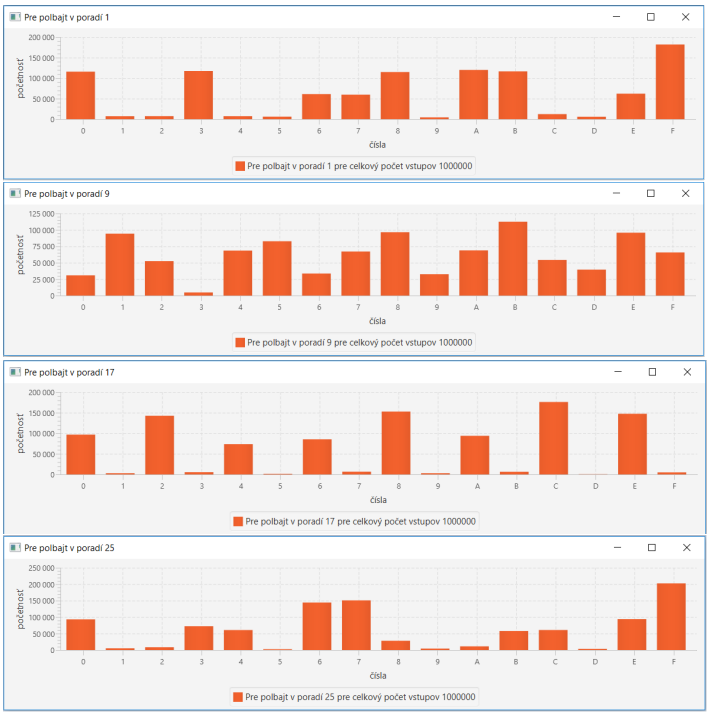
Tieto funkcie sú pre 32 bitové hodnoty a postupne sa v kolách menia hodnoty a, b, c, d; každá hodnota sa v každom kole zmení teda 4-krát. Každé kolo potom ešte používa po 16 konštánt pre operácie shift:
$s[0..15] := 7, 12, 17, 22, 7, 12, 17, 22, 7, 12, 17, 22, 7, 12, 17, 22$
$s[16..31] := 5, 9, 14, 20, 5, 9, 14, 20, 5, 9, 14, 20, 5, 9, 14, 20$
$s[32..47] := 4, 11, 16, 23, 4, 11, 16, 23, 4, 11, 16, 23, 4, 11, 16, 23$
$s[48..63] := 6, 10, 15, 21, 6, 10, 15, 21, 6, 10, 15, 21, 6, 10, 15, 21$
Po týchto kolách sa následne pripočítajú hodnoty a, b, c, d, tohto 512-bitového bloku k medzivýsledkom.
Nakoniec sa vytvorí 128 bitový výstup zo štyroch 32 bitových medzivýsledkových registrov.

Výsledky

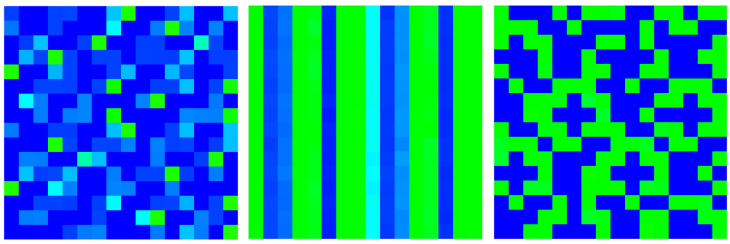
Vzhľadom k náročnosti a rozsiahlosti algoritmu, bolo ťažké vytvoriť jedinú chybu, ktorá by sama o sebe dostatočne zmenila nami pozorované vlastnosti.

1. Neinicializovanie konštánt ani **2. Iné hodnoty konštánt** nijako neovplyvnili vybrané vlastnosti, teda takúto zmenu odhaliť našim softvérom nevieme. Pri týchto 2 kategóriách sme skúšali zmeniť konštanty a, b, c, d a číslo, ktoré určovalo padding.

3. Zlé nainicializovanie poľa alebo tabuľky. Pri tomto druhu zmeny, sme zabudli nainicializovať tabuľku, ktorá určovala konštanty pre operácie shift. Výsledné vlastnosti však taktiež nepreukazovali žiadne zmeny na sledovaných vlastnostiach. Avšak ak sme takto implementovanú MD5 použili len na vstupy dĺžky 1-5, výsledky na výstupnom 1., 9., 17. a 25. znaku boli zjavne nerovnomerné. Ostatné znaky v našich testoch nepreukazovali zmeny vlastností. Na obr. 3.1 vidíme menované znaky 1, 9, 17 a 25 na výstupe. Na obr. 3.2 vidíme príklady heatmáp, 1. zobrazuje vzťah prvého znaku vstupu s prvým znakom výstupu, 2. zobrazuje štvrtý znak vstupu ku 25. znaku výstupu a posledná 9. znak vstupu ku 25. znaku výstupu.



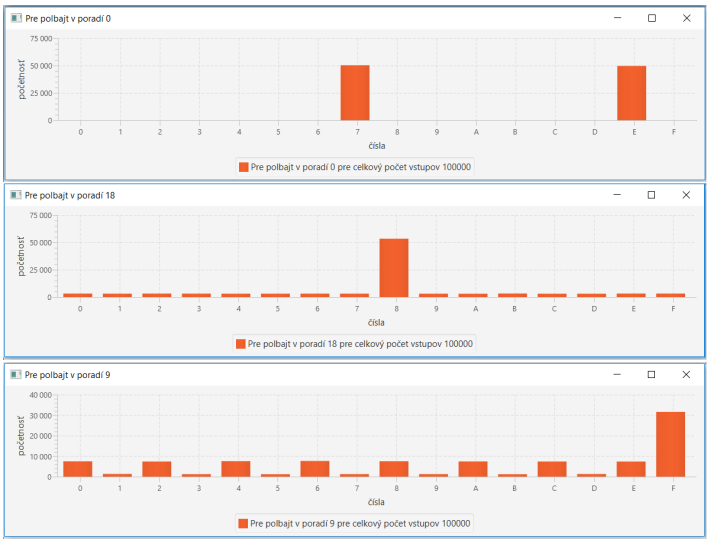
Obr. 3.1: MD5 neinicializovaná shift tabuľka, znaky 1, 9, 17 a 25.



Obr. 3.2: MD5 neinicializovaná shift tabuľka, príklady heatmáp.

4. Pokazený for-cyklus. Pokazený for cyklus v našej implementácii spôsobil, že sa namiesto 64 krokov (t.j. 4 kolá po 16 krokov) vykonal vždy len jeden krok. Pri takejto chybe potom vstupy dĺžky 1-10 dávali vždy rovnaký výsledok a to konkrétne reťazec $777777777f9fd4a0878888887431eda8$. Znaky dĺžky 10-100 už nedávali za výsledok jeden reťazec, ale znaky na výstupe boli stále ovplyvnené. Obr. 3.3 - znaky na pozícii 0 - 7 a 24 - 31 mali vždy len 2 pravdepodobné znaky výstupu (prvý obrázok, ukazuje ako príklad 0. pozíciu), znaky 8 - 23, okrem znaku 9, mali vždy na výstupe jeden vysoko pravdepodobný znak a ostatné mali veľmi malú pravdepodobnosť výskytu (druhý obrázok, ukazuje ako príklad 18. pozíciu). 9. znak výstupu môžme vidieť ako tretí z obrázkov.

Na väčších vstupoch, napr. dĺžky 200 - 600, už neboli pre túto chybu našim softvérom pozorovateľné zmeny.



Obr. 3.3: MD5 pokazený for cyklus, príklady znakov 0, 18, 9.

5. Nekorektná implementácia bežných operácií túto zmenu sme nevedeli na zvolenej implementácii demonštrovať.

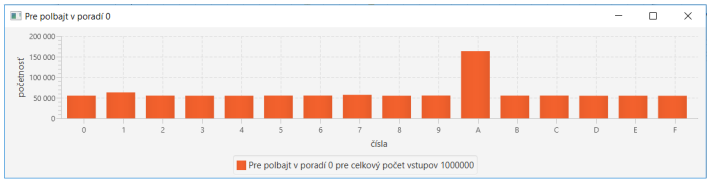
6. Iné preklepy. Ako prvú sme zvolili chybu zabudnutia pripočítania medzivýsledkov po každom kole. Toto samozrejme spôsobilo to, že výsledkom bol vždy ten istý reťazec. Ďalšou bola taká zmena, pri ktorej sa vykonalo len prvé kolo krokov, teda to s funkciou $F(a, b, c)$. Tu sme zase neboli schopní pozorovať žiadne zmeny vo výsledných vlastnostiach.

V zvolenej implementácii sme si všimli ešte 2 miesta, ktoré môžu byť nebezpečné. V oboch prípadoch je pointa rovnaká. Jeden z úsekov:

```
int j, i = 0;
for (j = 0; j < len; i++, j += 4)
    decodeBuffer[i] = (
        (buffer[j + offset] & 0xff)) |
        (((buffer[j + 1 + offset] & 0xff)) << 8) |
        (((buffer[j + 2 + offset] & 0xff)) << 16) |
        (((buffer[j + 3 + offset] & 0xff)) << 24
    );
```

Tento for cyklus pojednáva o premennej j. Okrem toho však vo svojej hlavičke inkrementuje aj premennú i. Toto sa dá pri prepisovaní ľahko nevšimnúť, nakoľko sa nejedná o štandardný zápis. Takýto for cyklus sa nachádza v dvoch funkciách. Prvou z nich je funkcia *encode*. Ak v tejto vynecháme inkrementáciu premennej i, výstupné reťazce zrazu nebudú mať 128 unikátnych bitov, ale len 4-krát zopakovaný prvý 32 bitový unikátny úsek. Táto skutočnosť uľahčí útočníkom hľadanie kolízií. Okrem toho pri použití, napr. v databáze, niekoľkonásobne spomalí celý systém, pretože budú častejšie vznikať kolízie aj pri bežných požiadavkách.

