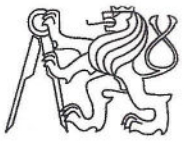
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ



ZADÁNÍ DIPLOMOVÉ PRÁCE

Název:	Příklad útoku na šifru RC4
Student:	Bc. Pavel Kocka
Vedoucí:	Ing. Jiří Buček
Studijní program:	Informatika
Studijní obor:	Počítačová bezpečnost (magisterský)
Katedra:	Katedra počítačových systémů
Platnost zadání:	do konce letního semestru 2014/15

Pokyny pro vypracování

Seznamte se s šifrou RC4 a existujícími útoky na tuto šifru (viz odkazy na literaturu). Vyberte existující útok a ověřte jeho funkčnost. Prozkoumejte praktické dopady vybraného útoku. Navrhněte a vytvořte školní úlohu, která bude založená na provedení vybraného útoku. Vytvořenou úlohu důkladně otestujte.

Seznam odborné literatury

http://www.isg.rhul.ac.uk/tls/
http://blog.cryptographyengineering.com/2013/03/attack-of-week-rc4-is-kind-of-broken-in.html
AlFardan, N., et al. "On the Security of RC4 in TLS and WPA." USENIX Security Symposium. 2013.

Ing. Tomáš Zahradnický, Ph.D.
vedoucí katedry

prof. Ing. Pavel Tvrdík, CSc.
děkan

V Praze dne 10. ledna 2014

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra počítačových systémů

Diplomová práce

Příklad útoku na šifru RC4

Bc. Pavel Kocka

Vedoucí práce: Ing. Jiří Buček

5. května 2014

Poděkování

Děkuji svému vedoucímu práce Ing. Jiřímu Bučkovi za ochotu, cenné rady a zpětnou vazbu při vypracovávání této práce. Zároveň chci poděkovat svým rodičům za bezmeznou podporu a půjčení notebooku na provedení časově náročných měření.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 5. května 2014 .....

České vysoké učení technické v Praze
Fakulta informačních technologií

Odkaz na tuto práci

Kocka, Pavel. *Příklad útoku na šifru RC4*. Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2014.

Abstrakt

Tato práce se zabývá útoky na proudovou šifru RC4. Obsahuje přehled publikovaných zranitelností, které byly objeveny za dobu její existence. Další částí práce je praktické ověření funkčnosti dvou útoků popsaných v [2]. Nakonec obsahuje návrh a implementaci školní úlohy, založené na ověřených útocích.

Klíčová slova broadcast útoky, kryptoanalýza, proudová šifra, pseudonáhodný generátor, RC4, rozvržení klíče, WEP, WPA

Abstract

This work deals with attacks on stream cipher RC4. It provides an overview of published vulnerabilities. Next part is a practical verification of attacks described in [2]. Lastly, it contains a design and implementation of school exercise, based on verified attacks.

Keywords broadcast attacks, cryptanalysis, key scheduling, pseudorandom generator, RC4, stream cipher, WEP, WPA

Obsah

Seznam obrázků

Seznam tabulek

Seznam algoritmů

Úvod

Šifra RC4 se na poli kryptologie vyskytuje již přes dvacet let. Přesto je stále hodně používaná především při šifrovaném přenosu webových stránek (TLS) a zabezpečení Wi-Fi (WEP, WPA).

O zranitelnostech a nedokonalostech šifry RC4 bylo publikováno mnoho vědeckých materiálů. Zorientovat se ve všech z nich se tak stalo časově náročným úkolem. Navíc nedávné publikace vedly k tomu, že klíčové společnosti IT průmyslu začínají šifru RC4 označovat za zastaralou a nedoporučují její používání.

Proto jsem se rozhodl vytvořit přehled nejdůležitějších publikovaných zranitelností a možných útoků na šifru RC4. Tato práce tak může posloužit jako upozornění pro ty, kteří šifru RC4 stále používají, aby měli možnost se včas porozhlédnout po nové alternativě.

V další části práce se pokusím implementačně ověřit funkčnost dvou útoků, které v rámci své práce [2] publikovali AlFardan a kol. Získané poznatky poté využiji k vytvoření školní úlohy, jejímž cílem bude interaktivním způsobem seznámit studenty se šifrou RC4 a ověřovanými útoky.

Kapitola 1

RC4

Šifru RC4 vytvořil v roce 1987 Ron Rivest z RSA Security. Přestože šifra nebyla nikdy odtajněna, její popis od anonymního autora se v roce 1994 objevil na internetu [3]. Poté bylo potvrzeno, že zveřejněný popis skutečně odpovídá originálu, protože výstup šifrování odtajněné šifry a komerčního produktu využívajícího RC4 byl totožný.

1.1 Proudové šifry

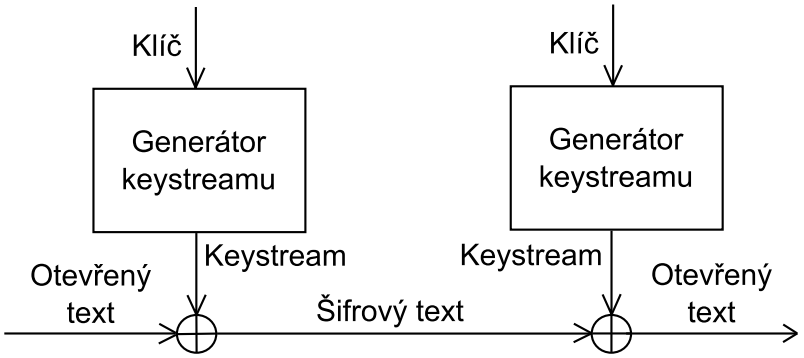
Základem šifrování proudových šifer bývá zpravidla operace `XOR` (binární sčítání modulo 2). Vstupní data otevřeného textu jsou postupně xorována s náhodnou posloupností bitů, kterou v této práci nazývám keystream.[1] Smyslem proudové šifry tak je generování náhodného keystreamu. Generovaný keystream je závislý na typu použité šifry, na tajném klíči a případně na inicializačních vektorech.

V praxi se ta část algoritmu proudové šifry, která slouží pro generování keystreamu, přirovnává k pseudonáhodným generátorům čísel (PRNG). Jsou na ni tedy kladeny stejné nároky jako na kryptograficky bezpečné PRNG [18]:

- dobré statistické vlastnosti (rovnoměrné rozdělení, maximální entropie, délka periody),
- vysoká rychlost generování,
- nízká složitost realizace (v hardwaru, v softwaru),
- bezpečnost (odolnost před útoky, testovatelnost zdroje entropie).

Šifrování a dešifrování u standardních proudových šifer probíhá stejným způsobem, jedná se tedy o symetrický typ šifrování. Zjednodušená ukázka funkce proudových šifer viz obrázek 1.1.

[1] Českým ekvivalentem může být slovo heslo. Rozhodl jsem se však použít anglický výraz keystream, protože lépe zapadá do textu této práce.



Obrázek 1.1: Ukázka proudové šifry, $\oplus$ značí operaci `XOR`

1.2 Princip šifry RC4

RC4 je proudová šifra. Její výhodou oproti jiným proudovým šifrám je, že operuje pouze na úrovni celých bytů. Dá se proto snadno implementovat jak v hardwaru, tak především v softwaru. Díky své jednoduchosti se použití RC4 v praxi velmi rozšířilo.

Průběh šifrování můžeme rozdělit na dvě hlavní části: algoritmus rozvržení klíče – KSA (key scheduling algorithm) a pseudonáhodné generování PRGA (pseudorandom generation algorithm):

- KSA (algoritmus 1.1) probíhá jako první a slouží jako příprava pro šifrování. Algoritmus přebírá na vstupu tajný klíč o délce standardně 5 až 32 bytů (tj. 40 až 256 bitů). Výstupem je počáteční vnitřní stav $st_0 = (i, j, S)$, kde S je nějaká permutace pole 256 bytů a i, j jsou indexy v tomto poli. Všimneme si, že šifra RC4 je tedy inicializována pouze tajným klíčem, nepodporuje standardní použití inicializačních vektorů.
- PRGA (algoritmus 1.2) přebírá na vstupu vnitřní stav st_r. Výstupem je potom nový vnitřní stav st_{r+1} a byte Z_r, který se použije pro zašifrování jednoho vstupního bytu otevřeného textu.

Pseudokódy algoritmů a značení jsem převzal z [2].

Algoritmus 1.1 RC4 algoritmus rozvržení klíče (KSA)

Vstup: klíč K délky l bytů
Výstup: počáteční vnitřní stav st_0

for $i = 0$ to 255 **do**
  $S[i] \leftarrow i$
end for
$j \leftarrow 0$
for $i = 0$ to 255 **do**
  $j \leftarrow j + S[i] + K[i \bmod l] \bmod 256$
  swap$(S[i], S[j])$
end for
$i, j \leftarrow 0$
$st_0 \leftarrow (i, j, S)$
return st_0

Algoritmus 1.2 RC4 pseudonáhodný generátor (PRGA)

Vstup: vnitřní stav st_r
Výstup: keystream byte Z_r, vnitřní stav st_{r+1}

$i \leftarrow i + 1 \bmod 256$
$j \leftarrow j + S[i] \bmod 256$
swap$(S[i], S[j])$
$Z_r \leftarrow S[S[i] + S[j] \bmod 256]$
$st_{r+1} \leftarrow (i, j, S)$
return (Z_r, st_{r+1})

Kapitola 2

Zranitelnosti a útoky na RC4

Zkoumání šifry RC4 bylo věnováno mnoho úsilí. Není proto divu, že za dobu své existence byly objeveny vážné i méně vážné zranitelnosti. Zatím nejpodrobnější seznam zranitelností RC4 ve své nedávné disertační práci (2013) prezentoval Sen Gupta [32]. Z jeho práce budu v této kapitole částečně vycházet.

Pokud chceme zranitelnosti RC4 nějak kategorizovat, můžeme je rozdělit do skupin podle toho, jestli jde o hledání slabin v rozvrhování klíče (KSA), pseudonáhodného generátoru (PRGA) nebo jejich kombinace, či další druhy slabin, které vyplývají přímo z chování proudových šifer.

Podobně můžeme klasifikovat i existující útoky podle toho, jaký je důsledek jejich úspěšného provedení. Některé útoky se zaměřují na obnovení použitého klíče ze známého vnitřního stavu šifry nebo keystreamu, dále obnovení vnitřního stavu z keystreamu, rozlišení od náhodného proudu bitů a nebo pokusy o obnovu otevřeného textu ze šifrových textů na základě nalezených statistických odchylek.

2.1 Běžné zranitelnosti proudových šifer

Přímo z funkce proudových šifer vyplývají některé zranitelnosti. Při šifrování nedochází k transpozici, pořadí znaků ve zprávě tak zůstává vždy stejné. Pokud známe část šifrované zprávy, můžeme tuto informaci využít. Samotné šifrování probíhá pomocí xorování. Vzhledem k vlastnostem operace xor nesmíme dopustit, aby dvě zprávy byly zašifrované stejným klíčem. Klíč použitý pro inicializaci šifry nesmí být předvídatelný.

2.1.1 Znovupoužití stejného klíče

Typická chyba při používání proudových šifer je znovupoužití stejného klíče. Pokud se útočníkovi podaří zachytit dvě zprávy šifrované stejným klíčem, může odhalit obsah obou zpráv.

Řekněme, že máme zprávy M_1 a M_2 obě zašifrované společným klíčem K. Výsledkem jsou zašifrované zprávy:

$$C_1 = M_1 \oplus K$$
$$C_2 = M_2 \oplus K.$$

Potom, když vyxorujeme zachycené šifrové texty C_1 a C_2 mezi sebou, dostaneme:

$$C_1 \oplus C_2 = M_1 \oplus K \oplus M_2 \oplus K = M_1 \oplus M_2.$$

Tajný klíč K se nám ztratí a zůstane nám xor zpráv M_1, M_2, což jsou smysluplné zprávy. To znamená, že se chovají podle určitých pravidel a rozdělení znaků v nich není rovnoměrné. Z výsledného xoru se nám tak může podařit získat obsah zpráv, třeba použitím frekvenční analýzy.

2.1.2 Zásah do šifrového textu

Při útoku MiM (Man In The Middle) můžeme zachytit a upravit šifrovanou zprávu. V případě, kdy je obsah zprávy předvídatelný, dokážeme zašifrovanou zprávu upravit tak, aby obsah po jejím dešifrování byl smysluplný.

Představme si zjednodušený případ, kdy zpráva obsahuje pouze jeden znak, buď A nebo N. V binární soustavě a ASCII kódování mají zprávy tyto hodnoty:

$$A = 01000001$$
$$N = 01001110.$$

Dále mějme tajný klíč:

$$K = 11001101,$$

zašifrujeme s ním zprávu A:

$$A \oplus K = 01000001 \oplus 11001101 = 10001100 = C_1.$$

Útočník chce obsah zprávy C_1 změnit z A na N, vezme tedy hodnotu:

$$A \oplus N = 01000001 \oplus 01001110 = 00001111,$$

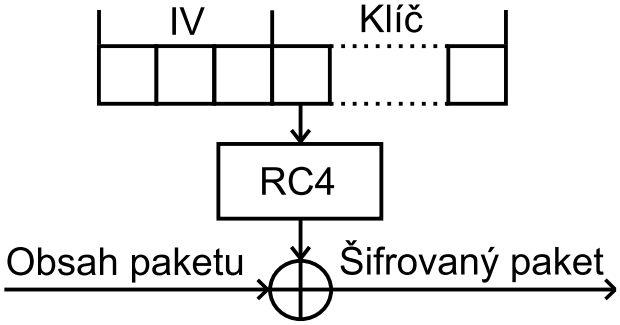
a přixoruje jí k zachycené zprávě C_1:

$$10001100 \oplus 00001111 = 10000011 = C_2,$$

Výslednou zprávu C_2 podstrčí namísto zprávy C_1. Po dešifrování zprávy C_2 původním klíčem K dostaneme N namísto A:

$$C_2 \oplus K = 10000011 \oplus 11001101 = 01001110 = N,$$

Obsah zprávy byl tedy změněn, a pokud nějakým způsobem nekontrolujeme její integritu, nemáme šanci si změny všimnout. Navíc, vzhledem k chování operace `XOR`, vůbec nezáleží na použitém klíči.



Obrázek 2.1: Ukázka WEP šifrování paketu

2.1.3 Použití předvídatelného klíče

Šifra RC4 je inicializována tajným klíčem, viz algoritmus 1.1. Proto musíme zaručit, aby použitý klíč byl skutečně náhodný. Pokud bychom dokázali klíč nebo jeho část nějakým způsobem předvídat, může to vyústit ke ztrátě celého zabezpečení. Asi nejznámější výskyt špatné inicializace klíče použitého pro šifrování se vyskytuje v zabezpečení bezdrátové komunikace WEP.

Pro ochranu před ztrátovostí paketů při přenosu je WEP navržen tak, aby každý datový paket šifroval odděleně. Proto pro šifrování nestačí pouze tajný klíč, je potřeba navíc přidat inicializační vektory (IV). Šifra RC4 však použití IV nepodporuje. To WEP řeší tak, že IV umístí na začátek jako součást klíče. Pro IV jsou vyhrazené pouze 3 byty a jejich hodnoty se většinou generují v čítačovém režimu. Jednoduchá ukázka WEP šifrování paketů je vidět na obrázku 2.1.

Od roku 2001 je WEP považován za nedokonalý způsob zabezpečení Wi-Fi. Útoky na něj se postupem času neustále zlepšují. Jako náhrada za WEP bylo koncem roku 2002 představeno zabezpečení WPA. To používá inicializační vektory o délce 48 bitů. Použitý klíč u RC4 se počítá jako hash z více informací, proto zranitelnosti aplikovatelné na WEP u WPA nefungují.

Krátkých IV u WEP zabezpečení v roce 2001 využili Fluhrer, Mantin a Shamir [11] pro navržení útoku, který teoreticky za znalosti dostatečného množství paketů (odhadem 4 miliony) dokáže najít použitý klíč s 50% pravděpodobností úspěchu. Stubblefield, Ioannidis a Rubin jejich útok prakticky ověřili a prokázali, že potřebný počet paketů se pohybuje kolem 5 až 6 milionů [39]. Útok využívá faktu, že některé byty paketů před zašifrováním jsou útočníkovi známé, jako například hlavičky paketů.

V roce 2004 se od uživatele vystupujícího pod přezdívkou KoreK na internetu objevila implementace útoku na WEP, které se později začalo přezdívat Aircrack-ng [21, 22]. Tento útok potřebuje pro 50% pravděpodobnost nalezení klíče jen 100 000 paketů, což je oproti předchozím výsledkům obrovský skok.

Další teoretický útok představil v roce 2006 Klein [19]. Klein odhadoval,

Rok	Odkaz	Typ útoku	Počet paketů
2001	FMS teoretický [11]	pasivní (odhad)	4 000 000
2001	FMS ověřený [39, 38]	pasivní (skutečný)	5 500 000
2004	Korek [21, 22]	pasivní (Aircrack-ng)	100 000
2006	Klein teoretický [19]	pasivní (odhad)	25 000
2006	Klein ověřený [36, 37]	pasivní (skutečný)	60 000
2007	TWP útok [41]	pasivní (skutečný)	40 000
2007	VV útok [43]	pasivní (skutečný)	32 768
2009	Tews-Beck [40]	interaktivní (Aircrack-ng)	24 200
2009	TB ověřený [36, 37]	ne-interaktivní (Aircrack-ng)	30 000
2010	SVV útok [36]	pasivní (odhad)	9 800
2010	SVV útok [37]	pasivní (odhad)	4 000
2013	SSVV útok [35]	pasivní (skutečný)	27 500
2013	SSVV útok [35]	ne-interaktivní (Aircrack-ng)	22 500
2013	SSVV útok [35]	interaktivní (Aircrack-ng)	19 800

Tabulka 2.1: Přehled útoků na WEP z pohledu potřebného množství paketů (Tabulka převzata z [32])

že pro provedení útoku bude potřeba pouze 25 000 paketů. Skutečné implementace se útok dočkal až o několik let později [36, 37] a potřebné množství paketů dosahuje 60 000.

Tews, Weinmann a Pyshkin v roce 2007 vylepšili Kleinův (v té době stále ještě prakticky neověřený) útok [41]. Kromě pasivního zachytávání paketů přišli také se způsobem, jak útok urychlit, vkládáním upravených ARP paketů do komunikace. Jejich útok dokáže prolomit 104bitový WEP během 60 vteřin a potřebuje na to kolem 40 000 paketů.

Podobný útok představili ten samý rok Vaudenay a Vuagnoux [43]. Navíc se jim podařilo objevit dvě nové slabiny týkající se operace modulo v KSA. Potřebný počet paketů k provedení útoku tak snížili na $2^{15} = 32768$.

Tento útok dále vylepšili ve své práci v roce 2009 Tews a Beck [40]. Útok implementovali jako rozšíření programu Aircrack-ng. Požadovaný počet paketů snížili na 24 200. Tews a Back zároveň v této práci popsali první praktický útok na WPA. Zaměřili se přitom na funkci TKIP (Temporal Key Integrity Protocol). TKIP je navržen tak, aby ho po nainstalování nového firmwaru nebo ovladačů, mohl použít starší hardware podporující pouze WEP.

