Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Kryptoanalýza prúdových šifier RC4 a Spritz

Diplomová práca

2016 **Bc. Martin Gábriš**

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Kryptoanalýza prúdových šifier RC4 a Spritz

Diplomová práca

Študijný program:	Informatika
Študijný odbor:	2508 Informatika
Školiace pracovisko:	Katedra Informatiky FMFI
Vedúci práce:	doc. RNDr. Martin Stanek, PhD.

Bratislava, 2016 **Bc. Martin Gábriš**

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Bc. Martin Gábriš
Študijný program: informatika (Jednoodborové štúdium, magisterský II. st., denná forma)
Študijný odbor: informatika
Typ záverečnej práce: diplomová
Jazyk záverečnej práce: slovenský
Sekundárny jazyk: anglický

Názov: Kryptoanalýza prúdových šifier RC4 a Spritz
Cryptanalysis of Stream Ciphers RC4 and Spritz

Cieľ: Naštudovať metódy kryptoanalýzy prúdových šifier. Analyzovať odolnosť vybraných šifier alebo ich prípadných modifikácií voči týmto metódam. Experimentálne overiť získané výsledky kryptoanalýzy.

Vedúci: doc. RNDr. Martin Stanek, PhD.
Katedra: FMFI.KI - Katedra informatiky
Vedúci katedry: doc. RNDr. Daniel Olejár, PhD.

Dátum zadania: 11.12.2014

Dátum schválenia: 12.12.2014 prof. RNDr. Branislav Rovan, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

Poďakovanie

Chcel by som vyjadriť veľkú vďaku vedúcemu mojej diplomovej práce Martinovi Stanekovi za jeho ochotu, podnetné konzultácie, cennú pomoc ako aj za mnohé rady a pripomienky k forme a obsahu práce.

Ďalej by som chcel poďakovať mojej rodine za podporu pri písaní práce.

Abstrakt

V tejto diplomovej práci sa venujeme útokom na vnútorný stav šifier RC4 a Spritz. Uvažujeme dva typy metód: použitie SMT solverov a prehľadávanie s návratom (backtracking).

Použitím SMT solverov na kryptoanalýzu RC4 a Spritz sme nadviazali na predchádzajúcu prácu o algebraickej analýze šifry RC4. Ukázali sme, ako popísať vzťah medzi stavom šifry a bežiacim kľúčom v jazyku SMT a ako tento popis využiť na kryptoanalýzu oboch šifier. Experimentálne sme overili metódu na redukovaných verziách šifier. Použite SMT solverov je oproti algebraickej analýze rýchlejšie a jednoduchšie na použitie.

Útokom na počiatočný stav šifier RC4 a Spritz pomocou prehľadávania s návratom sa v minulosti zaoberalo niekoľko prác. My sme pre šifru Spritz navrhli viaceré vylepšenia existujúceho prehľadávania a experimentálne preskúmali ich úspešnosť. Odvodili sme odhady zložitosti rôznych verzií prehľadávania pre obe šifry. Porovnaním nášho postupu s doteraz publikovanými analýzami ukazujú naše odhady mierne nižšiu zložitosť pri šifre RC4 a výrazne nižšiu zložitosť pre šifru Spritz.

Abstract

In this thesis we study state recovery attacks on ciphers RC4 and Spritz. We focus on two types of methods: cryptanalysis using SMT solvers and backtracking.

We use SMT solvers to recover intial state of RC4 and Spritz, similarly to algebraic analysis of RC4, a previous work using the same general idea. We show how to translate both ciphers to SMT language and how to use SMT solvers to find the state of ciphers using the known keystream. We validated this method by performing experiments on reduced versions of ciphers. Comparing the usage of SMT solvers to the algebraic analysis, our method is both faster and easier to use.

State recovery attacks using backtracking for both RC4 and Spritz were studied before. We propose various optimizations to existing backtracking for cipher Spritz and perform experiments to determine their improvement to complexity of the attack. We derive complexity extimates of different versions of backtracking for both ciphers. Comparing our estimates with existing results, our analysis shows an minor improvement for cipher RC4 and an significant improvement for cipher Spritz.

Obsah

Úvod

Cieľom tejto diplomovej práce je aplikovať vybrané metódy kryptoanalýzy na prúdové šifry RC4 a Spritz, experimentálne tieto metódy overiť a odhadnúť ich zložitosť.

Prúdové šifry sú jedným zo základných typov symetrických šifrových algoritmov. Prúdová šifra je pseudonáhodný generátor, skladajúci sa z vnútorného stavu šifry, aktualizačnej funkcie a výstupnej funkcie. Vnútorný stav šifry je inicializovaný pomocou symetrického kľúča a prípadne inicializačného vektora. V čase t je stav šifry modifikovaný aktualizačnou funkciou a pomocou výstupnej funkcie je zo stavu šifry odvodené slovo z_t. Slovo môže byť bit, bajt, 32-bitový reťazec a pod. Pseudonáhodná postupnosť slov $\{z_t\}_{t\geq 0}$ sa nazýva bežiaci kľúč (angl. keystream). Otvorený text je šifrovaný po slovách skombinovaním s bežiacim kľúčom. Kombinovanie slov otvoreného textu a bežiaceho kľúča je väčšinou jednoduchá operácia nezávislá od stavu šifry a kľúča, najčastejšie sa používa operácia `xor`.

Prúdové šifry sú v praxi používané hlavne vďaka ich rýchlosti a jednoduchosti hardvérovej implementácie. Príkladmi známych prúdových šifier sú RC4, A5/1 alebo Snow-3G.

Cieľom metód kryptoanalýzy prúdových šifier býva zvyčajne buď rozpoznať bežiaci kľúč od úplne náhodnej postupnosti alebo získať vnútorný stav šifry, napríklad hneď po inicializácii. Prvý prípad nám umožní tipovať otvorený text, pričom rozpoznávač nám vie povedať, či sme hádali správne. V druhom prípade, ak vieme počiatočný stav šifry, môžeme vygenerovať pseudonáhodnú postupnosť rovnakú, ako bola použitá pri šifrovaní a otvorený text s ňou skombinovať.

Obvykle uvažovaná situácia pri kryptoanalýze prúdových šifier je útok so znalosťou otvoreného textu (angl. KPA – Known Plaintext Attack). Znalosť otvoreného textu pre prúdové šifry znamená znalosť bežiaceho kľúča, keďže zvyčajne je otvorený text kombinovaný s bežiacim kľúčom veľmi jednoducho.

Najznámejšie typy útokov na prúdové šifry sú:

- Algebraické útoky sú útoky so znalosťou bežiaceho kľúča a snažia sa získať kľúč alebo počiatočný stav šifry. Hlavná idea je na základe popisu šifry vyrobiť sústavu rovníc, ktorá zahŕňa bežiaci kľúč a ako neznáme obsahuje počiatočný stav alebo tajný kľúč. Následne sa sústava vyrieši. Problém je práve vo vyriešení sústavy rovníc, ktoré by malo byť pre dobre navrhnutú šifru ťažké.

- Lineárne útoky sú modifikácia podobného prístupu pre blokové šifry. Táto metóda sa snaží nájsť v šifre časti spôsobujúce nelinearitu výstupu a aproximovať ich lineárnymi vzťahmi. Na rozdiel od algebraickej analýzy, kedy sústava rovníc platí vždy, tu sa snažíme nájsť vzťahy, ktoré platia s pravdepodobnosťou čo najviac rôznou od 1/2. Vzťahy zahŕňajúce známy bežiaci kľúč následne poslúžia na konštrukciu rozpoznávača.

- Korelačné útoky sú typ útokov na stav šifry využitím toho, že výstup niektorej vnútornej časti šifry koreluje s celkovým výstupom šifry. V takomto prípade vieme buď úplné preberanie alebo inú sofistikovanejšiu metódu zamerať iba na túto časť šifry a ostatok ignorovať.

- Ďalšie metódy (testovanie štatistických hypotéz, útok postranným kanálom, ...) a rôzne ad-hoc prístupy pre konkrétnu šifru.

V práci sa venujeme kryptoanalýze šifier RC4 a Spritz, pričom chceme získať počiatočný stav pri znalosti bežiaceho kľúča. V práci používame dva typy metód: generický útok pomocou SMT solverov podobný algebraickej kryptoanalýze a ad-hoc metódu použitím prehľadávania (backtracking) vymyslenú pôvodne pre RC4.

Organizácia práce je preto nasledovná: práca sa delí do 5 kapitol. V prvej kapitole zavedieme označenia pre celú prácu, detailnejšie popíšeme šifry RC4 a Spritz a uvedieme známe výsledky kryptoanalýzy. V druhej kapitole popíšeme SMT solvery a ich použitie na získanie počiatočného stavu RC4 alebo Spritz. Uvedieme, ako preložiť popis šifier do jazyka SMT a experimentálne overíme metódu na redukovaných verziách šifier. Našu metódu pre RC4 porovnáme s algebraickou analýzou RC4 v [21]. V tretej kapitole uvedieme všeobecný popis útoku na stav šifry pomocou prehľadávania, experimentálne overíme jeho korektnosť a porovnáme ho s implementáciou v práci [2], kde bol pre Spritz prvý krát publikovaný. V štvrtej kapitole predstavíme viaceré vylepšenia útoku pomocou prehľadávania na Spritz a experimentálne porovnáme ich úspešnosť. Piata kapitola je venovaná odhadom zložitosti útokov prehľadávaním. Najprv demonštrujeme našu techniku zlepšením odhadu pre RC4 z článku [9]. Následne rovnakú metódu aplikujeme na vybrané verzie útokov prehľadávaním pre Spritz, pričom zlepšíme analýzu z [2]. Ďalej zlepšíme odhad pre prehľadávanie začínajúce v špeciálnom stave šifry Spritz z práce [3] a na záver analyzujeme tradeoff zložitosti útoku prehľadávaním pre šifru Spritz.

KAPITOLA 1

Označenia a popis šifier

V tejto kapitole najprv uvedieme definície a označenia použité v celej práci. Ďalej detailnejšie popíšeme šifry RC4 a Spritz, ich fungovanie a známe výsledky kryptoanalýzy.

1.1 Všeobecné označenia

Symbolom lg budeme označovať funkciu logaritmu pri základe 2, t.j. $\log_2$. Symbolom $\mathbb{Z}$ budeme označovať celé čísla, $\mathbb{Z}_N$ označuje množinu $\{0, \ldots, N-1\}$ pre ľubovoľné prirodzené číslo $N > 0$. Symbol $\mathbb{F}_2$ bude označovať konečné pole $GF(2)$.

1.2 RC4

RC4 je známa prúdová šifra s jednoduchým návrhom a implementáciou. Šifra bola v minulosti v praxi často používaná v rôznych schémach a protokoloch.

Stav šifry predstavuje permutácia S množiny $\mathbb{Z}_N$, kde N je veľkosť permutácie. Typická hodnota N je 256. Ďalej šifra obsahuje dva registre $i, j \in \mathbb{Z}_N$, slúžiace na aktualizáciu stavu a zároveň na generovanie bežiaceho kľúča z permutácie. Celkový stav šifry má teda veľkosť $\lg(N!N^2)$ bitov.

Inicializácia a vygenerovanie slova bežiaceho kľúča sú uvedené v Algoritme 1.1. Všetky sčítania sú počítané modulo N.

Šifra je inicializovaná funkciou KSA pomocou postupnosti K nad $\mathbb{Z}_N$, ktorá v sebe obsahuje kľúč a prípadne aj inicializačný vektor. Vstupná postupnosť sa použije na zamiešanie permutácie S, následne sú oba registre nastavené na 0. Označme $z_t \in \mathbb{Z}_N$ slovo bežiaceho kľúča vygenerované šifrou v čase $t \geq 0$ a $Z = \{z_t\}_{t \geq 0}$ celú postupnosť bežiaceho kľúča. Slovo bežiaceho kľúča je odvodené funkciou Update, ktorá aktualizuje oba registre, vymení (v prípade, že $i \neq j$) dve hodnoty v permutácii a odvodí slovo bežiaceho kľúča.

```
1: function KSA(K)
2:     for i = 0 to N − 1
3:         S[i] = i
4:     j = 0
5:     for i = 0 to N − 1
6:         j = j + S[i] + K[i mod |K|]
7:         Swap(S[i], S[j])
8:     i = j = 0
```

```
1: function Update()
2:     i = i + 1
3:     j = j + S[i]
4:     Swap(S[i], S[j])
5:     z = S[S[i] + S[j]]
6:     return z
```

Algoritmus 1.1: RC4

1.2.1 Známe výsledky kryptoanalýzy RC4

Šifra RC4 je známa pomerne dlhú dobu (vymyslená v roku 1987, kód reverzným inžinierstvom získaný a zverejnený v roku 1994 [19]) a existuje množstvo prác zaoberajúcich sa jej kryptoanalýzou. V návrhu a implementácii RC4 bolo nájdených viacero chýb a zraniteľností [1, 10, 17, 12, 8, 18] a preto sa nepovažuje za vhodné ju v praxi používať.

V tejto práci nás budú zaujímať dva typy útokov na RC4, ktoré sa budeme snažiť vylepšiť a neskôr aplikovať na Spritz: algebraická analýza (respektíve jej základná idea) a extrakcia stavu šifry pomocou prehľadávania s návratom.

Algebraická analýza RC4 bola v minulosti skúmaná v práci [21]. V tomto článku autori odvodili rovnice nad $\mathbb{F}_2$ pre sčítanie bitových vektorov, extrakciu bitu z bitového vektora a výmenu dvoch bitov vektora. Pomocou týchto rovníc následne popísali odvodenie bežiaceho kľúča RC4 ako nelineárnu sústavu nad $\mathbb{F}_2$ a nechali túto sústavu pre fixný bežiaci kľúč vyriešiť. Na riešenie sústavy použili algoritmy založené na Gröbnerových bázach z knižnice *Magma* (pre detaily pozri [21]). Tento postup bol schopný úspešne nájsť počiatočný stav RC4 pre $N = 4$.

Hľadaním stavu šifry pomocou prehľadávania s návratom sa venovalo viacero prác. V [9] bola táto metóda prvý krát aplikovaná na RC4, s odhadom zložitosti prehľadávania 2^{779} krokov pre plnú verziu šifry. V [12] autori ukazujú inú metódu, založenú na hľadaní špecifických vzoriek v bežiacom kľúči, ktoré sa použijú na efektívnejšie nájdenie stavu šifry. Táto metóda má zložitosť 2^{241} krokov (pre realistické predpoklady, horný odhad je 2^{579}).

1.3 Spritz

Prúdová šifra Spritz bola navrhnutá ako náhrada RC4 [19]. Cieľom návrhu bolo zachovať základné komponenty RC4 ako použitie permutácie v stave šifry a výmena prvkov permutácie pri aktualizácii a pritom pozmeniť ostatné časti tak, aby bola šifra odolná voči známym útokom.

Stav šifry predstavuje, rovnako ako pri RC4, permutácia S množiny $\mathbb{Z}_N$ veľkosti N a registre $i, j, k, z, a, w \in \mathbb{Z}_N$. Registre i, j, k majú podobnú úlohu ako registre i, j pri RC4, register z obsahuje posledné vygenerované slovo bežiaceho kľúča, a sa používa iba pri inicializácii a w je inkrement registra i pri aktualizácii stavu. Stav má teda veľkosť $\lg(N!N^6)$ bitov.

Pseudokód inicializácie a generovania bežiaceho kľúča uvádzame v Algoritme 1.2.

Inicializácia šifry je inšpirovaná „sponge“ konštrukciami [6], kde je možné šifru inicializovať ľubovoľne dlhým reťazcom a pridanie ďalšieho vstupu je možné aj počas generovania bežiaceho kľúča. Pokiaľ predpokladáme, že inicializácia šifry prebehne iba pred generovaním bežiaceho kľúča a sme v situácii tesne po poslednom volaní SHUFFLE, niektoré hodnoty registrov sú známe. Register w je závislý iba na dĺžke absorbovaného reťazca a jeho hodnota je známa. Na začiatku inicializácie má hodnotu 1 a mení sa iba vo funkcii WHIP na najbližšiu hodnotu nesúdeliteľnú s N. Hodnota N aj dĺžka absorbovanej postupnosti (kľúč, inicializačný vektor) sú väčšinou známe. Počet zavolaní UPDATE pri inicializácii tiež závisí iba od dĺžky absorbovanej postupnosti, hodnota i je teda tiež známa. Hodnota z je 0 a a sa pri odvodzovaní bežiaceho kľúča nepoužíva. Jediné neznáme (okrem S) sú registre j a k.

V ďalších kapitolách budú z hľadiska kryptoanalýzy podstatné hlavne funkcie UPDATE a OUTPUT, ktoré aktualizujú stav šifry a zo stavu odvodia slovo bežiaceho kľúča.

Poznámka. Môžeme si všimnúť, že funkcia UPDATE sa dá krokovať aj smerom dozadu. V prípade, že vieme ako vyzerá stav šifry, predchádzajúce hodnoty k a i vypočítame odčítaním, hodnotu j určíme nájdením indexu hodnoty $j - k$ v S.

1.3.1 Známe výsledky kryptoanalýzy Spritz

Štatistickým odchýlkam v bežiacom kľúči a konštrukciám rozpoznávačov sa venuje niekoľko prác. V návrhu šifry [19] naznačujú, že na rozpoznanie bežiaceho kľúča od náhodnej postupnosti je potrebných 2^{81} bajtov dlhá postupnosť. V práci [22] bol navrhnutý rozpoznávač pre redukovanú verziu šifry ($N = 8$), no štatistické odchýlky neboli ani dokázané ani ukázaný podobný problém pre plnú verziu šifry. V článku [3] autori dokázali štatistickú odchýlku prvých dvoch slov bežiaceho kľúča a skonštruovali rozpoznávač pre Spritz, ktorý potrebuje buď $2^{44.8}$ dvojíc kľúč-inicializačný vektor alebo $2^{60.8}$ bajtov bežiaceho kľúča.

V oblasti extrakcie počiatočného stavu existujú dve predchádzajúce práce: v [2] predstavujú modifikáciu metódy prehľadávania s návratom pre RC4, pričom odhadovaná zložitosť pre plnú verziu šifry je 2^{1400} krokov a v práci [3] autori vylepšujú predchádzajúci útok v prípade, že prehľadávanie začína v špeciálnej triede počiatočných stavov s dodatočnými vlastnosťami. Vtedy je odhadovaná zložitosť 2^{1247} krokov.

```
1: function InitializeState()
2:     i = j = k = z = a = 0
3:     w = 1
4:     for v = 0 to N − 1
5:         S[v] = v

1: function Absorb(I)
2:     for v = 0 to I.length − 1
3:         AbsorbByte(I[v])

1: function AbsorbByte(b)
2:     AbsorbNibble(low(b))
3:     AbsorbNibble(high(b))
4:     for v = 0 to I.length − 1
5:         S[v] = v

1: function AbsorbNibble(x)
2:     if a = ⌊N/2⌋ then
3:         Shuffle()
4:     Swap(S[a], S[⌊N/2⌋] + x)
5:     a = a + 1

1: function AbsorbStop()
2:     if a = ⌊N/2⌋ then
3:         Shuffle
4:     a = a + 1

1: function Shuffle()
2:     Whip(2N)
3:     Crush()
4:     Whip(2N)
5:     Crush()
6:     Whip(2N)
7:     a = 0

1: function Whip(r)
2:     for v = 0 to r − 1
3:         Update()
4:     repeat
5:         w = w + 1
6:     until GCD(w, N) = 1

1: function Crush()
2:     for v = 0 to r − 1
3:         if S[v] > S[N − 1 − v] then
4:             Swap(S[v], S[N − 1 − v])

1: function Squeeze(r)
2:     if a > 0 then
3:         Shuffle()
4:     P = Array.New(r)
5:     for v = 0 to r − 1
6:         P[v] = Drip()
7:     return P

1: function Drip()
2:     if a > 0 then
3:         Shuffle()
4:     Update()
5:     return Output()

1: function Update()
2:     i = i + w
3:     j = k + S[j + S[i]]
4:     k = i + k + S[j]
5:     Swap(S[i], S[j])

1: function Output()
2:     z = S[j + S[i + S[k + z]]]
3:     return z
```

Algoritmus 1.2: Spritz

KAPITOLA 2

Kryptoanalýza pomocou SMT solverov

V tejto kapitole sa zaoberáme kryptoanalýzou šifier RC4 a Spritz pomocou SMT solverov. Ukážeme ako preložiť popis oboch šifier do jazyka SMT. Experimentujeme s rôznymi parametrami veľkosti stavu a dĺžky bežiaceho kľúča pre viacero SMT solverov a dva typy SMT logiky. Ukážeme, že na korektné zistenie vnútorného stavu po inicializácii je nutné mať určité množstvo bežiaceho kľúča, ktorého nárast ale výrazne zvyšuje čas výpočtu SMT solvera.