Druhou je funkcia *decode*. (Táto bola aj zobrazená v predchádzajúcej ukážke.) Vynechanie inkrementácie premennej i v nej, nespôsobí také viditeľné pokazenie ako pri prvej. Výsledné vlastnosti sú však mierne odlišné.



Obr. 3.4: MD5 chyba pri decode.

Testy boli prevedené na 1 000 000 náhodných vstupoch dĺžky 1-500 bajtoch. Obrázok 3.4 znázorňuje počet príslušných hexadecimálnych hodnôt na 0-tom výstupnom znaku. Pre všetky ostatné pozície (0-31) boli výsledky rovnaké, jeden znak sa vyskytoval 3x častejšie ako ostatné znaky. Ďalšími experimentami sme zistili, že nerovnomerné výsledky sú spôsobené tým, že každý reťazec dĺžky menej ako 56 sa pri tejto chybe v implementácií zobrazuje na výstupe ako string „ac1d1f03d08ea56eb767ab1f91773174“, string dĺžky 56-59 vrátane sa zobrazuje ako string „145b0dcd845d010449402e3132296610“ a string dĺžky 60 ako string „7b88c7635020730f39f617b0d1550151“. Väčšie dĺžky stringov nepreukazovali nami pozorovateľné zmeny vlastností.

7. Nesprávne pochopenie pseudokódu. Pri tomto druhu chyby sme skúsili pochopiť plus v krúžku ako plus a nie ako XOR. Táto zmena však nepriniesla nášmu programu pozorovateľné výsledky.

3.1.2 SHA-1

Všeobecné informácie

SHA alebo Secure Hash Algorithm je bloková hašovacia funkcia (1.1.6). Algoritmus bol vydaný v roku 1993 a bol uvedený do štandardov NIST. Neskôr bol upravený a publikovaný v roku 1995 pod menom SHA-1. Je veľmi podobnými algoritmu rodiny MD4, nakoľko bol z nej odvodený. Taktiež je nesprávne používaný ako „šifrovací“ na utajenie hesiel [20]. Prvá kolízia bola nájdená až vo februári tohto roka (t.j. 2017) [23].

Rodina SHA

Poznáme viacero algoritmov s označením SHA, napr. SHA-256, SHA-384, SHA-512. Tieto sa označujú ako SHA-2 a ich najzásadnejší rozdiel oproti SHA-1 je v tom, že výsledkom z nich sú odtlačky väčšej dĺžky, a teda poskytujú lepšiu ochranu proti útokom hrubou silou.

Okrem toho ešte poznáme algoritmy skupiny SHA-3. Sú to napr. SHA3-224, SHA3-256,

SHA3-384, SHA3-512. Tieto algoritmy už majú inú štruktúru oproti SHA-2. SHA-3 nie je pokračovaním, vylepšením ani náhradou SHA-2, v tomto smere je od nej nezávislá.

Informácie o algoritme [10]

SHA-1 vytvára 160 bitové odtlačky. Vstup spracúva po blokoch veľkosti 512 bitov. Algoritmus je veľmi podobný MD5 a môžeme ho tiež rozdeliť do 4 krokov:

1. Rovnako ako pri MD5, rozdelí vstup na 512 bitové bloky a doplní 1-512 bitov tak, aby v poslednom bloku ostalo len 448 bitov. Doplnené bity/bit majú prvý bit jednotku a ostatné nuly.

2. Do správy doplní na ostatých 64 bitov informáciu o dĺžke vstupu (bez doplnených bitov).

3. Inicializuje päť 32 bitových registrov pre ukladanie priebežných výsledkov a následne výsledku, ktorý potom poskladá do 160 bitového odtlačku. Prvé štyri hodnoty sú nainicializované rovnako ako pri MD5. a = 0x67452301; b = 0xefcdab89; c = 0x98badcfe; d = 0x10325476; e = 0xc3d2e1f0;

4. Spracováva vstup po 512 bitových blokoch. Každý blok prejde štyrmi kolami po 20 krokov. V každom kole sa aplikuje jedna zo 4 funkcií f_1, f_2, f_3, f_4:
   $f_1(a, b, c) = (a \wedge b) \vee (\neg a \wedge c)$,
   $f_2(a, b, c) = a \oplus b \oplus c$,
   $f_3(a, b, c) = (a \wedge b) \vee (a \wedge c) \vee (b \wedge c)$,
   $f_4(a, b, c) = a \oplus b \oplus c$.
   Tiež sa použije v každom kole jedna konštanta $k_1 = 0x5A827999$,
   $k_2 = 0x6ED9EBA1$, $k_3 = 0x8F1BBCDC$, $k_4 = 0xCA62C1D6$.
   Následne skombinuje výsledky s medzivýsledkami.
   Po týchto krokoch vytvorí z výsledkov v piatich 32 bitových registrov jeden 160 bitový reťazec.

Výsledky

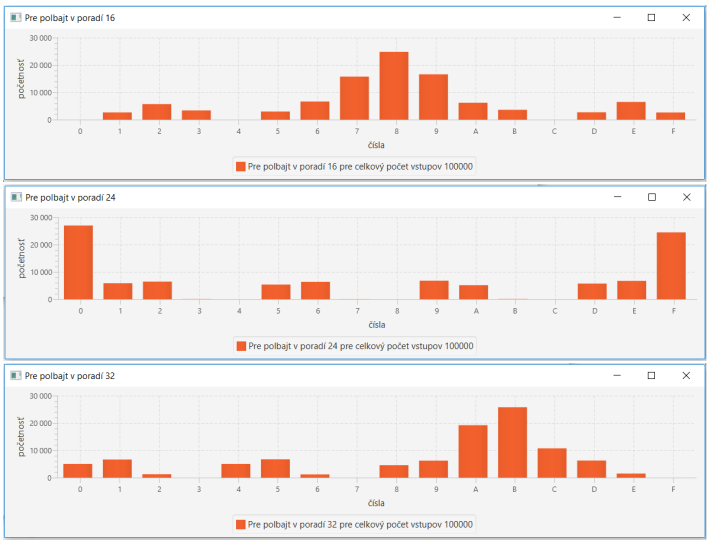
SHA-1 je na tom podobne ako MD5 aj s výsledkami. Vzhľadom ku komplexnosti kódu, nespôsobilo **1. Neinicializovanie konštanty** alebo **2. Iné hodnoty konštánt** žiaden negatívny dopad na nami pozorované vlastnosti. Nepodarilo sa nám ani vytvoriť žiadne zaujímavejšie výsledky.

3. Zlé nainicializovanie poľa alebo tabuľky. Vybraná implementáca neobsahuje tabuľky a polia, preto sme túto zmenu nemohli aplikovať.

4. Pokazený for-cyklus. Pri pokazení len prvého for cyklu sa neprejavili žiadne nami pozorovateľné zmeny vo vlastnostiach. Pri pokazení aj druhého for cyklu bol výsledkom vždy len jeden konštantný reťazec.

5. Nekorektná implementácia bežných operácií. Túto zmenu sme neaplikovali.

6. Iné preklepy. V tejto kategórii sme skúsili zmeniť operácie << za operácie <<< a naopak. Táto chyba priniesla zmeny vo vlastnostiach len na kratších vstupoch. Na obr. 3.5 vidíme výsledné haše na vstupoch dĺžky 1-5. Ovplyvnené boli len pozície znakov 16, 24 a 32, ktoré sú aj v tomto poradí vyzobrazené. Ostatné znaky vyzerali pre náš softvér náhodne. Tak isto aj pre vstupy väčších dĺžok tieto odlišné vlastnosti postupne zanikali.

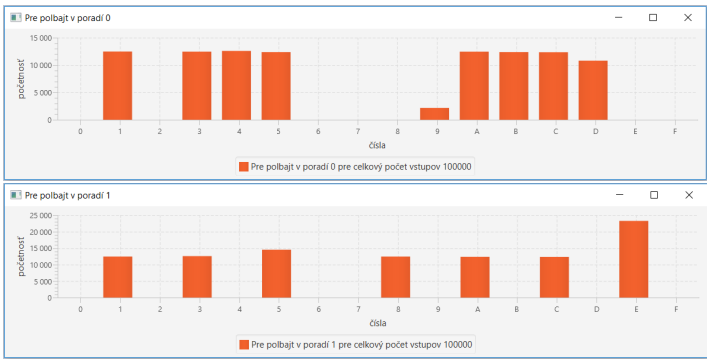


Obr. 3.5: SHA-1 výmena << za <<< a naopak na pozíciách 16, 24 a 32.

7. Nesprávne pochopenie pseudokódu. Pri zmene XORu na plus a *not* na − sme nespozorovali žiadne odlišné vlastnosti.

8. Úmyselné zmeny. Pri nahradení operácie *and* operáciou *or* vo funkcií plniacej pole, sme dosiahli to, že sa všetky vstupy zahašovali na reťazce zložené zo samých znakov F.

Naopak, keď sme nahradili všetky operácie *or* operáciou *and*, algoritmus vygeneroval hashe, ktoré mali totožných posledných 96 bitov (t.j. 24 znakov). Menilo sa len prvých 8 bajtov (t.j. 16 znakov). Aj tieto sa však menili nerovnomerne. (Obr. 3.6) Z prvého obrázka vidíme, že napr. hexadecimálne číslice 0, 2 či F neuvidíme nikdy ako prvý znak výstupného hashu. Na druhom obrázku vidíme, ktoré nikdy nebudú ako druhý znak. Každý znak z týchto prvých 16 mal nejaké nezobraziteľné znaky. Záujemcovia, ktorí chcú vidieť aj tie, si ich môžu pozrieť v programe, ktorý je ako príloha na CD.



Obr. 3.6: SHA-1 bez operácií OR.