Útok umožňuje čtení ARP požadavků nebo vložení až 7 paketů s libovolným obsahem do sítě. Nedokáže tedy u WPA obnovit použitý klíč. Přesto jde o první průlom bezpečnosti tohoto systému zabezpečení bezdrátové komunikace.

Další teoretické útoky na WEP objevili Sepehrdad, Vaudenay a Vuagnoux [36, 37]. Zároveň popsali nový útok na WPA. Tento útok dokáže obnovit 128bi-

tový klíč, potřebuje k tomu však 2^{38} paketů a odhadovaná časová složitost je 2^{96}. Útok tedy není praktický, ale jde o další pokrok směrem k prolomení WPA bez použití hrubé síly nebo slovníkových útoků.

Nejnovější útoky na WEP (2013) ve své práci představila čtveřice Sepehrdad, Sušil, Vaudenay a Vuagnoux [35]. Zaměřili se na útoky, které sami několik let před tím navrhli. Útoky dále vylepšili a předvedli jejich skutečnou implementaci, kterou dali k dispozici jako doplněk k programu Aircrack-ng. Nově ověřené hodnoty jsou vyšší než původní odhad, přesto se však jedná o doposud nejlepší známé útoky. V pasivním režimu stačí útoku zachytit 27 500 paketů, v interaktivním režimu pak pouze 19 800.

Kompletní přehled všech doposud publikovaných útoků na WEP lze vidět v tabulce 2.1.

2.2 Zranitelnosti algoritmu rozvržení klíče (KSA)

Algoritmus KSA, viz 1.1, nám vytvoří počáteční vnitřní stav šifry RC4 v závislosti na tajném klíči. Teoreticky dokáže vygenerovat 256! možných počátečních stavů, protože obsahuje pole s permutací 256 hodnot. Rozvrhování klíče šifry RC4 se tedy může zdát velmi silné. Výzkum však dokázal, že KSA obsahuje slabiny, které někdy vedou až k obnovení použitého klíče.

2.2.1 Slabé klíče

Slabý klíč je pro šifru takový klíč, který způsobí nežádané chování šifry. To může mít za následek neoprávněné získání otevřeného textu ze šifrového textu. Například u šifry DES jsou dokonce známy klíče, pro které je šifrový text totožný s otevřeným textem.

V ideálním případě by šifra neměla mít žádné slabé klíče. V praxi se však šifry se slabými klíči vyskytují. Potom chceme, aby počet těchto klíčů byl mnohonásobně menší než počet všech možných klíčů. Pokud jsme schopni slabé klíče identifikovat, můžeme šifru navíc vylepšit zákázáním jejich používání. Ne vždy jsme však toho schopni a ne vždy všechny slabé klíče známe.

Je zřejmé, že šifra RC4 obsahuje slabé klíče. V RC4 existuje $2^{8 \cdot 256} = 2^{2048}$ možných klíčů (každý klíč kratší než 256 bytů má 256bytový ekvivalent), ale pouze 256! $\sim 2^{1684}$ možných počátečních stavů. Každý počáteční stav má tedy v průměru 2^{364} klíčů, které ho generují. Šifra RC4 tedy obsahuje kolizní klíče, které při použití generují totožný keystream. Jakýkoliv otevřený text zašifrovaný pomocí takovýchto klíčů bude potom naprosto stejný. Z kryptografického hlediska je toto závažný nedostatek pro jakoukoliv šifru. Vzhledem k velikosti stavového prostoru ale není nalezení kolizních klíčů jednoduché.

V roce 2000 Grosul a Wallach [15] našli klíče velikostí blízké 256 bytům, které po průběhu KSA generují počáteční stavy, jenž se od sebe liší jen v několika málo bitech. Skutečné kolize klíčů nalezl o devět let později Mitsuru Matsui [27]. Dokázal, že RC4 má slabé klíče o menší délce než kolem 256

bytů. Matsui představil algoritmus pro hledání kolizí (nebo velmi podobných stavů) typu, kdy se od sebe pár klíčů liší pouze v jednom bitu, tzn. mají Hammingovu vzdálenost rovnou 1.

Nejkratší pár kolizních klíčů, který Matsui nalezl a zveřejnil ve své práci, má délku 24 bytů a zapsaný v šestnáctkové soustavě vypadá takto:

K_1 = 00 42 CE D3 DF DD B6 9D 41 3D BD 3A B1 16 5A 33 ED A2 CD 1F E2 8C 01 **76**,

K_2 = 00 42 CE D3 DF DD B6 9D 41 3D BD 3A B1 16 5A 33 ED A2 CD 1F E2 8C 01 **77**.

Oba klíče skutečně produkují totožný keystream, což jsem ověřil pomocí vlastní implementace šifry RC4. Z použitého algoritmu vyplynul zajímavý fakt, že nalezení kolizních párů u delších klíčů je jednodušší než u kratších klíčů. Z tohoto pohledu tedy použití delšího klíče u RC4 nezaručuje vyšší bezpečnost šifrování, jako je tomu u většiny šifer.

Jiageng Chen a Atsuko Miyaji [8] se postarali o nalezení nového typu kolizí, jejichž Hammingova vzdálenost se rovná 3. V další práci [7] tato dvojice vytvořila novou formalizaci kolizních párů klíčů RC4 vzhledem k tomu, jak probíhá KSA. Nakonec se jim podařilo vylepšit algoritmus vyhledávání kolizních klíčů [9]. Za pomoci nového algoritmu nalezli doposud nejkratší nalezený pár o délce 22 bytů:

K_1 = A2 27 43 A7 03 94 2F 17 75 BB A7 27 8F DD 3E 7B C6 A1 C7 **81** 02 5A,

K_2 = A2 27 43 A7 03 94 2F 17 75 BB A7 27 8F DD 3E 7B C6 A1 C7 **82** 02 5A.

2.2.2 Obnovení klíče z počátečního stavu

Andrew Roos byl první, kdo již v roce 1995 upozornil na korelaci mezi klíčem a počátečními byty stavu generovaného KSA [31]. Experimentálně určil, že i-tý prvek permutace S má pro nízké hodnoty i poměrně velkou pravděpodobnost, že se bude rovnat:

$$S[i] = \frac{i \cdot (i+1)}{2} + \sum_{x=0}^{i} K[x].$$

Přesnou pravděpodobnost pro tento jev později určili Paul a Maitra [30], kteří jsou zároveň autory algoritmu na obnovu $l$bytového klíče s konstantní pravděpodobností úspěchu. Princip algoritmu spočívá v hledání rovnic, které platí s určitou pravděpodobností a popisují závislost klíče na prvních bytech počátečního stavu. Za použití těchto rovnic se potom algoritmus snaží obnovit použitý klíč. Složitost navrženého algoritmu je $O(2^{4l})$, což je odmocnina počtu

všech možných klíčů délky l bytů. Bohužel pravděpodobnost úspěchu tohoto algoritmu je velmi nízká.

S lepším algoritmem přišli Eli Biham a Yaniv Carmeli [4]. Ten dosahuje pravděpodobnosti úspěchu při obnově 5bytového klíče až 86 % za 0,02 sekundy. Přesná složitost jejich algoritmu není v práci stanovena, čas běhu je však řádově nižší než u algoritmu z [30].

Další podobné slabiny KSA objevili Akgün, Kavak a Demirci [1] a v kombinaci s poznatky z [4, 30, 31] navrhli nový algoritmus, který je přesnější a rychlejší než jeho předchůdci. Rozdíl od předchozích algoritmů je mimo jiné v tom, že algoritmus se nemusí vždy snažit o obnovu celého klíče. Může obnovit pouze jeho část a zbytek klíče dopočítat hrubou silou. Klíč o délce 5 bytů najde algoritmus s 99% pravděpodobností za 0,011 s. Efektivně je schopen obnovit klíče z počátečního stavu až do délky 12 bytů.

Obnovení klíče z vnitřního stavu je tedy pro kratší klíče velmi často možné. Stačí nám znalost jakéhokoliv vnitřního stavu RC4. Nemusí se jednat jenom o počáteční stav, protože běh PRGA můžeme snadno obrátit a dostat tak z libovolného stavu stav počáteční.

2.2.3 Obnovení klíče z keystreamu

Obnovení klíče ze znalosti keystreamu je velmi obtížné. Proto zatím nebyl publikován žádný algoritmus, který by to dokázal. Z principu je tento úkol obtížnější, než obnovení klíče ze znalosti vnitřního stavu šifry.

Teoreticky by nám totiž stačil algoritmus, který by z keystreamu dokázal obnovit libovolný vnitřní stav šifry. Tento stav můžeme snadno převést do počátečního stavu šifry a potom bychom mohli použít algoritmus pro obnovení klíče z počátečního stavu, viz předchozí sekce 2.2.2. Bohužel, jak je uvedeno dále v sekci 2.3.1, žádný časově praktický algoritmus zatím není známý.

Existují některé útoky, které ze znalosti keystreamu jsou schopné obnovit použitý klíč. Jejich princip je však založen na využití nějaké chyby v implementaci šifry RC4. Nejznámější takový případ je implementace WEP, viz sekce 2.1.3.

2.3 Zranitelnosti pseudonáhodného generátoru (PRGA)

Algoritmus PRGA, viz 1.2, by se měl chovat jako PRNG (Pseudorandom Number Generator), proto se při zkoumání jeho slabin můžeme zaměřit na jeho statistické vlastnosti. Pokud u výstupu PRGA najdeme nějakou statistickou odchylku, můžeme ji potom dále využít k implementaci různých útoků.

2.3.1 Obnovení vnitřního stavu z keystreamu

Vnitřní stav RC4 se skládá z permutace 256 bytů a dvou ukazatelů na tuto permutaci. Dohromady tedy existuje $256! \cdot 256^2$ možných stavů. Při znalosti keystreamu máme k dispozici pouze posloupnost výstupních bytů Z_r, u kterých víme, že nabývají hodnot $S[S[i] + S[j]]$, viz obrázek 2.2. Teoreticky ještě můžeme určit hodnotu i, protože se chová jako čítač modulo 256 a dá se tak snadno předvídat. Žádné další informace o vnitřním stavu však nemáme. Proto je obnovení vnitřního stavu za znalosti keystreamu velmi obtížný úkol.

První, kdo se zabýval obnovením vnitřního stavu RC4 z keystreamu byli Knudsen, Meier a kol. [20]. Ve své práci v roce 1998 popsali důležitost metody *swap* v PRGA a jak by nižší používání této metody umožnilo silnější kryptoanalytické útoky. Zároveň představili první algoritmus na obnovu vnitřního stavu z keystreamu. Funkčnost algoritmu experimentálně ověřili na menší verzi RC4, která obsahuje pro vnitřní stav méně bytů. Složitost algoritmu analyticky odhadli jako menší než prohledání odmocniny všech možných stavů. Útok tedy pro běžnou RC4 není praktický.

V roce 2000 navrhnul Golić nový iterativní algoritmus [13]. Ten je založený na algoritmu z [20], ale používá složitější pravděpodobnostní analýzu. Algoritmus díky tomu potřebuje menší množství bytů keystreamu.

Další útok založený na algoritmu z [20] publikovali v roce 2007 Tomašević, Bojanić a Nieto-Taladriz [42]. Algoritmus ukládá hádané proměnné ve formě stromu a pak hledá požadovaný stav pomocí efektivního prohledávání stromu. Odhadovaná složitost útoku je 2^{731}.

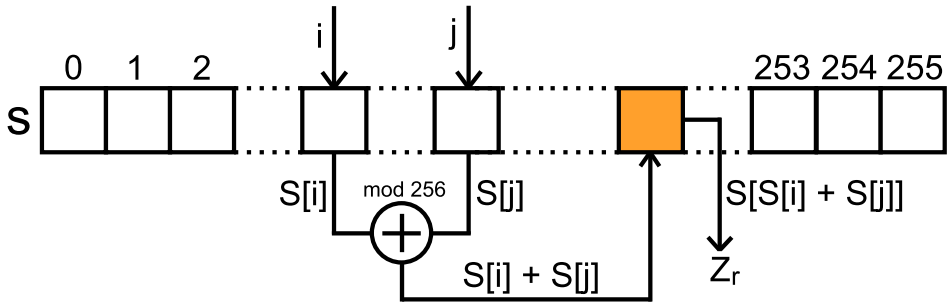
Nejlepší doposud nalezený útok prezentovali v roce 2008 Maximov a Khovratovich [28]. Ti objevili v průběhu PRGA opakující se vzory, které mohli použít pro určení některých parametrů. Tím se jim podařilo rapidně snížit potřebný počet testovaných stavů. Jejich odhadovaná složitost útoku je 2^{241}. Tuto složitost označili Golić a Morgari po analýze útoku za příliš optimistickou [14].

Dále existují útoky, které pro obnovení vnitřního stavu ze znalosti keystreamu používají postranní kanály. V roce 1994, velmi brzy po zveřejnění specifikace RC4, objevil Finney [10] stavy, do kterých se šifra RC4 nemůže při normálním použití nikdy dostat. Šifra RC4 je ve Finneyho stavu, když nabývá hodnot:

$$j = i + 1 \qquad a \qquad S[j] = 1.$$

Potom všechny následující stavy jsou také Finneyho. Tohoto faktu využili v roce 2005 Biham, Granboulan a Nguyên [5], když navrhli dva algoritmy založené na chybové analýze, které umožňují obnovení vnitřního stavu ze znalosti keystreamu. Tyto útoky mají praktickou složitost. Jsou však založené na myšlence, že v průběhu šifrování můžeme do vnitřního stavu šifry vkládat chyby.

V roce 2011 představili Chardin, Fouque a Leresteux jiný útok, který využívá postranních kanálů [6]. Ten dokáže obnovit vnitřní stav šifry RC4 za po-



Obrázek 2.2: Určení výstupního keystream bytu Z_r v PRGA

moci sledování vyrovnávací paměti procesoru (Cache Timing Analysis). Pro provedení útoku stačí znát v průměru 550 bytů keystreamu a mít možnost spustit na napadeném stroji vlastní proces.

2.3.2 Statistické odchylky

Při luštění jakékoliv zašifrované zprávy je důležité umět rozlišit, jaká šifra byla pro zašifrování použita. Potom jsme schopni použít odpovídající útoky, pokud existují, a eventuelně se dobrat k původnímu znění zprávy. Proto se u šifer hledají takzvané rozlišovače, pomocí nichž dokážeme šifru rozlišit od náhodného proudu bitů. Často se jedná právě o nějakou statistickou odchylku od normálu, díky které jsme ze šifrových textů schopni určit, že byla použita ta či ona šifra.

2.3.2.1 Odchylky závislé na klíči

V předchozí sekci týkající se zranitelností KSA 1.1 jsme si ukázali, že počáteční stav šifry je do jisté míry lineárně závislý na použitém klíči. Tato závislost se tedy logicky musí projevit i v pozdějším výstupu keystreamu z PRGA, protože přebírá počáteční stav a další byty keystreamu produkuje jeho modifikací.

V roce 2011 Sen Gupta, Maitra, Paul a Sarkar [34] prezentovali důkaz odchylek nalezených v [36] a použitých při praktickém útoku na WEP. Ve své další práci pak nalezli a dokázali závislost keystream bytu na délce použitého klíče [33]. Ukázali, že při použití klíče o délce l bytů, obsahuje byte keystreamu Z_l kladnou odchylku pro hodnotu $-l$. Například pro klíč délky 16 bytů určili odchylku:

$$P(Z_{15} = -16) = \frac{1}{2^8} + \frac{6}{2^{16}}.$$

(Byty Z_i jsou indexovány od 0.) Poznatky z jejich práce se dají využít pro určení délky použitého klíče.

Na předchozí práci [33] v roce 2013 částečně navázali Isobe, Ohigashi, Watanabe a Morii [16], když experimentálně objevili, že se odchylka závislá

na délce klíče netýká pouze keystream bytu Z_l. Tuto závislost rozšířili na tvar:

$$Z_{x \cdot l} = -x \cdot l.$$

Obecný vzorec pro určení pravděpodobnosti této odchylky a důkaz však v práci chybí.

Ten doplnil téhož roku ve své práci Sen Gupta [32]. Zároveň dokázal, že dříve objevená a zatím nevysvětlená negativní odchylka $Z_0 = 129$ [2] je také závislá na délce použitého klíče. Konkrétně se tato odchylka vyskytuje pouze při použití délky klíče $l = 2, 4, 8, 16, 32, 64, 128$ bytů, což jsou netriviální dělitelé 256.

2.3.2.2 Odchylky závislé na vnitřním stavu

Odchylky závislé na vnitřním stavu jsou užitečné při pokusech o obnovu vnitřního stavu ze znalosti keystreamu.

Jenkins [17] roce 1997 objevil první závislost mezi známými informacemi (i_r, Z_r) a tajnými informacemi (S, j_r). Zjistil, že jevy

$$S[j_r] = i_r - Z_r,$$

$$j_r - S[i_r] = Z_r$$

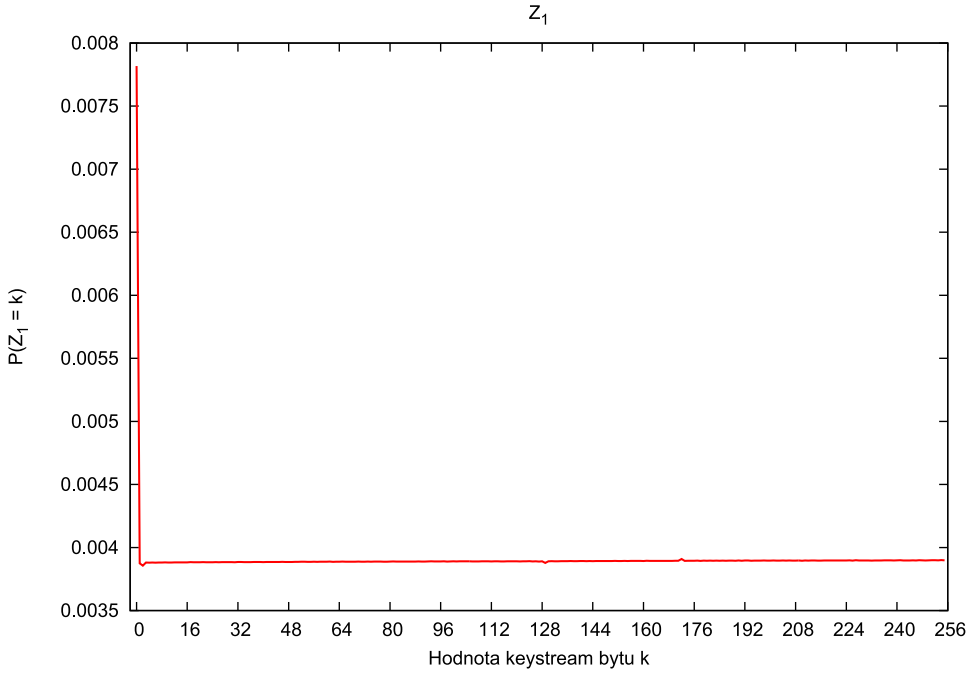
nastávají se zhruba dvojnásobnou pravděpodobností, než je jejich očekávaná hodnota. Tyto jevy teoreticky objasnil v roce 2001 Mantin ve své diplomové práci [24].