Našim cieľom je aplikovať metódu podobnú algebraickej kryptoanalýze, ktorá je pre RC4 popísaná v [21], na šifry RC4 a Spritz. Postupujeme podobne, no namiesto pracného popisu šifry algebraickými rovnicami používame jazyk SMT-LIB v2 [4] a jeho vstavané konštrukcie pre polia, ktoré nám umožňujú popis šifry vyjadriť jednoduchšie.

Predpokladáme znalosť dostatočného množstva bežiaceho kľúča a chceme zistiť počiatočný stav šifry, z ktorého bol vygenerovaný. Pre RC4 to znamená stav po inicializácii, keď sú oba registre znova inicializované na 0 a permutáciu nepoznáme. Pre Spritz predpokladáme, že absorbovanie vstupu sa udeje iba pred generovaním bežiaceho kľúča a útočíme na stav po poslednom zavolaní Shuffle a pred prvým Update, teda poznáme aj hodnoty registrov i, z, w.

2.1 SMT

SMT (Satisfiability modulo theories) je oblasť automatického dokazovania tvrdení zaoberajúca sa metódami dokazovania splniteľnosti formúl logiky prvého rádu ohraničenej na konkrétnu logickú teóriu. SMT problémy môžeme chápať ako zovšeobecnenia problému splniteľnosti booleovkých formúl (SAT) na niektoré rozhodnuteľné podmnožiny logiky prvého rádu. Na rozdiel od booleovskej formuly je pri SMT problémoch povolené na zápis problému používať aj nebinárne premenné (celé čísla, bitové vektory, polia,

...), funkcie, predikáty a iné. Konkrétne povolené typy premenných a funkcií závisia na použitej teórii.

SMT solver je nástroj riešiaci SMT problémy. Pre konkrétny problém sa solver pokúša zistiť, či existuje splniteľné ohodnotenie premenných a prípadne jedno takéto ohodnotenie hľadá. Realizuje to buď prekladom SMT programu na SAT formulu a následným použitím SAT solvera alebo využije špecializovaný postup pre danú teóriu, prípadne kombinuje predspracovanie s použitím SAT solvera. Rôzne SMT solvery podporujú rôzne teórie a ich kombinácie, napríklad teóriu celých čísel, reálnych čísel, lineárnych rovníc s celými číslami, polí, bitových vektorov atď.

SMT solvery sa v praxi často používajú napríklad pri verifikácii programov, plánovaní alebo pri automatickom generovaní testov [14].

2.2 Výber logiky

V tejto práci používame teórie obsahujúce podporu polí, do ktorých môžeme indexovať premennou. Podpora polí, sčítanie a modulo nad celými číslami sú operácie, ktoré v modelovaní šifier potrebujeme, keďže práve takto obe šifry fungujú. Preto ako vhodní kandidáti pripadajú logiky QF_AUFLIA a QF_ABV (v ďalšom ich budeme označovať *auflia* a *abv*). Logika *auflia* podporuje formuly bez kvantifikátorov s celými číslami a poliami, logika *abv* bitové vektory fixnej dĺžky s poliami, taktiež vo formulách bez kvantifikátorov.

SMT programy formulujeme vo formáte SMT-LIB v2, ktorý je podporovaný mnohými SMT solvermi. To nám umožňuje jednoducho porovnávať efektívnosť rôznych solverov.

2.3 Modelovanie šifier

Cieľom je popísať vzťah medzi počiatočným stavom šifry a bežiacim kľúčom, pričom počiatočný stav aj bežiaci kľúč budú vystupovať v SMT programe ako premenné. Následne vieme zafixovať bežiaci kľúč na konkrétne hodnoty a nechať solver takýto problém vyriešiť. Výsledné ohodnotenie premenných nám povie, ako vyzerá počiatočný stav, z ktorého sa bol zafixovaný bežiaci kľúč vygenerovaný. Pre kontrolu korektnosti SMT programu vieme zafixovať počiatočný stav a ako neznáme hodnoty nechať bežiaci kľúč, potom by mal výsledok solvera zodpovedať bežiacemu kľúču odvodenému z daného počiatočného stavu.

Kľúčová vec pre náš postup je použitie SMT teórií s poliami. V súvislosti s poliami využijeme nasledovné operácie: `select` z poľa vráti prvok na danom indexe, `store` vyrobí z poľa nové pole, pričom jeden prvok poľa zmení na novú hodnotu. Ďalej použijeme sčítanie, modulo a základné logické operácie. Operácia modulo je v prípade

logiky *abv* implicitne zabudovaná v sčítaní bitových vektorov, pri logike *auflia* túto operáciu niektoré solvery nepodporovali a preto sme ju implementovali sami (pozri časť 2.3.4).

V nasledujúcich častiach prevedieme jednotlivé súčasti šifier do formátu SMT-LIB v2. Vzhľadom na podobnosti šifier RC4 a Spritz aj oboch logík, detailnejšie uvedieme zápis šifry Spritz v logike *abv*, pre ostatné kombinácie rozoberieme najdôležitejšie časti, kde sa líšia. Úplné SMT programy pre vybrané parametre sa nachádzajú v prílohe A. Pre názornosť používame v nasledujúcich častiach parametre: veľkosť permutácie $N = 8$, počet kôl (počet odvodených slov bežiaceho kľúča) $r = 2$.

2.3.1 Premenné, konštanty, počiatočné podmienky

Pre počiatočný stav a každé kolo šifry máme premennú pre permutáciu:

```
(declare-fun S0 () (Array (_ BitVec 3) (_ BitVec 3)))
(declare-fun S1 () (Array (_ BitVec 3) (_ BitVec 3)))
(declare-fun S2 () (Array (_ BitVec 3) (_ BitVec 3)))
```

a premenné pre registre šifry (i, j, k, z, w):

```
(declare-fun i0 () (_ BitVec 3))
(declare-fun i1 () (_ BitVec 3))
(declare-fun i2 () (_ BitVec 3))
(declare-fun j0 () (_ BitVec 3))
(declare-fun j1 () (_ BitVec 3))
(declare-fun j2 () (_ BitVec 3))
(declare-fun k0 () (_ BitVec 3))
(declare-fun k1 () (_ BitVec 3))
(declare-fun k2 () (_ BitVec 3))
(declare-fun z0 () (_ BitVec 3))
(declare-fun z1 () (_ BitVec 3))
(declare-fun z2 () (_ BitVec 3))
(declare-fun w () (_ BitVec 3))
```

Register a sa pri generovaní bežiaceho kľúča nepoužíva.

Ďalej musíme zabezpečiť, že premenné $S0, S1, S2$ budú obsahovať permutáciu N prvkov a nie ľubovoľné hodnoty, čo docielime sadou podmienok:

```
(assert (not (= (select S0 #b000) (select S0 #b001) )))
(assert (not (= (select S0 #b000) (select S0 #b010) )))
(assert (not (= (select S0 #b000) (select S0 #b011) )))
...
(assert (not (= (select S0 #b110) (select S0 #b111) )))
```

Podmienky stačí aplikovať na $S0$, keďže ďalšie permutácie odvodzujeme iba pomocou operácie `store`, výmenou dvoch prvkov, teda následné premenné $S1, S2$ budú opäť permutácie.

Pri útoku na počiatočný stav máme zafixovaný známy bežiaci kľúč (premenné z) a známe hodnoty registrov i a w.

```
(assert (= i0 #b011))
(assert (= w #b001))
(assert (= z0 #b000))
(assert (= z1 #b011))
(assert (= z2 #b101))
```

V prípade simulácie šifrovania zafixujeme premenné pre počiatočnú permutáciu ($S0$) a registre (premenné registrov s indexom 0) na príslušné hodnoty.

2.3.2 Kolo šifry

V jednom kole šifry simulujeme funkciu odvodenia slova bežiaceho kľúča. Prvé kolo:

```
(assert (= i1 (bvadd i0 w)))
(assert (= j1 (bvadd k0 (select S0 (bvadd j0 (select S0 i1))))))
(assert (= k1 (bvadd i1 (bvadd k0 (select S0 j1)))))
(assert (= S1 (store (store S0 j1 (select S0 i1)) i1 (select S0 j1))))
(assert (= z1
   (select S1 (bvadd j1 (select S1 (bvadd i1 (select S1 (bvadd z0 k1))))))
))
```

2.3.3 Ostatné direktívy, cieľ

V SMT programe potrebujeme solveru povedať typ logiky, ktorý budeme používať a tiež, že má okrem zisťovania splniteľnosti aj jedno splniteľné ohodnotenie nájsť.

```
(set-option :produce-models true)
(set-logic QF_ABV)
```

V prípade útoku na stav šifry chceme vypísať počiatočné hodnoty permutácie a registrov:

```
(get-value ((select S0 #b000)))
(get-value ((select S0 #b001)))
...
(get-value ((select S0 #b111)))
(get-value (i0))
```

```
(get-value (j0))
(get-value (k0))
(get-value (z0))
(get-value (w))
```

a v prípade šifrovania hodnoty bežiaceho kľúča:

```
(get-value (z1))
(get-value (z2))
```

Na konci programu je príkaz

```
(exit)
```

2.3.4 Šifra RC4, logika *auflia*

SMT popis RC4 je veľmi podobný Spritz, rozdiely sú v jednoduchšom kole šifry a menšom počte premenných.

Pri logike celých čísel *auflia* potrebujeme zaručiť, že jednotlivé premenné nenadobudnú hodnoty väčšie ako maximálne možné hodnoty registrov (predpokladáme, že registre sú obmedzené na práve toľko bitov, aby umožnili indexovať do celej permutácie), inak by si SMT solver mohol do medzivýsledkov dosadiť ľubovoľne veľké čísla a prišiel by k nekorektnému výsledku. Preto potrebujeme ku každej hodnote pridať obmedzenia typu:

```
...
(assert (>= j1 0))
(assert (< j1 8))
...
(assert (>= (select S0 3) 0))
(assert (< (select S0 3) 8))
...
```

Ďalej si ešte potrebujeme definovať vlastné modulo, čo spravíme pomocou vstavanej konštrukcie podmienky. Keďže používame iba sčítanie s hodnotami, ktoré sú najviac N, modulo nám stačí implementovať ako jedno odčítanie, pokiaľ hodnota pretečie nad N.

```
(define-fun modulo ((x Int) (y Int)) Int (ite (>= x y) (- x y) x))
```

2.3.5 Testovanie jednoznačnosti

SMT solver sa pri riešení problému snaží zistiť, či je daný problém splniteľný a pokiaľ áno, nájsť spĺňajúce ohodnotenie. Toto ohodnotenie ale nemusí byť nutne jediné korektné, takých, ktoré spĺňajú všetky podmienky SMT programu môže byť veľa. Nás

bude okrem iného zaujímať aj to, či je výstup solveru jednoznačný, prípadne ak nie, či je možných spĺňajúcich ohodnotení málo (napríklad šifrovanie je deterministické a teda prítomnosť viacerých spĺňajúcich ohodnotení bežiaceho kľúča by indikovala chybu v našom SMT popise šifry).

Testovanie jednoznačnosti implementujeme jednoducho: po tom, ako solver vypíše výstup zoberieme rovnaký program, pričom k nemu pripojíme pravidlo, že príslušné premenné nesmú naraz nadobúdať hodnoty, ktoré sme práve dostali zo solvera a spustíme SMT solver znova. Tento postup iterujeme dostatočne veľa krát, prípadne kým nám solver nepovie, že program už nie je splniteľný.

Pre konkrétny stav by sa pri kontrole jednoznačnosti šifrovania po prvom spĺňajúcom ohodnotení pridal riadok:

```
(assert (not  (and (= z1 #b011) (= z2 #b101))))
```

2.3.6 Počty premenných

K zložitosti SMT programov okrem iného prispieva aj počet neznámych premenných, ktorých hodnoty musí solver zistiť. Pre porovnanie preto v ďalšom vyčíslime počty neznámych premenných pre oba typy problémov (šifrovanie a zisťovanie počiatočného stavu) a obe šifry.

Uvažujeme veľkosť permutácie $N = 2^n$ a r kôl šifry, tabuľka 2.1 ukazuje počty n-bitových neznámych premenných, ktoré musí solver zistiť.

	šifrovanie	extrakcia stavu
RC4	$rN + r$	$(r + 1)N + r$
Spritz	$rN + 4r$	$(r + 1)N + (2r + 2)$

Tabuľka 2.1: Počet n-bitových neznámych premenných

2.4 Testovacie prostredie, SMT solvery

Naše testy sme vykonali na OS Linux Mint 17 Qiana 64bit s procesorom Intel Core 2 Quad Q9400 (2.66 GHz) s 4GB pamäte.

Použili sme nasledujúce SMT solvery s príslušnými 64-bitovými verziami: Z3 4.4.0 [15], Yices 2.4.1 [7], CVC4 1.4 [5] a Boolector 2.1.1 [16]. Logiku *auflia* podporujú Z3 a CVC4, logiku *abv* Z3, Yices a Boolector. Pri všetkých solveroch sme použili štandardné nastavenia a solvery bežali na jednom jadre procesora. Na automatizované generovanie SMT programov, spúšťanie solverov, meranie časov a kontrolu korektnosti sme použili vlastný nástroj napísaný v jazyku Python, pozri prílohu B.

2.5 RC4

V tejto časti sa podrobnejšie zaoberáme šifrou RC4. Cieľom bude porovnať logiky *auflia* a *abv*, rôzne SMT solvery a celkovo náš postup s algebraickou analýzou v práci [21].

2.5.1 Generovanie bežiaceho kľúča

Generovanie bežiaceho kľúča sme testovali vygenerovaním náhodného počiatočného stavu, ten sme v SMT programe zafixovali a nechali solver vyriešiť ho pre neznáme premenné bežiaceho kľúča.

Použili sme veľkosť stavu $N = 256$ (čo je v praxi realistická hodnota) a štyri rôzne dĺžky vygenerovaného bežiaceho kľúča $r \in \{50, 100, 500, 1000\}$.

SMT solver		Priemerný čas [sekundy] $r = 50$	$r = 100$	$r = 500$	$r = 1000$
auflia	Z3	3.43	27.30	254.37	–
	CVC4	0.16	0.43	3.35	7.53
abv	Z3	0.02	0.04	0.41	1.04
	Boolector	0.01	0.03	0.18	0.51
	Yices	0.17	0.71	21.92	109.33

Pozn. 1: Hodnoty časov sú priemer z 10 experimentov.
Pozn. 2: Hodnota „-“ znamená, že žiadny experiment neskončil do 10 minút.

Tabuľka 2.2: Generovanie bežiaceho kľúča pre $N = 256$

Ako môžeme vidieť z tabuľky 2.2, pre rastúce r sa očakávane zvyšuje čas potrebný na nájdenie riešenia, keďže je nutné nájsť správne hodnoty pre čoraz viac premenných. Keďže šifrovanie je deterministické, tak očakávame, že solver vyrieši problém rýchlo. Hodnoty bežiaceho kľúča sa zhodovali s implementáciou šifry v jazyku Python a výsledky SMT solverov boli jedinečné, čo potvrdzuje korektnosť nášho modelu šifry v SMT-LIB v2.

Mierne nečakaný je nepomer medzi časmi rôznych kombinácií logika a solver. Keďže tento experiment mal za cieľ overiť korektnosť SMT programov a nejedná sa o kryptoanalýzu, tejto výchylke sme sa ďalej nevenovali.

2.5.2 Extrakcia stavu

Problém, ktorému sa chceme venovať hlbšie je extrakcia stavu šifry pri znalosti bežiaceho kľúča. Prvým cieľom bude porovnať z hľadiska rýchlosti obe SMT logiky a viacero SMT solverov a vybrať najvhodnejšiu kombináciu.

Nie je realistické očakávať, že extrakcia stavu bude fungovať v rozumnom čase pre reálne parametre šifry, preto budeme uvažovať redukovanú verziu šifry s malou veľkosťou stavu. Použijeme $N = 8$ a $r \in \{5, 10, 15, 20\}$.

Bežiaci kľúč pre všetky pokusy sme vygenerovali implementáciou šifry v jazyku Python z náhodného počiatočného stavu. Podotýkame, že hodnoty v tabuľkách sú časy nájdenia prvého počiatočného stavu, ktorý je konzistentný s bežiacim kľúčom. Tento stav šifry ale nemusí byť nutne ten, z ktorého sme problém vygenerovali.

SMT solver		Priemerný čas [sekundy] $r = 5$	$r = 10$	$r = 15$	$r = 20$
auflia	Z3	0.65	2.22	2.95	13.87
	CVC4	50.92	216.95	534.00	–
abv	Z3	0.14	0.21	0.29	0.71
	Boolector	0.16	0.62	1.66	8.06
	Yices	0.08	2.08	3.17	3.14

Pozn. 1: Hodnoty časov sú priemer z 10 experimentov.
Pozn. 2: Hodnota „-“ znamená, že žiadny experiment neskončil do 10 minút.

Tabuľka 2.3: Extrakciu stavu $N = 8$

SMT solver	Priemerný čas [sekundy] $r = 5$	$r = 6$	$r = 7$	$r = 8$	$r = 9$
Z3	0.05	0.14	0.62	2.01	13.52
Boolector	0.70	1.80	3.32	12.53	66.30
Yices	0.14	0.26	1.15	4.18	31.68

Pozn.: Hodnoty časov sú priemer z 10 experimentov.

Tabuľka 2.4: Extrakciu stavu $N = 16$, logika *abv*

Z tabuľky 2.3 vyplýva, že logika *abv* dosahuje lepšie výsledky. Menšie časy pre *abv* sú očakávané, keďže v logike *abv* sú operácie s číslami z obmedzeného rozsahu implicitne zabudované, zatiaľ čo pri *auflia* musíme obmedzený rozsah vynútiť sadou podmienok. Dodatočné podmienky jednak komplikujú SMT program a za druhé SMT solver implementujúci logiku *auflia* nevie dodatočnú informáciu o obmedzenom rozsahu všetkých premenných využiť tak efektívne ako pri logike *abv*.

Výsledky experimentu pre $N = 16$ sú uvedené v tabuľke 2.4. Ako vidíme z tabuliek 2.3 a 2.4, Z3 je spomedzi testovaných solverov najrýchlejší. V nasledujúcich experimentoch budeme vždy používať logiku *abv* a SMT solver Z3.

	Jednoznačné stavy (zo 100)		
r	$N = 4$	$N = 8$	$N = 16$
2	32	0	0
3	82	0	0
4	93	0	0
5	100	26	0
6	100	82	0
7	100	99	0
8	100	100	0
9	100	100	$*0$
10	100	100	$*0$

Pozn.: Hodnoty označené * sú z 10 stavov.