3.1.3 RC4

Všeobecné informácie

RC4 je prúdová (1.1.5) symetrická (1.1.1) funkcia, ktorá bola zostrojená v roku 1987. Je to prúdová šifra, teda správa sa tak, že XORuje vstup s nejakým „náhodne“ vygenerovaným reťazcom. Tiež je vďaka tomu pomerne rýchla. Algoritmus RC4 bol utajený po dobu siedmich rokov a v roku 1994 bol „anonymne“ zverejnený [9].

Používa sa napr. v protokoloch WEP, WPA, SSL a TLS. Aj Skype používa **modifikovanú** verziu RC4 [25]. Nakoľko je algoritmus veľmi rýchly a jednoduchý, je aj značne zraniteľný.

Rodina RC

Nakoľko je algoritmus RC4 veľmi zraniteľný, poznáme množstvo jeho pokusov o vylepšenie. Existujú napr. algoritmy RC4A, VMPC, $RC4^{+}$, Spritz,.. .

Informácie o algoritme [10]

Algoritmus RC4 spracováva vstup po 8 bitoch (teda 1 bajte). Obsahuje vstupný kľúč ľubovoľnej dĺžky od jedného po 256 bajtov. Podľa tohto kľúča robí následne permutácie pola. Je celkom jednoduchý a celý postup môžeme zhrnúť do 4 krokov:

1. Inicializuje bloky S a T o dĺžke 256 bajtov. Do S inicializuje hodnoty „i“ od 0 po 255 v postupnom poradí, čiže $S_i = i$. Do T vloží kľúč podľa potreby opakovane (ak je kratší ako 256 bajtov, vloží ho niekoľkokrát za sebou).

2. Permutuje pole S tak, že vymieňa S_i s S_j pre i = 0..255 a $j = (j + S_i + T_i)$ mod (256).

3. Opäť permutuje pole S pre $n = 0..$*(dĺžka správy)*, pričom teraz už počíta v každom kroku z pola aj výsledné čísla, ktoré použije na zašifrovanie v ďalšom kroku. Pre

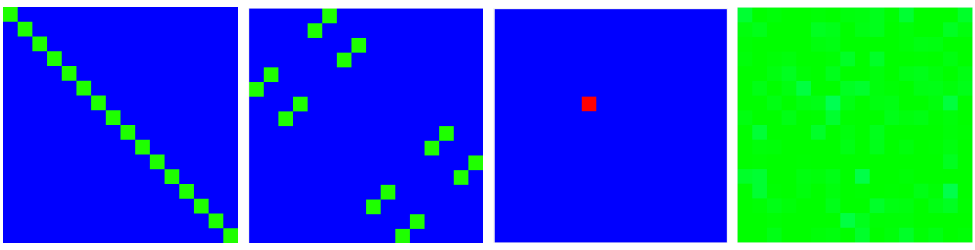
každé n teda robí: $i = (i + 1) \bmod (256)$, $j = (j + S_i) \bmod (256)$, vymení S_i a S_j, vypočíta finálny index, $fi = (S_i + S_j) \bmod (256)$ a číslo z poľa S na tomto indexe vyberie ako bajt „šifrovacieho kľúča“ (tento pojem sa v iných literatúrach takto nepoužíva, my ho takto v našej práci budeme označovať). Takto vytvorí „šifrovací kľúč“, ktorý teda môže obsahovať čísla z rozsahu 0 až 255.

4. Až v tomto kroku ideme robiť niečo so vstupom. Na vstupnú správu teraz použije XOR s vytvoreným „šifrovacím kľúčom“.

RC4 je symetrická šifra, teda na dešifrovanie stačí aplikovať na zašifrovaný text opäť rovnaký kľúč.

Výsledky

Nakoľko hlavná myšlienka šifry vytvorenej pomocou RC4 je, že sa vytvorí kľúč, ktorým sa vstup len XOR-uje, je veľmi dôležité nepoužívať rovnaký kľúč viackrát. Pri rovnakom kľúči sa totiž vždy rovnaký znak na rovnakom mieste zo vstupu zašifruje na rovnaký znak na výstupe. (Obr. 3.7) 2 obrázky zľava sú príklady náhodných vstupov pri konštantnom kľúči na všetkých vstupoch. Korelácia bitov vstupu a výstupu je zjavná pre x-tý znak vstupu a x-tý znak výstupu. Posledný obrázok je rovnaký, len pri x-tom znaku vstupu a y-tom znaku výstupu, pričom x != y. Vidíme teda, že takéto štvorbajtia nemajú na seba vplyv. Tretí obrázok znázorňuje RC4 tiež pri použití jedného a toho istého kľúča pre všetky vstupy, avšak vstupy neboli náhodné, ale boli to opakovania jedného písmena. Teda vidíme, že pri rovnakom kľúči sa rovnaký znak na rovnakej pozícii zašifruje vždy rovnako. Táto skutočnosť je aj dôvodom, že pri RC4 sme ďalej neskúmali výsledné zašifrované reťazce na vygenerovaných vstupoch, ale vstupy sme použili ako kľúče a reťazce, ktoré sme pomocou nich zašifrovali boli reťazce zložené zo znakov s ASCII kódom 00, teda znakov nulls, aby sme videli distribúciu kľúča, ktorá je pri RC4 podstatná.



Obr. 3.7: RC4 pri konštantnom kľúči.

RC4 neobsahuje konštanty, preto sme nemohli aplikovať zmeny **1. Neinicializovanie konštanty** a **2. Iné hodnoty konštánt**.

3. Zlé nainicializovanie poľa alebo tabuľky. Ak zabudneme na začiatku inicializovať pole, výsledkom budú pochopiteľne samé 0.

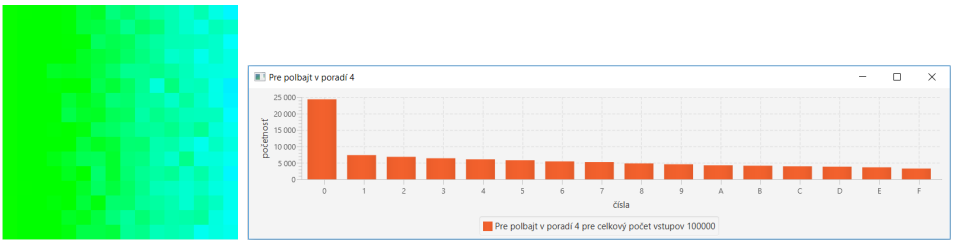
4. Pokazený for-cyklus. Táto chyba spôsobí výrazné zmeny. V našej implementácii má RC4 2 for cykly. Ak sa jej dopustíme vo for-cykle pri kódovaní, výsledný zašifrovaný reťazec bude mať unikátny len 1 bajt, ostatné hodnoty sú nuly. Ak vynecháme zátvorky for-cyklu pri inicializácií, budú niektoré znaky výstupu jednoznačne určené, ostatné pozície budú mať výrazne väčšiu pravdepodobnosť výskytu niektorého znaku. (obr. 3.8). Záujemcovia si jednotlivé výskyty znakov na ostatných pozíciách môžu pozreť v programe na priloženom CD. My uvádzame ako príklad pozíciu 63, kde môžme vidieť neschopnosť vygenerovania znakov A, D a E, ako aj veľmi malú pravdepodobnosť vygenerovania ostatných znakov, okrem znaku 5, ktorý má naopak veľmi vysokú pravdepodobnosť výskytu na tejto pozícii. Ostatné pozície majú podobné grafy ako táto.

Obr. 3.8: RC4 nekorektná inicializácia.

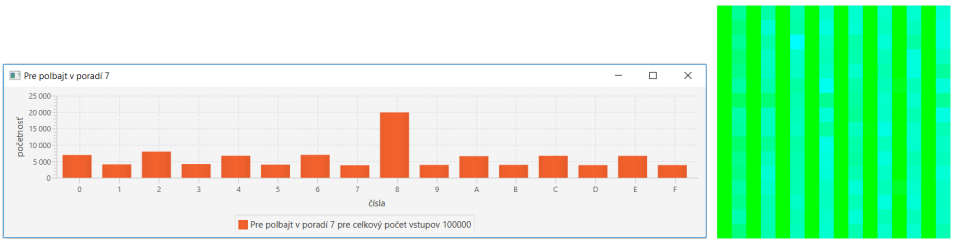
5. Nekorektná implementácia bežných operácií. V RC4 sa používa aj operácia výmeny dvoch premenných, swap. Táto sa dá nepozornosťou ľahko naimplementovať nesprávne. My sme vyskúšali dve jej nesprávne verzie. Uvedieme si len tú so zaujímavejšími výsledkami. Druhá je podobná.

```
private void swap(int i, int j, int[] sbox) {
        int temp = sbox[i];
        sbox[j] = sbox[i];
        sbox[i] = temp;
}
```

Vlastnosti výsledných zašifrovaných reťazcov po takto vnesenej chybe boli viditeľne rozdielne od vlastností neupravovanej RC4. Hodnoty znakov na párnych pozíciách (t.j. prvé štyri bity z každého bajtu) (obr. 3.9) a hodnoty znakov na nepárnych pozíciách (obr. 3.10) boli po niekoľkonásobnom opakovaní zakaždým na nových, na sebe nezávislých 100000 vstupoch vždy podobne nerovnomerne rozmiestnené.



Obr. 3.9: RC4 nekorektný swap, v poradí párny znak.



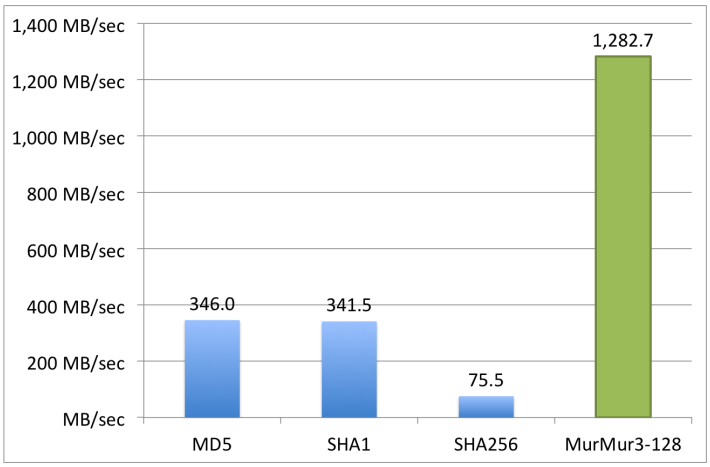
Obr. 3.10: RC4 nekorektný swap, v poradí nepárny znak.