Sepehrdad, Vaudenay a Vuagnoux v roce 2010 [36] navrhli způsob, jak automaticky odhalovat lineární závislosti v průběhu PRGA. Díky tomu objevili velké množství závislostí vnitřního stavu na keystreamu. Tyto závislosti poté použili v kombinaci se slabinami KSA k útoku na WEP. O důkaz těchto závislostí se postarali Sen Gupta, Maitra, Paul a Sarkar [33].

2.3.2.3 Odchylky počátečních bytů keystreamu

Nejznámější slabinu šifry RC4 objevili v roce 2002 Mantin a Shamir [26]. Experimentálně zjistili velmi význačnou odchylku u druhého bytu keystreamu. Ten nabývá hodnoty 0 až s dvojnásobnou pravděpodobností, viz obrázek 2.3. Tato informace se dá použít k poměrně snadnému rozlišení šifry RC4 od náhodného proudu bitů.

Ve své práci si kladou otázku, proč tato odchylka nebyla nalezena dříve, když RC4 byla tou dobou již podrobena velkému množství statistických testů. Dospěli k závěru, že dřívější testy byly zaměřené spíše na menší množství použitých klíčů a delší řetězy vygenerovaného keystreamu, kde se tento druh odchylky nedá dost dobře změřit. Jejich práce tak vzbudila všeobecný zájem o hledání dalších podobných odchylek v šifře RC4.

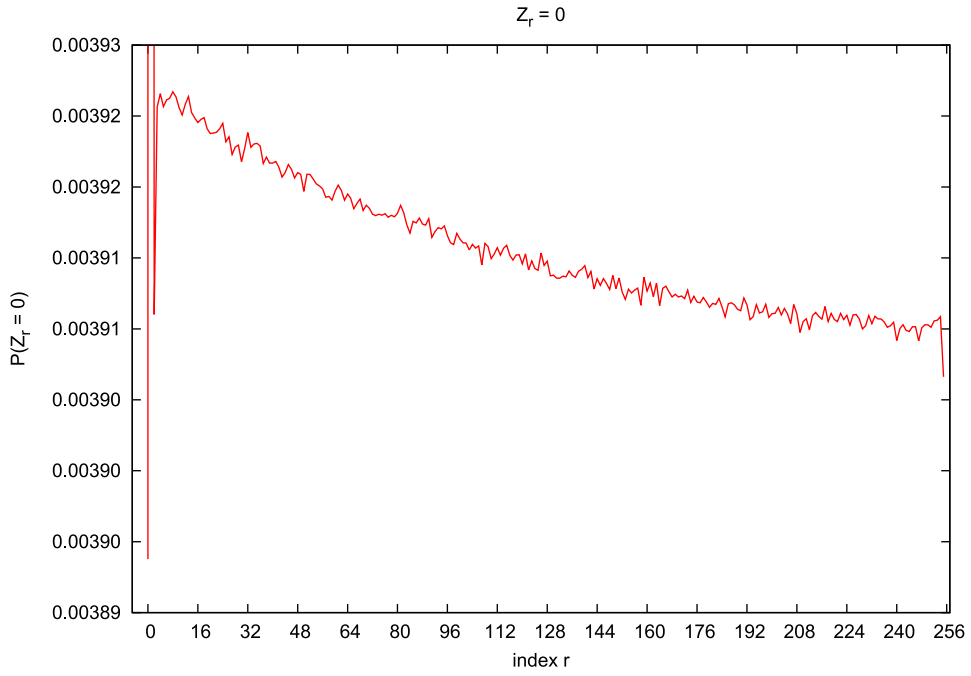


Obrázek 2.3: Rozdělení keystream bytů pro Z_1

Vzhledem k objeveným odchylkám v počátečních bytech se Mironov v roce 2002 pokusil stanovit počet bytů, které je potřeba pro zvýšení bezpečnosti šifry RC4 na začátku šifrování zahodit [29]. Identifikoval slabiny v zamíchání bytů u KSA a PRGA. Dále objevil odchylku u prvního bytu keystreamu. Doporučil zahazovat alespoň 756 bytů počátečního keystreamu. Konzervativnější odhad, jenž uvedl, je $12 \cdot 256$ čili 3072 bytů.

Maitra, Paul a Sen Gupta [23] dále v roce 2011 dokázali, že druhý byte keystreamu není jediný, který má vyšší tendenci nabývat hodnoty 0. I další byty keystreamu (od 3. do 255.) mají kladnou odchylku k 0, přestože není tak výrazná jako u druhého bytu. Na obrázku 2.4 lze vidět pravděpodobnost s jakou byte Z_r bude mít hodnotu 0. Obrázek je zúžen v ose y, kvůli příliš velké odchylce v Z_1.

V roce 2013 Isobe, Ohigashi, Watanabe a Morii objevili nové druhy odchylek v počátečních bytech [16]. Zajímavá je kladná odchylka $Z_r = r$ pro $r \in \langle 3, 255 \rangle$. Později téhož roku se AlFardan, Bernstein, Paterson, Poettering a Schuldt [2] pokusili provést kompletní analýzu všech odchylek v počátečních bytech keystreamu. Kromě již objevených odchylek se jim podařilo experimentálně objevit několik nových.



Obrázek 2.4: Pravděpodobnost, že keystream byte Z_r bude mít hodnotu 0

2.3.2.4 Dlouhodobé odchylky keystreamu

Potom, co bylo objeveno, že počáteční byty keystreamu šifry RC4 obsahují spoustu odchylek, začalo být obecně doporučeno při použití šifry RC4 zahazovat prvních N bytů. Proto se kryptoanalytici zabývali také dlouhodobými odchylkami šifry RC4, které se vyskytují po celé délce keystreamu.

V roce 2000 objevili Fluhrer a McGrew [12] dlouhodobé odchylky pro různé dvojice bytů. Na zmenšené verzi RC4 určili pravděpodobnostní rozdělení pro všechny možné hodnoty dvou po sobě jdoucích bytů keystreamu (Z_r, Z_{r+1}). Tímto způsobem identifikovali značnou množinu odchylek, kterou je možné rozšířit zpět na úplnou verzi RC4. Dále popsali, jak tyto odchylky mohou sloužit jako rozlišovač pro RC4, který potřebuje $2^{30.6}$ bytů keystreamu.

Další podobný druh odchylek objevil v roce 2005 Mantin [25]. Ten popsal kladnou odchylku vůči vzoru $ABSAB$, kde A a B jsou hodnoty bytů a S je krátký řetězec bytů, který může nabývat i délky 0. S tím, že čím kratší tento řetězec je, tím více významná je daná odchylka. Opět je možné tyto odchylky použít pro rozlišení RC4 od náhodného streamu bytů. Stačí k tomu 2^{26} bytů šifrového textu.

Dále Sen Gupta, Maitra, Paul a Sarkar [34, 32] objevili a později i teoreticky dokázali odchylku ve tvaru $Z_r = 0 \wedge Z_{r+2} = 0$.

2.3.3 Útoky na broadcast RC4

Pod pojmem útok na broadcast RC4 si představíme situaci, kdy jeden otevřený text je zašifrovaný vícekrát pomocí rozdílných klíčů, což nám vyprodukuje odpovídající množství šifrových textů. Tyto šifrové texty může útočník zachytit, třeba pomocí MiM útoku, a pokouší se z nich obnovit původní otevřený text.

První praktický útok na broadcast RC4 prezentovali v roce 2002 Mantin a Shamir [26]. Cílem útoku je pouze druhý byte, u kterého objevili odchylku ($Z_1 = 0$). Vzhledem k této odchylce, je při zachycení dostatečného množství šifrových textů (alespoň 256) velmi pravděpodobné, že hodnota bytu, která se na druhé pozici nejčastěji opakuje, je právě hodnota druhého bytu otevřeného textu.

Maitra, Paul a Sen Gupta poté v roce 2011 přišli s vylepšeným útokem [23], když dokázali, že byty následující druhý byte mají pozitivní odchylku k hodnotě 0. Myšlenku útoku z [26] tedy aplikovali i na následující byty, viz Algoritmus 2.1, teoreticky tak útok mohl obnovit dalších 253 bytů otevřeného textu. Pseudokód útoku jsem převzal z [2].

Praktická funkčnost tohoto útoku byla vyvrácena v [2], protože jak bylo později ukázáno, počáteční byty keystreamu RC4 obsahují i další odchylky, některé srovnatelně velké nebo i vetší než $Z_r = 0$. Útok tak velmi často určí odhadovaný byte otevřeného textu nesprávně.

Algoritmus 2.1 Základní útok na broadcast RC4 [23]

Vstup:
S nezávislých šifrových textů $(C_j)_{1 \leq j \leq S}$ otevřeného textu P
pozice r

Výstup: odhad P_r^* pro byte otevřeného text P_r

1: $N_{\texttt{0x00}} \leftarrow 0, \ldots, N_{\texttt{0xFF}} \leftarrow 0$
2: **for** $j = 1$ **to** S **do**
3: $\quad N_{C_{j,r}} \leftarrow N_{C_{j,r}} + 1$
4: **end for**
5: $P_r^* \leftarrow \arg\, max_{\mu \in \{\texttt{0x00},\ldots,\texttt{0xFF}\}} N_\mu$

Další útok navrhli v roce 2013 Isobe, Ohigashi, Watanabe a Morii [16]. Ten se snaží prvních 257 bytů otevřeného textu určit podobně jako předchozí útok, bere však v úvahu i jiné druhy odchylek než $Z_r = 0$. Následující byty otevřeného textu se útok snaží obnovit s využitím odchylky typu $ABSAB$ objevené v [25]. Útok vychází z předpokladu, že 3 ze 4 bytů na specifických pozicích jsou známé a díky tomu dokáže určit zbývající 4. byte. Pro pravděpodobnost úspěchu útoku blížící se 100 % je potřeba znát alespoň 2^{34} šifrových textů. Nevýhodou útoku je, že přestane fungovat v okamžiku, když dojde k zahození prvních 256 bytů keystreamu RC4.

Dva druhy útoků navrhli ve své práci AlFardan, Bernstein, Paterson, Poettering a Schuldt [2]. Jedná se o doposud nejlepší publikované útoky na broad-

cast RC4. Jejich bližší popis a ověření funkčnosti jsou náplní kapitoly 3.

AlFardan a kol. se tyto útoky v další části práce úspěšně pokusili aplikovat na komunikaci přes TLS a WPA. Počet šifrových textů pro úspěšné provedení prvního útoku s vysokou pravděpodobností je mezi 2^{28} a 2^{32}. Pro druhý útok je nutný počet kolem $13 \cdot 2^{30}$. Oba útoky se tedy pohybují na hranici praktičnosti.

Kapitola 3

Ověření útoků na RC4

AlFardan, Bernstein, Paterson, Poettering a Schuldt v roce 2013 navrhli dva praktické útoky na broadcast RC4 [2]. V této kapitole uvedu popis a vysvětlení principů obou útoků, tak jak jsou uvedené v [2]. Potom provedu vlastní implementaci těchto útoků a ověřím jejich funkčnost. Zároveň rozeberu složitost obou útoků a změřím jejich úspěšnost v závislosti na množství použitých šifrových textů a případně dalších vlastnostech.

Útoky se soustředí na obnovení části otevřeného textu, nedokáží tedy obnovit použitý klíč při šifrování. První útok je zaměřený na počátečních 256 bytů šifrových textů. Snaží se obnovovat postupně jeden byte otevřeného textu za druhým nezávisle na sobě, proto pro něj budeme používat pojmenování „jednobytový útok“. Druhý útok budeme označovat jako „dvoubytový útok“, protože pro nalezení nejpravděpodobnějšího otevřeného textu se zaměřuje na pravděpodobnost výskytu dvou po sobě jdoucích bytů.

3.1 Jednobytový útok

Princip tohoto útoku je založený na dříve navržených útocích z [26, 23]. Pro určení správného bytu však používá všechny odchylky v počátečních bytech RC4 keystreamu. Je tedy svým principem podobný první části útoku, který navrhli Isobe a kol. v [16], přestože oba útoky vznikly nezávisle na sobě. Útok je v [2] popsán následovně.

Prerekvizitou pro provedení útoku je znalost pravděpodobnostního rozdělení všech hodnot keystreamu pro požadovných r pozic, kde r značí počet bytů, které se pokusíme útokem obnovit. Toto rozdělení můžeme získat opakovaným generováním keystreamu RC4 s použitím velkého množství náhodných a vzájemně nezávislých klíčů. To znamená, že pokud chceme útočit na prvních 256 bytů šifry RC4, tak pro všechna $r \in \langle 0, 255 \rangle$ experimentálně určíme:

$$p_{r,k} := P(Z_r = k), k = \texttt{0x00}, \ldots, \texttt{0xFF},$$

s tím, že se omezíme na náhodné klíče s konstantní délkou, například 128 bitů. Pomocí tohoto rozdělení se můžeme pokusit obnovit otevřený text za znalosti dostatečného množství šifrových textů.

Řekněme, že máme k dispozici S šifrových textů $C_1, \ldots, C_S$. Pro zvolenou pozici r a všechny kandidáty na byte otevřeného textu μ mějme pole hodnot $(N^{(\mu)}_{\texttt{0x00}}, \ldots, N^{(\mu)}_{\texttt{0xFF}})$, kde

$$N^{(\mu)}_k = |\{j \mid C_{j,r} = k \oplus \mu\}_{1 \le j \le S}| \qquad (\texttt{0x00} \le k \le \texttt{0xFF}).$$

Toto pole hodnot potom odpovídá rozdělení Z_r bytů, které bychom potřebovali, abychom zašifrovali μ na dané šifrové byty $\{C_{j,r}\}_{1 \le j \le S}$. Porovnáním tohoto pole pro všechny kandidáty μ s rozdělením $p_{r,\texttt{0x00}}, \ldots, p_{r,\texttt{0xFF}}$ stanovíme $P_r = \mu$, tedy pravděpodobnost, že na r-té pozici otevřeného textu se nachází právě byte μ.

Dále pozorujeme, že pravděpodobnost λ_μ, že byte otevřeného textu μ je zašifrován na šifrové texty $\{C_{j,r}\}_{1 \le j \le S}$ se chová jako multinomiální rozdělení a může být přesně určena jako

$$\lambda_\mu = \frac{S!}{N^{(\mu)}_{\texttt{0x00}}! \ldots, N^{(\mu)}_{\texttt{0xFF}}!} \prod_{k \in \{\texttt{0x00},\ldots,\texttt{0xFF}\}} p_{r,k}^{N^{(\mu)}_k}.$$

Nejpravděpodobnější byte otevřeného textu potom určíme tím, že vypočteme λ_μ pro všechna $\texttt{0x00} \le \mu \le \texttt{0xFF}$ a najdeme mezi nimi μ s maximální hodnotou λ_μ.

Všimneme si, že pro každou pozici r a šifrové texty $\{C_{j,r}\}_{1 \le j \le S}$ můžeme hodnoty $N^{(\mu)}_k$ spočítat z hodnot $N^{(\mu')}_k$ jako $N^{(\mu)}_k = N^{(\mu')}_{k \oplus \mu' \oplus \mu}$. Jinými slovy $(N^{(\mu)}_{\texttt{0x00}}, \ldots, N^{(\mu)}_{\texttt{0xFF}})$ a $(N^{(\mu')}_{\texttt{0x00}}, \ldots, N^{(\mu')}_{\texttt{0xFF}})$ jsou vzájemné permutace, proto ze vzorce pro výpočet λ_μ můžeme vynechat část $S!/(N^{(\mu)}_{\texttt{0x00}}! \ldots, N^{(\mu)}_{\texttt{0xFF}}!)$, aniž by to ovlivnilo maximální hodnotu λ_μ. Pro další zjednodušení výpočtu můžeme počítat $\log(\lambda_\mu)$ namísto λ_μ.

Pseudokód útoku viz Algoritmus 3.1.

3.1.1 Složitost útoku

Zamysleme se nyní nad paměťovou a časovou složitostí celého útoku. Útok můžeme rozdělit do tří následujících částí:

1. vygenerování pravděpodobnostního rozdělení keystreamu,
2. zachycení a zpracování šifrových textů,
3. nalezení nejpravděpodobnějšího otevřeného textu za použití dat z předchozích bodů.

Algoritmus 3.1 Jednobytový útok na broadcast RC4 [2]

Vstup:
$\{C_{j,r}\}_{1 \leq j \leq S}$ – S nezávislých šifrových textů otevřeného textu P
r - pozice bytu
$(p_{r,k})_{(\texttt{0x00} \leq k \leq \texttt{0xFF})}$ - rozdělení keystream bytů na pozici r
Výstup: odhad P_r^* pro byte otevřeného textu P_r

1: $N_{\texttt{0x00}} \leftarrow 0, \ldots, N_{\texttt{0xFF}} \leftarrow 0$
2: **for** $j = 1$ to S **do**
3: $\quad N_{C_{j,r}} \leftarrow N_{C_{j,r}} + 1$
4: **end for**
5: **for** $\mu = \texttt{0x00}$ **to** $\texttt{0xFF}$ **do**
6: $\quad$ **for** $k = \texttt{0x00}$ **to** $\texttt{0xFF}$ **do**
7: $\quad\quad N_k^{(\mu)} \leftarrow N_{k \oplus \mu}$
8: $\quad$ **end for**
9: $\quad \lambda_\mu \leftarrow \sum_{k=\texttt{0x00}}^{\texttt{0xFF}} N_k^{(\mu)} \log p_{r,k}$
10: **end for**
11: $P_r^* \leftarrow \arg\, max_{\mu \in \{\texttt{0x00},\ldots,\texttt{0xFF}\}} \lambda_\mu$
12: **return** P_r^*

Potřebná délka keystreamu v první části útoku nebyla v původní práci [2] přesně stanovena. Logicky chceme, aby tato hodnota byla co nejvyšší, aby se odchylky v keystreamu co nejvíce projevily. Jsme tedy v tomto případě omezeni pouze vlastními zdroji, protože generování je časově náročné. Výhodou je, že tuto tabulku nám stačí vygenerovat pouze jednou, potom ji můžeme použít u všech provedených útoků.

Složitost z hlediska času je lineárně závislá na tom, kolik nezávislých náhodných klíčů K použijeme. Generujeme keystream byte pro prvních 256 pozic, proto jí můžeme stanovit jako $\mathcal{O}(2^8 K)$. Autoři útoku ve své práci [2] použili pro vygenerování rozdělení keystream bytů celkem 2^{44} nezávislých náhodných klíčů. Dohromady tedy vygenerovali 2^{52} bytů.

Paměťová složitost však není tak velká, protože nemusíme ukládat celé keystreamy. Stačí si nám pamatovat, kolikrát se pro každou pozici vyskytla každá hodnota bytu. Z této hodnoty můžeme potom vypočítat $p_{r,k}$. Máme tedy 2^8 možných hodnot a 2^8 požadovaných pozic, proto potřebujeme celkem uložit 2^{16} hodnot.