Tabuľka 2.5: Počty jednoznačných stavov pre rôznu dĺžku bežiaceho kľúča

2.5.3 Jednoznačnosť počiatočného stavu

V predchádzajúcich experimentoch sme sa nezaoberali tým, či bol nájdený stav bežiacim kľúčom jednoznačne určený. Pri malej dĺžke bežiaceho kľúča sa môže stať, že bude existovať viacero počiatočných stavov, ktoré vyprodukujú ten istý bežiaci kľúč a solver nájde konzistentnú, ale zlú odpoveď.

Z náhodného stavu vygenerujeme bežiaci kľúč a necháme solver riešiť problém nájdenia počiatočného stavu. Jednoznačnosť výsledku kontrolujeme tak, ako sme popísali v časti 2.3.5. Počiatočné stavy pre fixné N sú pre rôzne k vždy rovnaké, SMT programy sa líšia iba dĺžkou známeho bežiaceho kľúča.

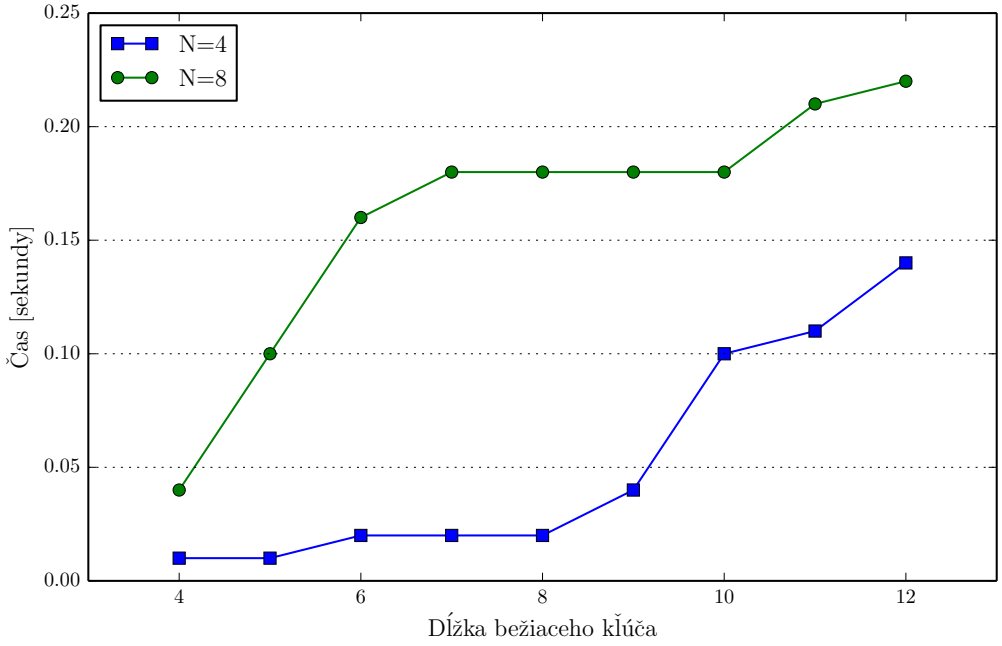
V tabuľke 2.5 môžeme pre rôzne N vidieť dĺžky bežiaceho kľúča, kedy už boli všetky počiatočné stavy z experimentu jednoznačne určené. Pre $N = 16$ a $r > 10$ bola extrakcia stavu príliš pomalá a pre tieto parametre sme jednoznačnosť netestovali.

2.5.4 Dĺžka bežiaceho kľúča a zložitosť extrakcie stavu

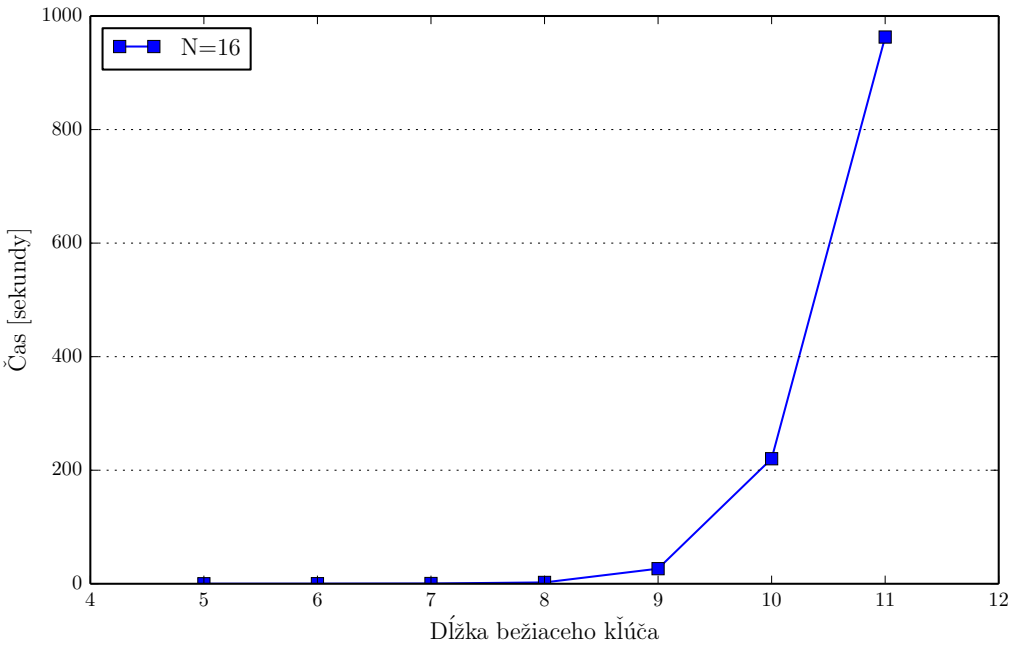
V tejto časti experimentálne overíme, ako so zväčšujúcim sa bežiacim kľúčom rastie čas potrebný na nájdenie počiatočného stavu šifry. Budeme uvažovať veľkosti permutácií $N \in \{4, 8, 16\}$ a variabilnú dĺžku bežiaceho kľúča r.

Hodnoty v grafoch sú priemery časov nájdenia počiatočného stavu (prvého konzistentného s bežiacim kľúčom).

V experimentoch pre $N \in \{4, 8\}$ sme vykonali 100 pokusov namiesto 10 z dôvodu kolísania časov výpočtu, kedy sa zväčšením bežiaceho kľúča znížil priemerný čas výpočtu solvera. Toto správanie môžeme pripísať veľmi malým hodnotám času výpočtu, prípadne sme narazili na situáciu, kde pridanie podmienok do SMT programu pomôže solveru k nájdeniu riešenia. Pri vykonaní 100 pokusov sa v priemernej hodnote času výpočtu tieto výchylky už neprejavili.



Obrázok 2.1: Porovnanie extrakcie stavu pre $N \in \{4, 8\}$, priemer zo 100 pokusov



Obrázok 2.2: Extrakcia stavu pre $N = 16$, priemer z 10 pokusov

Pre hodnoty $N \in \{4, 8\}$, ako môžeme vidieť na obrázku 2.1, SMT solver vyrieši problém veľmi rýchlo aj pre dĺžky bežiaceho kľúča, keď už očakávame jednoznačnosť výsledku. Čas výpočtu by pri rastúcej dĺžke bežiaceho kľúča mal stúpať, keďže rastie počet premenných a pridávame ďalšie podmienky na ohodnotenie, ktoré musí solver zachovať.

Na obrázku 2.2 môžeme pozorovať, že pri zväčšení stavu sa riešenie SMT problému skomplikuje. Na rozdiel od menších veľkostí N, pre $N = 16$ rastie s dĺžkou bežiaceho kľúča čas riešenia rýchlejšie, pričom k hodnotám, kedy je stav jednoznačný sme sa ani nedostali.

Pri testoch s $N = 16$ sme pri hodnote $r = 11$ pozorovali pomerne veľké výchylky v dĺžkach výpočtu. Z 10 testov sa SMT programy rozdelili do dvoch kategórií: v jednej bolo vždy prvé nájdené ohodnotenie iné ako to, z ktorého sme pokus vygenerovali a časy výpočtu boli 600 až 800 sekúnd, v druhej kategórii sa SMT solveru podarilo nájsť správne ohodnotenie, ale čas výpočtu bol oveľa dlhší (nad 1200 sekúnd). Domnievame sa, že to bolo spôsobené tým, že pri dlhotrvajúcich pokusoch sme trafili na jednoznačný stav (podľa experimentov v tabuľke 2.5 odhadujeme, že pri $r = 11$ by sa jednoznačné stavy už mohli vyskytovať), pričom v týchto prípadoch si SMT solver nemohol vyberať prvé ohodnotenie, ktoré nemalo konflikty a teda mu výpočet trval dlhšie.

Pre $N = 16$ sa nám pre malé hodnoty r síce podarilo nájsť počiatočný stav, no tento s veľkou pravdepodobnosťou nebude jedinečný a nebude to ten, ktorý bol použitý na vygenerovanie bežiaceho kľúča. Z našich experimentov pre RC4 teda vyplýva, že $N = 16$ je veľkosť stavu šifry, kedy časová zložitosť nájdenia počiatočného stavu začne exponenciálne stúpať už pre malé dĺžky bežiaceho kľúča, pričom tieto ešte nestačia na jednoznačné určenie stavu. Exponenciálny nárast pozorujeme pri hodnotách dĺžky bežiaceho kľúča 10 a 11.

2.6 Spritz

V tejto časti zopakujeme rovnaké experimenty ako v predchádzajúcich častiach kapitoly, tentoraz na šifre Spritz. Rovnako budeme používať solver Z3 a logiku *abv*.

2.6.1 Generovanie bežiaceho kľúča

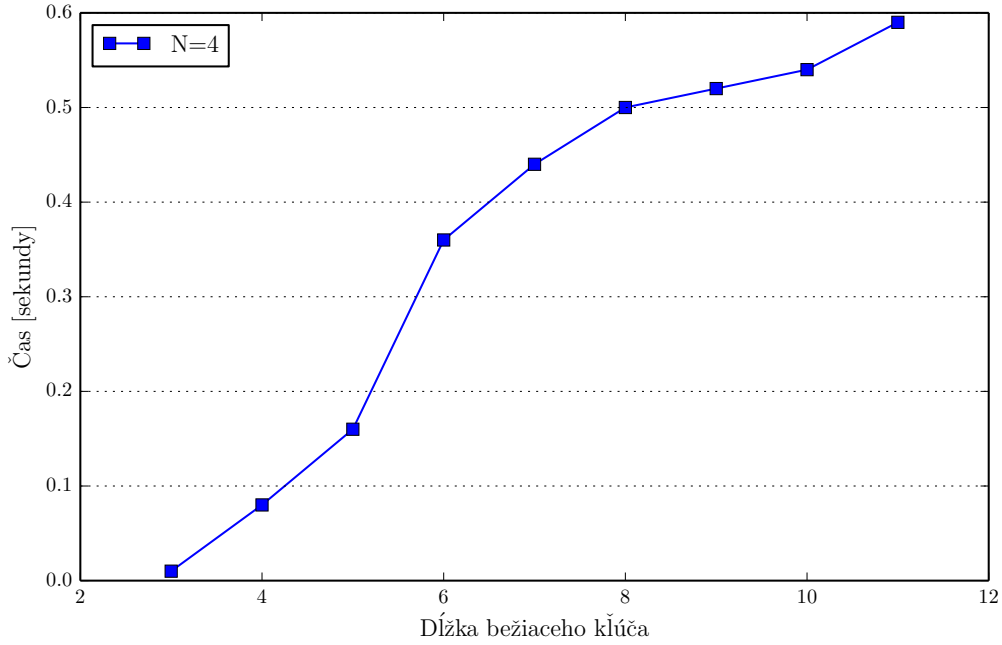
Najprv otestujeme korektnosť SMT popisu šifry vypočítaním bežiaceho kľúča SMT solverom a porovnaním ho s implementáciou šifry v jazyku Python. Použité parametre sú $N = 256$ a $r \in \{50, 100, 500, 1000\}$

Všetky vypočítané bežiace kľúče boli jednoznačné a zhodovali sa s hodnotami vygenerovanými šifrou v jazyku Python.

	Priemerný čas [sekundy]			
SMT solver	$r = 50$	$r = 100$	$r = 500$	$r = 1000$
Z3	0.40	0.44	1.06	2.21

Pozn.: Hodnoty časov sú priemer z 10 experimentov.

Tabuľka 2.6: Generovanie bežiaceho kľúča pre $N = 256$



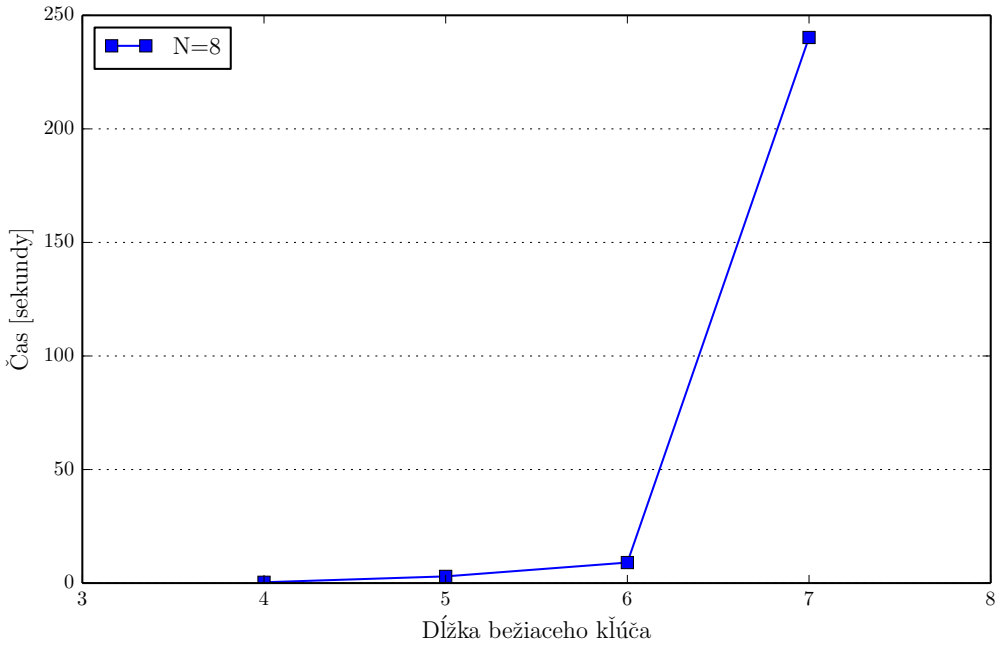
Obrázok 2.3: Extrakcia stavu pre $N = 4$

2.6.2 Dĺžka bežiaceho kľúča a zložitosť extrakcie stavu

Vykonali sme rovnaké experimenty ako v časti 2.5.4. Na grafoch 2.3 a 2.4 sú znázornené časy nájdenia prvého riešenia pre $N \in \{4, 8\}$ a rôzne r (priemer z 10 pokusov).

Z experimentov môžeme vidieť, že hodnoty so stúpajúcimi kolami šifry očakávane rastú. Pri porovnaní s šifrou RC4 sú časy pre obe hodnoty N vyššie, keď už pre $N = 8$ sa dostávame do rádu minút. Zvýšenie zložitosti je očakávané, keďže na rozdiel od RC4 je v programoch použitých viac premenných a taktiež odvodenie slova bežiaceho kľúča zo stavu šifry je komplikovanejšie. Do situácie, keď čas oproti predchádzajúcej hodnote rapídne narástol sme sa tiež dostali pre menšie parametre.

Experimentovali sme s extrakciou stavu aj pre parametre $N = 8$ a $k = 8$ (tieto hodnoty už nie sú v grafe), no pre tieto parametre solver pri viacerých pokusoch neskončil výpočet ani po 35 minútach, po ktorých sme ho ukončili ručne. Predpokladáme, že tieto hodnoty parametrov sa už blížia maximu, čo sa uvedenou metódou na našom hardvéri dá dosiahnuť.



Obrázok 2.4: Extrakcia stavu pre $N = 8$

2.6.3 Jednoznačnosť počiatočného stavu

Jednoznačnosť počiatočného stavu šifry Spritz sme testovali rovnakým spôsobom ako pri RC4, výsledky experimentov sú v tabuľke 2.7.

	Jednoznačné stavy (zo 100)	
r	$N = 4$	$N = 8$
4	0	0
6	44	0
8	96	–
10	100	–
12	100	–

Tabuľka 2.7: Počty jednoznačných stavov pre rôznu dĺžku bežiaceho kľúča

Pre $N = 8$ a $r \geq 8$ bola extrakcia stavu už príliš pomalá a pre tieto parametre sme jednoznačnosť netestovali.

Porovnaním experimentov pre RC4 v tabuľke 2.5 a Spritz v tabuľke 2.7 môžeme pozorovať, že pri šifre Spritz je pre jednoznačný stav potrebný dlhší známy bežiaci kľúč. To je spôsobené komplikovanejším odvodením slova bežiaceho kľúča, kde znalosť jedného slova odhaľuje menej informácie o počiatočnom stave než pri šifre RC4. Navyše, pri extrakcii stavu okrem permutácie sú neznáme aj hodnoty registrov j a k.

2.7 Zhrnutie výsledkov

Ukázali sme, ako preložiť šifry RC4 a Spritz do jazyka SMT a ako tento popis použiť na nájdenie počiatočného stavu šifry zo znalosti bežiaceho kľúča. Na jednoznačné nájdenie počiatočného stavu je potrebný dostatočne dlhý známy bežiaci kľúč, čo ale spomaľuje výpočet SMT solvera. Experimentálne sme preskúmali na redukovaných verziách šifier hranicu, kedy začne čas výpočtu SMT solvera exponenciálne stúpať.

Použitie SMT solverov na extrakciu počiatočného stavu šifry RC4 sa ukázalo efektívnejšie než algebraická analýza v práci [21]. V [21] autori úspešne našli počiatočný stav pre $N = 4$, no pre $N = 8$ bol tento postup výpočtovo príliš náročný. Našim postupom sme boli schopní nájsť jednoznačné riešenie pre $N = 8$ v čase pod 0.3 sekundy na našom hardvéri. Zároveň modelovanie šifier pomocou jazyka SMT-LIB v2 je omnoho jednoduchšie a menej pracné než metóda použitá v [21].

KAPITOLA 3

Extrakcia stavu pomocou prehľadávania

V tejto kapitole predstavíme triedu útokov na počiatočný stav šifry pomocou prehľadávania s návratom (backtracking). Detailne popíšeme tento typ útoku na šifre Spritz a experimentálne overíme jeho korektnosť. Porovnáme naše experimenty s prácou [2].

Cieľom tohto útoku je zistiť počiatočný stav šifry pri znalosti bežiaceho kľúča. Zameriame sa na šifru Spritz, pre RC4 je tento typ útoku veľmi podobný a okrem toho sú známe efektívnejšie spôsoby extrakcie stavu, napríklad [12]. Útok sa opiera o nasledujúce predpoklady:

- Dostatočne dlhý známy bežiaci kľúč. V experimentoch sme pozorovali, že stačí $3N$ slov bežiaceho kľúča. Samotné prehľadávanie nikdy nepotrebovalo viac než N slov a ďalších $2N$ bolo použitých na kontrolu korektnosti v situáciách, keď prehľadávanie vyplnilo celý stav.
- Absorbovanie vstupu sa udeje iba pred generovaním bežiaceho kľúča, hľadaný stav je stav po poslednom zavolaní Shuffle a pred prvým volaním Update.
- Poznáme hodnoty všetkých registrov stavu, ktorý hľadáme (neznáma je iba permutácia). Hodnoty registrov i, z, w sú známe počas celého generovania bežiaceho kľúča, hodnoty j a k zistíme napríklad úplným preberaním.