3.2 Nekryptografické

3.2.1 MurmurHash

Všeobecné informácie

Murmur hash je nekryptografická hašovacia funkcia (1.1.6). Vytvorená bola v roku 2008. Vytvorenie kolízie je príliš jednoduché, preto nie je využiteľná na kryptografické účely. Jej výhodou však je jednoduchosť, rýchlosť a vysoká odolnosť voči kolíziám (na náhodných vstupoch), ako aj fakt, že pre 4-bajtové kľúče nie je možné vytvoriť kolíziu. Austin Appleby, tvorca murmurhashu sám povedal na základe meraní a experimentov: [1]: „Murmur is close enough to random to be suitable for virtually all keysets, though it does have the interesting property that it produces 0 collisions if the keyset is all 4-byte integers from 0 to $2^{32} - 1$ - this is a side effect of it doing mixes on the full 4-byte hash state using only reversible operations.“ Jej rýchlosť dokazuje fakt, že je 37% rýchlejšia ako SuperFastHash a lookup3 a 307% rýchlejšia ako FNV (potiaľto čerpané z [1]). A tiež to dokazujú porovnania rýchlosti Murmur3 s funkciami SHA-1 a MD5 naimplementovanými v jazyku C# (obr. 3.11). [7]



Obr. 3.11: Porovnanie rýchlosti MD5, SHA1 a Murmur hash.

Meno tejto funkcie vzniklo z jednoduchej postupnosti operácií, ktoré používa -

MUltiply - násobenie a **R**otate - rotáciu. Opakovaním tejto postupnosti cca 15-krát využitím správnych hodnôt na násobenie a rotáciu, je výsledok pekne pseudo-náhodný. Pozn.: V skutočnosti autor využíva aj operáciu XOR, ale ako sám povedal, meno Murmur znie lepšie ako Musxmusx [2].

Okrem Murmur1 poznáme 2 varianty tejto hašovacej funkcie a to Murmur2 a Murmur3. Pri ich implementáciách záleží hlavne na architektúre, na ktorej ju chcem implementovať. Murmur3 pri dĺžke kľúča 128 bitov produkuje rozdielne hodnoty hašov pri x86 a x64 verzii. Murmur2 má zase 2 verzie: Murmur64A - pre 64 bitovú architektúru a Murmur64B pre 32 bitovú architektúru.

Vďaka jej vlastnostiam je jej použitie vskutku užitočné pre haš tabuľky, pre hľadanie a porovnávanie rovnakých súborov - teda na overenie integrity aj pre napr. emaily.

Informácie o algoritme

Algoritmus Murmur 2.0 je veľmi jednoduchý a používa len operácie sčítania/odčítania, násobenia, rotácie a operáciu XOR pri vhodne zvolených konštantách. Bližšie sa nedá tak ľahko popísať, preto uvádzame jeho kód. Kód je v 64 bit verzii jazyka C.

```
uint64_t MurmurHash2 (const void * key, int len, uint64_t seed) {
    const uint64_t m = (0xc6a4a793 << 32) | 0x5bd1e995;
    const int r = 24;
    uint64_t h = seed ^ len;
    const unsigned char * data = (const unsigned char *)key;
    while(len >= 8)  {
        uint64_t k = *(uint64_t*)data;
        k *= m;
        k ^= k >> r;
        k *= m;
        h *= m;
        h ^= k;
        data += 8;
        len -= 8;
    }

    switch(len)  {
    case 7: h ^= data[6] << 48;
    case 6: h ^= data[5] << 40;
    case 5: h ^= data[4] << 32;
    case 4: h ^= data[3] << 24;
    case 3: h ^= data[2] << 16;
```

```
    case 2: h ^= data[1] << 8;
    case 1: h ^= data[0];
        h *= m;
    };

    h ^= h >> 13;
    h *= m;
    h ^= h >> 15;
    return h;
}
```

Výsledky

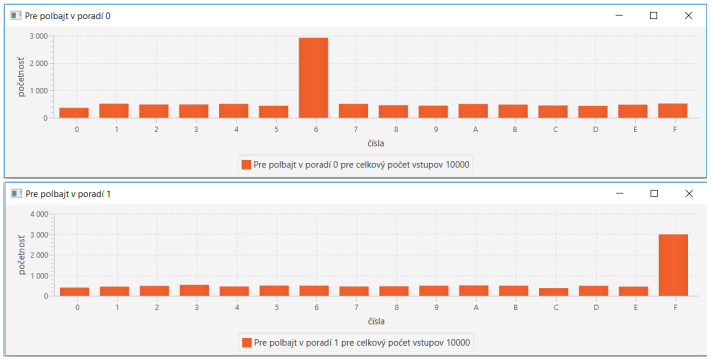
1. Neinicializovanie konštánt. Algoritmus obsahuje 3 konštanty, m, r a h. Pri zabudnutí inicializácie prvej z nich, sú výsledkom samé nuly. Pri zabudnutí priradenia hodnoty premenným r alebo h samostatne sme nezistili žiadne výrazné zmeny vlastností. Avšak ak sme priradili hodnotu 0 aj r aj h súčasne, niektoré výstupy obsahovali samé 0, niektoré vyzerali náhodne. Zhruba štvrtina vstupov sa zahešovala na samé 0. Môžme to vidieť aj na obrázku 3.12, ktorý zobrazuje 0. znak výstupu. Veľmi podobne na tom sú aj ostatné pozície výstupu.

Obr. 3.12: Murmur neinicializovane r a h, pozícia 0.

2. Iné hodnoty konštánt. Zmena hodnoty m na 1 nepreukazovala žiadne nami pozorovateľné zmeny vo vlastnostiach. Zmena konštánt r ani h tiež nepreukazovali zmeny.

3. Zlé nainicializovanie poľa alebo tabuľky. Algoritmus neobsahuje svoje polia a tabuľky, preto sme túto chybu nemohli implementovať.

4. Pokazený for-cyklus. Táto chyba spôsobila zmeny vo výsledkoch len na vstupoch dĺžky 1-5, kde vždy jeden znak mal väčšiu pravdepodobnosť vyskytnutia sa ako ostatné. Obr. 3.13 zobrazuje príklady výstupných znakov na pozícii 0 a 1.



Obr. 3.13: Murmur pokazený for cyklus na krátkych vstupoch, pozície 0 a 1.

Ďalšími pozorovaniami sme zistili, že je to takto preto, lebo vstupy dĺžky 4 sa zahešovali vždy na jeden a ten istý reťazec $6f4778db$.

5. Nekorektná implementácia bežných operácií. Tento druh zmeny sme do algortimu vniesť nemohli.

6. Iné preklepy. V tejto kategórii sme navrhli len zámenu >> za >>> a naopak. Táto zmena spôsobila len to, že ako 0. znak výstupu mohli byť len znaky 0-7. Ostatné pozície zostali neovplyvnené z pohľadu našich pozorovaných vlastností.

7. Nesprávne pochopenie pseudokódu. Ako vnesenú chybu sme opäť použili zámenu symbolu plus za XOR. Táto zmena nám však nepriniesla žiadne zmeny v pozorovaných vlastnostiach.

8. Úmyselné zmeny. Tu sme vyskúšali zámenu operácie AND za operáciu OR. Táto spôsobila zmeny vo výsledných vlastnostiach opäť len pre krátke vstupy. Toto spôsobil fakt, že väčšina reťazcov dĺžky 4 (ale nie všetky) sa zahešovala na reťazec $14acb3da$.

3.2.2 Fowler-Noll-Vo

Všeobecné informácie

FNV je nekryptografická hašovacia funkcia. Základná myšlienka FNV algoritmu bola vzatá z komentárov ľudí poslaných ako review na IEEE POSIX P1003.2 v roku 1991. Komentáre boli poslané Glenn Fowlerom a Phong Voom. Algoritmus vzal a upravil Landon Curt Noll. Meno FNV teda vzniklo z priezvisk ich tvorcov. Dnes má algoritmus celkom rozsiahle použitie [14].

FNV algoritmus je navrhnutý tak, aby bol rýchly a zároveň mal nízku mieru kolízií na náhodných vstupoch. Vďaka vysokému rozptylu hashov je vhodný pre podobné/takmer zhodné reťazce (ako sú napr. URL adresy, IP adresy,..). Nie je vhodný na kryptografické aplikácie hlavne kvôli týmto trom vlastnostiam [5]:

1. „Sticky state“ - kryptografická funkcia by nemala mať stav, v ktorom sa pre „normálne vyzerajúci“ vstup zasekne. Ak je FNV hash v priebehu výpočtu zrazu

nula a na vstupe nasleduje sekvencia núl, hash zostane nulový, až pokým nepríde nenulový vstup a výsledný hash nie je ovplyvnený počtom týchto núl na vstupe.

2. „Diffusion“ - v kryptografickej funkcii by mal každý bit zo vstupu nejako ovplyvniť/prejaviť sa vo výstupe. Pri FNV tomu tak nie je, nakoľko najmenej významný bit hashu je XOR najmenej významných bitov z každého bajtu zo vstupu, pričom nezáleží na ostatných bitoch vstupu. Takúto podobnú nevýhodu majú aj bity dva až sedem. Len najvrchnejší bit zavisí na všetkých bitoch vstupu.

3. „Work factor“ - kryptografická funkcia by mala byť na výpočet drahá a náročnejšia, aby spomalila bruteforce útok (útok s postupným skúšaním všetkých možností). FNV je však veľmi lacná pre výpočet procesorom.