Druhá část útoku má stejnou časovou i paměťovou složitost jako ta první. Tuto část ovšem musíme provádět při každém útoku. Namísto generování keystreamu v této části zachytáváme šifrové texty. Ty opět nemusíme ukládat celé, ale stačí nám čítače pro pozici a byte textu.

Složitost třetí fáze útoku je v porovnání s předchozími zanedbatelná. Ze souboru načteme rozdělení z první části, tj. 2^{16} hodnot. Potom načteme rozdělení bytů v šifrových textech z druhé části, opět 2^{16} hodnot. Dále je doba útoku závislá na počtu bytů n, které chceme obnovit. Pro každý byte určíme

jeho hodnotu po zašifrování všemi možnými byty klíče, tj. 2^8 hodnot. Každou tuto hodnotu vynásobíme pravděpodobností a spočteme jejich sumu (λ_μ), to znamená, že provedeme dalších 2^8 operací. Ze všech bytů n najdeme ten s nejvyšší hodnotou, takže hledáme maximum z pole n hodnot. Když toto všechno sečteme dohromady, dostaneme složitost:

$$\mathcal{O}(2^{16} + 2^{16} + 2 \cdot 2^8 \cdot n + n) = \mathcal{O}(2^{17} + 2^9 \cdot n + n).$$

A protože při útoku nikdy neobnovujeme více než 256 (2^8) bytů, můžeme horní mez útoku určit jako:

$$\mathcal{O}(2^{17} + 2^9 \cdot 2^8 + 2^8) = \mathcal{O}(2^{18}).$$

Časově nejnáročnější část celého útoku je generování pravděpodobnostního rozdělení keystreamu (první část). To nám však stačí vygenerovat pouze jednou a navíc si ho můžeme připravit před provedením útoku. Proto můžeme první část při odhadování celkové složitosti zanedbat.

Podle [2] potřebujeme pro úspěšné určení otevřeného textu zhruba kolem 2^{30} šifrových textů. Zachytávání těchto textů má pro nás lineární časovou náročnost. Vidíme tedy, že celková složitost jednobytového útoku je přímo závislá na množství potřebných šifrových textů k provedení útoku, protože maximální složitost třetí části útoku se nám podařilo vyjádřit konstantou (2^{18}) mnohem menší než 2^{30}.

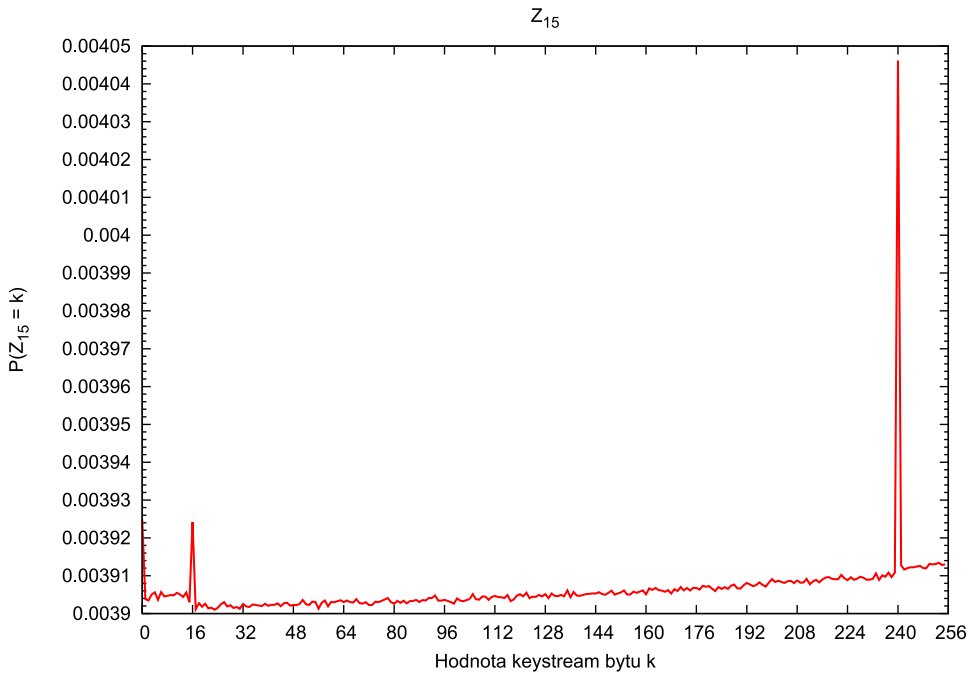
Paměťová složitost útoku je $\mathcal{O}(2^{17})$ hodnot, protože potřebujeme v paměti uchovat dvě pole o velikosti 2^{16} hodnot. Ostatní prvky mají všechny řádově menší velikost. Pokud bychom chtěli určit přesnou velikost paměti, kterou útok potřebuje, museli bychom dále určenou složitost vynásobit velikostí paměti potřebné pro uložení jedné hodnoty.

3.1.2 Implementace útoku

Pro implementaci potřebných programů jsem použil programovací jazyk `C++` s kompilátorem `g++` verze `4.8.1`. Jako testovací stroj jsem použil notebook s čtyřjádrovým procesorem Intel Core i7 (Ivy Bridge), 8 GB RAM a 64bitovým operačním systémem `Ubuntu 13.10`.

Nejdříve ze všeho jsem naimplementoval vlastní verzi šifry RC4. Její funkčnost jsem ověřil pomocí testovacích vektorů. Výstup jsem porovnával s výstupem z programu OpenSSL. Tím jsem ověřil, že má implementace pracuje správně.

Potom jsem program modifikoval do druhé verze, která slouží pro generování tabulky rozdělení keystream bytů $p_{r,k}$, viz `rc4-sbaKG.cpp`. Program generuje keystream byty pro náhodné klíče, jejichž počet se programu předává jako vstupní argument. Jednotlivé keystream byty zpracovává ve formě čítače, kde rozhodující je index a hodnota bytu. Pro přehlednost uvádím pseudokód



Obrázek 3.1: Rozdělení keystream bytů pro Z_{15}

3.2. Výstupem programu jsou výsledné hodnoty čítačů. Z jednotlivých hodnot můžeme vypočítat příslušnou pravděpodobnost tím, že hodnotu vydělíme celkovým počtem použitých klíčů.

Všimněme si, že generování rozdělení keystream bytů je velmi snadno škálovatelné. Jelikož jsou výstupem hodnoty čítačů, můžeme vygenerovaná rozdělení sjednotit jednoduchým součtem odpovídajících hodnot. Pro sjednocení rozdělení jsem vytvořil program `mergeKD.cpp`. Generování jsem tak velmi jednoduše zparalelizoval pomocí skriptu, který spustí několik procesů generování najednou a jejich výsledky sjednotí.

Pro všechny klíče jsem použil konstantní délku 16 bytů (128 bitů). Hodnotu klíče generuji pomocí náhodného generátoru byte po bytu. Je velmi důležité, aby použitý náhodný generátor měl dostatečně dlouhou periodu. Pokud například chceme použít dohromady 2^{32} odlišných klíčů, kde každý klíč má 2^4 bytů, potřebujeme generátor s periodou delší než je 2^{36} hodnot. Jinak by se nám klíče začaly opakovat a vygenerované rozdělení by nemělo odpovídající hodnotu. Já jsem v programu použil náhodný generátor `Mersenne Twister`, který je součástí knihovny `<random>` aktuálního standardu `C++11`.

Jako semínko (seed) pro náhodný generátor jsem použil funkci `time`, která vrací aktuální časovou známku v systému (unix timestamp). Předvídatelnost zvoleného semínka nás v tomto případě nemusí zajímat, důležité je pouze, aby

se seed při každém spuštění programu lišil, jinak by došlo k vygenerování identického klíče a tím pádem i keystreamu. Při spouštění více procesů generování najednou musíme tedy dávat pozor, abychom procesy generování nespustili hned po sobě (ve stejnou vteřinu).

Pro vygenerování testovacího rozdělení jsem použil 2^{34} náhodných klíčů. Pomocí vygenerovaných dat můžeme mimo jiné ověřit odchylky v keystreamu, viz obrázky 2.3, 2.4 a 3.1. Připomínám, že pozici bytu v keystreamu indexuji od 0. Na obrázku 2.3 lze pozorovat až dvojnásobnou odchylku druhého bytu keystreamu ($Z_1 = 0$). Na obrázku 3.1 pro Z_{15} je vidět odchylka závislá na délce klíče 16 bytů ($Z_{x \cdot K} = 256 - x \cdot K$) a odchylka závislá na indexu ($Z_r = r$).

Poznámka: Pokud pro generování použijeme velké množství klíčů, přesáhne jejich počet maximální hodnotu `unsigned int` ($2^{32} - 1$), proto je potřeba dávat pozor, jakým způsobem data ukládáme. V implementaci jsem použil typ long, který má na použitém systému velikost 8 bytů (64 bitů), velikost je tak dostatečná i pro mnohem větší množství klíčů.

Algoritmus 3.2 Generování pravděpodobnostního rozdělení keystream bytů

Vstup: K – počet klíčů

Výstup: $p_{r,k}$ - rozdělení keystream bytů, r – index, k – hodnota

1: $p[r][k] \leftarrow 0$ **for all** $0 \leq r, k \leq 255$
2: **for** $cnt = 0$ **to** K **do**
3: $KSA(newKey())$
4: **for** $r = 0$ **to** 255 **do**
5: $Z_r = PRGA()$
6: $p[r][Z_r] \leftarrow p[r][Z_r] + 1$
7: **end for**
8: **end for**
9: **return** $p_{r,k}$

Další program (`rc4-sba.cpp`) vznikl modifikací původní implementace šifry RC4. Program jsem přepsal tak, aby otevřený text šifroval vícekrát s různými náhodnými klíči a aby místo šifrového textu vracel tabulku rozdělení bytů u šifrových textů vzhledem k pozicím bytů. Vstupem programu je počet provedených šifrování a otevřený text. Výstupem jsou hodnoty čítačů stejně jako u předchozího programu.

Pro první pokusy jsem použil text, který se skládá pouze z písmene „a“ opakovaného 256krát. Pro otestování funkčnosti útoku jsem data připravil z 2^{30} šifrových textů generovaných nahodnými klíči.

Program `sba.cpp`, určený pro obnovu otevřeného textu ze zachycených dat jsem naprogramoval podle pseudokódu (Algoritmus 3.1). Z nachystaných testovacích dat se podařilo z 256 bytů úspěšně obnovit 207 bytů. Kdybychom byty vybírali náhodně, v průměru bychom ze 256 hodnot trefili jen 1 správně. To je zcela jasná ukázka toho, že útok skutečně funguje.

Počet šifrových textů	2^{20}	2^{24}	2^{26}	2^{28}	2^{30}	2^{32}
Úspěšnost pro prvních 16 bytů v %	12	41	72	87	94	100
Úspěšnost pro prvních 32 bytů v %	8	40	73	91	97	100
Úspěšnost pro prvních 64 bytů v %	5	31	71	95	98	100
Úspěšnost pro prvních 128 bytů v %	4	21	56	92	99	100
Celková úspěšnost útoku v %	3	12	31	57	81	97

Tabulka 3.1: Úspěšnost jednobytového útoku v závislosti na počtu šifrových textů při použití 2^{34} různých klíčů pro vygenerování rozdělení keystreamu

3.1.3 Měření statistik útoku

Nejprve jsem změřil čas potřebný pro provedení útoku. Vygenerování 2^{30} šifrových textů v jednom procesu zabere 2 hodiny. Nalezení kandídáta na otevřený text zabere vždy méně než 1 sekundu.

Dále jsem se zaměřil na testování úspěšnosti útoku z hlediska různých aspektů. Jako první jsem změřil úspěšnost útoku v závislosti na počtu vygenerovaných šifrových textů. Pro všechny zvolené hodnoty jsem měření zopakoval osmkrát a výsledky jsem zprůměroval. Naměřenou úspěšnost lze vidět v tabulce 3.1.

Když si prohlédneme ještě jednou pravděpodobnostní rozdělení keystream bytů, zjistíme, že odchylky u počátečních bytů jsou mnohem výraznější než odchylky u pozdějších bytů. Proto jsem do tabulky kromě celkové úspěšnosti přidal i řádky vyjadřující úspěšnost pro prvních n bytů, kde n jsem zvolil jako mocniny 2 počínaje 16. Díky tomu si můžeme všimnout několika zajímavých faktů:

- Stačí nám zachytit 2^{26} šifrových textů, abychom úspěšně obnovili prvních 128 bytů otevřeného textu s pravděpodobností větší než 50 %.
- Při zachycení 2^{28} šifrových textů, kde celková úspěšnost dosáhla 57 %, je úspěšnost obnovení prvních 128 bytů až 92 %.
- Znalost 2^{32} šifrových textů nám zaručuje úspěšné obnovení otevřeného textu, s výjimou několika bytů.

Dále jsem změřil závislost úspěšnosti útoku na počtu klíčů použitých pro vygenerování pravděpodobnostního rozdělení keystreamu. V [2] nebyla uvedena žádná požadovaná hodnota, proto jsem se tímto pokusil určit absolutní minimum, pro které mají pokusy o obnovu otevřeného textu ještě smysl. Pod tím si představuji alespoň 50% celkovou úspěšnost útoku. Odhadem bych tuto hodnotu umístil někam mezi 2^{26} a 2^{32}.

Pro měření jsem použil 2^{30} šifrových textů z předchozího měření. Výsledek měření viz tabulka 3.2. Všimneme si, že i když použijeme rozdělení vygene-

Počet klíčů	2^{26}	2^{28}	2^{30}	2^{32}	2^{34}	2^{36}
Úspěšnost pro prvních 16 bytů v %	62	71	77	90	94	99
Úspěšnost pro prvních 32 bytů v %	55	73	81	94	97	99
Úspěšnost pro prvních 64 bytů v %	50	76	90	97	98	99
Úspěšnost pro prvních 128 bytů v %	30	58	86	97	99	99
Celková úspěšnost útoku v %	16	31	50	65	81	89

Tabulka 3.2: Úspěšnost jednobytového útoku pro 2^{30} šifrových textů v závislosti na počtu klíčů použitých pro vygenerování pravděpodobnostního rozdělení keystreamu

rované s 2^{26} náhodnými klíči, pořád jsme schopni obnovit prvních 64 bytů otevřeného textu s 50% pravděpodobností. Pro dosažení celkové úspěšnosti 50 % je zapotřebí použít alespoň 2^{30} klíčů, tuto hodnotu bych označil jako minimum, pro které ještě dosáhneme přijatelné úspěšnosti.

Jako další jsem ověřil, jak moc se zlepší úspěšnost obnovy otevřeného textu, pokud se omezíme pouze na určitou sadu znaků. Původní program (`sba.cpp`) jsem modifikoval tak, aby uvažoval, že se v otevřeném textu mohou vyskytovat pouze malá písmena abecedy.

Omezením znakové sady se celková úspěšnost obnovení otevřeného textu zlepší v průměru o 34 %, viz tabulka 3.3. Pro malé množství šifrových textů (2^{20}) však toto zlepšení nemá moc velký význam, protože v průměru obnovíme o 8 znaků více. Zároveň pro velké množství šifrových textů (2^{32}) dosahuje útok tak velké úspěšnosti, že už se téměř nemůže dál zlepšovat.

Proto za více relevantní průměr považuji hodnoty pro 2^{26} až 2^{30} šifrových textů, kde je průměrné 12% zlepšení. To je poměrně velká hodnota. Ukazuje nám, že pokud máme nějakou dodatečnou informaci o struktuře otevřeného textu, můžeme úspěšnost útoku dále vylepšit.

V [2] svůj útok zaměřili na komunikaci v TLS, proto se omezili pouze na délky klíče 16 bytů (128 bitů). Z toho důvodu jsem ověřil funkčnost útoku pro náhodnou délku klíče. Vygeneroval jsem keystream rozdělení pro 2^{32} různých klíčů o náhodné délce v intervalu 5 až 20 bytů. Potom jsem vygeneroval 2 rozdělení 2^{30} šifrových textů se stejným nastavením pro náhodné klíče.

S těmito daty jsem dosáhl průměrné úspěšnosti obnovy všech 256 bytů 65 %. Pro prvních 128 bytů byla úspěšnost 99 %. Dále jsem použil rozdělení šifrových textů 2^{30} z předchozích měření (s klíčem vždy o délce 16 bytů) a dosáhl jsem 51% úspěšnosti pro 256 bytů a 76% úspěšnosti pro 128 bytů. Vidíme tedy, že jednobytový útok může fungovat i v případě, kdy neznáme délku použitého klíče při šifrování. Předpokládám, že úspěšnost útoku by se dále zvýšila, pokud bych pro vygenerování rozdělení keystream bytů použil ještě větší množství klíčů.

Počet šifrových textů	2^{20}	2^{24}	2^{26}	2^{28}	2^{30}	2^{32}
Úspěšnost původního útoku v %	3	12	31	57	81	97
Úspěšnost modifikovaného útoku v %	6	20	38	62	84	98
Zlepšení v %	100	67	23	9	4	1

Tabulka 3.3: Srovnání úspěšnosti jednobytového útoku při použití omezené abecedy s původním algoritmem

3.1.4 Vyhodnocení útoku

Ukázali jsme si, že počáteční byty šifry RC4 obsahují kritické slabiny. Při znalosti dostatečného množství šifrových textů, generovaných s různými klíči pro stejný otevřený text, jsme schopni tento text obnovit. Stačí nám znalost 2^{30} šifrových textů, abychom obnovili otevřený text dlouhý 256 bytů s pravděpodobností 50 % a prvních 128 bytů textu dokonce s pravděpodobností 99 %. Útok jsme navíc schopni s vysokou pravděpodobností úspěchu provést i bez znalosti délky použitého klíče při šifrování.

Množství potřebných šifrových textů je na hranici praktičnosti. Při měření mi vygenerování 2^{30} šifrových textů zabralo zhruba 2 hodiny. Jedná se však o laboratorní hodnotu. V průběhu skutečné komunikace by, v závislosti na způsobu generování šifrových textů, tato doba byla pravděpodobně mnohem delší. Přesto si myslím, že skutečná komunikace může být tímto útokem ohrožena, pokud nedojde k vynechání počátečních bytů.

Z tabulky 3.1 je vidět, že útok má větší než 50% pravděpodobnost úspěchu pro prvních 128 bytů při znalosti pouze 2^{26} šifrových textů, což je 16× méně šifrových textů než 2^{30}. Pokud by tedy generování skutečné komunikace bylo 16× pomalejší, stále jsme schopni útok provést s měřitelnými výsledky během dvou hodin.

Nezapomínejme dále, že útok je zaměřen čistě na slabiny šifry RC4, obsahuje tak prostor pro zlepšení. Pokud známe strukturu otevřeného textu můžeme jednak omezit abecedu otevřeného textu, zároveň však máme další možnosti vylepšení, jako je frekvenční analýza dvojic bytů dle struktury otevřeného textu a podobně. Úspěšnost útoku je tak možno dále vylepšit.