3.1 Popis útoku

Útok bol v základnej verzii navrhnutý v [2] a prebieha ako rekurzívne prehľadávanie priestoru všetkých permutácií nad $\mathbb{Z}_N$. Jediná neznáma v počiatočnom stave je permutácia S, čo znamená, že všetky jej hodnoty sú na začiatku nastavené na None. Prehľadávanie bude simulovať funkcie Update a Output, pričom ak je potrebná nejaká hodnota $S[x]$, ktorú nepoznáme, tipneme si ju – do $S[x]$ postupne dosadíme všetky

prípustné hodnoty (ktoré v S ešte nie sú použité, keďže S je permutácia) a pre každý tip pokračujeme ďalej v prehľadávaní.

V algoritme 3.1 uvádzame pseudokód základnej verzie prehľadávania – rekurzívnu funkciu simulujúcu UPDATE a tipujúcu hodnoty S.

Počas jedného prechodu funkciami UPDATE a OUTPUT si potrebujeme tipnúť maximálne 5 neznámych hodnôt: $S[i]$, $S[j + S[i]]$, $S[j]$, $S[z + k]$ a $S[i + S[z + k]]$ (presne v tomto poradí). V pseudokóde sú to riadky 4, 7, 10, 15 a 18.

Hodnoty bežiaceho kľúča použijeme na kontrolu korektnosti – funkcia OUTPUT musí vrátiť správny výstup. Označme $d = j + S[i + S[z + k]]$. Poznamenajme, že táto hodnota je známa, všetky neznáme hodnoty potrebné na jej výpočet už boli v algoritme určené. Nech prehľadávanie práve simuluje t-ty prechod funkciami UPDATE a OUTPUT a z_t je známe slovo bežiaceho kľúča, ktoré by malo byť vygenerované v aktuálnom kroku. Môžu nastať štyri prípady:

1. $S[d]$ je známe a $z_t = S[d]$: Výsledok OUTPUT je známy a rovnaký ako bežiaci kľúč. Pokračujeme vo výpočte ďalej, riadok 22 v pseudokóde.

2. $S[d]$ je známe a $z_t \neq S[d]$: Výsledok OUTPUT je iný než bežiaci kľúč na príslušnom mieste, našli sme konflikt s bežiacim kľúčom. Riadok 24 v pseudokóde.

3. $S[d]$ je neznáme a $z_t \notin S$: Jediná možnosť, ako môže byť stav konzistentný s bežiacim kľúčom je, že z_t bude v $S[d]$. Priradíme $S[d] = z_t$ a pokračujeme vo výpočte. Riadky 27 a 28 v pseudokóde.

4. $S[d]$ je neznáme a $z_t \in S$: Na mieste d nemôže byť z_t lebo S je permutácia, našli sme konflikt. Riadok 30 v pseudokóde.

V prípade konfliktu (prípady 2 a 4) nebudeme pokračovať v ďalšom prehľadávaní tejto vetvy výpočtu. Vyhnutím sa prehľadávaniu vetiev výpočtu, kde sme niečo tipli zle (a vieme to zistiť) znížime časovú zložitosť oproti úplnému preberaniu všetkých permutácií.

Keď v prehľadávaní vyplníme celú permutáciu, overíme korektnosť tohto stavu nasledovne: budeme z neho generovať slová bežiaceho kľúča a porovnávať ich so známym bežiacim kľúčom (v experimentoch sme pri kontrole korektnosti porovnávali $2N$ slov). Pokiaľ sa všetky rovnajú, našli sme správny stav. Počiatočný stav vypočítame invertovaním funkcie UPDATE.

Vstup: Bežiaci kľúč $\{z_t\}_{t\geq 0}^{3N}$, N, prázdna permutácia S
Výstup: Permutácia S pred odvodením z_0

1: **function** RecoverState(t, S, i, j, k, w)
2: $i_{next} = i + w$
3: **if** $S[i_{next}] =$ None **and** $g_1 \notin S$ **then**
4: Priraď $S[i_{next}] = g_1$ ▷ **for** $g_1 = 0$ **to** $N - 1$
5: $a = j + S[i_{next}]$
6: **if** $S[a] =$ None **and** $g_2 \notin S$ **then**
7: Priraď $S[a] = g_2$ ▷ **for** $g_2 = 0$ **to** $N - 1$
8: $j_{next} = j + S[a]$
9: **if** $S[j_{next}] =$ None **and** $g_3 \notin S$ **then**
10: Priraď $S[j_{next}] = g_3$ ▷ **for** $g_3 = 0$ **to** $N - 1$
11: $k_{next} = k + i_{next} + S[j_{next}]$
12: Swap($S[i_{next}], S[j_{next}]$)
13: $b = z_{t-1} + k_{next}$
14: **if** $S[b] =$ None **and** $g_4 \notin S$ **then**
15: Priraď $S[b] = g_4$ ▷ **for** $g_4 = 0$ **to** $N - 1$
16: $c = i_{next} + S[b]$
17: **if** $S[c] =$ None **and** $g_5 \notin S$ **then**
18: Priraď $S[c] = g_5$ ▷ **for** $g_5 = 0$ **to** $N - 1$
19: $d = j_{next} + S[c]$
20: **if** $S[d] \neq$ None **then**
21: **if** $z_t = S[d]$ **then**
22: RecoverState($t + 1, S, i_{next}, j_{next}, k_{next}, w$)
23: **else**
24: Konflikt, skúsiť iné priradenie
25: **else**
26: **if** $z_t \notin S$ **then**
27: Priraď $S[d] = z_t$
28: RecoverState($t + 1, S, i_{next}, j_{next}, k_{next}, w$)
29: **else**
30: Konflikt, skúsiť iné priradenie

Algoritmus 3.1: Základné prehľadávanie s návratom (prebrané a upravené z [2])

Poznámka. Všimnime si, že prehľadávanie môžeme rovnako dobre začať pre stav v ľubovoľnom čase pred vykonaním funkcie UPDATE. Predpokladáme dostatok známeho bežiaceho kľúča, registre i, z, w poznáme v ľubovoľnom čase a hodnoty j, k musíme získať úplným preberaním pre ľubovoľný začiatok prehľadávania.

Poznámka. Pre RC4 je útok prehľadávaním veľmi podobný. Predpoklady sú iba dostatočne dlhý známy kľúč, keďže v RC4 sú oba registre pred generovaním bežiaceho kľúča nastavené na 0 (pozri funkciu KSA v algoritme 1.1). Samotné prehľadávanie prebieha podobne, jediná zmena je v nutnosti tipovať maximálne tri neznáme hodnoty permutácie: $S[i]$, $S[j]$ a $S[i] + S[j]$ (v tomto poradí). Pri kontrole korektnosti sa podobne rozoberú štyri možnosti hodnôt z_t a $d = S[i] + S[j]$.

3.2 Experimenty

Prehľadávanie s návratom sme implementovali ako rekurzívnu funkciu v jazyku Python, pre detaily pozri prílohu B. Implementáciu sme otestovali pre redukované parametre šifry.

Experimenty sme vykonali na náhodne vygenerovaných stavoch. Registre i, j, k a permutácia S boli zvolené náhodne, register w bol náhodný, ale nesúdeliteľný s N. Registre a, z boli nastavené na 0, t.j rovnako, ako po inicializácii šifry (pozri algoritmus 1.2). V prípade väčších hodnôt N, kedy by bolo prehľadávanie príliš pomalé sme prednastavili niekoľko (označujeme x) hodnôt permutácie, čím sme hľadanie urýchlili, keďže prednastavené hodnoty netreba tipovať. Indexy prednastavených hodnôt boli vybrané náhodne a hodnoty boli nastavené korektne (podľa permutácie, ktorú sa prehľadávanie snaží nájsť).

Ako metriku zložitosti prehľadávania uvažujeme celkový počet priradení hodnôt do permutácie počas behu algoritmu. V tabuľke 3.1 a uvádzame niekoľko experimentálnych hodnôt zložitostí a porovnanie s počtom všetkých permutácií. Počet priradení je očakávane menší než prislúchajúce úplné preberanie všetkých permutácií.

N	$\lg(N!)$	Prehľadávanie
6	9.49	8.21
8	15.30	12.09
10	21.79	16.40
12	28.84	21.11

Pozn.: Posledný stĺpec predstavuje „lg“ počtu priradení, priemer z 1000 experimentov.

Tabuľka 3.1: Zložitosť základného prehľadávania

N	x	$\lg((N-x)!)$	[2]	Prehľadávanie
8	0	15.30	11.70	12.09
16	5	25.25	18.50	15.87
32	19	32.53	21.30	14.42
64	47	48.33	20.80	15.14

Pozn. 1: Posledné dva stĺpce predstavujú „lg“ počtu priradení.
Pozn. 2: Posledný stĺpec je priemer z 1000 experimentov.

Tabuľka 3.2: Porovnanie základného prehľadávania s implementáciou v [2]

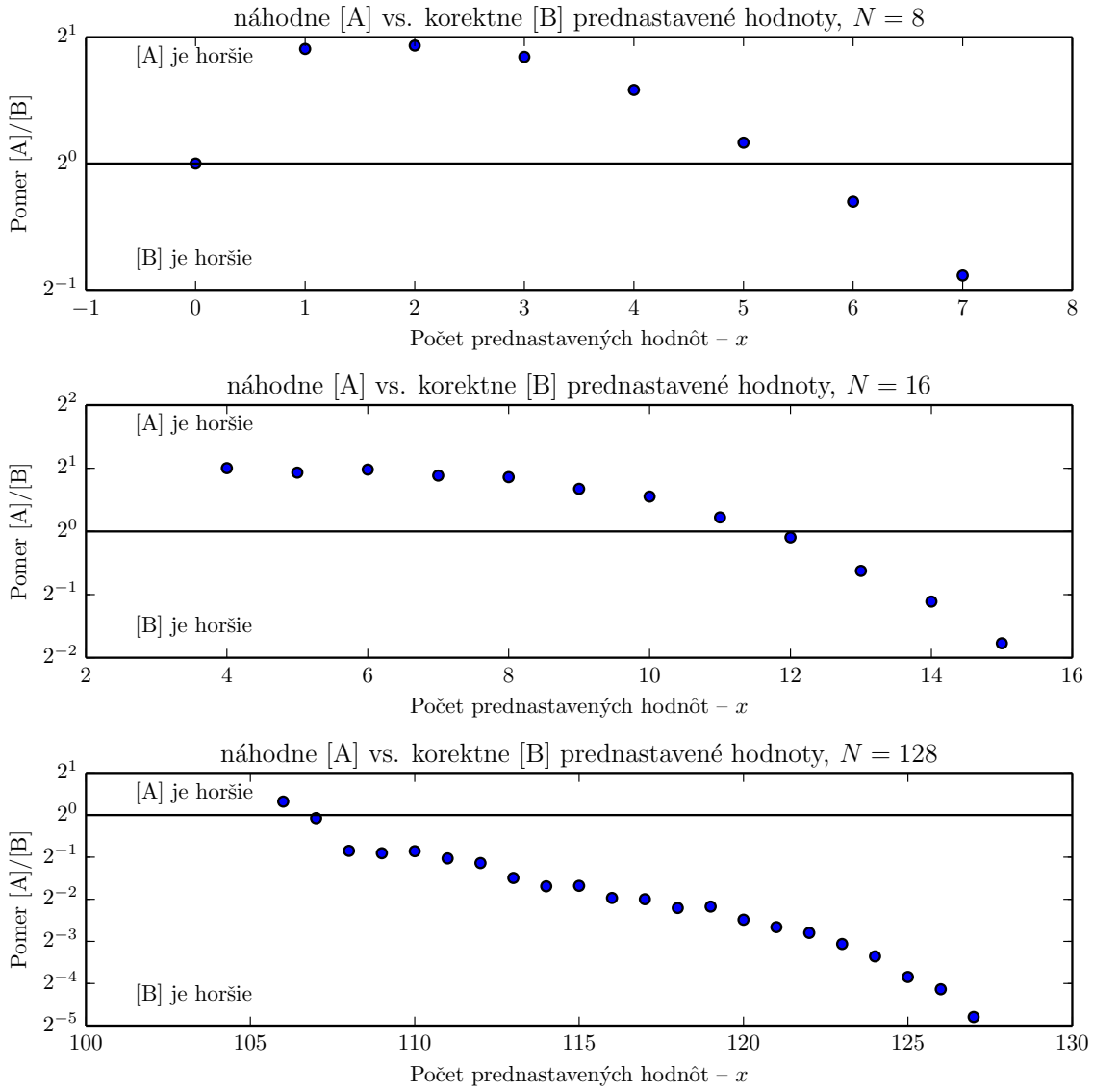
V tabuľke 3.2 môžeme vidieť porovnanie s rovnakým prehľadávaním z práce [2]. Pri týchto experimentoch sme pozorovali pomerne veľký rozdiel v zložitosti našej implementácie a tej v práci [2]. V oboch prípadoch sa jedná o ten istý algoritmus a preto by sme očakávali, že až na odchýlku kvôli iným testovacím dátam by mali byť experimentálne zložitosti rovnaké. Pozorovaný rozdiel nevieme zdôvodniť, no zložitosti získané z našich experimentov súhlasia s teoretickou analýzou v kapitole 5 a preto ich považujeme za korektné.

Pozorovania. Z experimentov vyplýva, že rôzne inštancie problému s rovnakými parametrami majú veľmi rôznu zložitosť, pri všetkých experimentoch sme pozorovali veľkú disperziu počtu priradení. Ďalej sme si všimli, že prehľadávanie nikdy nepotrebovalo viac ako N hodnôt bežiaceho kľúča (nepočítajúc kontrolu korektnosti, keď je celá permutácia vyplnená). Zároveň sa iba málokrát stalo to, že prehľadávanie bez konfliktu vyplnilo celú permutáciu.

3.2.1 Náhodne prednastavené hodnoty, nezastavenie pri výsledku

V predchádzajúcich experimentoch sme v prípade väčších hodnôt N, kedy by bolo prehľadávanie príliš pomalé, prednastavili niekoľko prvkov permutácie na korektné hodnoty hľadaného stavu. Zároveň ak prehľadávanie našlo korektný počiatočný stav, zastavili sme ho. Pri experimentoch v kapitole 5 budeme uvažovať aj situácie, keď prednastavené hodnoty stavu sú náhodné a prehľadávanie necháme dopočítať úplne do konca. Preto v tejto časti experimentálne preskúmame vplyv týchto predpokladov na počet priradení.

Náhodne prednastavené hodnoty. V prípade, že permutácii prednastavíme hodnoty náhodne je veľmi pravdepodobné, že táto inštancia nebude mať riešenie. Počiatočná permutácia je totiž z veľkou pravdepodobnosťou nezávislá od známeho bežiaceho



Obrázok 3.1: Pomer priradení pre náhodne vs. korektne prednastavené hodnoty S

kľúča. Očakávame teda, že prehľadávanie, keďže nezastaví pri nájdení správneho stavu, bude potrebovať viac priradení, lebo musí prejsť všetky nekonfliktné vetvy výpočtu. Ak predpokladáme, že v priemere nájde prehľadávanie správny stav v polovici výpočtu, pre náhodne prednastavené prvky by malo potrebovať približne dvakrát viac priradení.

Na obrázku 3.1 uvádzame pomer priemerného počtu priradení pre náhodne a korektne prednastavené hodnoty. Priemery sú vždy z 1000 experimentov. Vidíme, že pre x idúce k 0 potrebujú náhodne prednastavené inštancie naozaj dvakrát viac priradení. Na druhú stranu pre veľa prednastavených hodnôt sú náhodné hodnoty oveľa lepšie. To je spôsobené tým, že pre permutáciu, ktorá je skoro celá známa a nezávislá od bežiaceho kľúča sa budú konflikty objavovať veľmi často.

Nezastavenie pri výsledku. Predpokladáme, že prehľadávanie nájde správny počiatočný stav v priemere v polovici výpočtu. V tabuľke 3.3 uvádzame experimentálne

výsledky pre niekoľko hodnôt N, ktoré tento predpoklad potvrdzujú.

N	Zastaviť	Nezastaviť
6	8.21	9.22
8	12.09	13.05
10	16.40	17.40
12	21.11	22.09

Pozn. 1: Posledné dva stĺpce predstavujú „lg“ počtu priradení (priemer z 1000 experimentov).

Tabuľka 3.3: Porovnanie zastavenia prehľadávania pri nájdení správneho stavu

KAPITOLA 4

Vylepšenia útoku prehľadávaním

V tejto kapitole navrhujeme niekoľko vylepšení prehľadávania s návratom popísaného v kapitole 3 a experimentálne overujeme ich dopad na zložitosť útoku. Prehľadávanie z kapitoly 3 budeme vzhľadom k vylepšeniam nazývať „jednoduché prehľadávanie“.

Vylepšenia sme implementovali modifikáciou rekurzívnej funkcie použitej v časti 3.2. Experimenty boli vykonané na náhodne vygenerovaných stavoch – registre i, j, k ako aj permutácia boli zvolené náhodne, register w náhodne tak, aby bol nesúdeliteľný s N. Registre a a z boli nastavené na 0, t.j rovnako, ako v situácii po inicializácii šifry. Zo stavu bol implementáciou Spritz odvodený bežiaci kľúč. Prehľadávanie na vstupe dostalo stav so známymi registrami a neznámou permutáciou. V prípade, že sme permutácii prednastavili niekoľko hodnôt, boli ich pozície vybrané náhodne a hodnoty boli nastavené korektne (teda tie zo správneho počiatočného stavu).

V prípade porovnávania rôznych optimalizácií bola sada počiatočných stavov v oboch experimentoch rovnaká. Okrem priemerného počtu priradení pri experimentoch uvádzame aj distribúciu zlepšenia pre jednotlivé stavy. Pre každý počiatočný stav v experimente máme počet priradení c_b bez optimalizácie a c_s s optimalizáciou. Potom hodnota $\lg(c_b/c_s)$ vyjadruje, o koľko sa zníži logaritmus zložitosti pri použití optimalizácie. Distribúciu týchto hodnôt v experimentoch pre rôzne parametre N a x uvádzame na obrázkoch 4.1, 4.2, 4.3 a 4.4.