Ak však je FNV vo svojej pôvodnej podobe použité pre hash tabuľky, na serveroch, databázach a podobne, útočníci vedia ľahko posielať požiadavky, aby vytvorili kolízie a tým spôsobili spomalenie aplikácie. Preto aj samotní autori odporúčajú spraviť v algoritme malé úpravy. [5]

Informácie o algoritme

Funkcia FNV je veľmi krátka a jednoduchá. Preto uvádzame celý jej pseudokód:

```
hash = offset
for each octet_of_data
        hash = hash xor octet_of_data
        \begin{lstlisting}
        hash = hash * FNV_Prvočíslo0 \bmod 2^{x}
        \end{lstlisting}
return hash
```

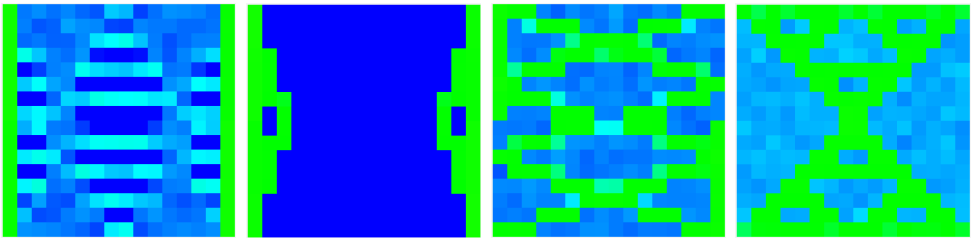
Hodnoty konštánt Offset a Prvočíslo závisia od dĺžky hashu. Číslo x, ktorým modulujeme výsledok závisí od verzie FNV a použitej architektúry, môže mať hodnotu napr. 64. Tento pseudokód znázorňuje algoritmus funkcie FNV-1a. Algoritmus FNV-1 má rovnaké inštrukcie a konštanty, len má vymenené poradie operácií. Algoritmus FNV-0 je rovnaký ako FNV-1, len má inú konštantu Offset, a to nula. FNV-0 algoritmus sa používa na výpočet Offsetu pre FNV-1a. [5]

Výsledky

Skúmali sme verziu FNV-1a 64 bit.

1. Neinicializovanie konštanty. FNV algoritmus je veľmi krátky, s malým počtom inštrukcií, a preto aj malé chyby prinášajú fatálne následky. Na začiatku sa inicializuje offset a prvočíslo. Ak zabudneme inicializovať prvočíslo, a teda bude mať

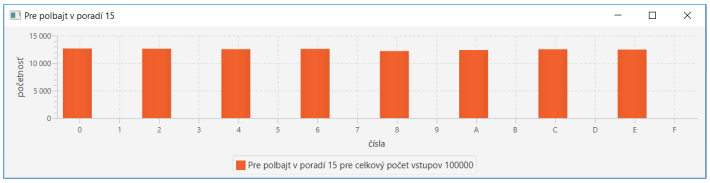
hodnotu 0, všetky hašové výstupy budú tiež 0. Pri zabudnutí inicializácie offsetu sme našimi testami žiadne anomálie výstupu na rôzne dlhých a náhodne vygenerovaných reťazcoch nezaznamenali. Avšak na vstupoch opakujúceho sa znaku null, boli všetky výstupy 0. Podobne zle na tom boli náhodné vstupy reťazcov dĺžky 1-3. Tieto vykazovali značné korelácie znakov medzi vstupom a výstupom takmer každého znaku s každým (Obr. 3.14). Na prvom obrázku vidíme 0. vstupný znak ku 0. výstupnému, na druhom 0. vstupný znak k 7. výstupnému, na treťom 0. vstupný a 10. výstupný, posledný obrázok znázorňuje 2. vstupný znak a 4. výstupný.



Obr. 3.14: FNV na vstupoch dĺžky 1-3.

Toto ale nie je nič zvláštne, pretože FNV-1a vykazuje podobné korelácie znakov vstupu a výstupu pre takéto krátke vstupy aj pri korektnej implementácii.

2. Iné hodnoty konštánt. Nahradenie prvočísla párnym číslom spôsobí absenciu nepárnych čísel na konci výstupu, teda posledný znak výstupu môže byť len párny (obr. 3.15). Iné odchýlky sme nespozorovali.



Obr. 3.15: FNV párne číslo namiesto prvočísla, posledný znak.

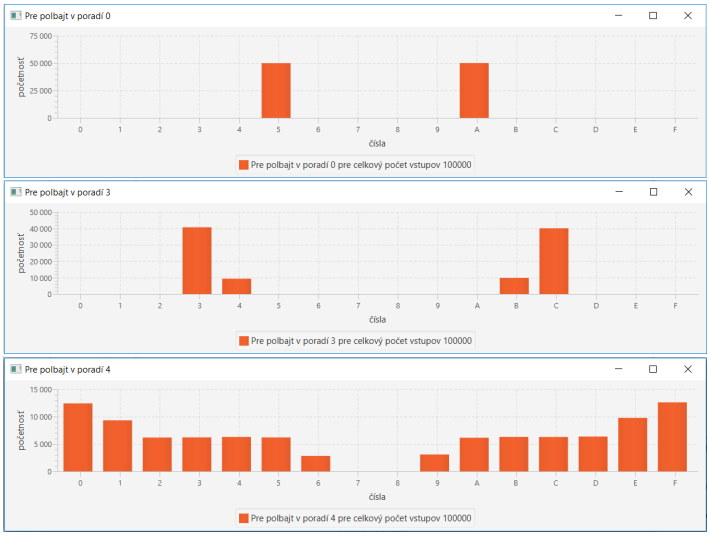
3. Zlé nainicializovanie poľa alebo tabuľky. Implementácia neobsahuje polia a tabuľky.

4. Pokazený for-cyklus. Ak sa spravíme chybu v cykle typu zabudnuté zátvorky {} alebo preklikneme hneď za for cyklus bodkočiarku,

```
int i=0;
for(i = 0; i < len; i++)
        rv ^= k[i];
        rv *= FNV_Prvočíslo;
```

zanechá to výrazné zmeny na výsledných vlastnostiach. Na obrázkoch (3.16) môžeme pozorovať vybraté anomálie správania. Hodnota 0.tého znaku môže byť len 5 alebo A.

Pravdepodobnosti výskytu mnohých ďalších znakov na svojich pozíciách sú tiež značne poškodené.



Obr. 3.16: FNV nekorektný for cyklus.

Zmeny v kategóriách **5. Nekorektná implementácia bežných operácií** a **6. Iné preklepy** sme neimplementovali.

7. Nesprávne pochopenie pseudokódu. Vniesli sme zmenu operácie XOR operáciou plus. Táto zmena opäť nespôsobila žiadne pozorovateľné zmeny vo vlastnostiach.

3.2.3 Pearson hashing

Všeobecné informácie

Pearson hashing je hašovacia funkcia (1.1.6). Je tiež veľmi jednoduchá a nepoužíva žiadne rozsiahlejšie inštrukcie, na ktoré by bolo treba špeciálny hardvér. Autor uviedol hlavné výhody a nevýhody tohoto hašovania. Výhody:

- Nie je žiadne obmedzenie na dĺžke textu.
- Dĺžku textu nie je nutné poznať dopredu.
- Na každom znaku sa vykonáva len málo aritmetiky.
- Je nepravdepodobné, že podobné reťazce budú vytvárať kolízie.
- Perfektné hašovacie funkcie[1] a minimálne perfektné hašovacie funkcie[2] môžu byť postavené na tejto konštrukcii.

Nevýhody:

- Je náročné poskytnúť výstupné hodnoty, ktoré nie sú mocninou 2.

- Treba umiestniť do pamäte pomocnú tabuľku s veľkosťou 256 bajtov.

[15]

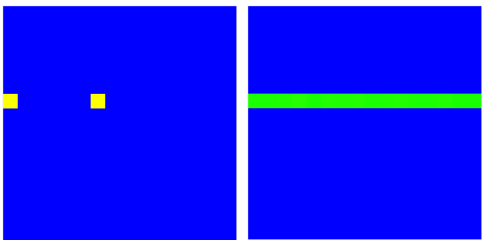
Informácie o algoritme

Na začiatku si algoritmus uloží do pamäte tabuľku s 256 bajtami, v ktorej sú čísla od 0 po 255, každé práve raz, v hocijakom/náhodnom usporiadaní. Podľa tejto tabuľky potom postupne nezávisle po bajtoch vypočítava celú 8 bajtovú odpoveď - hash, a to takým spôsobom, že vyberá náhodný index ako prechádza tabuľkou a vyberie toto posledné číslo ako jeden výstupný bajt.

Výsledky

Pearson hashing algoritmus neobsahuje žiadne konštanty, preto preskočíme zmeny **1. Neinicializovanie konštanty** a **2. Iné hodnoty konštánt**.

3. Zlé nainicializovanie poľa alebo tabuľky. Pri nechaní všetkých hodnôt tabuľky 0, bude celý výstup tiež nulový. Ak sme nechali pole lineárne, teda *pole[i]* obsahuje hodnotu i pre $i=0..255$, zmeny vlastností na náhodných dlhých vstupoch sme nespozorovali. Čo však bolo badateľné, bolo použitie takto upraveného algoritmu na vstupy zložené z jedného opakujúceho sa písmena. Na vstupoch rôznej dĺžky zložených len z písmena „a“ Pearsonov hash s lineárnym polom vyprodukoval rozdielne výsledky ako ten, ktorý mal pole náhodne usporiadné (pozn. tu náhodne != lineárne). Zatiaľ čo nelineárne pole výsledky rovnomerne rozhádzalo, lineárne znaky na danej pozícii zobrazilo vždy len do 2 konštánt. (obr. 3.17)



Obr. 3.17: Pearson hash - porovnanie lineárneho pola (vľavo) a nelineárneho na tom istom vstupe zloženého z opakovania jedného písmena.