Na konci celého měření jsem ještě provedl test pro 2^{36} klíčů a 2^{32} šifrových textů. Dosažená úspěšnost byla 100 % s výjimkou posledních 3 bytů. Byte s indexem 254 byl obnovem s 50% pravděpodobností úspěchu. Byty s indexem 253 a 255 se nepodařilo obnovit vůbec. Předpokládám, že pro dosažení vyšší úspěšnosti obnovy u těchto bytů je potřeba použít ještě větší množství klíčů pro generování rozdělení keystream bytů.

Pár hodnot	Podmínka na index i	Pravděpodobnost
$(0, 0)$	$i = 1$	$2^{-16}(1 + 2^{-9})$
$(0, 0)$	$i \neq 1, 255$	$2^{-16}(1 + 2^{-8})$
$(0, 1)$	$i \neq 0, 1$	$2^{-16}(1 + 2^{-8})$
$(i + 1, 255)$	$i \neq 254$	$2^{-16}(1 + 2^{-8})$
$(255, i + 1)$	$i \neq 1, 254$	$2^{-16}(1 + 2^{-8})$
$(255, i + 2)$	$i \neq 0, 253, 254, 255$	$2^{-16}(1 + 2^{-8})$
$(255, 0)$	$i = 254$	$2^{-16}(1 + 2^{-8})$
$(255, 1)$	$i = 255$	$2^{-16}(1 + 2^{-8})$
$(255, 2)$	$i = 0, 1$	$2^{-16}(1 + 2^{-8})$
$(129, 129)$	$i = 2$	$2^{-16}(1 + 2^{-8})$
$(255, 255)$	$i \neq 254$	$2^{-16}(1 - 2^{-8})$
$(0, i + 1)$	$i \neq 0, 255$	$2^{-16}(1 - 2^{-8})$

Tabulka 3.4: Odchylky keystreamu RC4 pro dva po sobě následující byty, i je proměnná z vnitřního stavu RC4, tabulka převzata z [2]

3.2 Dvoubytový útok

Předchozí jednobytový útok je zaměřen pouze na prvních 256 bytů šifry RC4, proto ho při jejich zahození není možné dále použít. Největší výhoda dále popsaného dvoubytového útoku je, že dokáže obnovit část otevřeného textu na libovolné pozici i z pozdějších dat keystreamu. Nedokážeme se tak proti němu bránit, tím že zahodíme prvních $n \cdot 256$ bytů.

Navíc tento útok nepotřebuje, aby všechny šifrové texty vznikly použitím různých klíčů. Umožňuje použít data šifrovaná jedním klíčem, kde se námi napadená část otevřeného textu periodicky opakuje po 256 bytech.

Útok je založený na použití odchylek v pravděpodobnostním rozdělení keystreamu pro dvě po sobě jdoucí hodnoty bytů (Z_r, Z_{r+1}). Tyto odchylky objevili v roce 2000 Fluhrer a McGrew [12] a vyskytují se po celé délce keystreamu. Přehled těchto odchylek lze vidět v tabulce 3.4. Při rovnoměrném rozdělení by pro každou dvojici měla být pravděpodobnost výskytu 2^{-16}.

Předpokládá se, že ostatní dvojice hodnot jsou rozděleny rovnoměrně. Jinak řečeno jsme schopni stanovit pravděpodobnostní rozdělení p_{r,k_1,k_2} pro $1 \leq r \leq 256$, $\texttt{0x00} \leq k_1, k_2 \leq \texttt{0xFF}$ takové, že

$$\begin{aligned} p_{r,k_1,k_2} &= P[(Z_{256q+r}, Z_{256q+r+1}) = (k_1, k_2)] \\ &\approx P[(\texttt{RC4}(r, j, S), \texttt{RC4}^{+1}(r, j, S)) = (k_1, k_2)], \end{aligned}$$

kde první pravděpodobnost je myšlena pro všechny možné klíče RC4 a libovolné volby $q \in N$ a druhá pravděpodobnost představuje všechny možné konfigurace vnitřního stavu RC4 a $\texttt{RC4}(i, j, S), \texttt{RC4}^{+1}(i, j, S)$ představují dva

po sobě jdoucí byty, vystupující z RC4 PRGA. Podle výsledků z [12] by tyto pravděpodobnosti měly být přibližně stejné.

AlFardan a kol. popisují svůj dvoubytový útok v [2] následovně: Nechť L je nějaký celý násobek 256. Dále uvažujeme otevřený text P skládající se z L bytů takový, že $P = P_1||\ldots||P_L$ je zašifrován opakovaně s použitím jednoho klíče. Jinak řečeno máme šifrový text C, který jsme získali zašifrováním $P||\ldots||P$. (Pro provedení útoku dokonce stačí, aby se opakoval na stejné pozici pouze nějaký podřetězec.) Nechť C_j značí podřetězec C odpovídající j-té kopii P a $C_{j,r}$ značí r-tý byte v této kopii. To znamená, že $C_{j,r}$ odpovídá bytu C na indexu $(j-1)\cdot L + r$.

Potom se můžeme pokusit o obnovu otevřeného textu podobným způsobem jako u jednobytového útoku (algoritmus 3.1), že pro každou pozici r můžeme nejpravděpodobnější dvojici (μ_r, μ_{r+1}) spočítat ze šifrových textů $\{(C_{j,r}, C_{j,r+1})\}_{1\leq j\leq S}$ a pravděpodobnostního rozdělení $\{p_{r,k_1,k_2}\}_{\texttt{0x00}\leq k_1,k_2\leq\texttt{0xFF}}$. Jinak řečeno, kandidáta na dvojici bytů otevřeného textu bychom získali rozdělením šifrového textu C po dvojicích bytů a individuálně bychom spočítali nejpravděpodobnější dvojici pro každou pozici.

Můžeme však nalézt přesnějšího kandidáta na otevřený text, když budeme navíc uvažovat překrývající se páry bytů. Potom pro kandidáta na otevřený text $P' = \mu_1||\ldots||\mu_L$ spočítáme pravděpodobnost $\lambda_{P'} = \lambda_{\mu_1||\ldots||\mu_L}$, že P' je skutečný otevřený text pomocí rekurze:

$$\lambda_{\mu_1||\ldots||\mu_{\ell-1}||\mu_\ell} = \delta_{\mu_\ell|\mu_{\ell-1}} \cdot \lambda_{\mu_1||\ldots||\mu_{\ell-1}} \qquad (\ell \leq L),$$

kde $\delta_{\mu_\ell|\mu_{\ell-1}}$ značí pravděpodobnost, že $P_\ell = \mu_\ell$ za předpokladu, že $P_{\ell-1} = \mu_{\ell-1}$ a $\lambda_{\mu_1||\ldots||\mu_{\ell-1}}$ je stanovená pravděpodobnost, že $\mu_1||\ldots||\mu_{\ell-1}$ je odpovídajících $(\ell-1)$ bytů otevřeného textu P.

Tento výraz implicitně předpokládá, že μ_ℓ může záviset na $\mu_{\ell-1}$, ale nezávisí na žádném z předchozích bytů. Níže je ukázáno, jak můžeme spočítat $\delta_{\mu_\ell|\mu_{\ell-1}}$ ze znalosti bytů šifrového textu $\{(C_{j,\ell-1}, C_{j,\ell})\}_{1\leq j\leq S}$ a pravděpodobnostního rozdělení $\{p_{\ell-1,k_1,k_2}\}_{\texttt{0x00}\leq k_1,k_2\leq\texttt{0xFF}}$.

Popisovaný algoritmus hledá kandidáta na otevřený text $P^* = \mu_1||\ldots||\mu_\ell$ s maximální hodnotou λ_{P^*}. Základní myšlenkou algoritmu je iterativně skládat možné P^* z předchozích P^* kratších o jeden byte. Nové kandidáty delší o jeden byte dostaneme tím, že pro každou možnou hodnotu bytu zvolíme předchozího kandidáta, pro nějž má testovaný byte maximální hodnotu.

Algoritmus pro dosažení vyšší úspěšnosti předpokládá znalost prvního (μ_1) a posledního (μ_L) bytu hledaného otevřeného textu. Může být však modifikován i tak, aby tato znalost nebyla potřeba. Pokud neznáme μ_1, hledáme nejpravděpodobnější páry (μ_1, μ_2) pro všechny hodnoty μ_2. Poslední byte μ_L můžeme také najít vyzkoušením všech 256 možných hodnot.

Hodnotu pravděpodobnosti $\delta_{\mu_{i+1}|\mu_i}$ spočítáme podobně, jako při hledání maximální pravděpodobností v předchozím (jednobytovém) útoku (viz algoritmus 3.1). Každá možná kombinace indexu i, páru (μ_i, μ_{i+1}) a bytů šifrového

textu $\{(C_{j,i}, C_{j,i+1})\}_{1 \leq j \leq S}$ indukuje pravděpodobnostní rozdělení keystream bytů $\{(Z_{(j-1)L+i}, Z_{(j-1)L+i+1})\}_{1 \leq j \leq S}$. To můžeme reprezentovat jako pole hodnot $(N_{i,\texttt{0x00},\texttt{0x00}}, \ldots, N_{i,\texttt{0xFF},\texttt{0xFF}})$, kde

$$N_{i,k_1,k_2} = |\{1 \leq j \leq S | (C_{j,i}, C_{j,i+1}) = (k_1 \oplus \mu_i, k_2 \oplus \mu_{i+1})\}|.$$

Toto pole je, podobně jako u jednobytového útoku, multinomiálně rozdělené a pravděpodobnost, že $(N_{i,\texttt{0x00},\texttt{0x00}}, \ldots, N_{i,\texttt{0xFF},\texttt{0xFF}})$ nastane (tj. pravděpodobnost, že dvojice (μ_i, μ_{i+1}) je shodná s byty otevřeného textu s indexy i a $(i+1)$ je daná vzorcem:

$$P[P_i = \mu_i \wedge P_{i+1} = \mu_{i+1} | C] = \prod_{k_1,k_2 \in \{\texttt{0x00},\ldots,\texttt{0xFF}\}} p_{i,k_1,k_2}^{N_{i,k_1,k_2}}.$$

Poznámka: Z rovnice jsem vynechal část $S!/(N_{i,\texttt{0x00},\texttt{0x00}}!, \ldots, N_{i,\texttt{0xFF},\texttt{0xFF}}!)$, protože při porovnávání dvou pravděpodobností stejně jako v algoritmu 3.1 nehraje žádnou roli.

Můžeme tedy spočítat $\delta_{\mu_{i+1}|\mu_i}$ jako:

$$\begin{aligned} \delta_{\mu_{i+1}|\mu_i} &= P[P_{i+1} = \mu_{i+1} | P_i = \mu_i \wedge C] \\ &= \frac{P[P_i = \mu_i \wedge P_{i+1} = \mu_{i+1} | C]}{P[P_i = \mu_i | C]}. \end{aligned}$$

Předpokládáme, že v keystreamu nejsou přítomné žádné významné jednobytové odchylky, tj. že $P[P_i = \mu_i | C]$ je rovnoměrně rozdělená přes všechny možné hodnoty μ_i. Při platnosti této podmínky můžeme z rovnice dále vynechat $1/P[P_i = \mu_i | C]$, protože při porovnávání pravděpodobností nebude hrát roli.

Pseudokód dvoubytového útoku viz algoritmus 3.3.

3.2.1 Složitost útoku

Podobně jako u jednobytového útoku i zde provedu zhodnocení časové a paměťové složitosti. Celý útok si opět rozdělíme do tří částí:

1. vygenerování pravděpodobnostního rozdělení keystream bytů,
2. zachycení a zpracování šifrových textů,
3. nalezení nejpravděpodobnějšího otevřeného textu za použití dat z předchozích dvou bodů.

První část dvoubytového útoku není tak náročná jako u jednobytového, protože zde negenerujeme keystream experimentálně pro náhodné klíče, nýbrž použijeme teoreticky stanovené odchylky. Úkolem algoritmu je tedy pouze spočítat hodnotu pro každou položku generovaného pole. Časová složitost odpovídá počtu hodnot. Generujeme pole pro všech 256 (2^8) možných indexů

Algoritmus 3.3 Dvoubytový útok na broadcast RC4 [2]

Vstup:

C – S nezávislých šifrových textů otevřeného textu P

$C_{j,r}$ značí r-tý byte v C u j-té kopie P

L – délka P v bytech (musí být násobek 256)

μ_1 a μ_L – první a poslední byte P

$\{p_{r,k_1,k_2}\}_{1 \leq r \leq L-1, \texttt{0x00} \leq k_1,k_2 \leq \texttt{0xFF}}$ – pravděpodobnostní rozdělení keystreamu

Výstup: odhad P^* pro otevřený text P

// $max_2(Q)$ značí $(P, \lambda) \in Q$ takové, že $\lambda \geq \lambda' \forall (P', \lambda') \in Q$

1: $N_{r,k_1,k_2} \leftarrow 0$ **for all** $1 \leq r \leq L-1, \texttt{0x00} \leq k_1, k_2 \leq \texttt{0xFF}$
2: **for** $j = 1$ **to** S **do**
3: **for** $r = 1$ **to** $L-1$ **do**
4: $N_{(r,C_{j,r},C_{j,r+1})} \leftarrow N_{(r,C_{j,r},C_{j,r+1})} + 1$
5: **end for**
6: **end for**
7: $Q \leftarrow \{(\mu_1, 0)\}$
8: **for** $r = 1$ **to** $L-2$ **do**
9: $Q_{ext} \leftarrow \{\}$ // Seznam kandidátů otevřeného textu o délce $r+1$
10: **for** $\mu_{r+1} = \texttt{0x00}$ **to** $\texttt{0xFF}$ **do**
11: $Q_{\mu_{r+1}} \leftarrow \{\}$ // Seznam kandidátů končících bytem μ_{r+1}
12: **for each** $(P', \lambda_{P'}) \in Q$ **do**
13: $P' \rightarrow \mu_1 || \ldots || \mu_r$
14: $\lambda_{P'||\mu_{r+1}} \leftarrow \lambda_{P'} + \sum_{k_1=\texttt{0x00}}^{\texttt{0xFF}} \sum_{k_2=\texttt{0x00}}^{\texttt{0xFF}} N_{(r,k_1 \oplus \mu_r, k_2 \oplus \mu_{r+1})} \cdot \log p_{(r,k_1,k_2)}$
15: $Q_{\mu_{r+1}} \leftarrow Q_{\mu_{r+1}} \cup \{(P'||\mu_{r+1}, \lambda_{P'||\mu_{r+1}})\}$
16: **end for**
17: $Q_{ext} \leftarrow Q_{ext} \cup \{max_2(Q_{\mu_{r+1}})\}$
18: **end for**
19: $Q \leftarrow Q_{ext}$
20: **end for**
21: $Q_{\mu_L} \leftarrow \{\}$ // Seznam kanditátů otevřeného textu končících bytem μ_L
22: **for each** $(P', \lambda_{P'}) \in Q$ **do**
23: $P' \rightarrow \mu_1 || \ldots || \mu_{L-1}$
24: $\lambda_{P'||\mu_L} \leftarrow \lambda_{P'} + \sum_{k_1=\texttt{0x00}}^{\texttt{0xFF}} \sum_{k_2=\texttt{0x00}}^{\texttt{0xFF}} N_{(r,k_1 \oplus \mu_{L-1}, k_2 \oplus \mu_L)} \cdot \log p(r, k_1, k_2)$
25: $Q_{\mu_L} \leftarrow Q_{\mu_L} \cup \{(P'||\mu_L, \lambda_{P'||\mu_L})\}$
26: **end for**
27: $(P^*, \lambda_{P^*}) \leftarrow max_2(Q_{\mu_L})$
28: **return** P^*

a všechny možné dvojice keystream bytů, tj. 256^2 (2^{16}) hodnot. Paměťová a časová složitost první části je tedy 2^{24}.

U druhé části je složitost lineárně závislá na počtu a délce zachycených šifrových textů. Šifrové texty opět nemusíme ukládat celé, ale stačí nám pole čítačů pro všechny indexy a hodnoty keystream bytů. Paměťová složitost je tedy stejně jako u předchozího bodu 2^{24} hodnot. Časovou složitost druhé části útoku můžeme vyjádřit následovně.

Řekněme, že zachytíme k šifrových textů, každý šifrovaný pro stejný otevřený text odlišným klíčem. Šifrový text se skládá z n opakování 256bytového textu. Potom celkovou složitost zachycení a zpracování těchto dat vyjádříme jako $\mathcal{O}(2^8 \cdot k \cdot n)$. Pokud bychom navíc místo k šifrových textů uvažovali pouze jeden dlouhý zřetězený šifrový text, můžeme výslednou složitost zapsat jako $\mathcal{O}(2^8 \cdot n)$, což je stejná složitost jako u druhé části jednobytového útoku.

Třetí část útoku začíná načtením pole s pravděpodobnostním rozdělením z první části, potom načteme pole šifrových dat z druhé části. V obou případech máme konstantní složitost načtení 2^{24} hodnot. Dále je složitost závislá na délce obnovovaného textu.

Nejprve budeme uvažovat, že známe hodnotu prvního a posledního obnovovaného bytu. Na začátku uložíme do paměti prvního kandidáta na otevřený text, obsahující zatím pouze první známý byte. Potom iterativně budujeme nejpravděpodobnější kandidáty na otevřený text byte po bytu od délky 1 do $L - 2$, kde L je počet obnovovaných znaků (včetně těch známých).

Pro každou délku řetězce vybereme pro všech 256 možných hodnot bytu z paměti nejpravděpodobnějšího kandidáta tím, že ke všem kandidátům umístíme na konec zkoušený byte a vybereme kandidáta s maximální pravděpodobností. Pravděpodobnost spočítáme přes všechny dvojice bytů, tj. 2^{16} hodnot. Po prvním průběhu (nalezení kandidátů délky 2 byty) budeme v paměti vždy až do konce algoritmu uchovávat minimálně 256 kandidátů. Po doběhnutí hlavní smyčky zkusíme, který z kandidátů na otevřený text má nejvyšší pravděpodobnost, pokud by končil posledním bytem, jehož hodnotu známe.

Počet aktuálně uložených kandidátů pro nějakou délku může být větší než 256, protože teoreticky pro některý testovaný byte můžeme najít 2 a více kandidátů, pro které má testovaný byte stejnou pravděpodobnost a tato hodnota je maximální ze všech jeho pravděpodobností. V praxi však najdeme pro každý byte ve většině případů právě jednoho kandidáta s maximální hodnotou, proto budeme při odhadu složitosti počítat s 256 kandidáty.

Pokud bychom neznali hodnotu prvního bytu, složitost algoritmu se příliš nezmění. V prvním kroku, při budování kandidátů o délce 2 byty bychom vybírali místo z 1 možného kandidáta z 256 kandidátů. Při neznalosti posledního bytu se složitost algoritmu také moc nezvýší. Namísto zkoušení 1 hodnoty pro 256 kandidátů, zkoušíme všech 256 hodnot pro 256 kandidátů a z nich vybereme kandidáta s maximální pravděpodobností.