4.1 Zmena poradia

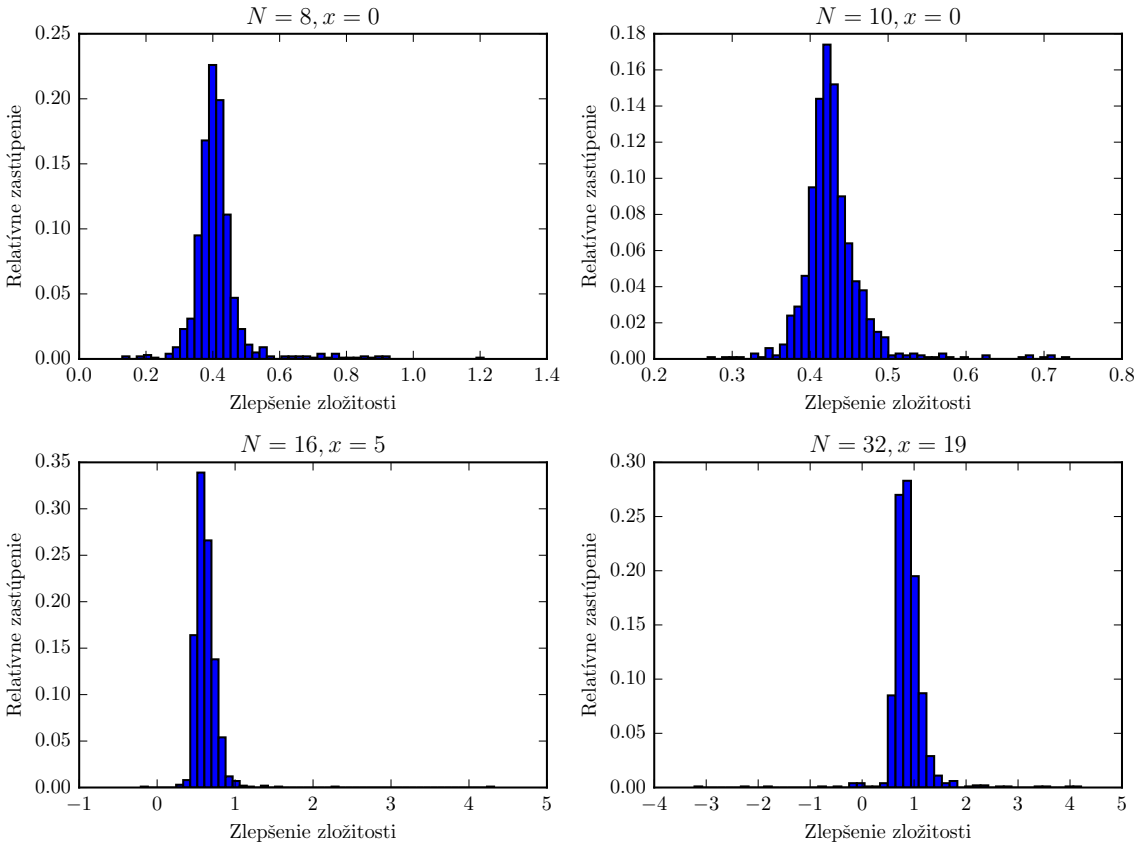
Táto optimalizácia bola pôvodne navrhnutá pre šifru RC4 v práci [9]. Jej myšlienkou je zmeniť poradie, v akom sú jednotlivé hodnoty S v prehľadávaní použité. Namiesto tipovania hodnôt $S[i]$, $S[j + S[i]]$, $S[j]$, $S[z + k]$, $S[i + S[z + k]]$ (v tomto poradí) a následného overenia konzistentnosti s bežiacim kľúčom môžeme postupovať nasledovne: pred spracovaní hodnoty $S[i + S[z + k]]$ skontrolujeme, či sa z_t nachádza v S. Ak áno, t.j. existuje $d \in \mathbb{Z}_N$ také, že $z_t = S[d]$, potom musí platiť $S[i + S[z + k]] = d - j$.

Skontrolujeme konzistentnosť – ak je $S[i+S[z+k]]$ známa hodnota, musí sa rovnať $d-j$, ak je neznáma, potom hodnota $d-j$ nesmie byť na inom mieste v S a pokračujeme do ďalšej iterácie UPDATE. V prípade, že sa z_t nenachádza v S, postupujeme rovnako ako v jednoduchom prehľadávaní – v prípade potreby tipneme $S[i+S[z+k]]$, skontrolujeme korektnosť ($S[j+S[i+S[z+k]]]$ musí byť buď neznáme alebo z_t), ak treba priradíme z_t na správne miesto a pokračujeme v prehľadávaní.

Oproti jednoduchému prehľadávaniu šetríme priradenia v prípade, že sa z_t nachádza v S: $S[i+S[z+k]]$ môže obsahovať iba hodnotu $d-j$, teda nemusíme skúšať všetky možnosti.

V tabuľke 4.1 vidíme porovnanie priemeru zložitosti jednoduchého prehľadávania a zmeny poradia. Zmena poradia naozaj znižuje počet priradení, aj keď rozdiel nie je príliš veľký. Na obrázku 4.1 porovnávame zlepšenie pre každý stav zvlášť.

Pre experimenty s prednastavenými hodnotami táto optimalizácia v niekoľko málo prípadoch počet priradení zvýšila. Príčinu tohto zvýšenia počtu priradení sa nám nepodarilo vysvetliť. Na druhej strane bolo jednak takýchto stavov málo (1 pre parametre $N = 16, x = 5$ a 8 pre $N = 32, x = 19$ z celkovo 1000 stavov v experimente) a zvýšenie zložitosti sme pozorovali iba pri prednastavených hodnotách v permutácii, pričom reálny útok bude pravdepodobne začínať z neznámej permutácie.



Obrázok 4.1: Zlepšenie zložitosti pri zmene poradia

		Jednoduché		Zmena poradia	
N	x	priradenia	lg(priradenia)	priradenia	lg(priradenia)
8	0	4286.00	12.07	3228.19	11.66
10	0	87035.78	16.41	64654.06	15.98
16	5	56687.06	15.79	37675.75	15.20
32	19	23801.75	14.54	13815.45	13.75

Pozn.: Hodnoty (okrem prvých dvoch stĺpcov) predstavujú priemer z 1000 experimentov

Tabuľka 4.1: Porovnanie jednoduchého prehľadávania a zmeny poradia

4.2 Poradie skúšaných hodnôt

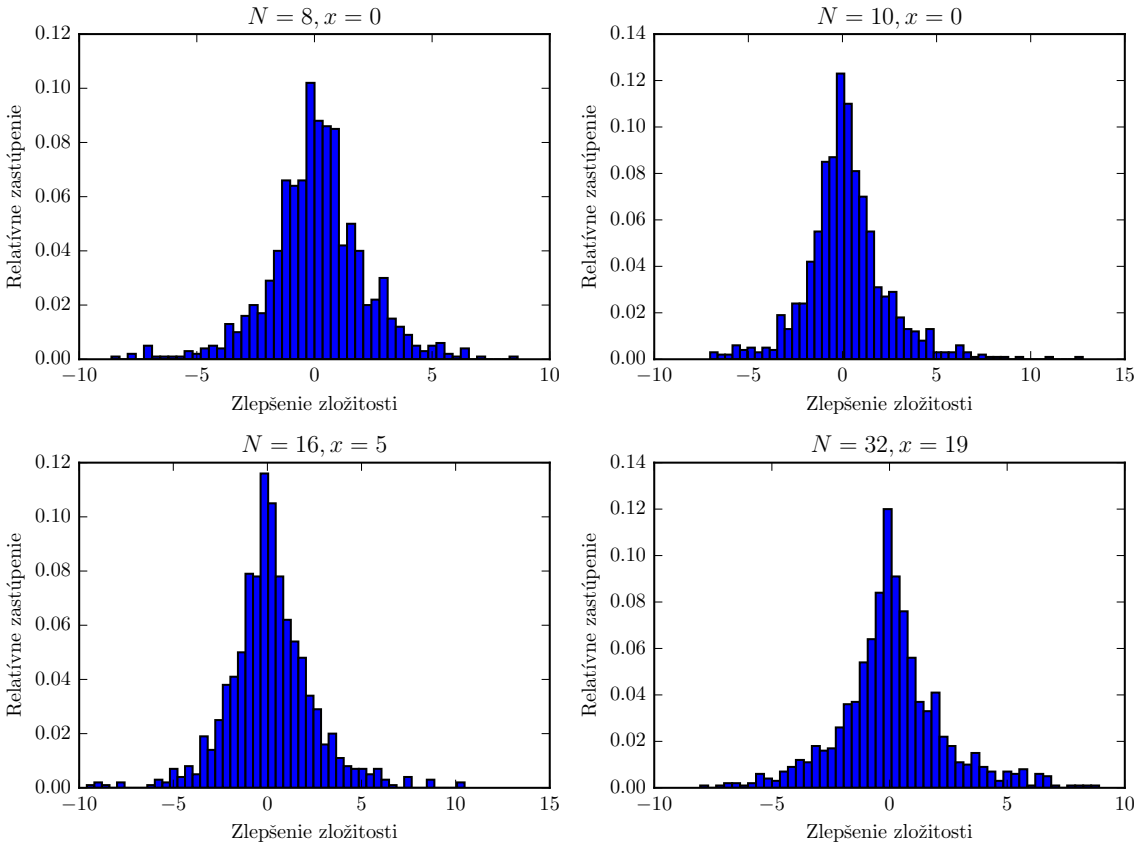
K zníženiu zložitosti prehľadávania by mohla viesť aj zmena poradia, v akom sa skúšajú hodnoty na neznáme miesta permutácie. Zmenou poradia skúšaných hodnôt a teda aj poradia vetiev prehľadávania, ktoré sa budú spracovávať môžeme docieliť, že vetva vedúca k správnej permutácii bude prehľadaná skôr. Naša idea je skúšať najprv tie hodnoty, ktoré sa nachádzajú v bežiacom kľúči (od začiatku bežiaceho kľúča a presne v takom poradí, bez duplikátov) a po nich ostatné, čo chýbajú do $\mathbb{Z}_N$ (sekvenčne). Toto poradie by malo byť výhodné spolu s predchádzajúcou optimalizáciou. Priraďovať najprv hodnoty z bežiaceho kľúča bude mať za následok, že pri kontrole konzistentnosti skôr nastanú situácie, že sa z_t nachádza v S. To by mohlo šetriť počet priradení, keďže v týchto situáciách sa nikdy neskúšajú všetky možnosti.

Z tabuľky 4.2 vidíme, že táto optimalizácia v priemere pomáha len veľmi málo. Obrázok 4.2 ukazuje, že síce existujú počiatočné stavy, kedy poradie hodnôt časovú zložitosť zlepší, no v ostatných sa zložitosť buď zhorší alebo ostane nezmenená. Hoci v prehľadávaní s touto optimalizáciou budú zo začiatku behu nastávať situácie $z_t \in S$, z experimentov usudzujeme, že rovnako často, ako existujú počiatočné stavy, kde pre nájdenie správneho stavu je treba v prehľadávaní tipnúť najprv hodnoty z bežiaceho kľúča, budú existovať prípady, kde je potrebné najprv priradiť iné hodnoty než tie z bežiaceho kľúča (a teda správny počiatočný stav nájde prehľadávanie neskôr).

		Zmena poradia		+ Poradie hodnôt	
N	x	priradenia	lg(priradenia)	priradenia	lg(priradenia)
8	0	3228.19	11.66	3117.03	11.61
10	0	64654.06	15.98	61874.40	15.92
16	5	37675.75	15.20	36056.66	15.14
32	19	13815.45	13.75	13360.41	13.71

Pozn.: Hodnoty (okrem prvých dvoch stĺpcov) predstavujú priemer z 1000 experimentov

Tabuľka 4.2: Porovnanie zmeny poradia a poradia skúšaných hodnôt



Obrázok 4.2: Zlepšenie zložitosti s iným poradím skúšaných hodnôt

4.3 Konzistentnosť s prefixom

Idea tejto optimalizácie je začať prehľadávanie na posunutej pozícii známeho bežiaceho kľúča a použiť preskočený prefix bežiaceho kľúča na dodatočnú kontrolu konzistencie. V prehľadávaní predpokladáme znalosť bežiaceho kľúča $Z = \{z_t\}_{t \geq 0}$ vygenerovaného z počiatočného stavu S_0. Predchádzajúce prehľadávania sa vždy snažili nájsť S_0. Označme S_p pre $p \geq 0$ permutáciu po p iteráciách funkcie Update. Pokiaľ predpokladáme znalosť hodnôt i, z, w v stave S_0, poznáme hodnoty týchto registrov aj po p krokoch (pozri časť 1.3.1). Hodnoty j a k zistíme úplným preberaním rovnako, ako pri začiatku v S_0.

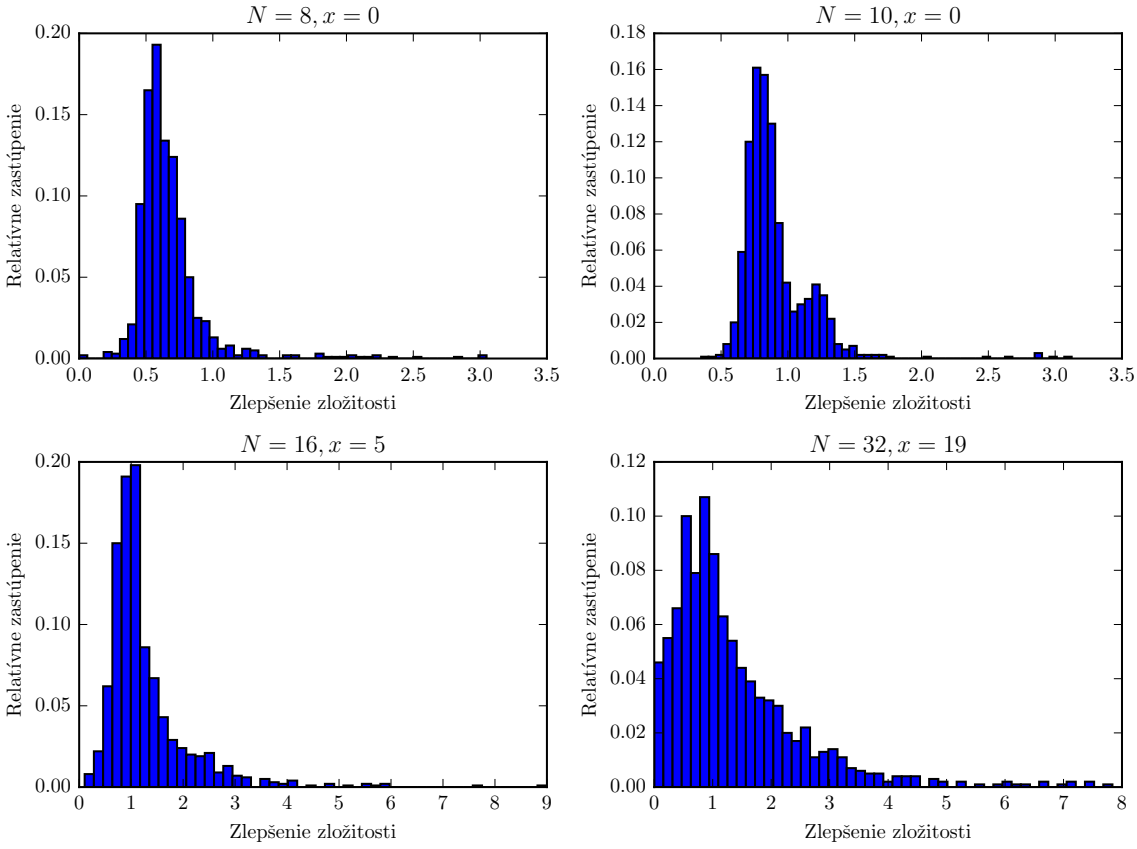
V prehľadávaní s návratom použijeme bežiaci kľúč začínajúci p-tym slovom, pričom sa snažíme získať stav S_p. Uvedomme si, že funkciu Update ide krokovať aj dozadu (pozri aj časť 1.3). Pokiaľ máme v prehľadávaní čiastočne vyplnený stav (s x známymi hodnotami), vieme ho vrátiť do pozície p, keďže všetky hodnoty na to potrebné sme do S už priradili. Takýto stav obsahuje tých istých x hodnôt, možno na iných miestach kvôli operácii Swap vo funkcii Update. Následne sa pokúsim stav ďalej vracať, pričom kontrolujeme, či je konzistentný s hodnotami $z_{t-1}, z_{t-2}, \ldots$. Ak nie je, vieme, že niektoré priradenie v prehľadávaní je zlé a v tejto vetve výpočtu nemusíme pokračovať. Poznamenajme, že pri spätnom krokovaní Update od pozície p sa môže stať, že

potrebujeme hodnoty z S, ktoré nepoznáme. V takejto situácii nemôžeme pokračovať v kontrole prefixu, žiadny konflikt sme nenašli a pokračujeme v prehľadávaní.

Poznámka. Pokiaľ pri kontrole prefixu narazíme na neznámu hodnotu vo výpočte z, nemusíme kontrolu hneď skončiť. Výpočtom z zo stavu kontrolujeme iba konzistentnosť s aktuálnym slovom bežiaceho kľúča a ďalšie krokovanie späť funkcie UPDATE na tejto hodnote nezávisí (správnu hodnotu z vieme z prefixu bežiaceho kľúča). Teda túto kontrolu môžeme preskočiť dúfajúc, že konflikt nájdeme na inom mieste prefixu.

Poznámka. Ak pri kontrole prefixu potrebujeme neznámu hodnotu $S[j + S[i + S[z + k]]]$ a aktuálne z_t z prefixu bežiaceho kľúča je nie je v S, môžeme spraviť priradenie $S[j + S[i + S[z + k]]] = z_t$, keďže toto je jediný prípad, kedy môže byť aktuálny stav konzistentný s prefixom, čím zvýšime šancu, že nájdeme konflikt ďalej pri kontrole na inom mieste prefixu.

V experimentoch sme použili $p = 8$, napriek tomu väčšina kontrol prefixu skončila po prvých troch slovách bežiaceho kľúča. Z tabuľky 4.3 a obrázka 4.3 vidíme, že táto optimalizácia zlepšuje počet priradení, zároveň je o čosi lepšia, než zmena poradia, no stále sa jedná o malé zlepšenie. Pri prednastavených hodnotách je zlepšenie väčšie. To je spôsobené korektným prednastavením permutácie, keďže viac známych hodnôt v stave S_p zvyšuje šancu, že kontrola prefixu nájde konflikt.



Obrázok 4.3: Zlepšenie zložitosti pre kontrolu prefixu

		Jednoduché		Kontrola prefixu	
N	x	priradenia	lg(priradenia)	priradenia	lg(priradenia)
8	0	4170.11	12.03	2650.62	11.37
10	0	84155.77	16.36	45387.84	15.47
16	5	57927.48	15.82	25465.33	14.64
32	19	24301.18	14.57	10675.66	13.38

Pozn.: Hodnoty (okrem prvých dvoch stĺpcov) predstavujú priemer z 1000 experimentov

Tabuľka 4.3: Porovnanie jednoduchého prehľadávania a kontroly prefixu

4.4 Kombinácia optimalizácií

Z troch navrhovaných optimalizácií zlepšovala počet priradení kontrola prefixu a zmena poradia. V tabuľke 4.4 a na obrázku 4.4 môžeme vidieť experimentálne výsledky pre obe optimalizácie naraz. Keďže každá zlepšuje inú časť prehľadávania, je očakávané, že spolu bude zlepšenie väčšie než samostatne. Zlepšenie v počte priradení je od dvojnásobku po štvornásobok (pre experimenty s prednastavenými hodnotami). Zriedkavé prípady, keď sú optimalizácie horšie než jednoduché prehľadávanie, sa nám nepodarilo vysvetliť. Je ale pravdepodobné, že príčina je rovnaká, ako pri podobnom zhoršení v experimentoch so zmenou poradia. Väčšie zlepšenie pozorované pri prednastavených hodnotách je pravdepodobne spôsobené tým, že korektné hodnoty pomáhajú kontrole prefixu.