4. Pokazený for-cyklus. Pokazenie for cyklu spôsobilo, že na výstupe bol jedinečný len posledný bajt. Ostatné pozície boli zložené len z 0.

[1] Perfektné hašovacie funkcie sú injektívne - pre každý vstup dajú iný výstup a netvoria teda žiadne kolízie [8]

[2] Minimálne perfektné hašovacie funkcie sú tiež injektívne a ak má hašovacia funkcia N možných výstupov, tak pre N rôznych vstupov použije všetkých N možných výstupov [8]

5. Nekorektná implementácia bežných operácií. Ako ďalšie sme znázornili nekorektné rozhádzanie pola. Toto rozhadzovalo pole pomocou funkcie:

```
for (int i=0;i<n;i++){
        x = (int) (Math.random()*256);
        y = (int) (Math.random()*256);
        T[x]=T[y];
        T[y]=T[x];
    }
```

Výsledky takejto funkcie sú rôzne podľa premennej n a podľa toho ako veľmi sa dokáže pokaziť pole T. Ak by sa dokázalo pokaziť až tak, že by obsahoval jednu konštantnú hodnotu 255-krát, tak výstup bude len jeden a ten istý pre ľubovoľný vstup. Postupne s počtom rôznych čísel v tabuľke sa zvyšuje aj počet možných rôznych výstupov.

6. Iné preklepy. Implementovali sme preklep v tele for cykla, kde sa počíta výsledok z pola $x[i]$, i sa postupne inkrementuje vo for cykle. Preklep je taký, že sa namiesto $x[i]$ počíta vždy s $x[0]$. Táto chyba však nepriniesla nami pozorovateľnú zmenu vlastností.

7. Nesprávne pochopenie pseudokódu. Opäť sme implementovali zámenu plus za XOR. Táto chyba tiež nepriniesla zmenu v pozorovaných vlastnostiach.

Kapitola 4

Generátory náhodných čísel

V tejto kapitole si povieme niečo o náhodnosti, pseudo náhodnosti, skutočnej náhodnosti či hardvérovom generovaní náhodných čísel.

Ako sme mohli pocítiť už od prvej kapitoly s definíciami, generovanie náhodnosti ovplyvňuje bezpečnosť kryptografických algoritmov veľmi výrazne. Generovanie náhodných čísel je mimoriadne náročné. Pri tomto postupe sa snažíme generovať nejaké čísla, ktoré budú mať čo najviac vlastností náhodných čísel. Nakoľko sa tvorenie náhodných čísel často zakladá len na deterministických (predpovedateľných) výpočtoch, nemôžeme hovoriť o skutočnej náhodnosti. Takéto čísla voláme pseudonáhodné. Generované číslo býva obvykle reálne číslo z intervalu 0 až 1. Čísla iných intervalov sa dajú z nich ľahko odvodiť. [24]

Okrem deterministických algoritmov poznáme aj nedeterministické. Tieto sa označujú aj ako skutočne náhodné a používajú skratku TRNG (True Random Number Generator). Týmto generátorom postupne s časom rastie aj miera entropie. Sú však značne pomalšie, nakoľko na generovanie postupnosti náhodných čísel využívajú fyzikálne javy ako napríklad polčas rozpadu rádioaktívnych látok[1], šum polovodičových látok, termálny šum a podobne. Tieto javy nám však nedokážu vygenerovať dostatočné množstvo čísiel, „kedykoľvek si zmyslíme“. Deterministické algoritmy majú však konečný počet stavov, to znamená, že vygenerovaný reťazec náhodných čísel je periodický, teda bude sa opakovať po určitom počte vygenerovaných čísel. A naviac, už aj prvý reťazec sa dá predpočítať, lebo takýto algoritmus môžme opísať vnútorným stavom generátora a deterministickou funkciou.

Vygenerovaná postupnosť náhodných čísel by mala mať dve základné vlastnosti, a to **náhodnosť** a **nepredpovedateľnosť priebehu**. Nepredpovedateľnosť znamená, že z jedného alebo viacerých po sebe idúcich čísiel z postupnosti nevieme predpovedať nasledujúce číslo. Náhodnosť môžme definovať pomocou 2 vlastností, *rovnomernosť*

[1]Dva atómy aj toho istého prvku sa môžu rozpadať úplne iný čas. Jednému to môže trvať pár sekúnd, zatiaľ čo druhému niekoľko stoviek rokov.

výstupu - teda počet výskytov by mal byť pre každé číslo rovnaký a *nezávislosť* - čísla v postupnosti nie sú medzi sebou nijako závislé. Zatiaľ čo rovnomernosť výstupu sa dá jednoducho overiť, nezávislosť členov postupnosti je náročná. [10]

Hardvérový generátor náhodných čisiel

Okrem softvérových generátorov pseudo-náhodných čísiel, môžme použiť aj hardvérový. Tento sa považuje za True RNG. Je to zariadanie, ktoré sa pripojí do počítača a následne generuje skutočne náhodnú sériu bitov, ktoré sú štatisticky nezávislé. Vyprodukované bity sa potom už ľahko dajú poskladať do rôznych typov premenných. Hardvér používa zväčša takú metódu, že zosilňuje šum rezistora alebo polovodičovej diódy a tento šum ďalej posiela na komparátor. [4]

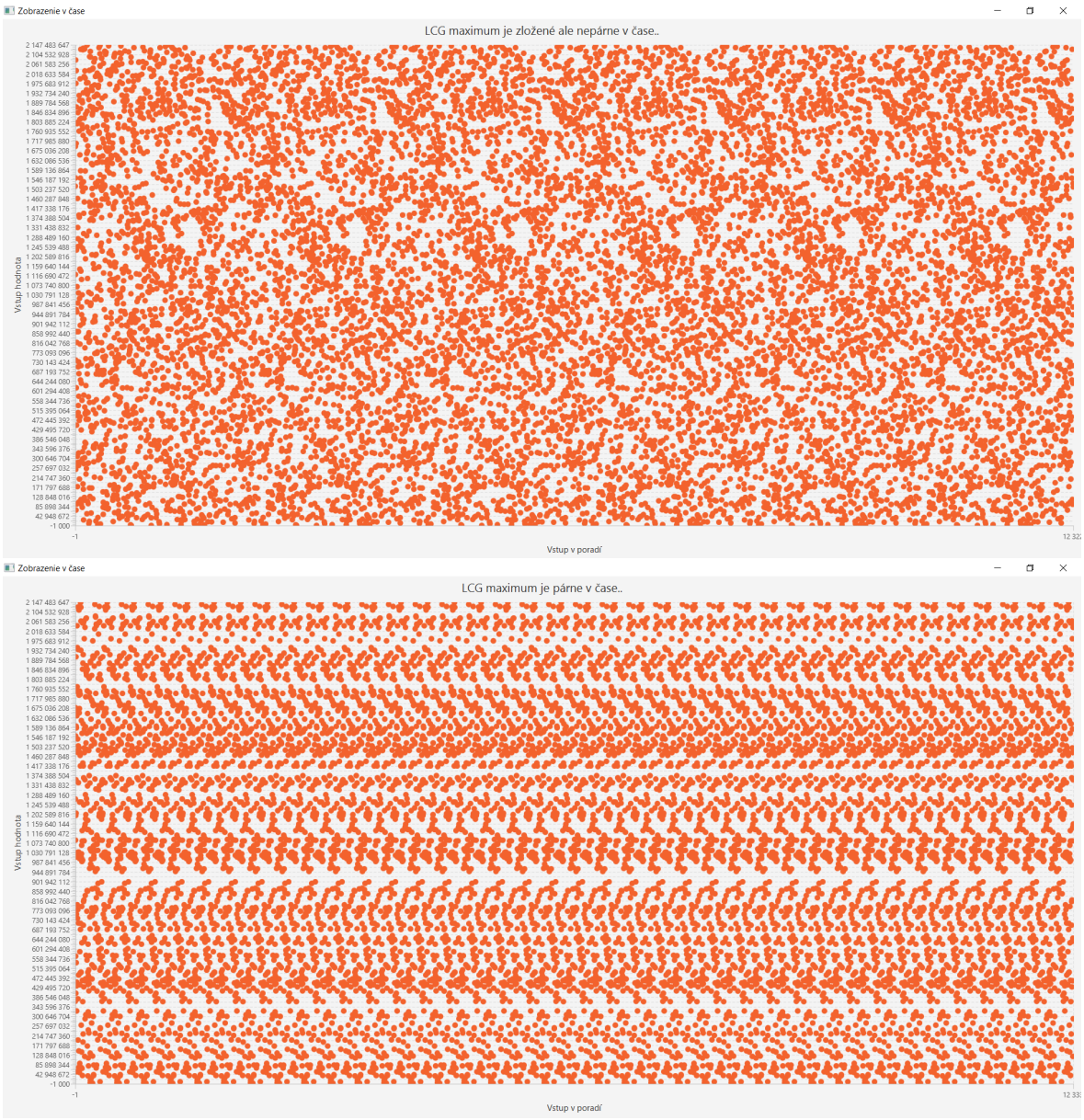
4.1 Lineárny kongurentný generátor

Generátory, ktoré pracujú na báze lineárnej kongurencie patria medzi najjednoduchšie a najznámejšie. Tieto generátory generujú pseudonáhodné čísla podľa vzťahu:
$x_{n+1} = (a \cdot x_n + b) \bmod (m)$. Premenné a, b, m sú konštanty, číslo m by malo byť veľmi veľké, maximálne možné, typicky sa rovná $2^{32} - 1$, pretože sa často používa 32 bitová aritmetika. Ak by b bolo nula, potom je vhodné zvoliť a tak, aby perióda mala dĺžku $m - 1$, pretože je dôležité, aby postupnosť mohla obsahovať všetky čísla z intervalu 0 až $m - 1$. Kvalita takéhoto generátora závisí na zvolení konštánt a, b, a m. Postupnosť potom závisí na voľbe prvého x_0. Avšak pri takomto druhu generátora náhodných čísel stačí odchytiť 4 po sebe idúce čísla a dostaneme 3 rovnice s tromi neznámymi. Útočník si následne vie predpočítať celú nasledujúcu postupnosť pseudonáhodnej postupnosti. Prípadne ak pozná a, b aj m, stačí mu jedna hodnota a vie si ďalej vypočítať celú postupnosť. Tieto generátory preto majú veľmi malú kryptografickú bezpečnosť. [10]

Výsledky

LCG pri seede 0, generuje všetky čísla nulové.