Celkovou složitost třetí části útoku při znalosti prvního a posledního bytu otevřeného textu tedy můžeme vyjadřit jako:

$$\mathcal{O}(2^{25} + (L-2) \cdot 2^8 \cdot 2^8 \cdot 2^{16} + 2 \cdot 2^{24}) \sim \mathcal{O}(2^{32}L),$$

kde L je počet bytů obnovovaného textu. Pokud neznáme ani počáteční ani koncový byte otevřeného textu je složitost rovná:

$$\mathcal{O}(2^{25} + L \cdot 2^8 \cdot 2^8 \cdot 2^{16}) \sim \mathcal{O}(2^{32}L).$$

Všimneme si, že vygenerování rozdělení keystreamu v první části útoku má stejnou složitost, jako jeho načtení ze souboru ve třetí části. Je tedy výhodnější toto rozdělení generovat při každém útoku znovu, protože načítání ze souboru při stejné složitosti bude vždy pomalejší.

Vidíme, že největší asymptotickou složitost má třetí část útoku. Počet bytů, které budeme útokem obnovovat však nebude větší než 16. Ve výsledku tedy provedeme 2^{36} výpočtů. Pro dosažení vysoké úspěšnosti potřebuje podle [2] útok zachytit kolem $12 \cdot 2^{30}$ šifrových textů. To pro 256 opakujících se bytů v textu dohromady dělá $3 \cdot 2^{40}$ operací. Provedení útoku tak bude časově záviset především na druhé části, tj. zachycení, případně generování a zpracování šifrových textů.

Celková paměťová složitost útoku je 2^{25} hodnot. 2^{24} hodnot pro uchování rozdělení pro dvojice keystream bytů a 2^{24} pro uchování pole čítačů u zachycených šifrových textů. Ostatní uložené hodnoty jsem pro větší jednoduchost zanedbal, žádná z nich nedosahuje velikosti srovnatelné s 2^{24}. Skutečná paměťová náročnost útoku je dále závislá na velikosti použitých typů pro uchování jednotlivých hodnot.

3.2.2 Implementace útoku

Pro implementaci dvoubytového útoku jsem použil zdrojové kódy z předchozího (jednobytového) útoku. Jejich modifikací jsem vytvořil potřebné programy pro otestování funkčnosti útoku.

Program `rc4-dba.cpp` má na starosti vygenerování a zpracování šifrových textů. Program přebírá v uvedeném pořadí 3 vstupní parametry:

1. počet opakování otevřeného textu při zašifrování,
2. počet klíčů použitých k šifrování,
3. cesta k souboru, který obsahuje otevřený text.

Výstupem programu jsou hodnoty čítačů (pole definované trojicí (i, k_i, k_{i+1}), kde i je index v keystreamu, k_i je hodnota prvního keystream bytu, k_{i+1} hodnota následujícího keystream bytu). Na začátku šifrování je zahozeno prvních 2048 bytů keystreamu, to představuje ochranu proti předchozímu, jednobytovému útoku.

Očekávaná délka otevřeného textu v programu je 256 bytů. Program je možné dále snadno modifikovat, aby umožňoval použití kratších a případně i delších otevřených textů.

Pro otestování funkčnosti útoku jsem vygeneroval $14 \cdot 2^{30}$ dat, tj. šifrové texty pro 14 různých klíčů, kde každý šifrový text se skládá z 2^{30} opakování otevřeného textu. Jako testovací otevřený text jsem opět použil 256 znaků „a“. Abych generování urychlil, celý proces jsem paralelizoval podobně jako u předchozího útoku.

Pro obnovení otevřeného textu slouží program `dba.cpp`. Vstupem programu je 5 parametrů:

1. index počátku obnovovaného textu,
2. index konce obnovovaného textu,
3. první znak obnovovaného textu,
4. poslední znak obnovovaného textu,
5. cesta k souboru, který obsahuje hodnoty čítačů pro šifrové texty (výstup programu `rc4-dba.cpp`).

Příklad: Program se po spuštění s parametry

```
./rc4-dba 0 9 a z ct.txt
```

pokusí obnovit otevřený text o délce 10 bytů za použití dat ze souboru `ct.txt`. Na index 0 dosadí znak „a“ a na index 9 znak „z“. Program se tedy snaží určit hodnotu zbylých 8 bytů. Výstupem programu je nejpravděpodobnější otevřený text.

První testování útoku dopadlo následovně. Pro $14 \cdot 2^{30}$ dat dokázal program z 8 odhadovaných znaků úspěšně obnovit 3 znaky, což je poměrně málo. V [2] uvádí pro $14 \cdot 2^{30}$ dat úspěšnost blízkou 100 %. Měření jsem zopakoval pro různé pozice v textu, ale průměrná úspěšnost se přibližovala pouze 40 %. Dále jsem útok otestoval na větším množství dat, pro $30 \cdot 2^{30}$ dosáhl 80% úspěšnosti.

Útok tedy funguje, ale není tak přesný, jak je uvedeno v [2]. Nabízí se nám jen dvě možná vysvětlení:

1. úspěšnost útoku není ve skutečnosti tak vysoká, jak je uvedeno v [2],
2. moje implementace útoku obsahuje nějakou nepostřehnutou chybu.

Pravděpodobnější je v tomto případě druhá možnost, proto jsem vynaložil veškeré úsilí k tomu, abych odstranil jakýkoliv zdroj nepřesnosti. Kromě programu `dba.cpp` jsem pečlivě zkontroloval, že generování šifrových textů pomocí programu `rc4-dba.cpp` neobsahuje žádnou chybu.

V programu pro obnovení otevřeného textu používám pro počítání pravděpodobností typ `double`, který má omezenou přesnost. Proto moje první hypotéza vedla k tomu, že za ztrátou přesnosti stojí právě použití typu `double`.

Nejprve jsem se pokusil typ `double` nahradit třídou pro reálná čísla s nastavitelnou přesností z opensource knihovny `mpfr`. Ukázalo se však, že použití této knihovny útok zpomalilo za hranici praktické použitelnosti.

Obnovení jediného bytu zabralo kolem 5 minut. Navíc při obnově jednoho bytu zkoušíme hodnoty pouze pro jednoho počátečního kandidáta, u všech následujících bytů by možných kandidátů bylo 256, proto by jejich obnovení trvalo mnohem déle. Možnost použití desetinných čísel s nastavitelnou přesností jsem tedy zavrhnul.

Jako další se nabízelo použít pro výpočet pravděpodobností vyjádření v racionální formě. Když se podíváme na odchylky využité v útoku, viz tabulka 3.4, vidíme, že je všechny dokážeme vyjádřit pomocí zlomku. Pro provedení útoku si pak dále vystačíme s operacemi sčítání, násobení a dělení celými čísly nebo jinými zlomky, proto je možné všechny pravděpodobnosti počítat ve tvaru zlomku a zachovat si tak maximální přesnost výpočtu.

Pro vyjádření zlomku jsem naprogramoval vlastní třídu `Rational`. Útok jsem potom upravil do druhé verze `dba-rational.cpp` tak, aby místo typu `double` používal tuto třídu. Pro reprezentaci čitatele a jmenovatele jsem použil třídu `int128_t`, která je součástí knihovny `boost`.

Nebyl jsem si zcela jistý, jestli 128bitový integer bude stačit a nedojde při výpočtech k přetečení, proto jsem se snažil operátory ve třídě `Rational` navrhnout tak, aby udržovaly použité hodnoty pokud možno co nejmenší. Například při sčítání dvou zlomků nepoužívám nejjednodušší vzorec:

$$\frac{n_1}{d_1} + \frac{n_2}{d_2} = \frac{n_1 \cdot d_2 + n_2 \cdot d_1}{d_1 \cdot d_2},$$

nýbrž:

$$\frac{n_1}{d_1} + \frac{n_2}{d_2} = \frac{n_1 \cdot (\frac{d_2}{\texttt{GCD}(d_1,d_2)}) + n_2 \cdot (\frac{d_1}{\texttt{GCD}(d_1,d_2)})}{\texttt{LCM}(d_1, d_2)},$$

kde `GCD` je největší společný dělitel d_1, d_2 a `LCM` je nejmenší společný násobek d_1, d_2. Pro výpočet `GCD` jsem použil Euklidův algoritmus. `LCM` počítám jako $d_1/\texttt{GCD}(d_1, d_2) \cdot d_2$. Zároveň jsem pro kontrolu přetečení přidal do programu kontrolní podmínky, které v případě přetečení vypíšou varování na standardní výstup. V takovém případě bych věděl, že výsledek algoritmu je zkreslený.

Výsledný program `dba-rational.cpp` je rychlejší než verze s knihovnou pro přesná reálná čísla, přesto však obnovení prvního bytu trvá necelou minutu. Program je tak stále pomalý pro plnou verzi útoku, můžeme ho však urychlit v případě, kdy omezíme abecedu otevřeného textu na menší počet znaků.

Když jsem abecedu omezil na malá písmena od „a“ do „z“, dokáže program obnovit první byte během 8 vteřin a pro obnovu 16 bytů potřebuje zhruba 17

minut. Můžeme tak program použít alespoň pro porovnání úspěšnosti u útoků s omezenou abecedou. Naměřená úspěšnost a srovnání s `dba.cpp` viz další sekce 3.2.3

Další možné zrychlení programu mě napadlo při sledování průběhu útoku původním programem `dba.cpp`. Všimnul jsem si, že kandidát odpovídající správné části otevřeného textu většinou nabývá v každém kroku větších než průměrných hodnot. Útok by tak šlo urychlit tím, že v každém kroku vezmeme místo všech 256 kandidátů pouze n kandidátů s nejvyšší pravděpodobností.

Tuto myšlenku jsem ověřil jak na programu `dba.cpp`, tak na programu `dba-rational.cpp`. Zrychlení bylo podle očekávání v závislosti na počtu vybíraných kandidátů dost velké. Bohužel měření ukázalo, že takto upravený útok má nižší úspěšnost oproti původní verzi, proto jsem tento způsob zrychlení již dále nepoužíval.

3.2.3 Měření statistik útoku

Prvním měřením jsem ověřil, že mnou stanovený odhad časové složitosti třetí částí dvoubytového útoku (obnovování otevřeného textu) skutečně platí. A že tato složitost bude menší, než časová složitost druhé části útoku (generování a zpracování šifrových textů).

Program pro obnovu otevřeného textu jsem pro měření modifikoval tak, aby obsahoval čítač, jehož hodnota se přičte pokaždé, když se v programu počítá pravděpodobnost pro nějakou dvojici bytů. Hodnota tohoto čítače by neměla přesahovat stanovenou horní mez $\mathcal{O}(2^{32}L)$.

Dále jsem do programu přidal hodnotu obsahující maximální počet kandidátů, kterého bylo v průběhu programu dosaženo. Tím jsem chtěl zjistit, zdali hodnota kandidátů někdy přesáhne jisté minimum 256 kandidátů.

Útok jsem několikrát zopakoval, pro různé délky obnovovaného textu. Měření ukázalo, že v každém běhu bylo maximum kandidátů rovno 256, to znamená, že počet kandidátů je vždy s výjimkou začátku běhu 256. Při obnovování textu dlouhého 4 byty (z toho 2 známé) bylo změřeno 8623489024 operací, což odpovídá našemu odhadu.

Dokonce můžeme přesně tuto hodnotu z pseudokódu určit. Počáteční a koncový byte je známý, na začátku a na konci algoritmu tedy máme jen 1 kandidáta a provedeme tak v obou případech 2^{24} operací. Pro zbylé dva byty zkoušíme všech 256 hodnot, tj. $2 \cdot 2^{32}$ operací. Dohromady máme $2^{33} + 2^{25} = 8623489024$. Časová složitost algoritmu v předchozí sekci 3.2.1 je tedy určena správně.

Nyní porovnáme dobu běhu druhé a třetí části útoku. Vygenerování šifrových textů s počtem opakování otevřeného textu rovno 2^{30} a s použitím jediného klíče zabralo při testování zhruba 2 a půl hodiny. Zatímco pokus o obnovu otevřeného textu délky 16 bytů (2 známé byty) při použití programu `dba.cpp` zabral 4 minuty. Pokud tedy máme rychlou implementaci třetí části útoku, druhá část bude časově vždy náročnější.

Počet šifrových textů	Počet obnovovaných bytů				
	2 byty	4 byty	6 bytů	8 bytů	16 bytů
$4 \cdot 2^{30}$	60	32	22	16	11
$6 \cdot 2^{30}$	67	33	26	19	12
$8 \cdot 2^{30}$	66	36	29	22	12
$12 \cdot 2^{30}$	72	45	33	27	17
$14 \cdot 2^{30}$	80	50	40	36	23
$16 \cdot 2^{30}$	83	56	43	38	27
$18 \cdot 2^{30}$	82	63	52	45	40
$30 \cdot 2^{30}$	100	85	84	82	78

Tabulka 3.5: Úspěšnost dvoubytového útoku v procentech z hlediska počtu obnovovaných znaků a v závislosti na počtu použitých šifrových textů

Podívejme se dále na naměřenou úspěšnost dvoubytového útoku z hlediska počtu úspěšně obnovených bytů. Pro měření jsem použil program `dba.cpp`, připouštím tak, že mohlo dojít ke ztrátě přesnosti kvůli použití typu `double`. Měření jsem provedl pro různé počty šifrových textů a různé počty obnovovaných bytů. Útok jsem vždy provedl pro 32 různých pozic v textu a naměřené výsledky jsem zprůměroval. Výsledky měření lze vidět v tabulce 3.5.

Vidíme, že změřená úspěšnost útoku není příliš velká. Při použití $14 \cdot 2^{30}$ šifrových textů obnovíme 4 byty otevřeného textu s 50% úspěšností, zatímco v [2] uvádí při stejném množství šifrových textů pravděpodbnost 100 % pro 16bytový otevřený text.

Dvoubytový útok jsem dále upravil omezením abecedy otevřeného textu na program `dba-az.cpp`. Při omezení pouze na znaky od „a“ do „z“ došlo k velmi výraznému zvýšení úspěšnosti útoku, viz tabulka 3.6. Zároveň si můžeme všimnout, že rozdíly v úspěšnosti v závislosti na délce otevřeného textu se při omezení abecedy zmenšily.

Při obnovování 2bytových otevřených textů není zlepšení tak výrazné, jako pro delší texty. Bez omezení abecedy je pro $14 \cdot 2^{30}$ šifrových textů při obnově 4bytových otevřených textů změřená 50% úspěšnost a u 16bytových otevřených textů 23% úspěšnost, s omezením abecedy je tato úspěšnost 90% pro 4 byty a 82% pro 16 bytů. V obou případech tak došlo k průměrnému zlepšení o zhruba 50 %.

Použití $30 \cdot 2^{30}$ šifrových textů vedlo k téměř 100% úspěšnosti pro všechny měřené délky obnovovaného textu a to i včetně délek větších než 16 bytů. Při pokusu o obnovu všech 256 bytů byly nalezeny pouze 2 chybné byty, ostatních 254 bytů bylo odhaleno správně.

Jako další jsem provedl porovnání výsledku při obnovování 8 bytů s omezenou abecedou pomocí programů `dba-az.cpp` a `dba-rational.cpp`, viz tabulka 3.7. Z tabulky je vidět, že oba programy v určitých případech mají

Počet šifrových textů	Počet obnovovaných bytů				
	2 byty	4 byty	6 bytů	8 bytů	16 bytů
$4 \cdot 2^{30}$	69	45	31	19	20
$6 \cdot 2^{30}$	70	56	47	40	31
$8 \cdot 2^{30}$	77	64	55	47	48
$12 \cdot 2^{30}$	88	81	76	77	75
$14 \cdot 2^{30}$	95	90	88	84	82
$16 \cdot 2^{30}$	97	93	91	89	86
$18 \cdot 2^{30}$	98	93	90	90	89
$30 \cdot 2^{30}$	100	99	99	100	99

Tabulka 3.6: Úspěšnost dvoubytového útoku v procentech z hlediska počtu obnovovaných znaků a v závislosti na počtu použitých šifrových textů při omezení abecedy v otevřeném textu

Počet šifrových textů	$8 \cdot 2^{30}$	$12 \cdot 2^{30}$	$14 \cdot 2^{30}$	$16 \cdot 2^{30}$
Úspěšnost `dba.cpp` v %	47	77	84	89
Úspěšnost `dba-rational.cpp` v %	52	78	86	88

Tabulka 3.7: Srovnání úspěšnosti programů dba-az.cpp a dba-rational.cpp při obnově 8bytových otevřených textů s omezenou abecedou

Počet šifrových textů	$4 \cdot 2^{30}$	$8 \cdot 2^{30}$	$12 \cdot 2^{30}$	$16 \cdot 2^{30}$
Úspěšnost `dba-az.cpp` v %	45	64	81	93
Úspěšnost `dba-3max-az.cpp` v %	58	87	96	100

Tabulka 3.8: Srovnání úspěšnosti programů dba-az.cpp a dba-3max-az.cpp při obnově 4bytových otevřených textů s omezenou abecedou

odlišné výstupy. V průměru dosáhl program `dba-rational.cpp` lepších výsledků. To znamená, že použitím typu `double` skutečně dochází k určité ztrátě přesnosti.

Rozdíl výsledků však není příliš velký, pokud by typ `double` byl jediným zdrojem chyby, měly by výsledky programu `dba-rational.cpp` být výrazně lepší. To nasvědčuje tomu, že moje implementace dvoubytového útoku může obsahovat nějaký další zdroj chyby.

Původní programy `dba.cpp`, `dba-az.cpp` jsem dále modifikoval do verzí `dba-3max.cpp`, `dba-3max-az.cpp`, které vrací po doběhnutí 3 nejpravděpodobnější kandidáty na otevřený text místo jednoho. Ke zlepšení výsledků došlo při hledání kratších otevřených textů (od 2 do 8 bytů), při obnovování 16 bytů je naměřené zlepšení zanedbatelné. Srovnání s `dba.cpp` při hledání otevřeného textu o délce 4 byty s omezenou abecedou viz tabulka 3.8.

Jako poslední jsem vyzkoušel, jestli úspěšnost útoku nějak závisí na počtu klíčů použitých k vygenerování šifrových dat. U předchozích měření jsem data generoval po sadách, kde pro vygenerování 2^{30} hodnot byl použit jeden klíč. Pro data $12 \cdot 2^{30}$ bylo tedy použito 12 odlišných klíčů.

Původní program pro generování šifrových dat `rc4-dba.cpp` jsem modifikoval do verze `rc4-dba-onekey.cpp`, která generuje šifrová data za použití jediného klíče. Program kromě vygenerovaných dat navíc na chybový výstup vytiskne po doběhnutí aktuální vnitřní stav šifry RC4.

Tento stav jde potom spolu s daty předat programu jako vstupní argument. Šifrování tak může pokračovat tam, kde skončilo. Tím jsem docílil toho, že jsem generování dat mohl rozdělit na části. Pokud by toto generování nebylo rozděleno, zabralo by v závislosti na délce šifrového textu i více než 24 hodin.

Pro měření jsem vygeneroval 4 sady dat, každou o délce $14 \cdot 2^{30}$ opakování otevřeného textu. Průměrná úspěšnost útoku je pro data generovaná pouze jedním klíčem o trochu lepší. Řádově se však jedná o jednotky procent. Závislost úspěšnosti na počtu použitých klíčů při generování je tedy změřitelná, ale ne příliš výrazná.