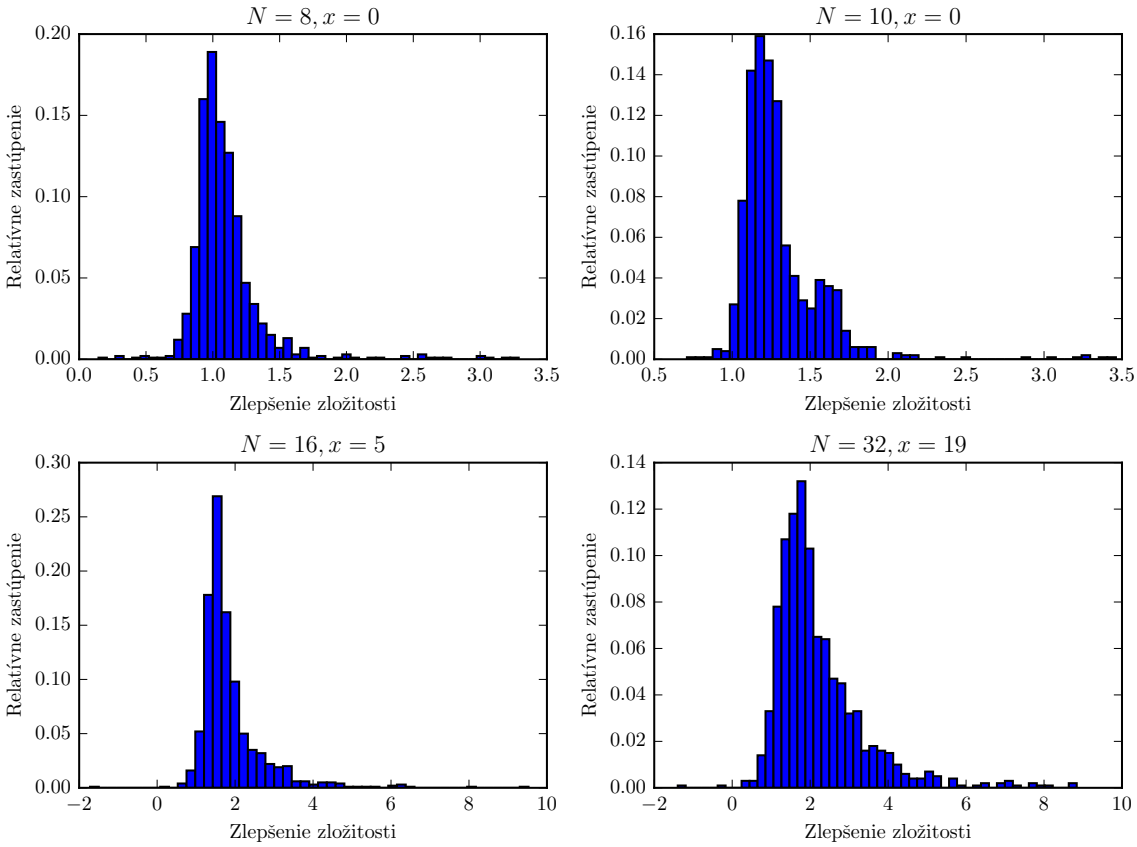
		Jednoduché		Prefix a zmena poradia	
N	x	priradenia	lg(priradenia)	priradenia	lg(priradenia)
8	0	4170.11	12.03	1977.26	10.95
10	0	84155.77	16.36	34576.96	15.08
16	5	57927.48	15.82	17241.99	14.07
32	19	24301.18	14.57	6286.73	12.62

Pozn.: Hodnoty (okrem prvých dvoch stĺpcov) predstavujú priemer z 1000 experimentov

Tabuľka 4.4: Porovnanie jednoduchého prehľadávania a kontroly prefixu spolu so zmenou poradia

4.5 Zhrnutie výsledkov

Experimentovali sme s viacerými vylepšeniami jednoduchého prehľadávania z kapitoly 3. Experimentálne sme overili, že kontrola prefixu a zmena poradia naozaj zlepšujú počet priradení a tiež, že sa dajú použiť spoločne. V tomto prípade sme pozorovali najväčšie zlepšenie oproti jednoduchému prehľadávaniu a to dvoj až štvornásobne. Na



Obrázok 4.4: Zlepšenie zložitosti pre kontrolu prefixu spolu so zmenou poradia

druhej strane poradie skúšaných hodnôt odvodené z bežiaceho kľúča rovnako často zlepšuje zložitosť ako ju zhoršuje a na očakávaný počet priradení vplyv nemá.

KAPITOLA 5

Odhady zložitosti útokov prehľadávaním

V tejto kapitole sa zaoberáme teoretickými odhadmi zložitosti viacerých variantov prehľadávania s návratom pre šifry RC4 a Spritz. Uvádzame odhad zložitosti prehľadávania z kapitoly 3 a odhady optimalizácie so zmenou poradia z kapitoly 4 pre šifru RC4, čím opravíme a zlepšíme analýzu z práce [9]. Ďalej opravíme odhad prehľadávania z kapitoly 3 pre Spritz z práce [2] a uvedieme odhad prehľadávania s optimalizáciou zmeny poradia. V článku [3] autori uvádzajú odhad prehľadávania za predpokladu, že prehľadávanie začína v špeciálnom stave. Tento odhad taktiež zlepšíme. Budeme tiež experimentovať s odhadmi a celkovou zložitosťou prehľadávania v situácii, keď začíname prehľadávanie s čiastočne predvyplnenou permutáciou.

Pri odhadoch používame techniku z práce [9] navrhnutú pre šifru RC4. Zložitosť prehľadávania odhadneme ako očakávaný počet priradení počas celého behu algoritmu. Zavedieme premenné, ktoré počítajú očakávaný počet priradení v prípade, že začneme v konkrétnom kroku prehľadávania. Premenné použijeme v rovniciach, ktoré reflektujú logiku konkrétneho prehľadávania a očakávanú zložitosť útokov získame vyriešením týchto rovníc. Výpočty rovníc sme vykonali pomocou nástroja SageMath [20].

Zložitosti odhadov v tejto kapitole porovnávame aj s experimentálnymi hodnotami, pre ktoré sme implementovali prehľadávanie (aj s príslušnými optimalizáciami) v C++. Vzhľadom na parametre použité v experimentoch predchádzajúcich prác, s ktorými naše odhady porovnávame bola naša implementácia prehľadávania v jazyku Python príliš pomalá. Pre detaily implementácie riešenia rovníc a prehľadávania pozri prílohu B.

Poznámka. Odhady predstavujú zložitosť celého prehľadávania, započítané sú všetky vetvy výpočtu od prvého po posledné priradenie. Touto technikou totiž nie je možné zistiť, že prehľadávanie našlo správny počiatočný stav a následne „rovnice zastaviť“.

Vplyv zastavenia prehľadávania v prípade nájdenia správneho stavu na zložitosť útoku je experimentálne preskúmaný v časti 3.2.1.

5.1 RC4

Začneme analýzou zložitosti prehľadávania pre RC4. Cieľom je demonštrovať techniku na jednoduchšej šifre RC4, podobne budeme v časti 5.2 postupovať aj pre šifru Spritz.

5.1.1 Jednoduché prehľadávanie

Základná verzia prehľadávania je v kapitole 3 podrobne popísaná pre šifru Spritz. Modifikácia pre RC4 je pomerne jednoduchá, hlavné rozdiely sú popísané v časti 3.1.

Používame, podobne ako v [9], premenné $c_1(x)$, $c_2(x)$ a $c_3(x)$ pre $x \in \mathbb{N}$ s nasledujúcim významom: premenná $c_i(x)$ označuje očakávaný počet priradení, ktoré spraví prehľadávanie, ak S obsahuje práve x známych (nie nutne správnych) hodnôt a algoritmus začne:

- $c_1(x)$ na začiatku prehľadávania, t.j. pred tým, ako je potrebná hodnota $S[i]$;
- $c_2(x)$ pred tým, ako je potrebná hodnota $S[j]$;
- $c_3(x)$ pred tým, ako treba hodnotu $S[S[i] + S[j]]$.

Celková zložitosť algoritmu je potom $c_1(0)$. Hraničné prípady sú pre $x = N$, keď máme všetky prvky S už určené a netreba robiť žiadne priradenia, teda $c_1(N) = c_2(N) = c_3(N) = 0$. Ak vieme hodnoty premenných $c_1(x+1)$, $c_2(x+1)$ a $c_3(x+1)$, hodnoty $c_1(x)$, $c_2(x)$ a $c_3(x)$ spočítame nasledovnými rovnicami:

$$\begin{aligned} c_1(x) &= (x/N) \cdot c_2(x) + (1 - x/N) \cdot (N - x) \cdot (1 + c_2(x+1)) \\ c_2(x) &= (x/N) \cdot c_3(x) + (1 - x/N) \cdot (N - x) \cdot (1 + c_3(x+1)) \\ c_3(x) &= (x/N) \cdot (1/N \cdot c_1(x)) + (1 - x/N)^2 \cdot (1 + c_1(x+1)) \end{aligned}$$

Počet priradení v $c_1(x)$ môžeme rozdeliť na dva prípady – buď je $S[i]$ už známa hodnota, čo nastane s pravdepodobnosťou x/N, alebo neznáma (pravdepodobnosť $1 - x/N$). V prvom prípade nerobíme žiadne priradenie a pokračujeme do druhého kroku s rovnakým počtom známych prvkov S, teda očakávaný počet priradení bude $c_2(x)$. V druhom prípade vyskúšame do $S[i]$ priradiť všetkých $N - x$ hodnôt, ktoré ešte v S nie sú použité. Pre každú hodnotu započítame jedno priradenie a pokračujeme do druhého kroku s o jedna viac známymi prvkami v S. Rovnicu pre $c_2(x)$ skonštruujeme analogicky.

Počet priradení v $c_3(x)$ môžeme tiež rozdeliť na dva prípady – buď je hodnota $S[i] + S[j]$ známa (pravdepodobnosť x/N) alebo nie (pravdepodobnosť $1 - x/N$). V prvom prípade pokračujeme v ďalšej iterácii funkcie Update (započítajúc $c_1(x)$ priradení) iba v prípade, že sa hodnota $S[i] + S[j]$ rovná aktuálnemu slovu bežiaceho

kľúča z_t (pravdepodobnosť $1/N$). V opačnom prípade nastal konflikt a ďalšie priradenia nerobíme, teda táto situácia do počtu priradení nijako neprispieva a v rovnici nie je. V druhom prípade, keď je hodnota $S[i] + S[j]$ neznáma, spravíme jedno priradenie $S[i] + S[j] \leftarrow z_t$ a pokračujeme v ďalšej iterácii funkcie UPDATE (započítajúc $c_1(x+1)$ priradení), no iba v prípade, že z_t nie je v S na inom mieste (pravdepodobnosť $1-x/N$). Inak nastal konflikt a ďalej nepokračujeme.

Sústavy troch rovníc vyriešime postupne pre klesajúce x od $x = N-1$ po $x = 0$, čím dostaneme očakávaný počet priradení $c_1(0)$. Pre $N = 256$ dostaneme zložitosť $c_1(0) \approx 2^{775}$. Odhady pre ďalšie hodnoty N sú uvedené v tabuľke 5.1.

V rovniciach odhadujeme pravdepodobnosť, že hodnota $S[\cdot]$ je známa ako x/N, keďže predpokladáme rovnomerné náhodné rozdelenie hodnôt, ktorými indexujeme do S. Tento predpoklad bol použitý aj v odhadoch v [9] pre RC4 a [2, 3] pre Spritz.

5.1.2 Optimalizácia so zmenou poradia

Idea tejto optimalizácie je pre Spritz popísaná v kapitole 4. Jej modifikácia pre RC4 je priamočiara: v prípade, že sa z_t nachádza v S a hodnotu $S[i]$ vieme, $S[j]$ sa musí rovnať $d-j$, kde $S[d] = z_t$. Pri prehľadávaní teda pred tipovaním $S[j]$ skontrolujeme, či $z_t \in S$ a ak áno, hodnotu $S[j]$ vypočítame bez skúšania všetkých možností. Kompletný popis optimalizácie je uvedený v [9]. V ďalšom odvodíme a popíšeme rovnice pre jednotlivé kroky prehľadávania:

1. Ak je hodnota $S[i]$ známa (pravdepodobnosť x/N pre x známych hodnôt v S), pokračujeme ďalej. Inak (s pravdepodobnosťou $1 - x/N$) do $S[i]$ postupne priradíme každú z $N - x$ možných hodnôt a pokračujeme ďalej.

   Tento krok je rovnaký ako pri jednoduchom prehľadávaní a teda aj rovnica pre $c_1(x)$ bude rovnaká:

   $$c_1(x) = (x/N) \cdot c_2(x) + (1 - x/N) \cdot (N - x) \cdot (1 + c_2(x+1))$$

2. V prípade, že sa z_t nachádza v S (pravdepodobnosť x/N) existuje také $d \in \mathbb{Z}_N$, že $z_t = S[d]$. Označme $e = d - S[i]$ a pozrime sa na $S[j]$:

   (1) $S[j]$ je neznáma & $e \notin S$ $S[j] \leftarrow e$ a pokračujeme v ďalšej iterácii;
   (2) $S[j]$ je neznáma & $e \in S$ konflikt;
   (3) $S[j]$ je známa & $S[j] = e$ pokračujeme v ďalšej iterácii UPDATE;
   (4) $S[j]$ je známa & $S[j] \neq e$ konflikt.

   Ak sa z_t nenachádza v S (pravdepodobnosť $1 - x/N$) pokračujeme ďalej.

Rovnica pre $c_2(x)$ (s vyznačenými prípadmi 1 a 3):

$$c_2(x) = \underbrace{(x/N)}_{z_t \in S} \cdot \Big(\underbrace{(1 - x/N)^2 \cdot (1 + c_1(x+1))}_{(1)} + \underbrace{(x/N) \cdot (1/N) \cdot c_1(x)}_{(3)} \Big) + \underbrace{(1 - x/N)}_{z_t \notin S} \cdot c_3(x)$$

3. Vieme, že sa z_t nenachádza v S. Môžu nastať dva prípady: buď je $S[j]$ neznáme (pravdepodobnosť $1 - x/N$) alebo známe (pravdepodobnosť x/N).

   V prvom prípade vyskúšame všetkých $N - x$ možností pre hodnotu $S[j]$. Rozlišujeme dva prípady podľa toho, akú hodnotu skúšame do $S[j]$ priradiť: (1) z_t alebo (2) inú hodnotu:

   (1) ak platí $S[S[i] + S[j]] = z_t$ (pravdepodobnosť $1/N$):
   pokračujeme ďalšou iteráciou
   inak: konflikt

   (2) ak je $S[S[i] + S[j]]$ neznáme (pravdepodobnosť $1 - (x+1)/N$):
   priradíme $S[S[i] + S[j]] \leftarrow z_t$ a pokračujeme ďalšou iteráciou
   inak: konflikt

   V druhom prípade ($S[j]$ je známe) sa pozeráme, či je pozícia $S[S[i] + S[j]]$ voľná. Ak áno, priradíme do nej z_t a pokračujeme v ďalšej iterácii UPDATE, inak máme konflikt. Rovnica pre $c_3(x)$:

$$c_3(x) = \underbrace{(1 - x/N)}_{S[j] \text{ neznáme}} \cdot \Big[\underbrace{1 + (1/N) \cdot c_1(x+1)}_{(1)} + \underbrace{(N - x - 1) \cdot (1 - (x+1)/N) \cdot (1 + c_1(x+2))}_{(2)} \Big] + \underbrace{(x/N)}_{S[j] \text{ známe}} \cdot \underbrace{(1 - x/N)}_{S[S[i]+S[j]] \text{ neznáme}} \cdot (1 + c_1(x+1))$$

Hraničné prípady sú $c_1(x) = c_2(x) = c_3(x) = 0$ pre $x \geq N$. Vyriešením rovníc pre $N = 256$ dostaneme odhad $c_1(0) \approx 2^{772}$, čo súhlasí s tým, že táto zmena v prehľadávaní by mala ušetriť niektoré priradenia. Odhady pre ďalšie hodnoty N sú uvedené v tabuľke 5.1. Poznamenávame, že existujú efektívnejšie útoky na stav RC4, napríklad [12].

Rovnice uvedené v [9], ktoré tiež analyzujú očakávaný počet priradení prehľadávania so zmenou poradia, sú rozdielne od tu uvedených, no obsahujú viaceré chyby:

– Zložitosť 2^{779} pre $N = 256$ je horšia než zložitosť jednoduchého prehľadávania z časti 5.1.1 (podobne pre $N = 128$), čo nezodpovedá zlepšeniu zložitosti pre prehľadávanie so zmenou poradia.

– Nech $N = 4$ (podobný argument platí ak pre iné hodnoty, napríklad $N = 8$ a $N = 16$). Zoberme si situáciu, keď prehľadávanie začína v S, ktorá má iba jednu neznámu hodnotu, t.j. 3 zo 4 hodnôt sú už (náhodne) zvolené. Keď použijeme rovnice z [9], dostaneme očakávaný počet priradení v tejto situácii 0.136. Na druhú stranu, prehľadávanie začne pozretím na hodnotu $S[i]$, ktorá je z pravdepodobnosťou 1/4 neznáma a teda hneď v prvom kroku spravíme 0.25 priradenia. Očakávaný počet priradení teda musí byť aspoň 0.25 (poznamenajme, že ak je $S[i]$ známa, môžu neskôr nastať ďalšie priradenia). Z našich rovníc dostávame $c_1(3) \approx 0.528$ pre jednoduché prehľadávanie a $c_1(3) \approx 0.410$ pre zmenu poradia.

	Teória			Experimenty	
N	Jednoduché	Zmena poradia	[9]	Jednoduché	Zmena poradia
8	9.47	8.56	8.65	9.69	9.32
12	15.38	14.28	14.56	15.72	15.19
16	22.02	20.77	21.25	22.53	21.90
64	132.28	130.33	132.65	–	–
128	322.85	320.55	324.76	–	–
256	775.03	772.39	779.68	–	–

Pozn. 1: Čísla (okrem prvého stĺpca) predstavujú „lg“ zložitosti prehľadávania.
Pozn. 2: Posledné dva stĺpce sú priemer z 1000 experimentov.

Tabuľka 5.1: RC4 – zložitosť rôznych verzií prehľadávania

Naša analýza ukazuje, že zmena poradia naozaj zrýchľuje prehľadávanie. Faktor zrýchlenia sa pohybuje od približne 2 pre malé N až po približne 5 pre $N = 128$ alebo 6 pre $N = 256$. Rozdiel medzi našimi odhadmi a tými v článku [9] je pomerne malý.

Poznámka. Aby sme porovnali experimentálne hodnoty s teoretickými odhadmi, experimenty sme vykonali nasledovne: prehľadávanie začína s prázdnou permutáciou a nezastaví sa pri nájdení správneho počiatočného stavu, ale necháme ho počítať, až kým nevyskúša každé nekonfliktné priradenie. Experimenty v časti 3.2.1 ukazujú, že zastavenie pri nájdení správneho stavu v priemere znižuje zložitosť o polovicu (t.j. „lg“ počtu priradení klesne približne o 1).

Rozdiely medzi teoretickými odhadmi a príslušnými experimentálnymi hodnotami sú spôsobené predpokladom náhodnej distribúcie hodnôt indexov v rovniciach, ako poznamenávame v časti 5.1.1.

5.2 Spritz

5.2.1 Jednoduché prehľadávanie

Odhadneme očakávaný počet priradení základnej verzie prehľadávania pre Spritz popísanej v kapitole 3. Budeme používať premenné $c_1(x), \ldots, c_6(x)$ pre $x \in \mathbb{N}$. Premenné označujú očakávaný počet priradení prehľadávania, ak S obsahuje práve x známych (nie nutne korektných) hodnôt a algoritmus začne: na začiatku prehľadávania ($c_1(x)$) alebo pred tým, ako je potrebná druhá ($c_2(x)$), ..., šiesta ($c_6(x)$) hodnota z S. Celková očakávaná zložitosť je $c_1(0)$. Hraničné prípady sú $c_i(N) = 0$ pre $i \in \{1, \ldots, 6\}$. Pokiaľ poznáme hodnoty $c_1(x+1), \ldots, c_6(x+1)$, rovnice pre výpočet hodnôt $c_1(x), \ldots, c_6(x)$ zostrojíme rovnako, ako pre jednoduché prehľadávanie RC4:

$$c_i(x) = (x/N) \cdot c_{i+1}(x) + (1 - x/N) \cdot (N - x) \cdot (1 + c_{i+1}(x+1)), \quad \text{pre } i = 1, \ldots, 5$$
$$c_6(x) = (x/N) \cdot (1/N \cdot c_1(x)) + (1 - x/N)^2 \cdot (1 + c_1(x+1))$$

Postupným vyriešením sústavy pre klesajúce $x = N-1, \ldots, 0$ dostaneme očakávanú zložitosť nájdenia počiatočného stavu šifry Spritz jednoduchým prehľadávaním. Pre $N = 256$ dostávame $c_1(0) \approx 2^{1122}$, ďalšie hodnoty sú uvedené v tabuľke 5.2.