Vo svojom algoritme má LCG určené svoje maximum, ktoré môže vygenerovať. Jeho hodnota je $2^{31} - 1$. Nie je náhoda, že toto číslo je prvočíslo. Po zmenení maxima na nejaké zložené číslo, môžeme pozorovať vzory v zobrazení čísel v lineárnom čase. Ak bolo číslo nepárne aj ak bolo párne (obr.4.1). Ak bolo párne, môžeme si dokonca všimnúť aj neschopnosť vygenerovania niektorých čísel už na prvý zbežný pohľad. Takže toto nie je tá správna cesta ako meniť maximum intervalu, z ktorého chceme generovať čísla.

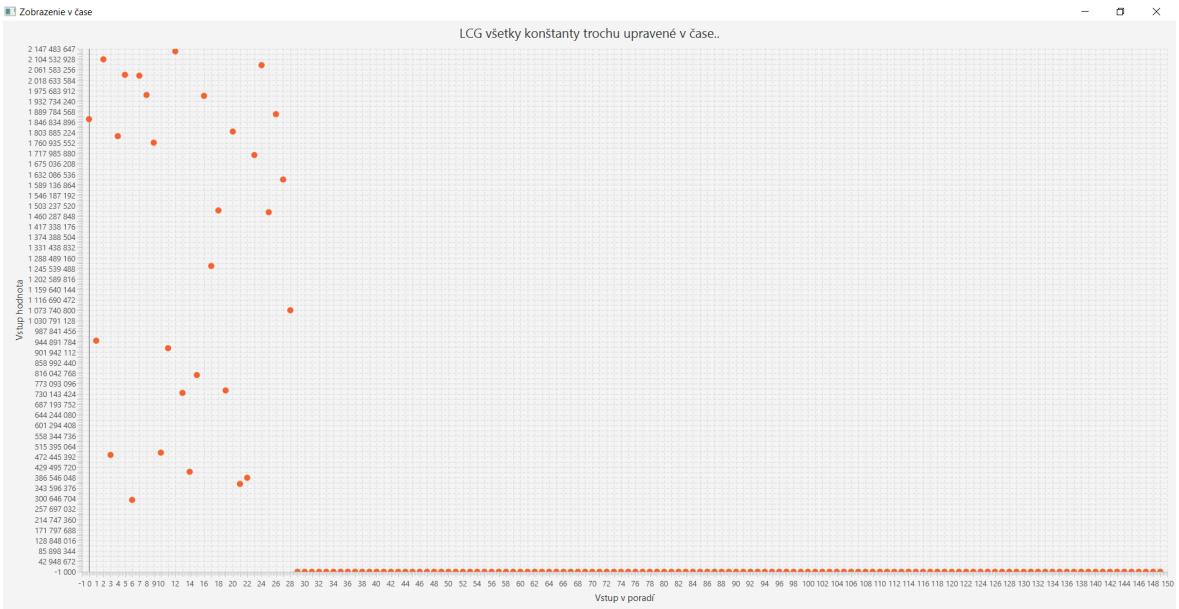


Obr. 4.1: LCG vzory pri nepárnom zloženom čísle (hore) a párnom čísle (dole) ako maximum.

Ďalej sme sa pozreli na konštantu a, ktorá sa používa na násobenie pre ďalšie „náhodné“ čísla. Ak bude hodnota tejto konštanty rovná nule, tak budú aj všetky vygenerované čísla rovné nule. Ak by sme ju zmenšili na číslo 2, výsledok bude, že generátor bude opakovať dokola 32 čísel po sebe (obr. 4.2).

Obr. 4.2: LCG pri konštante a = 2.

Ešte sa pozrieme na jednu zaujimavú úpravu algoritmu, a to takú, že si opäť zvolíme konštanty podľa seba. Algoritmus začal generovať náhodné čísla, avšak od istého okamihu začal generovať samé nuly. Stalo sa to vtedy, keď dosiahol presne hodnotu svojho maxima. Vtedy vypočítal hodnotu násobiča a ako aktuálne číslo modulo maximum, pričom vieme, že platí $x \; mod \; x = 0$. Násobič potom už navždy násobil nulou. (obr. 4.3)



Obr. 4.3: LCG pomenené konštanty.

4.2 Mersenne Twister

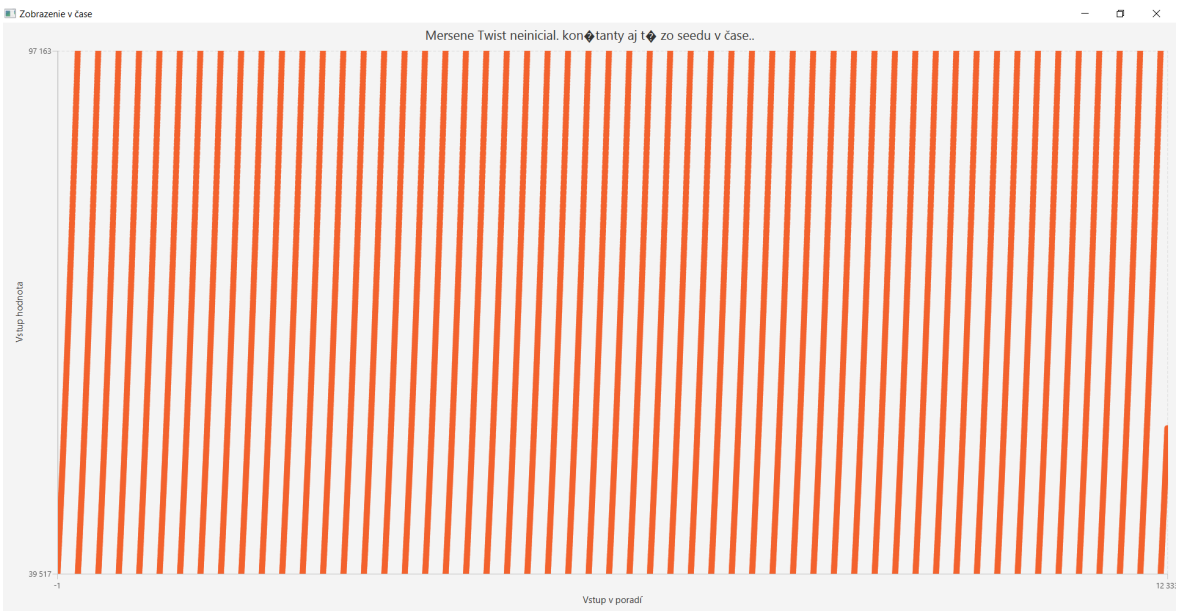
Je pseudonáhodný generátor čísel. Bol vytvorený v roku 1997 japoncami Makoto Matsumoto a Takuji Nishimura. Jeho meno pochádza z pojmu Mersenne prime.

Mersenne prime - je také prvočíslo, ktoré má tvar $MP = 2^n - 1$. V januári 2016 bolo objavené najväčšie MP, a to $2^{74207281} - 1$ a zároveň je aj doposiaľ najväčším známym prvočíslom. Toto číslo je 49. obajvené Mersenovo prvočíslo. Šesť najväčších známych prvočísiel sú Mersenove prvočísla.

MT algoritmus využíva Mersenovo prvočíslo $2^{19937}-1$ a zároveň je to aj jeho perióda. Táto závratná perióda patrí medzi veľké výhody, aj keď len perióda nezaručuje kvalitu. Okrem toho MT používa len pamäť veľkosti 624 slov. Tiež úspešne prešiel mnohými testami náhodnosti ako napríklad aj testami Diehard (popísané v 2. kapitole).
Algoritmus má aj nevýhody. Je veľmi náročný na pamäť, potrebuje až 2.5 KiB stavový buffer. (Na vyriešenie tohto problému bol navrhnutý algortimus TinyMT, ktorý používa len 127 bitov.) Dnešným dňom je MT trochu pomalý. [12]

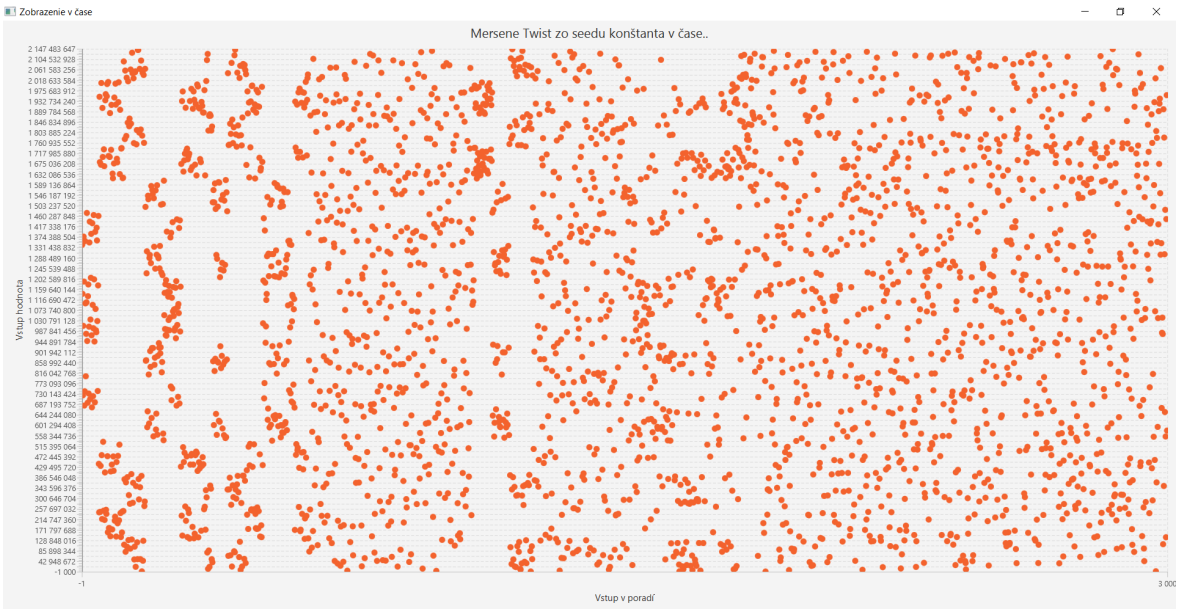
Výsledky

Ak algoritmu MT zabudneme inicializovať masky a v inicializácií seedu nepoužijeme „zázračnú“ konštantu, jeho vygenerované čísla budú lineárne (obr. 4.4).