3.2.4 Vyhodnocení útoku

Naměřená úspěšnost dvoubytového útoku je nižší než v [2]. Je možné, že jsem se při implementaci někde dopustil chyby, kterou se mi nepodařilo objevit. Zároveň mohlo dojít ke snížení úspěšnosti použitím typu `double` pro výpočet pravděpodobností. Srovnání programů `dba.cpp` a `dba-rational.cpp` však nasvědčuje tomu, že nízká přesnost typu `double` není jediným faktorem, který snižuje úspěšnost útoku.

Přesto, že jsem nedosáhl tak velké úspěšnosti jako v [2], bych označil experimentální ověření funkčnosti dvoubytového útoku za úspěšné. Z naměřených dat jasně vyplývá, že dvoubytový útok funguje. Navíc při omezení vstupní abecedy došlo k podstatnému zvýšení úspěšnosti.

Z výsledků měření pro obnovované texty delší než je 16 bytů jsem si všimnul zajímavé vlastnosti útoku. Útok je schopný se za určitých okolností částečně opravit z chyby. Například mezi kandidáty o délce 4 znaky je správný kandidát „aaaa“, pro délku 5 znaků se však pro znak „a“ na konci jako nejpravděpodobnější kandidát vybere „axyza“, mezi kandidáty o délce 5 znaků se dále nachází další kandidát „aaaac“, potom se velmi často stane, že algoritmus dál pro znak „a“ správně vybere kandidáta „aaaac“, který má celkovou menší chybu. K úplné opravě však již dojít nemůže, protože mezi kandidáty již nebude řetězec skládající se z pěti znaků „a“.

Časová složitost dvoubytového útoku je poměrně velká – $2^{32} \cdot L$, kde L je počet obnovovaných bytů. Jako možné pokračování této práce tak přichází v úvahu paralelizace celého útoku. V této práci jsem k paralelizaci neměl důvod, protože jsem při měření spouštěl vždy větší množství útoků najednou.

Použití paralelizované verze útoků by tak vedlo pouze k souboji procesů o dostupné zdroje.

Pokud uvažujeme otevřený text s omezenou abecedou, pro jeho obnovení s pravděpodobností úspěchu větší než 80 % potřebujeme $14 \cdot 2^{30}$ šifrových dat. Vygenerování 2^{30} dat v jednom procesu zabralo 2 a půl hodiny, data potřebná pro útok by se tak generovala 35 hodin. Doba nalezení nejpravděpodobnějšího kandidáta otevřeného textu se pak pohybuje v řádu jednotek minut.

Dvoubytový útok tedy zatím nepovažuji za hrozbu pro reálnou komunikaci. Útok však obsahuje prostor pro zlepšení stejně jako jednobytový útok. Navíc je tu možnost zkombinovat útok s dalšími odchylkami, které se v keystreamu šifry RC4 vyskytují po celé jeho délce.

Kapitola 4

Školní úloha

Náplní této kapitoly je návrh, implementace a testování školní úlohy. Před provedením návrhu jsem stanovil požadavky, které by úloha měla splňovat:

- Úloha má za úkol studenty seznámit se šifrou RC4 a s útoky ověřenými v kapitole 3.
- Pro vyřešení úlohy by studentovi měly stačit základní znalosti počítačové bezpečnosti a programování.
- Generování úlohy by mělo být zautomatizované, s nastavitelným řešením. Tím se zvýší znovupoužitelnost úlohy v dalších semestrech.

4.1 Návrh

Celá úloha je svým obsahem zasazena do fiktivního příběhu. Příběh popisuje souboj dvou informatických skupin, jejichž názvy jsou „`MI101`“ a „`Evil 110`“. Student si bude moci vybrat, na čí stranu se přikloní, což povede k odlišnému zadání.

Budou tedy existovat 2 odlišná zadání úloh, podle toho, pro kterou skupinu je určeno. Každá úloha bude rozdělena na tři dílčí podúlohy. Tyto úlohy na sebe budou navazovat. Zadání úloh bude zaheslované a heslo k jejich otevření bude zpřístupněno vyřešením předchozí úlohy.

Princip úloh a texty týkající se příběhu zůstanou při znovupoužití úlohy stejné. Klíče použité k odemčení úloh však bude možné měnit. Pro tyto účely vytvořím generátor úloh. K tomu by měl stačit jednoduchý skript. Pomocí tohoto skriptu bude možné vygenerovat nové zadání jakékoliv z dílčích úloh a se zadanými specifickými parametry.

Úlohy jsem navrhnul tak, aby celková složitost řešení byla pro obě skupiny přibližně stejná.

4.1.1 Úlohy pro skupinu „MI101“

První úloha pro „MI101“ bude založená na zranitelnosti vyplývající z použití stejného klíče pro zašifrování dvou odlišných zpráv. Student dostane k dispozici dva šifrové texty a informaci, že pro jejich zašifrování byl použit stejný klíč. Pro zjednodušení úlohy může být navíc dodáno prvních pár znaků jedné ze zpráv.

Text zašifrovaných zpráv bude v českém jazyce, bez diakritiky. Zprávy by měly být zvoleny tak, aby po vyxorování jejich šifrových textů byla úloha řešitelná za pomocí pera a papíru. Součástí generátoru zadání bude i několik připravených textů.

Druhá úloha bude využívat zásahu do obsahu šifrového textu, viz 2.1.2. Student dostane k dispozici aplikaci a soubor se šifrovým textem, jehož strukturu bude částečně znát. Aplikace bude tento šifrový soubor načítat a v závislosti na jeho obsahu bude vracet tři možné odpovědi:

- **OK** – při použití validního vstupu, který ovšem není řešením úlohy,
- **Heslo** – při použití validního vstupu, který je řešením úlohy, součástí výstupu bude heslo k další úloze,
- **Chyba** – při použití chybného vstupu.

Student tedy bude mít za úkol upravit šifrový text tak, aby výstupem programu byla zpráva s heslem.

Třetí úloha bude vyžadovat implementaci jednobytového útoku z kapitoly 3. Student dostane data připravená pro jednobytový útok z již zpracovaných šifrových textů. Zároveň bude mít k dispozici vygenerované rozdělení keystream bytů. Úkolem studenta bude z těchto dat obnovit otevřený text. Ten bude obsahovat poslední heslo, které slouží jako důkaz vyřešení všech úloh.

4.1.2 Úlohy pro skupinu „Evil 110“

První úlohou pro „Evil 110“ bude implementace samotné šifry RC4. Student dostane zadaný šifrový text a klíč použitý při šifrování. Jeho úkolem bude použít šifru RC4 k dešifrování. Součástí otevřeného textu bude heslo k další úloze.

Ve druhé úloze dostane student opět šifrový text, tentokrát však místo použitého klíče bude mít k dispozici aplikaci pro generování klíčů, včetně zdrojového kódu. Úkolem studenta bude tento zdrojový kód zanalyzovat. Aplikace bude navržená tak, aby umožňovala pouze omezenou inicializaci náhodného generátoru, výstupem tak bude například pouze 2^{10} možných klíčů.

Student tak bude schopný všechny tyto klíče vygenerovat a použitím implementace šifry RC4 z předchozí úlohy může s jednoduchým skriptem nebo programem otestovat jejich správnost.

Smyslem třetí úlohy bude implementace dvoubytového útoku z kapitoly 3. Student dostane data připravená pro dvoubytový útok z již zpracovaných šifrových textů a část zdrojového kódu, který bude generovat potřebné rozdělení dvojic keystream bytů. Úkolem studenta je pomocí dvoubytového útoku obnovit část otevřeného textu, jenž bude obsahovat heslo pro dokončení úlohy.

4.2 Implementace

Nyní popíšu výslednou implementaci školní úlohy. Implementační prostředí zůstává stejné jako v předchozí kapitole (`g++` verze `4.8.1`, 64bitový OS `Ubuntu 13.10`).

Pro uživatelsky přívětivé vygenerování nového zadání úloh slouží skript `generate.sh`. Tento skript umožňuje vygenerování všech úloh najednou. Vygenerování všech úloh zabere dohromady asi půl minuty. Zároveň skript umožňuje generování úloh jednotlivě, se zadanými parametry.

Použité parametry při generování včetně hesel uloží generovací skript do výstupního souboru `generovani-YYYY-MM-DD.log`, kde `YYYY-MM-DD` odpovídá aktuálnímu datumu v době generování. Tento soubor může sloužit později jako kontrola správných řešení.

Rozdělení úloh jsem docílil pomocí komprese ve formátu `zip`. Každá úloha pro danou skupinu s výjimkou té první je zaheslovaná. Heslo pro odemčení archivu další úlohy student získá úspěšným vyřešením předchozí úlohy. Tím je dosaženo toho, že úlohy budou řešeny postupně.

Pro přípravu potřebných šifrových textů použitých v úlohách slouží program `rc4.cpp`. Program v sobě obsahuje šifrování pomocí RC4. Vstupem programu je identifikační číslo a parametry úlohy, jejíž šifrová data se mají vygenerovat. Výstup programu je pak závislý na zvolené úloze. Použitá identifikační čísla nemusí uživatel znát, protože o obsluhu programu se stará generovací skript.

4.2.1 Úlohy pro skupinu „`MI101`“

Pro vygenerování zadání první úlohy stačí zašifrovat dva otevřené texty stejným klíčem a uložit jejich šifrové texty. Program očekává, že oba texty mají stejnou délku. Výstupem je soubor, který obsahuje oba šifrové texty oddělené znakem nového řádku. Na dalších dvou řádcích jsou tyto šifrové texty zapsané v šestnáctkové soustavě.

Generovací skript umožňuje použití náhodné dvojice z předem připravených textů. Připravené dvojice otevřených textů svým obsahem korespondují s navrženým příběhem. Zároveň je možné zadat vlastní otevřené texty. Pokud chceme použít vlastní texty, uložíme je oba do jednoho souboru. Jeden z textů by měl v sobě obsahovat heslo pro odemčení další úlohy. Toto heslo by se v souboru mělo také vyskytovat dále na třetím řádku, odkud ho může

snadno přečíst generovací skript bez potřeby složitého prohledávání otevřených textů.

Druhá úloha se skládá ze šifrového textu `msg.txt` a programu `validator`. Když přesměrujeme šifrový text na standardní vstup programu, dostaneme jako odpověď jednu ze tří možností uvedených v předchozí sekci 4.1.1.

Rozhodující pro řešení úlohy jsou dva binární řetězce, které generující skript dokáže vygenerovat náhodně nebo je zadá uživatel. Tyto řetězce spolu s heslem pro další úlohu se vloží na konec hlavičkového souboru `validator.h` a generující skript poté zkompiluje `validator.cpp`, aby výsledný program `validator` obsahoval nastavené řešení.

Třetí úloha je celá o provedení jednobytového útoku ověřeného v předchozí kapitole. Zadání třetí úlohy tak vyžaduje rozdělení šifrových textů v podobě čítačů pro jeden otevřený text. Viděli jsme, že vygenerování dostatečného množství šifrových textů trvá poměrně dlouhou dobu.

Naštěstí jsem objevil způsob, jak celé generování obejít. Stačí nám znalost dříve vygenerovaného rozdělení šifrových textů pro jiný otevřený text, který taktéž celý známe. Potom můžeme toto rozdělení převést na nové rozdělení pro libovolný otevřený text pomocí jednoduché změny pořadí prvků.

Pokud například máme hodnotu v rozdělení $N[i][j]$ a původní byte otevřeného textu měl hodnotu p_1, pro byte p_2 z nového otevřeného textu získáme index odpovídající hodnoty následovně:

$$N[i][j \oplus p_1 \oplus p_2] \leftarrow N[i][j].$$

Provedením této operace pro všechny hodnoty i a j vytvoříme nové rozdělení, které svým obsahem odpovídá, jako bychom šifrovali nový otevřený text. Místo časově náročného šifrování jsme tak schopni vygenerovat zadání celé úlohy během zlomku vteřiny.

Další výhodou kromě zrychlení je i fakt, že úspěšnost útoku pro toto nové rozdělení zůstává identická vzhledem k původnímu rozdělení. Předem tedy máme informaci o tom, které byty otevřeného textu se nám podaří obnovit. To je pro nás velmi důležité v tom, že víme, kam přibližně v otevřeném textu umístit heslo pro dokončení úlohy.

4.2.2 Úlohy pro skupinu „`Evil 110`“

Zadání první úlohy obsahuje šifrový text a použitý klíč. Vygenerování zadání tak není nic jiného, než zašifrování otevřeného textu a vytisknutí šifrového textu a klíče na standardní výstup. Klíč použitý pro šifrování se generuje náhodně. Otevřený text se předává jako parametr. Na standardní výstup je vytištěn šifrový text a použitý klíč, nejprve v podobě textu a potom v podobě hodnot šestnáctkové soustavy.

Druhá úloha opět obsahuje šifrový text. Místo klíče je však součástí zadání úlohy program `badRandom.cpp`, který byl použit pro vygenerování šifrového

klíče. Vstupem programu je hodnota náhodného semínka (seed), která je použita pro inicializaci náhodného generátoru a může nabývat hodnot od 0 do $2^{32}-1$. Program používá lineární kongruentní generátor náhodných čísel s rovnicí:

$$x_{n+1} \equiv 111111 \cdot x_n + 3 \mod 2^{32},$$

který se sám o sobě jeví v pořádku. Počet možných klíčů se však vzhledem k algoritmu jejich generování drasticky sníží.

Klíč je generován byte po bytu tak, že pro vygenerování každého bytu vezmeme výstupní hodnotu náhodného generátoru modulo 256. Vidíme, že použitý náhodný generátor počítá hodnoty modulo 2^{32} a tato hodnota je dělitelná 256. Výsledkem je, že jsme schopni opakovaným spuštěním programu pro všech 2^{32} hodnot náhodného semínka vygenerovat dohromady pouze 256 odlišných klíčů.

Pokud totiž použijeme dvě hodnoty náhodného semínka, které se od sebe liší o nějaký násobek hodnoty 256, dostaneme totožný klíč. To bude platit pro libovolné parametry lineárního kongruentního generátoru, pokud použitý modul bude dělitelný 256.

Zadání třetí úlohy obsahuje připravený program s rozdělením keystreamu pro dvojice po sobě jdoucích bytů a rozdělení vzniklé ze zachycení šifrových textů. Podobně jako u třetí úlohy pro skupinu „`MI101`“, i toto rozdělení není třeba generovat od začátku. Pro urychlení použijeme opět existující rozdělení šifrových textů. Tentokrát však mámé trojrozměrné pole $N[i][j][k]$, pro byty původního textu $p_{1,1}, p_{1,2}$ a byty nového textu $p_{2,1}, p_{2,2}$ bude transformace probíhat následovně:

$$N[i][j \oplus p_{1,1} \oplus p_{2,1}][k \oplus p_{1,2} \oplus p_{2,2}] \leftarrow N[i][j][k].$$

Úspěšnost útoku zůstane pro transformované rozdělení šifrových textů zase stejná. A jelikož dvoubytový útok nemá úplně 100% úspěšnost obnovení otevřeného textu, jedná se o ještě větší výhodu než u třetí úlohy pro „`MI101`“ (jednobytového útoku). Určíme totiž přesnou pozici, kam v otevřeném textu uložit konečné heslo a zaručíme tím řešitelnost úlohy.

4.3 Testování

Nejprve jsem testoval funkčnost generovacího skriptu `generate.sh`. Celkový počet nabídek ve skriptu není tak velký, proto jsem je mohl všechny otestovat ručně. Automatické generování úloh zabere dohromady zhruba půl minuty. U všech vygenerovaných úloh jsem zkontroloval správnost zadání. Každou úlohy jsem dále zkusil vyřešit.

Složitost úloh mi pro obě skupiny přišla celkově vyrovnaná. Skupina „`MI101`“ má první dvě úlohy o něco obtížnější. Třetí úloha je však mnohem jednodušší na naprogramování.

4.3.1 Úlohy pro skupinu „MI101“

První úlohu pro skupinu „MI101“ na znovupoužití stejného klíče při šifrování jsem řešil pomocí tužky a papíru. Počítač jsem použil pouze k vyxorování zadaných šifrových textů mezi sebou. Obsah otevřených textů mi nebyl předem znám, protože jsem požádal další osobu o jejich napsání.

Řešení úlohy je celkem snadné, především proto, že abeceda otevřeného textu nepokrývá všechny možné hodnoty ASCII kódování. Ve výsledném xoru šifrových textů tak snadno můžeme indentifikovat například interpunkční znaménka.

Z provedení tohoto testu jsem dospěl k závěrů, že složitost úlohy je dostatečně nízká. Do zadání úlohy tak není potřeba vkládat žádné dodatečné informace o použitých otevřených textech.

Druhá úloha už ke svému vyřešení vyžaduje programování. Funkčnost jsem ověřil za pomocí jednoduchého skriptu, který vyzkoušel všech 2^{12} možných vstupů. Heslo pro další úroveň se zobrazí skutečně jen v případě, kdy je validačnímu programu zadána správná odpověď, tj. dodaný soubor byl správně modifikován. Také jsem vyzkoušel, že při poškození souboru program vypíše odpovídající chybovou hlášku.

Úlohu je možné vyřešit i alternativním způsobem. Pro jednoduchost implementace se řešení úlohy vkládá ve formátu řetězce do validačního programu před jeho kompilací. Text tedy po kompilaci není v programu nijak maskován a můžeme ho v něm najít.

Pro otestování třetí úlohy jsem použil implementaci jednobytového útoku z předchozí kapitoly. Princip útoku je poměrně jednoduchý, proto se řešení úlohy dá naprogramovat celkem rychle.

4.3.2 Úlohy pro skupinu „Evil 110“

První úloha (implementace šifry RC4) je velmi jednoduchá. Dokonce je možné ji vyřešit bez vlastní implementace šifry RC4, když použijeme nějakou existující implementaci (třeba v OpenSSL).

Druhá úloha vyžaduje vyzkoušení 256 možných klíčů k dešifrování textu. Toho jsem při testování docílil jednoduchým skriptem. Pro řešení úlohy lze opět využít OpenSSL.

Třetí úlohu jsem podobně jako u skupiny „MI101“ otestoval pomocí implementace dvoubytového útoku z předchozí kapitoly. Díky specifickému umístění hesla v textu je při správné implementaci dvoubytového útoku řešitelnost úlohy zaručena.

Závěr

Cílem této práce bylo seznámit se s šifrou RC4, prozkoumat její zranitelnosti, ověřit funkčnost útoků popsaných v [2] a vytvořit školní úlohu.

V kapitole 2 jsem uvedl rozsáhlý seznam doposud objevených a publikovaných zranitelností šifry RC4. Je patrné, že inicializační část (KSA) obsahuje velké slabiny. Počáteční keystream byty generované šifrou RC4 tak vykazují jasně změřitelné statistické odchylky. Určité nedokonalosti však již byly objeveny i u generátoru keystreamu (PRGA).