Jednoduché prehľadávanie bolo pre Spritz popísané a analyzované v práci [2]. Analýza ale vykazuje viaceré problémy:

- Zložitosti prezentované v práci nie sú konzistentné s uvedenými rovnicami. Napríklad vyriešením rovníc z [2] pre $N = 256$ dostávame zložitosť $\approx 2^{1575}$, kým v práci je prezentovaná zložitosť 2^{1400}.

- Podobne ako v časti 5.1.2 môžeme ukázať, že rovnice z [2] sú nesprávne. Majme $N = 4$ a $x = 3$ (podobný argument platí aj pre iné hodnoty N) a vypočítajme očakávanú zložitosť pomocou rovníc z [2]. Dostaneme 0.1083, no $c_1(3) \geq 0.25$, keďže predpokladáme náhodný stav s 3 známymi hodnotami a teda prehľadávanie spraví 0.25 priradenia hneď v prvom kroku priradením do $S[i]$. Z našich rovníc dostávame $c_1(3) \approx 0.81$.

5.2.2 Prehľadávanie so zmenou poradia

V ďalšom odhadneme zložitosť prehľadávania so zmenou poradia. Táto optimalizácia je pre Spritz popísaná v kapitole 4. Analýza je z veľkej časti rovnaká ako pre RC4 zo časti 5.1.2. Odvodíme a popíšeme rovnice pre jednotlivé kroky prehľadávania:

1. Prehľadávanie začneme kontrolou prvých štyroch (potenciálne neznámych) hodnôt $S[i]$, $S[j + S[i]]$, $S[j]$, $S[z + k]$, tipujúc tie hodnoty, ktoré sú neznáme. Ak je konkrétna hodnota známa (pravdepodobnosť x/N), pokračujeme ďalej. V opačnom prípade (pravdepodobnosť $1 - x/N$) vyskúšame priradiť každú z $N - x$

možných hodnôt a pokračujeme ďalej. Tieto štyri kroky sú rovnaké, ako pre jednoduché prehľadávanie:

$$c_i(x) = (x/N) \cdot c_{i+1}(x) + (1 - x/N) \cdot (N - x) \cdot (1 + c_{i+1}(x+1)), \quad \text{pre } i = 1, \ldots, 4.$$

2. Ak sa z_t nachádza v S (pravdepodobnosť x/N), bude existovať $d \in \mathbb{Z}_N$ také, že $z_t = S[d]$. Označme $e = d - j$ a $f = S[i + S[z + k]]$. Poznamenajme, že f môže byť buď neznáma hodnota alebo prvok zo $\mathbb{Z}_N$. Ďalej si všimnime, že hodnota $S[z + k]$ je známa (ak by bola neznáma, v predchádzajúcom kroku by sme ju tipli), takže index $i + S[z + k]$ je tiež známy. Môžu nastať štyri prípady:

   (1) f je neznáma & $e \notin S$ $S[i + S[z + k]] \leftarrow e$, ďalšia iterácia UPDATE;
   (2) f je neznáma & $e \in S$ konflikt;
   (3) f je známa & $f = e$ pokračujeme v ďalšej iterácii funkcie UPDATE;
   (4) f je známa & $f \neq e$ konflikt.

   Ak sa z_t nenachádza v S (pravdepodobnosť $1 - x/N$) pokračujeme ďalej. Celkovo dostávame nasledovnú rovnicu:

$$c_5(x) = \underbrace{(x/N)}_{z_t \in S} \cdot \Big(\underbrace{(1 - x/N)^2 \cdot (1 + c_1(x+1))}_{(1)} + \underbrace{(x/N) \cdot (1/N) \cdot c_1(x)}_{(3)} \Big) + \underbrace{(1 - x/N)}_{z_t \notin S} \cdot c_6(x)$$

3. V tomto kroku sa z_t nenachádza v S. Opäť označme $f = S[i + S[z + k]]$. Hodnota f je buď neznáma (pravdepodobnosť $1 - x/N$) alebo známa (pravdepodobnosť x/N). V prvom prípade vyskúšame všetkých $N - x$ možností pre $S[i + S[z + k]]$. Rozlíšime dva prípady podľa toho, akú hodnotu do $S[i+S[z+k]]$ skúšame priradiť: (1) z_t alebo (2) inú:

   (1) ak sa $S[j + S[i + S[z + k]]] = z_t$ (pravdepodobnosť $1/N$):
       pokračujeme ďalšou iteráciou
       inak: konflikt
   (2) ak je $S[j + S[i + S[z + k]]]$ neznáma (pravdepodobnosť $1 - (x + 1)/N$):
       priradíme do nej z_t a pokračujeme ďalšou iteráciou
       inak: konflikt

   V druhom prípade je hodnota f známa. Skontrolujeme, či pozícia, na ktorej má byť z_t, je prázdna, t.j. či je hodnota $S[j + S[i + S[z + k]]]$ neznáma (pravdepodobnosť $1 - x/N$). Ak je, priradíme do nej z_t a pokračujeme ďalšou iteráciou funkcie UPDATE.

Rovnica pre $c_6(x)$:

$$\begin{aligned} c_6(x) = &\underbrace{(1 - x/N)}_{f \text{ neznáma}} \cdot \Big[\underbrace{1 + (1/N) \cdot c_1(x+1)}_{(1)} \\ &+ \underbrace{(N - x - 1) \cdot (1 - (x+1)/N) \cdot (1 + c_1(x+2))}_{(2)}\Big] \\ &+ \underbrace{(x/N)}_{f \text{ známa}} \cdot \underbrace{(1 - x/N)}_{S[j+S[i+S[z+k]]] \text{ neznáma}} \cdot (1 + c_1(x+1)) \end{aligned}$$

Poznamenávame, že rovnice pre $c_5(x)$ a $c_6(x)$ sú rovnaké ako rovnice $c_2(x)$ a $c_3(x)$ pre RC4 v časti 5.1.2.

Hraničné podmienky sú $c_i(x) = 0$ pre $i \in \{1, \ldots, 6\}$ a $x \geq N$. Celkovú zložitosť $c_1(0)$ vypočítame podobne ako v predchádzajúcich častiach. Zmena poradia očakávane znížila zložitosť prehľadávania, napríklad pre $N = 256$ je $c_1(0) \approx 2^{1121}$, čo je iba o málo lepšie než jednoduché prehľadávanie. V tabuľke 5.2 sú uvedené odhady pre vybrané hodnoty N a x.

		Teória			Experimenty	
N	x	Jednoduché	[2]	Zmena poradia	Jednoduché	Zmena poradia
8	0	13.02	13.40	12.54	13.05	12.64
10	0	17.34	19.36	16.87	17.40	16.97
16	5	16.76	22.49	16.14	16.81	16.23
64	48	13.65	40.19	12.67	13.59	12.61
128	114	6.07	32.82	5.13	5.97	5.09
256	240	4.47	40.24	3.64	4.51	3.62
64	0	195.81	272.03	194.83	–	–
128	0	473.34	664.35	472.18	–	–
256	0	1122.34	1575.43	1121.01	–	–

Pozn. 1: Čísla (okrem prvých dvoch stĺpcov) predstavujú „lg“ zložitosti prehľadávania.
Pozn. 2: Posledné tri stĺpce sú priemer z 1000 experimentov.

Tabuľka 5.2: Spritz – zložitosť rôznych verzií prehľadávania

Poznámka. Experimenty sme vykonali čo najbližšie k situácii z teoretického odhadu, t.j. prehľadávanie sa nezastavilo pri nájdení správneho stavu a pokiaľ začínalo s prednastavenými hodnotami, ich pozície a hodnoty boli náhodné.

5.2.3 Špeciálny stav

Banik a Isobe [3] pozorovali viaceré zaujímavé vlastnosti šifry Spritz, pokiaľ sa šifra nachádza v „špeciálnom“ stave. Stav šifry sa nazýva špeciálny, ak sú splnené nasledovné podmienky:

1. $S[t] \equiv 0 \pmod 2$, pre všetky nepárne $t \in \mathbb{Z}_N$ a
2. $S[t] \equiv 1 \pmod 2$, pre všetky párne $t \in \mathbb{Z}_N$ a
3. $j \equiv 0 \pmod 2$ a $k \equiv 0 \pmod 2$.

Ak predpokladáme párne N, autori v [3] ukázali, že šifra začínajúca v špeciálnom stave má nasledovné vlastnosti:

1. Stav šifry po štyroch iteráciách funkcie UPDATE bude znova špeciálny.
2. Parita hodnôt v S sa nemení – po každej iterácii UPDATE budú na párnych indexoch nepárne hodnoty a na nepárnych indexoch párne hodnoty.
3. Bežiaci kľúč je periodický s periódou 4: $z_t \equiv z_{t+4} \pmod 2$ pre všetky $t \geq 0$.

Vlastnosť 3 umožňuje jednoduchú identifikáciu špeciálneho stavu. Pravdepodobnosť, že budeme pozorovať špeciálny stav je malá (približne $2^{-253.7}$ pre $N = 256$), dá sa ale využiť na zrýchlenie prehľadávania. Vylepšené prehľadávanie používa vlastnosť 2 na redukovanie počtu hodnôt, ktoré je potreba skúšať pri prehľadávaní, ak narazíme na neznámu hodnotu. Prehľadávanie musí zohľadniť paritu indexu použitého vo funkcii UPDATE pri tipovaní hodnôt S – parita závisí od pozície v cykle periódy bežiaceho kľúča. Parity rôznych výrazov použitých vo funkcii UPDATE sú uvedené v tabuľke 5.3.

	Offsets			
Index	$v = 0$	$v = 1$	$v = 2$	$v = 3$
$i \leftarrow i + w$ *	1	0	1	0
$j + S[i]$ *	0	0	0	0
$j \leftarrow k + S[j + S[i]]$ *	1	0	1	0
$k \leftarrow k + i + S[j]$	1	0	1	0
$z + k$ *	1	0	0	1
$i + S[z + k]$ *	1	1	0	0
$j + S[i + S[z + k]]$ *	1	0	0	1
$z \leftarrow S[j + S[i + S[z + k]]]$	0	1	1	0

Pozn. 1: Hodnoty označené * sú použité pri tipovaní hodnôt v prehľadávaní.
Pozn. 2: Číslovanie pozície v cykle bežiaceho kľúča začína od 0.

Tabuľka 5.3: Špeciálny stav – parity indexov počas 4 iterácii UPDATE, prebrané z [3]

Analýza zložitosti prehľadávania využívajúceho špeciálny stav v práci [3] má viaceré problémy:

- Uvedená zložitosť prehľadávania pre $N = 256$ je $\approx 2^{1233}$ (nepočítajúc úplné preberanie hodnôt registrov j a k), čo je horšie než zložitosť $\approx 2^{1122}$ jednoduchého prehľadávania z časti 5.2.1. Použitie špeciálneho stavu teda neukazuje žiadne zlepšenie.

- Vyriešením rovníc uvedených v [3] dostávame rozdielne výsledky ako uvedené v článku. Napríklad pre $N = 256$ (nepočítajúc úplné preberanie hodnôt registrov j a k) je uvedená zložitosť $\approx 2^{1233}$. Na druhej strane našim výpočtom pre $N = 256$ dostávame zložitosť 2^{1235}. Podobne aj ďalšie hodnoty z tabuľky 5.4 sú rozdielne od hodnôt uvedených v [3, Obr. 4].
- Hraničné podmienky $c_i(N/2 - 1) = 1$ (pre detaily pozri [3]) sú chybné – očakávaný počet priradení v prípade, že chýba práve 1 párna/nepárna hodnota v S je ostro menší než 1, keďže existuje nenulová pravdepodobnosť, že počas jedného priebehu funkcie UPDATE chýbajúcu hodnotu nebudeme potrebovať a potom nastane konflikt z bežiacim kľúčom (t.j. nič sme neprirаďovali a hneď skončíme).

Opravená analýza. Definujeme si premenné $c_u^v(x, y)$, pre $u \in \{1, \ldots, 6\}$, $v \in \mathbb{Z}_4$, a $x, y \in \mathbb{Z}_{N/2+1}$. Premenná $c_u^v(x, y)$ označuje očakávaný počet priradení v nasledujúcej situácii:

x	počet známych hodnôt v S na párnych indexoch;
y	počet známych hodnôt v S na nepárnych indexoch;
u	prehľadávanie začína pred u-tym priradením;
v	pozícia v cykle periódy bežiaceho kľúča.

Celková zložitosť prehľadávania začínajúceho v špeciálnom stave je $c_1^0(0, 0)$. Hraničné prípady sú: $c_u^v(N/2, N/2) = 0$, a $c_u^v(x, N/2 + 1) = c_u^v(N/2 + 1, y) = 0$ pre všetky $u \in \{1, 2, \ldots, 6\}$, $v \in \mathbb{Z}_4$ a $x, y \in \mathbb{Z}_{N/2+1}$. Rovnice musia zohľadňovať paritu indexov pri prístupe k hodnotám S, pozri tabuľku 5.3. Označme $N' = N/2$. Nasledujúce rovnice sú zostrojené rovnakým spôsobom ako rovnice pre jednoduché prehľadávanie z časti 5.2.1.

Prípad $u = 1$ (potrebujeme hodnotu $S[i]$, parita indexu: $1, 0, 1, 0$):

$$c_1^v(x, y) = y/N' \cdot c_2^v(x, y) + (1 - y/N') \cdot (N' - y) \cdot (1 + c_2^v(x, y + 1)), \text{ pre } v \in \{0, 2\}$$
$$c_1^v(x, y) = x/N' \cdot c_2^v(x, y) + (1 - x/N') \cdot (N' - x) \cdot (1 + c_2^v(x + 1, y)), \text{ pre } v \in \{1, 3\}$$

Prípad $u = 2$ (potrebujeme hodnotu $S[j + S[i]]$, parita indexu: $0, 0, 0, 0$):

$$c_2^v(x, y) = x/N' \cdot c_3^v(x, y) + (1 - x/N') \cdot (N' - x) \cdot (1 + c_3^v(x + 1, y)), \text{ pre } v \in Z_4$$

Prípad $u = 3$ (potrebujeme hodnotu $S[j]$, parita indexu: $1, 0, 1, 0$):

$$c_3^v(x, y) = y/N' \cdot c_4^v(x, y) + (1 - y/N') \cdot (N' - y) \cdot (1 + c_4^v(x, y + 1)), \text{ pre } v \in \{0, 2\}$$
$$c_3^v(x, y) = x/N' \cdot c_4^v(x, y) + (1 - x/N') \cdot (N' - x) \cdot (1 + c_4^v(x + 1, y)), \text{ pre } v \in \{1, 3\}$$

Prípad $u = 4$ (potrebujeme hodnotu $S[z + k]$, parita indexu: $1, 0, 0, 1$):

$$c_4^v(x, y) = y/N' \cdot c_5^v(x, y) + (1 - y/N') \cdot (N' - y) \cdot (1 + c_5^v(x, y + 1)), \text{ pre } v \in \{0, 3\}$$
$$c_4^v(x, y) = x/N' \cdot c_5^v(x, y) + (1 - x/N') \cdot (N' - x) \cdot (1 + c_5^v(x + 1, y)), \text{ pre } v \in \{1, 2\}$$

Prípad $u = 5$ (potrebujeme hodnotu $S[i + S[z + k]]$, parita indexu: $1, 1, 0, 0$):

$$c_5^v(x, y) = y/N' \cdot c_6^v(x, y) + (1 - y/N') \cdot (N' - y) \cdot (1 + c_6^v(x, y + 1)), \text{ pre } v \in \{0, 1\}$$
$$c_5^v(x, y) = x/N' \cdot c_6^v(x, y) + (1 - x/N') \cdot (N' - x) \cdot (1 + c_6^v(x + 1, y)), \text{ pre } v \in \{2, 3\}$$

Case $u = 6$ (potrebujeme hodnotu $S[j + S[i + S[z + k]]]$, kontrolujeme korektnosť s bežiacim kľúčom, parita indexu: $1, 0, 0, 1$):

$$c_6^0(x, y) = y/N' \cdot 1/N' \cdot c_1^1(x, y) + (1 - y/N')^2 \cdot (1 + c_1^1(x, y + 1))$$
$$c_6^1(x, y) = x/N' \cdot 1/N' \cdot c_1^2(x, y) + (1 - x/N')^2 \cdot (1 + c_1^2(x + 1, y))$$
$$c_6^2(x, y) = x/N' \cdot 1/N' \cdot c_1^3(x, y) + (1 - x/N')^2 \cdot (1 + c_1^3(x + 1, y))$$
$$c_6^3(x, y) = y/N' \cdot 1/N' \cdot c_1^0(x, y) + (1 - y/N')^2 \cdot (1 + c_1^0(x, y + 1))$$

Výsledný systém má pre $N = 256$ približne 400 000 rovníc. Namiesto riešenia celého systému naraz môžeme postupne riešiť sústavy 24 rovníc pre 24 premenných $c_u^v(x, y)$ so zafixovaným x a y. Vhodne zvoleným poradím parametrov x a y docielime, že pri riešení sústavy budeme mať vždy hodnoty $c_u^v(x + 1, y)$ a $c_u^v(x, y + 1)$ už vypočítané.

Výpočet sústavy (aj s príkladom poradia parametrov x, y) uvádzame v algoritme 5.1. Na uloženie už vypočítaných hodnôt premenných, aby sme ich mohli následne použiť v ďalších sústavách, využijeme slovník (v algoritme 5.1 označený *memo*). V ňom sú pred výpočtom nastavené hodnoty premenných $c_u^v(N/2, N/2)$, $c_u^v(x, N/2 + 1)$ a $c_u^v(N/2 + 1, y)$ pre $u \in \{1, 2, \ldots, 6\}$, $v \in \mathbb{Z}_4$ a $x, y \in \mathbb{Z}_{N/2+1}$ na 0 (pozri hraničné podmienky rovníc). Počas výpočtu sú do *memo* pridávané hodnoty premenných z riešení jednotlivých sústav rovníc. Celková zložitosť sa na konci výpočtu bude nachádzať v *memo* pri premennej $c_1^0(0, 0)$.

Tabuľka 5.4 porovnáva výsledky našej teoretickej analýzy s rovnicami použitými v [3]. Tabuľka tiež ukazuje niektoré experimentálne hodnoty pre vybrané parametre. Celkovú zložitosť prehľadávania využívajúceho špeciálny stav pre $N = 256$ odhadujeme na $\approx 2^{941}$. V tomto odhade je započítané aj úplné preberanie hodnôt j a k (faktor $(N/2)^2 = 2^{14}$).

Poznámka. Pri experimentoch so špeciálnym stavom sme pozorovali, že je potrebný dlhší známy bežiaci kľúč než pri predošlých verziách prehľadávania. Postačujúcich bolo $5N$ slov bežiaceho kľúča ($3N$ na prehľadávanie a $2N$ na kontrolu konzistentnosti) na rozdiel od $3N$ slov pre prehľadávanie bez špeciálneho stavu.

			Teória		
N	x	y	Naša anlýza	[3]	Experimenty
16	0	0	23.02	24.80	23.08
18	0	0	27.02	29.80	27.14
20	0	0	31.19	35.10	∗31.28
64	22	22	13.99	34.50	14.07
128	50	50	15.19	59.32	15.26
256	110	110	12.37	87.60	12.13
64	0	0	152.62	194.11	–
128	0	0	381.03	500.05	–
256	0	0	927.00	1235.32	–

Pozn. 1: Čísla v posledných troch stĺpcoch predstavujú „lg“ zložitosti prehľadávania.

Pozn. 2: Posledný stĺpec je priemer z 1000 experimentov, okrem hodnoty označenej * (priemer zo 100 experimentov).