Obr. 4.4: MT neinicializované masky a neprenásobenie inicializácie seedu konštantou.

Ak len neprenásobíme pri inicializovaní seedu pole „zázračnou“ konštantou a všetko ostatné spravíme správne, vygenerovaná postupnosť bude prvých pár stoviek čísiel viditeľne odlišná od náhodnej postupnosti (obr. 4.5).



Obr. 4.5: MT neprenásobenie inicializácie seedu konštantou.

Záver

V našej práci sme sa zaoberali vplyvom rôznych zmien, zavedených do známych a používaných algortimov, na ich výsledné vlastnosti.

V prvej kapitole sme si popísali základné definície z oblasti kryptológie, ktoré čitateľ pre pochopenie ďalšieho textu potreboval. Definície sme sa snažili popísať čo najjednoduchšie a najzrozumiteľnejšie.

Ďalej sme si popísali spôsob testovania kryptografických a štatistických hašovacích funkcií a generátorov náhodných čísel. Stručne sme popísali aj niektoré známe a štandardizované testy. Následne sme si povedali viac o implementácií praktickej časti, aké vlastnosti sme spravili my a priblížili sme si heatmapu, resp. našu legendu k nej. Na konci kapitoly sme pojednali o rôznych druhoch zmien a chýb, ktoré môžu pri implementácii algoritmu vzniknúť a roztriedili sme si ich pre ďalšie použitie v práci.

V tretej kapitole sme vždy popísali vybraný kryptografický algortimus najskôr všeobecne, potom sme sa pozreli na jeho fungovanie a nakoniec popísali našu experimentálnu časť a naše zistenia. Vybrané algoritmy boli: MD5, SHA-1, RC4, Murmur hash, FNV-1a, Pearson hashing. V štvrtej kapitole sme spravili to isté pre 2 vybrané generátory náhodných čísel, LCG a Mersenne Twister.

Okrem teoretickej časti sme mali aj praktickú, v ktorej sme vytvorili softvérové aplikácie. Postupne sme generovali rôzne vstupné reťazce, na ktoré sme aplikovali nami vybrané kryptografické a hašovacie funkcie. Potom sme premýšľali nad fungovaním algoritmov funkcií a vnášali do nich rôzne chyby, ktoré by mohli vzniknúť zlyhaním ľudského faktora alebo chyby, ktoré by mohli vyzerať neškodne. Tieto upravené algoritmy sme opäť aplikovali na vstupy. Následne sme sledovali ich vlastnosti oproti vlastnostiam pôvodných, neupravených algoritmov. Pri generátoroch náhodných čísiel sme postupovali podobne. Vytvorené grafy sme použili ďalej v práci.

Našim cieľom bolo poukázať na fakt, že aj malé chyby v implementácií algoritmu môžu mať za následok veľké zmeny v ich výsledných vlastnostiach, a upozorniť tak programátorov, aby si dávali pri implementácii veľký pozor a nezanedbali žiadne maličkosti. Zároveň sme prácou chceli varovať aj tých programátorov, ktorí by chceli vnášať chyby do svojich algoritmov úmyselne, na znemožnenie použitia napr. predpočítaných tabuliek útočníkmi, že takéto ich snaženie môže byť niekedy skôr na škodu ako na úžitok a svojim nerozvážnym konaním môžu naopak útočníkom ich prácu niekoľkonásobne

uľahčiť. Cieľ sa nám pri niektorých algoritmoch podaril splniť viac, u niektorých menej. Dôvodom je aj fakt, že je nemožné zhromaždiť konečnú množinu testov na overenie všetkých možných vzorov a závislostí vo výstupnych postupnostiach.

Nenašli sme žiadne predchádzajúce práce s podobným cieľom alebo kontextom. Práce, ktoré sa zaoberali konkrétnymi kryptografickými algoritmami, boli orientované skôr na ich slabé stránky a na útoky na tieto funkcie z pohľadu útočníka. My sa pozeráme na kryptografické funkcie skôr z pohľadu programátora a dôležitosti ich korektnej implementácie.

Literatúra

[1] Austin Appleby. MurmurHash statistical analysis done, reports posted. 11.03.2008. https://groups.google.com/forum/#!msg/sci.crypt/canqF49amtE/1F1XRY0TmdEJ.

[2] Austin Appleby. MurmurHash, 2008. https://sites.google.com/site/murmurhash/.

[3] Rebecca Blank, Acting Secretary, Patrick D Gallagher, Elaine Barker, William Barker, William Burr, William Polk, and Miles Smid. Nist special publication 800-57, recommendation for key management part 1: General (revision 3). 2012.

[4] Robert Davies. True random number generators, 2013. http://www.robertnz.net/true_rng.html a http://www.robertnz.net/hwrng.htm.

[5] G. Fowler et al. The FNV Non-Cryptographic Hash Algorithm, 2016. https://tools.ietf.org/html/draft-eastlake-fnv-12#page-93.

[6] Ján Guniš. Elektronický podpis, kryptografia, PGP. http://gkmke.sk/informatika/4.rocnik/SifrovanieInformacii.pdf, originál: https://di.ics.upjs.sk/informatika_na_zs_ss/studijny_material/sifry/pgp/pgp1.htm.

[7] Adam Horvath. MurMurHash3, an ultra fast hash algorithm for C# / .NET. Adam Horvath's blog, 2012. http://blog.teamleadnet.com/2012/08/murmurhash3-ultra-fast-hash-algorithm.html.

[8] Jakub Jantošík. Hashovanie - funkcia hash, 2012. http://jantosik2.devblog.matfyz.sk/p22756-hashovanie-funkcia-hash.

[9] Poonam Jindal and Brahmjit Singh. RC4 Encryption-A Literature Survey. *Procedia Computer Science*, 46:697 – 705, 2015. Proceedings of the International Conference on Information and Communication Technologies, ICICT 2014, 3-5 December 2014 at Bolgatty Palace and Island Resort, Kochi, India.

[10] Dušan Levický. *Kryptografia v komunikačnej bezpečnosti.* elfa, 2014.

[11] Markus Gäbler Luigi Accardi. Statistical analysis of random number generators, 2010. `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.915.1369&rep=rep1&type=pdf`.

[12] Makoto Matsumoto and Takuji Nishimura. Mersenne Twister: A 623-dimensionally Equidistributed Uniform Pseudo-random Number Generator. *ACM Trans. Model. Comput. Simul.*, 8(1):3–30, January 1998.

[13] NIST. About NIST. `https://www.nist.gov/about-nist`.

[14] Landon Curt Noll. FNV Hash, 2015. `http://www.isthe.com/chongo/tech/comp/fnv/index.html#history`.

[15] Peter K. Pearson. Fast Hashing of Variable-length Text Strings. *Commun. ACM*, 33(6):677–680, June 1990.

[16] Daryna Polevyk. *Moderní kryptografické metody, Bakalářská práce.* 2013. `https://is.bivs.cz/th/19520/bivs_b_a2/metody_BP_Daryna_Polevyk.pdf`.

[17] Fahmida Y. Rashid. Stop signing JAR files with MD5. *InfoWorld*, 2017. `http://www.infoworld.com/article/3159186/security/oracle-to-java-devs-stop-signing-jar-files-with-md5.html`.

[18] Michal Rjaško. Kryptológia - Úvod. 2015. `http://new.dcs.fmph.uniba.sk/files/uib/crypto_2015.pdf`.

[19] Andrew Rukhin, Juan Soto, James Nechvatal, Elaine Barker, Stefan Leigh, Mark Levenson, David Banks, Alan Heckert, James Dray, San Vo, Andrew Rukhin, Juan Soto, Miles Smid, Stefan Leigh, Mark Vangel, Alan Heckert, James Dray, and Lawrence E Bassham Iii. Statistical test suite for random and pseudorandom number generators for cryptographic applications, NIST SP 800-22 Rev. 1a, 2010.

[20] Robert Sanek. Password Storage in Databases: Best Practices. pages 33–39, 2015. `https://www.auburn.edu/undgres/documents/aujus/2015/articles/article5.pdf`.

[21] Martin Stanek. *Kryptológia, Pragmatický pohľad.* 2004.

[22] Martin Stanek. *Kryptológia, Základy.* 2015.

[23] Marc Stevens et al. The first collision for full SHA-1. *CWI Amsterdam, Google Research*, 2017. `http://shattered.io/static/shattered.pdf`.

[24] Andreas Schwill Volker Claus. *Lexikón informatiky.* Slovenské pedagogické nakladateľstvo, 1986.

[25] Chris von Eitzen. Skype's encryption procedure partly exposed. *The H security*, 2010. `http://www.h-online.com/security/news/item/Skype-s-encryption-procedure-partly-exposed-1034577.html`.

[26] Xiaoyun Wang, Dengguo Feng, Xuejia Lai, and Hongbo Yu. Collisions for Hash Functions MD4, MD5, HAVAL-128 and RIPEMD. Cryptology ePrint Archive, Report 2004/199, 2004. `http://eprint.iacr.org/2004/199`.

Prílohy

Príloha na priloženom CD.