Dále jsem v kapitole 2 uvedl praktické útoky na specifické části šifry RC4. Nejvíce pozornosti bylo podle mého názoru věnováno především útokům na WEP. Ten je však od roku 2001 považován za prolomený, proto se pozornost ubírala i dalšími směry. Do popředí se nyní dostávají útoky na broadcast vysílání zpráv šifrovaných pomocí RC4. Funkčnost dvou útoků popsaných v [2] jsem ověřil v kapitole 3.

Naměřená úspěšnost jednobytového útoku je srovnatelná s výsledky uvedenými v [2]. Pro 2^{32} šifrových textů obnovíme otevřený text se 100% jistotou s výjimkou bytů na indexu 253 a 255. Pro 2^{28} šifrových textů je celková pravděpodobnost úspěchu větší než 50 % a pravděpodobnost úspěchu pro prvních 128 bytů je dokonce přes 90 %.

Jednobytový útok tak můžeme považovat za praktický. V reálném provozu je vygenerování 2^{28} šifrových textů stále ještě dost náročné. Útok však obsahuje další prostor pro zlepšení, například kombinací s frekvenční analýzou nebo složitějšími jazykovými modely, které by nějakým způsobem využily pro zlepšení úšpěšnosti strukturu otevřeného textu.

Jedinou nevýhodou jednobytového útoku je, že funguje pouze pro prvních 256 bytů keystreamu. Je tedy možné se proti němu s jistotou ubránit tím, že počáteční byty keystreamu při šifrování zahodíme.

U dvoubytového útoku jsem naměřil nižší úspěšnost, než je uvedeno v [2]. Příčinu nižší úspěšnosti se mi nepodařilo odhalit. Cílem této práce však bylo dokázat, že tento útok prakticky funguje, a to se mi povedlo. Při omezení možných počtu znaků v otevřeném textu a zachycení $30 \cdot 2^{30}$ šifrových textů

je úspěšnost útoku téměř 100%.

Počet potřebných šifrových textů k provedení útoku s vysokou pravděpodobností úspěchu je mnohem větší než u jednobytového útoku. Proto zatím považuji dvoubytový útok za nepraktický. Podobně jako jednobytový útok však obsahuje další prostor pro zlepšení. Kromě frekvenční analýzy či složitějších jazykových modelů může být pro vylepšení útoku v budoucnu použita znalost dalších odchylek, které se vyskytují po celé délce keystreamu.

V kapitole 4 jsem navrhnul, vytvořil a otestoval školní úlohu, která se tématicky zabývá šifrou RC4. Úloha je rozdělená do dvou skupin a každá skupina má odlišné zadání skládající se ze tří dílčích úloh. Celková náročnost úloh pro obě skupiny je srovnatelná. Pro vygenerování zadání jednotlivých úloh jsem vytvořil jednoduchý skript. Celý proces je tak kompletně zautomatizovaný a zabere dohromady kolem půl minuty.

Zadání úloh, týkajících se jednobytového a dvoubytového útoku, se generuje za použití již existujících, zpracovaných šifrových dat. Tím jsem docílil toho, že nové zadání pro libovolný otevřený text vygenerujeme velmi rychle. Zároveň jsem tímto způsobem zaručil, že úlohy jdou při správné implementaci vyřešit.

Všechny cíle práce se mi podařilo úspěšně splnit. Pro pokračování v této práci bych doporučil paralelizaci dvoubytového útoku, jenž je časově dost náročný. Zajímavý úkol může být snaha o zvýšení úspěšnosti obou útoků za pomocí kombinace s frekvenční analýzou nebo složitějším jazykovým modelem. Za velmi přínosné téma považuji také zkoumání dlouhodobých odchylek v keystreamu šifry RC4. Při identifikaci nového druhu odchylek by mohlo dojít k dalšímu vylepšení existujících útoků.

Literatura

[1] Akgün, M.; Kavak, P.; Demirci, H.: New Results on the Key Scheduling Algorithm of RC4. In *Proceedings of the 9th International Conference on Cryptology in India: Progress in Cryptology*, INDOCRYPT '08, Berlin, Heidelberg: Springer-Verlag, 2008, ISBN 978-3-540-89753-8, s. 40–52, doi:10.1007/978-3-540-89754-5_4. Dostupné z: `http://dx.doi.org/10.1007/978-3-540-89754-5_4`

[2] AlFardan, N. J.; Bernstein, D. J.; Paterson, K. G.; aj.: On the Security of RC4 in TLS. In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, Berkeley, CA, USA: USENIX Association, 2013, ISBN 978-1-931971-03-4, s. 305–320. Dostupné z: `http://dl.acm.org/citation.cfm?id=2534766.2534793`

[3] Anonymous: RC4 Source Code. 1994. Dostupné z: `http://cypherpunks.venona.com/date/1994/09/msg00304.html`

[4] Biham, E.; Carmeli, Y.: Fast Software Encryption. kapitola Efficient Reconstruction of RC4 Keys from Internal States, Berlin, Heidelberg: Springer-Verlag, 2008, ISBN 978-3-540-71038-7, s. 270–288, doi:10.1007/978-3-540-71039-4_17. Dostupné z: `http://dx.doi.org/10.1007/978-3-540-71039-4_17`

[5] Biham, E.; Granboulan, L.; Nguyên, P. Q.: Impossible Fault Analysis of RC4 and Differential Fault Analysis of RC4. In *Proceedings of the 12th International Conference on Fast Software Encryption*, FSE'05, Berlin, Heidelberg: Springer-Verlag, 2005, ISBN 3-540-26541-4, 978-3-540-26541-2, s. 359–367, doi:10.1007/11502760_24. Dostupné z: `http://dx.doi.org/10.1007/11502760_24`

[6] Chardin, T.; Fouque, P.-A.; Leresteux, D.: Cache Timing Analysis of RC4. In *Proceedings of the 9th International Conference on Applied Cryptography and Network Security*, ACNS'11, Berlin, Heidelberg: Springer-

Verlag, 2011, ISBN 978-3-642-21553-7, s. 110–129. Dostupné z: http://dl.acm.org/citation.cfm?id=2025968.2025977

[7] Chen, J.; Miyaji, A.: Generalized RC4 Key Collisions and Hash Collisions. In *Proceedings of the 7th International Conference on Security and Cryptography for Networks*, SCN'10, Berlin, Heidelberg: Springer-Verlag, 2010, ISBN 3-642-15316-X, 978-3-642-15316-7, s. 73–87. Dostupné z: http://dl.acm.org/citation.cfm?id=1885535.1885544

[8] Chen, J.; Miyaji, A.: A New Class of RC4 Colliding Key Pairs with Greater Hamming Distance. In *Proceedings of the 6th International Conference on Information Security Practice and Experience*, ISPEC'10, Berlin, Heidelberg: Springer-Verlag, 2010, ISBN 3-642-12826-2, 978-3-642-12826-4, s. 30–44, doi:10.1007/978-3-642-12827-1_3. Dostupné z: http://dx.doi.org/10.1007/978-3-642-12827-1_3

[9] Chen, J.; Miyaji, A.: How to Find Short RC4 Colliding Key Pairs. In *Proceedings of the 14th International Conference on Information Security*, ISC'11, Berlin, Heidelberg: Springer-Verlag, 2011, ISBN 978-3-642-24860-3, s. 32–46. Dostupné z: http://dl.acm.org/citation.cfm?id=2051002.2051006

[10] Finney, H.: An RC4 cycle that can't happen. 1994.

[11] Fluhrer, S. R.; Mantin, I.; Shamir, A.: Weaknesses in the Key Scheduling Algorithm of RC4. In *Revised Papers from the 8th Annual International Workshop on Selected Areas in Cryptography*, SAC '01, London, UK, UK: Springer-Verlag, 2001, ISBN 3-540-43066-0, s. 1–24. Dostupné z: http://dl.acm.org/citation.cfm?id=646557.694759

[12] Fluhrer, S. R.; McGrew, D. A.: Statistical Analysis of the Alleged RC4 Keystream Generator. In *Proceedings of the 7th International Workshop on Fast Software Encryption*, FSE '00, London, UK, UK: Springer-Verlag, 2001, ISBN 3-540-41728-1, s. 19–30. Dostupné z: http://dl.acm.org/citation.cfm?id=647935.740916

[13] Golić, J.: Iterative Probabilistic Cryptanalysis of RC4 Keystream Generator. In *Information Security and Privacy*, *Lecture Notes in Computer Science*, ročník 1841, editace E. Dawson; A. Clark; C. Boyd, Springer Berlin Heidelberg, 2000, ISBN 978-3-540-67742-0, s. 220–233, doi:10.1007/10718964_18. Dostupné z: http://dx.doi.org/10.1007/10718964_18

[14] Golić, J.; Morgari, G.: Iterative Probabilistic Reconstruction of RC4 Internal States. Cryptology ePrint Archive, Report 2008/348, 2008, http://eprint.iacr.org/.

[15] Grosul, W. D., A.L.: A Related-Key Cryptanalysis of RC4. In *Technical Report TR-00-358*, 2000.

[16] Hlaváč, J.: Bezpečnost a technické prostředky: Generátory (pseudo)náhodných čísel. ČVUT FIT, 2012. Dostupné z: `https://edux.fit.cvut.cz/courses/MI-BHW/`

[17] Isobe, T.; Ohigashi, T.; Watanabe, Y.; aj.: Full plaintext recovery attack on broadcast RC4. *Proc. the 20th International Workshop on Fast Software Encryption (FSE 2013).(March 2013)*, 2013.

[18] Jenkins, R. J.: ISAAC and RC4s. 1997, `http://burtleburtle.net/bob/rand/isaac.html`.

[19] Klein, A.: Attacks on the RC4 Stream Cipher. *Des. Codes Cryptography*, ročník 48, č. 3, Září 2008: s. 269–286, ISSN 0925-1022, doi: 10.1007/s10623-008-9206-6. Dostupné z: `http://dx.doi.org/10.1007/s10623-008-9206-6`

[20] Knudsen, L. R.; Meier, W.; Preneel, B.; aj.: Analysis Methods for (Alleged) RCA. In *Proceedings of the International Conference on the Theory and Applications of Cryptology and Information Security: Advances in Cryptology*, ASIACRYPT '98, London, UK, UK: Springer-Verlag, 1998, ISBN 3-540-65109-8, s. 327–341. Dostupné z: `http://dl.acm.org/citation.cfm?id=647094.716579`

[21] KoreK: Need security pointers. 2004. Dostupné z: `http://archive.is/Tm9as`

[22] KoreK: Next generation of WEP attacks? 2004. Dostupné z: `http://www.netstumbler.org/news/next-generation-of-wep-attacks-t12277.html`

[23] Maitra, S.; Paul, G.; Gupta, S. S.: Attack on Broadcast RC4 Revisited. In *Proceedings of the 18th International Conference on Fast Software Encryption*, FSE'11, Berlin, Heidelberg: Springer-Verlag, 2011, ISBN 978-3-642-21701-2, s. 199–217. Dostupné z: `http://dl.acm.org/citation.cfm?id=2022159.2022176`

[24] Mantin, I.: *Analysis of the stream cipher RC4*. Diplomová práce, The Weizmann Institute of Science, Israel, 2001.

[25] Mantin, I.: Predicting and Distinguishing Attacks on RC4 Keystream Generator. In *Proceedings of the 24th Annual International Conference on Theory and Applications of Cryptographic Techniques*, EUROCRYPT'05, Berlin, Heidelberg: Springer-Verlag, 2005, ISBN 3-540-25910-4, 978-3-540-25910-7, s. 491–506, doi:10.1007/11426639_29. Dostupné z: `http://dx.doi.org/10.1007/11426639_29`

[26] Mantin, I.; Shamir, A.: A Practical Attack on Broadcast RC4. In *Revised Papers from the 8th International Workshop on Fast Software Encryption*, FSE '01, London, UK, UK: Springer-Verlag, 2002, ISBN 3-540-43869-6, s. 152–164. Dostupné z: http://dl.acm.org/citation.cfm?id=647936.741069

[27] Matsui, M.: Fast Software Encryption. kapitola Key Collisions of the RC4 Stream Cipher, Berlin, Heidelberg: Springer-Verlag, 2009, ISBN 978-3-642-03316-2, s. 38–50, doi:10.1007/978-3-642-03317-9_3. Dostupné z: http://dx.doi.org/10.1007/978-3-642-03317-9_3

[28] Maximov, A.; Khovratovich, D.: New State Recovery Attack on RC4. In *Proceedings of the 28th Annual Conference on Cryptology: Advances in Cryptology*, CRYPTO 2008, Berlin, Heidelberg: Springer-Verlag, 2008, ISBN 978-3-540-85173-8, s. 297–316, doi:10.1007/978-3-540-85174-5_17. Dostupné z: http://dx.doi.org/10.1007/978-3-540-85174-5_17

[29] Mironov, I.: (Not So) Random Shuffles of RC4. In *Proceedings of the 22Nd Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '02, London, UK, UK: Springer-Verlag, 2002, ISBN 3-540-44050-X, s. 304–319. Dostupné z: http://dl.acm.org/citation.cfm?id=646767.704312

[30] Paul, G.; Maitra, S.: Permutation After RC4 Key Scheduling Reveals the Secret Key. In *Proceedings of the 14th International Conference on Selected Areas in Cryptography*, SAC'07, Berlin, Heidelberg: Springer-Verlag, 2007, ISBN 3-540-77359-2, 978-3-540-77359-7, s. 360–377. Dostupné z: http://dl.acm.org/citation.cfm?id=1784881.1784904

[31] Roos, A.: A Class of Weak Keys in the RC4 Stream Cipher. 1995. Dostupné z: http://marcel.wanda.ch/Archive/WeakKeys

[32] Sen Gupta, S.: *Analysis and Implementation of RC4 Stream Cipher*. Dizertační práce, INDIAN STATISTICAL INSTITUTE Kolkata, 2013.

[33] Sen Gupta, S.; Maitra, S.; Paul, G.; aj.: Proof of Empirical RC4 Biases and New Key Correlations. In *Proceedings of the 18th International Conference on Selected Areas in Cryptography*, SAC'11, Berlin, Heidelberg: Springer-Verlag, 2012, ISBN 978-3-642-28495-3, s. 151–168, doi:10.1007/978-3-642-28496-0_9. Dostupné z: http://dx.doi.org/10.1007/978-3-642-28496-0_9

[34] Sen Gupta, S.; Maitra, S.; Paul, G.; aj.: (Non–)random sequences from (non–)random permutations – analysis of RC4 stream cipher. *Journal of Cryptology*, 2013: s. 10–1007.

[35] Sepehrdad, P.; Sušil, P.; Vaudenay, S.; aj.: Smashing WEP in A Passive Attack. 2013.

[36] Sepehrdad, P.; Vaudenay, S.; Vuagnoux, M.: Discovery and Exploitation of New Biases in RC4. In *Proceedings of the 17th International Conference on Selected Areas in Cryptography*, SAC'10, Berlin, Heidelberg: Springer-Verlag, 2011, ISBN 978-3-642-19573-0, s. 74–91. Dostupné z: `http://dl.acm.org/citation.cfm?id=1964441.1964448`

[37] Sepehrdad, P.; Vaudenay, S.; Vuagnoux, M.: Statistical Attack on RC4 Distinguishing WPA. In *Proceedings of the 30th Annual International Conference on Theory and Applications of Cryptographic Techniques: Advances in Cryptology*, EUROCRYPT'11, Berlin, Heidelberg: Springer-Verlag, 2011, ISBN 978-3-642-20464-7, s. 343–363. Dostupné z: `http://dl.acm.org/citation.cfm?id=2008684.2008712`

[38] Stubblefield, A.; Ioannidis, J.; Rubin, A. D.: Using the Fluhrer, Mantin, and Shamir Attack to Break WEP. 2001, s. 17–22.

[39] Stubblefield, A.; Ioannidis, J.; Rubin, A. D.: A key recovery attack on the 802.11b wired equivalent privacy protocol (wep). *ACM Transactions on Information and System Security*, ročník 7, 2004: s. 319–332.

[40] Tews, E.; Beck, M.: Practical Attacks Against WEP and WPA. In *Proceedings of the Second ACM Conference on Wireless Network Security*, WiSec '09, New York, NY, USA: ACM, 2009, ISBN 978-1-60558-460-7, s. 79–86, doi:10.1145/1514274.1514286. Dostupné z: `http://doi.acm.org/10.1145/1514274.1514286`

[41] Tews, E.; Weinmann, R.-P.; Pyshkin, A.: Breaking 104 Bit WEP in Less Than 60 Seconds. In *Proceedings of the 8th International Conference on Information Security Applications*, WISA'07, Berlin, Heidelberg: Springer-Verlag, 2007, ISBN 3-540-77534-X, 978-3-540-77534-8, s. 188–202. Dostupné z: `http://dl.acm.org/citation.cfm?id=1784964.1784983`

[42] Tomašević, V.; Bojanić, S.; Nieto-Taladriz, O.: Finding an Internal State of RC4 Stream Cipher. *Inf. Sci.*, ročník 177, č. 7, Duben 2007: s. 1715–1727, ISSN 0020-0255, doi:10.1016/j.ins.2006.10.010. Dostupné z: `http://dx.doi.org/10.1016/j.ins.2006.10.010`

[43] Vaudenay, S.; Vuagnoux, M.: Passive-only Key Recovery Attacks on RC4. In *Proceedings of the 14th International Conference on Selected Areas in Cryptography*, SAC'07, Berlin, Heidelberg: Springer-Verlag, 2007, ISBN 3-540-77359-2, 978-3-540-77359-7, s. 344–359. Dostupné z: `http://dl.acm.org/citation.cfm?id=1784881.1784903`

Příloha A

Seznam použitých zkratek

AES Advanced Encryption System

ARP Address Resolution Protocol

GCM Galois/Counter Mode

IV Inicializační Vektory

KSA Key Scheduling Algorithm

MiM Man In The Middle

PRGA Pseudorandom Generation Algorithm

PRNG Pseudorandom Number Generator

TKIP Temporal Key Integrity Protocol

TLS Transport Layer Security

WEP Wired Equivalent Privacy

WPA Wi-Fi Protected Access

XOR Exclusive Or

Příloha **B**

Obsah přiloženého CD

- `readme.txt` ..... popis obsahu CD, včetně návodu k použití programů
- `bin`
  - `dba` . zkompilované programy pro dvoubytový útok a výsledky měření
    - `ct` ..... vygenerovná šifrová data
    - `ct_onekey` .... vygenerovaná šifrová data za použití jediného klíče
    - `mereni` ..... výsledky provedení dvoubytového útoku a změřená úspěšnost
  - `sba` .zkompilované programy pro jednobytový útok a výsledky měření
    - `grafy_k32` .... grafy rozdělení keystream bytů pro zvolenou pozici bytu za použití 2^{32} náhodných klíčů
    - `keystream` ..... vygenerovaná keystream rozdělení
    - `mereni` ..... provedené jednobytové útoky a měření úspěšnosti
    - `tabulky` .... změřené úspěšnosti jednobytového útoku v přehledné podobě
- `src`
  - `dba` ..... zdrojové kódy programů pro dvoubytový útok
  - `sba` ..... zdrojové kódy programů pro jednobytový útok
- `zadání úloh` ..... ukázková zadání školní úlohy
- `text` ..... text práce
  - `src` ..... zdrojová forma práce ve formátu LaTeX