Tabuľka 5.4: Spritz – zložitosť prehľadávania využívajúceho špeciálny stav

1: **function** GenEquations($x, y, memo$)
2: Vygeneruj rovnice pre fixné x, y a $\forall v \in \mathbb{Z}_4$, $\forall u \in \{1, \ldots, 6\}$ s použitím hodnôt z $memo$

3: **function** SolveEquations($eqs, memo$)
4: Vyrieš sústavu rovníc eqs
5: Hodnoty vypočítaných premenných pridaj do $memo$

6: **function** Solve()
7: $memo$ = hraničné podmienky
8: **for** $x = N/2$ **to** 0
9: **for** $y = N/2$ **to** 0
10: eqs = GenEquations($x, y, memo$)
11: SolveEquations($eqs, memo$)
12: **if** y = x **then**
13: **break**
14: eqs = GenEquations(y, x)
15: SolveEquations($eqs, memo$)
16: **return** $memo[c_1^0(0, 0)]$

Algoritmus 5.1: Riešenie sústavy rovníc pre špeciálny stav

5.3 Tradeoff pre Spritz

Útok prehľadávaním je možné okrem spustenia na neznámej počiatočnej permutácii vykonať aj iným spôsobom: zafixujeme si niekoľko (označme m) pozícii v permutácii a pre každú možnosť m rôznych hodnôt zo $\mathbb{Z}_N$ spustíme prehľadávanie na stave predvyplnenom týmito m hodnotami. Všetkých možností, ako vyplniť m prvkov na fixných miestach v permutácii veľkosti N je $N^{\underline{m}} = N(N-1)\dots(N-m+1)$. Samotné predvyplnenie stavu zníži zložitosť prehľadávania, celková zložitosť takéhoto prehľadávania v závislosti od počtu predvyplnených prvkov je znázornená na obrázku 5.1. Ako zložitosť prehľadávania je použitý odhad prehľadávania so zmenou poradia (pozri časť 5.2.2).

Ako vidíme z obrázku 5.1, tento postup nezlepšuje zložitosť prehľadávania, najefektívnejšie je spustiť prehľadávanie na úplne neznámom stave.

Predvyplnenie môžeme urobiť aj iným spôsobom. Namiesto zafixovania pozícií a úplného preberania ich hodnôt zoberieme prvých m rôznych hodnôt bežiaceho kľúča, tieto použijeme ako predvyplnené hodnoty a vyskúšame všetky možnosti, na ktorých miestach týchto m zafixovaných hodnôt môže byť v permutácii (takýchto možností je opäť $N^{\underline{m}}$). V prípade použitia prehľadávania so zmenou poradia z časti 5.2.2 to má nasledovnú výhodu: prvých m iterácií funkcie UPDATE v prehľadávaní sa bude z_t vždy nachádzať v S, čím sa vyhneme potenciálnemu skúšaniu všetkých možností do $S[i + S[z + k]]$ (prípad 3 v časti 5.2.2). Rovnice pre takýto odhad sú pre $x \geq 2m$ rovnaké ako v časti 5.2.2, pre $m \leq x < 2m$ sú rozdiely nasledovné:

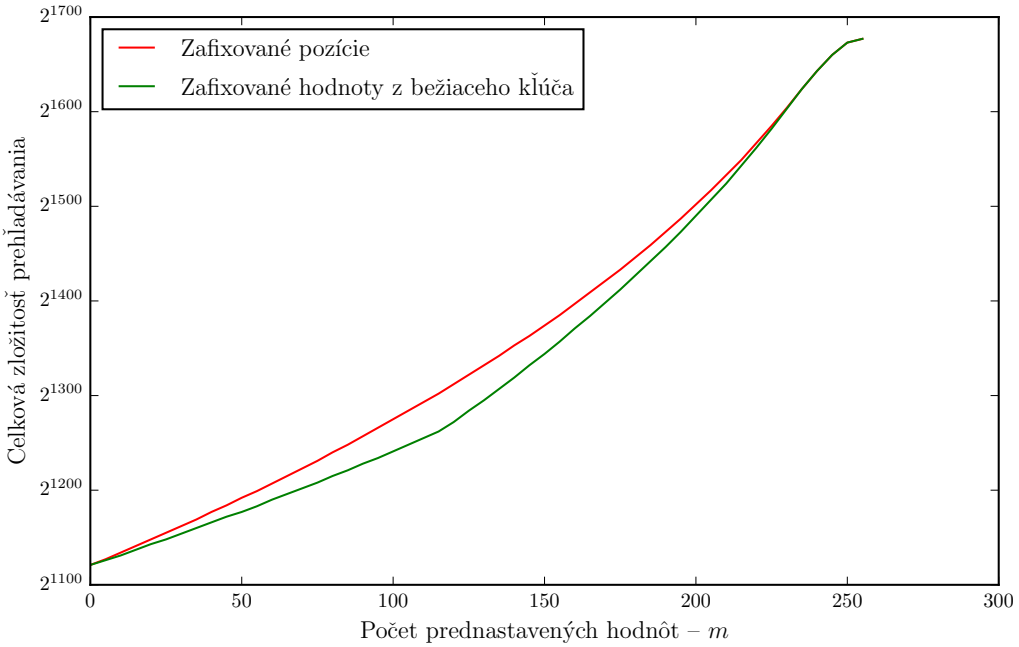
1. Situácia, že sa z_t nenachádza v S nikdy nenastane a preto rovnica pre $c_5(x)$ vyzerá nasledovne:

$$c_5(x) = (1 - x/N)^2 \cdot (1 + c_1(x + 1)) + (x/N) \cdot (1/N) \cdot c_1(x)$$

2. Rovnicu pre $c_6(x)$ nemusíme uvažovať, keďže táto situácia v prehľadávaní nikdy nenastane. Táto hodnota nám zároveň nebude v rovniciach „chýbať“, keďže jediné miesto, kde bola potrebná, je pôvodná rovnica pre $c_5(x)$.

Rovnice pre $x < m$ nemusíme uvažovať, keďže prehľadávanie začíname s m prednastavenými hodnotami. Zložitosť vypočítame obvyklým spôsobom od hraničných hodnôt pre $x \geq N$ po $x = m$.

Celková zložitosť takto upraveného prehľadávania v závislosti od m je znázornená taktiež na obrázku 5.1. Vidíme, že predvyplnenie permutácie hodnotami z bežiaceho kľúča má pre väčšinu hodnôt m za následok menší počet priradení než zafixovanie pozícií s úplným preberaním hodnôt, keďže pre $x < 2m$ v kontrole konzistentnosti nikdy neskúšame všetky možnosti hodnôt $S[i + S[z + k]]$. Pre $m = 0$ z oboch metód dostaneme obyčajné prehľadávanie popísané v predchádzajúcich častiach. Pre $m > 0$ je celková zložitosť útoku vždy vyššia, takže najefektívnejšie je začať prehľadávanie v úplne neznámom stave.



Obrázok 5.1: Tradeoff predvyplnenia počiatočného stavu, $N = 256$

Záver

V tejto práci sme sa zaoberali útokmi na vnútorný stav šifier RC4 a Spritz. Brali sme do úvahy dva typy metód: použitie SMT solverov a prehľadávanie s návratom.

Ukázali sme, ako popísať vzťah medzi počiatočným stavom šifry a bežiacim kľúčom v jazyku SMT a ako tento popis spolu so znalosťou bežiaceho kľúča použiť na zistenie stavu šifry. Ukázali sme, že na korektné určenie počiatočného stavu je potrebné určité množstvo bežiaceho kľúča, ktorého nárast ale výrazne zvyšuje čas výpočtu SMT solvera. Pre RC4 sme porovnali použite SMT solverov s algebraickou analýzou [21], pričom náš postup bol jednak schopný nájsť počiatočný stav pre väčšie parametre šifry a použitie jazyka SMT bolo omnoho jednoduchšie.

Ďalej sme sa zaoberali útokmi na vnútorný stav šifry Spritz pomocou prehľadávania s návratom. Navrhli sme vylepšenia prehľadávania, založené na zmene poradia krokov prehľadávania, na použití prefixu bežiaceho kľúča na dodatočnú kontrolu a na zmene poradia skúšaných hodnôt v prehľadávaní. Experimentálne sme ich overili a pre kombináciu prvých dvoch optimalizácií sme v experimentoch na redukovaných verziách šifry pozorovali dvojnásobné až štvornásobné zlepšenie zložitosti. Zmena poradia skúšaných hodnôt zložitosť v priemernom prípade nezlepšovala. Pokiaľ nepredpokladáme dodatočné vlastnosti počiatočného stavu, je prehľadávanie so zmenou poradia krokov a použitím prefixu bežiaceho kľúča najlepším doteraz známym útokom na počiatočný stav šifry Spritz.

V predchádzajúcich prácach, ktoré odhadovali časovú zložitosť prehľadávania pre RC4 a Spritz sme pozorovali viaceré nedostatky. Navrhli sme vlastné odhady viacerých verzií prehľadávania. Mierne sme zlepšili odhad prehľadávania pre RC4 so zmenou poradia krokov [9]. Pre Spritz sme výrazne zlepšili odhad prehľadávania bez vylepšení [2], odhadli sme zložitosť prehľadávania so zmenou poradia krokov a zlepšili sme odhad prehľadávania v prípade, že šifra začína v špeciálnom stave [3]. Experimentovali sme s odhadom zložitosti v prípade, keď časť stavu šifry pred útokom predvyplníme. Takýto tradeoff ale nie je výhodný a efektívnejšie je začínať s „prázdnym“ stavom.

Ďalší výskum v tejto oblasti vidíme hlavne v zlepšení prehľadávania a odhadov zložitosti pre šifru Spritz. Jedna z možností je navrhnúť iné optimalizácie prehľadávania buď využitím vlastností šifry Spritz alebo modifikáciou podobných útokov na šifru RC4. Prínosom by bol aj teoretický odhad zložitosti optimalizácie s využitím prefixu bežiaceho kľúča. Zaujímavým otvoreným problém pri teoretických odhadoch zložitos-

ti prehľadávania je presnejšie určenie pravdepodobností, že konkrétna hodnota stavu šifry je neznáma, keďže tieto pravdepodobnosti ovplyvňujú odhad celkovej zložitosti útoku. Inou možnosťou ďalšieho výskumu je modifikácia prehľadávania na ďalšie typy prúdových šifier.

DODATOK A

Ukážka SMT popisu

A.1 RC4

Ukážka SMT popisu hľadania počiatočného stavu RC4 pre logiku *auflia* s parametrami $N = 4$, dĺžka bežiaceho kľúča 2, počiatočný stav $S = [2, 0, 3, 1]$, známy bežiaci kľúč $[3, 0]$.

```
(set-option :produce-models true)
(set-logic QF_AUFLIA)
; declaration of modulo function
(define-fun modulo ((x Int) (y Int)) Int (ite (>= x y) (- x y) x))
; declarations of variables for each round
(declare-fun j0 () Int)
(declare-fun j1 () Int)
(declare-fun j2 () Int)
(declare-fun S0 () (Array Int Int))
(declare-fun S1 () (Array Int Int))
(declare-fun S2 () (Array Int Int))
; constraints on registers
(assert (= j0 0))
(assert (>= j1 0))
(assert (< j1 4))
(assert (>= j2 0))
(assert (< j2 4))
; constraints on S0
(assert (>= (select S0 0) 0))
(assert (< (select S0 0) 4))
(assert (>= (select S0 1) 0))
(assert (< (select S0 1) 4))
```

```
(assert (>= (select S0 2) 0))
(assert (< (select S0 2) 4))
(assert (>= (select S0 3) 0))
(assert (< (select S0 3) 4))
; asserting S0 is permutation on 4 elements
(assert (= 6
    (+ (select S0 0) (+ (select S0 1) (+ (select S0 2) (select S0 3))))
))
(assert (not (= (select S0 0) (select S0 1) )))
(assert (not (= (select S0 0) (select S0 2) )))
(assert (not (= (select S0 0) (select S0 3) )))
(assert (not (= (select S0 1) (select S0 2) )))
(assert (not (= (select S0 1) (select S0 3) )))
(assert (not (= (select S0 2) (select S0 3) )))
; rounds of cipher
; round 1
(assert (= j1 (modulo (+ j0 (select S0 1)) 4)))
(assert (= S1 (store (store S0 j1 (select S0 1)) 1 (select S0 j1))))
(assert (= 3 (select S1 (modulo (+ (select S1 1) (select S1 j1)) 4))))
; round 2
(assert (= j2 (modulo (+ j1 (select S1 2)) 4)))
(assert (= S2 (store (store S1 j2 (select S1 2)) 2 (select S1 j2))))
(assert (= 0 (select S2 (modulo (+ (select S2 2) (select S2 j2)) 4))))
(check-sat)
; getting values of S0
(get-value ((select S0 0)))
(get-value ((select S0 1)))
(get-value ((select S0 2)))
(get-value ((select S0 3)))
(exit)
```

A.2 Spritz

Ukážka SMT popisu hľadania počiatočného stavu Spritz pre logiku *abv* s parametrami $N = 4$, dĺžka bežiaceho kľúča 2, počiatočný stav $S = [1, 2, 0, 3]$, registre v poradí i, j, k, z, a, w sú $2, 2, 3, 0, 0, 3$, známy bežiaci kľúč $[2, 3]$.

```
(set-option :produce-models true)
(set-logic QF_ABV)
; declaration of w
(declare-fun w () (_ BitVec 2))
; declaration of state variables for each round
(declare-fun S0 () (Array (_ BitVec 2) (_ BitVec 2)))
(declare-fun S1 () (Array (_ BitVec 2) (_ BitVec 2)))
(declare-fun S2 () (Array (_ BitVec 2) (_ BitVec 2)))
(declare-fun i0 () (_ BitVec 2))
(declare-fun i1 () (_ BitVec 2))
(declare-fun i2 () (_ BitVec 2))
(declare-fun j0 () (_ BitVec 2))
(declare-fun j1 () (_ BitVec 2))
(declare-fun j2 () (_ BitVec 2))
(declare-fun k0 () (_ BitVec 2))
(declare-fun k1 () (_ BitVec 2))
(declare-fun k2 () (_ BitVec 2))
(declare-fun z0 () (_ BitVec 2))
(declare-fun z1 () (_ BitVec 2))
(declare-fun z2 () (_ BitVec 2))
; asserting S0 is permutation on 4 elements
(assert (not (= (select S0 #b00) (select S0 #b01) )))
(assert (not (= (select S0 #b00) (select S0 #b10) )))
(assert (not (= (select S0 #b00) (select S0 #b11) )))
(assert (not (= (select S0 #b01) (select S0 #b10) )))
(assert (not (= (select S0 #b01) (select S0 #b11) )))
(assert (not (= (select S0 #b10) (select S0 #b11) )))
(assert (= z0 #b00))
(assert (= i0 #b10))
(assert (= w #b11))
; fixed keystream [2, 3]
(assert (= z1 #b10))
(assert (= z2 #b11))
; rounds of cipher
```

```
; round 1
(assert (= i1 (bvadd i0 w)))
(assert (= j1 (bvadd k0 (select S0 (bvadd j0 (select S0 i1))))))
(assert (= k1 (bvadd i1 (bvadd k0 (select S0 j1)))))
(assert (= S1 (store (store S0 j1 (select S0 i1)) i1 (select S0 j1))))
(assert (= z1
   (select S1 (bvadd j1 (select S1 (bvadd i1 (select S1 (bvadd z0 k1))))))
))
; round 2
(assert (= i2 (bvadd i1 w)))
(assert (= j2 (bvadd k1 (select S1 (bvadd j1 (select S1 i2))))))
(assert (= k2 (bvadd i2 (bvadd k1 (select S1 j2)))))
(assert (= S2 (store (store S1 j2 (select S1 i2)) i2 (select S1 j2))))
(assert (= z2
   (select S2 (bvadd j2 (select S2 (bvadd i2 (select S2 (bvadd z1 k2))))))
))
(check-sat)
;getting values of starting state
(get-value ((select S0 #b00)))
(get-value ((select S0 #b01)))
(get-value ((select S0 #b10)))
(get-value ((select S0 #b11)))
(get-value (i0))
(get-value (j0))
(get-value (k0))
(get-value (z0))
(get-value (w))
(exit)
```

DODATOK B

Zdrojové kódy

Zdrojové kódy programov použitých v práci sú dostupné na

`https://github.com/mgabris/master-thesis-attachment`

a na priloženom CD. Informácie o použití jednotlivých programov sú v príslušných adresároch prílohy.

Príloha obsahuje:

smt/
: Nástroj na generovanie SMT programov pre RC4 a Spritz (časť 2.3) a spúšťanie SMT solverov.

generate/
: Nástroj na generovanie náhodných stavov šifier RC4 a Spritz pre experimenty so SMT solvermi a prehľadávaním.

backtrack/equations/
: Výpočty rovníc odhadov zložitostí z kapitoly 5.

backtrack/spritz-py/
: Implementácia prehľadávania s návratom z kapitoly 3 a vylepšení z kapitoly 4 pre šifru Spritz v jazyku Python.

backtrack/spritz-cpp/
: Implementácia prehľadávania s návratom z kapitoly 3 a vylepšenia použitím špeciálneho stavu z časti 5.2.3 pre šifru Spritz v jazyku C++.

backtrack/rc4-cpp/
: Implementácia prehľadávania s návratom z kapitoly 3 a vylepšenia so zmenou poradia z časti 4.1 modifikované pre šifru RC4 v jazyku C++.

Literatúra

[1] Nadhem J. AlFardan et al.: *On the Security of RC4 in TLS*. In: *USENIX Security*. 2013, pp. 305–320. URL: https://www.usenix.org/conference/usenixsecurity13/technical-sessions/paper/alFardan.

[2] Ralph Ankele, Stefan Kölbl, Christian Rechberger: *State-Recovery Analysis of Spritz*. In: *LATINCRYPT*. Vol. 9230. Lecture Notes in Computer Science. Springer, 2015, pp. 204–221. DOI: 10.1007/978-3-319-22174-8_12.

[3] Subhadeep Banik, Takanori Isobe: *Cryptanalysis of the Full Spritz Stream Cipher*. In: *IACR Cryptology ePrint Archive* 2016 (2016), p. 92. URL: http://eprint.iacr.org/2016/092.

[4] Clark Barrett, Aaron Stump, Cesare Tinelli: *The SMT-LIB Standard: Version 2.0*. In: *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK)*. Ed. by A. Gupta, D. Kroening. 2010.

[5] Clark Barrett et al.: *CVC4*. In: *CAV*. Vol. 6806. Lecture Notes in Computer Science. Springer, 2011, pp. 171–177. DOI: 10.1007/978-3-642-22110-1_14.

[6] Guido Bertoni et al.: *Cryptographic sponge functions*. http://sponge.noekeon.org. 2011.

[7] Bruno Dutertre: *Yices 2.2*. In: *CAV*. Vol. 8559. Lecture Notes in Computer Science. Springer, 2014, pp. 737–744. DOI: 10.1007/978-3-319-08867-9_49.

[8] Scott R. Fluhrer, Itsik Mantin, Adi Shamir: *Weaknesses in the Key Scheduling Algorithm of RC4*. In: *SAC*. Vol. 2259. Lecture Notes in Computer Science. Springer, 2001, pp. 1–24. DOI: 10.1007/3-540-45537-X_1.

[9] Lars R. Knudsen et al.: *Analysis Methods for (Alleged) RC4*. In: *ASIACRYPT*. Vol. 1514. Lecture Notes in Computer Science. Springer, 1998, pp. 327–341. DOI: 10.1007/3-540-49649-1_26.

[10] Itsik Mantin, Adi Shamir: *A Practical Attack on Broadcast RC4*. In: *FSE*. Vol. 2355. Lecture Notes in Computer Science. Springer, 2001, pp. 152–164. DOI: 10.1007/3-540-45473-X_13.

[11] Alexander Maximov: *Some Words on Cryptanalysis of Stream Ciphers*. eng. PhD thesis. Lund University, 2006, p. 256. ISBN: 91-7167-039-4. URL: http://lup.lub.lu.se/record/25433.

[12] Alexander Maximov, Dmitry Khovratovich: *New State Recovery Attack on RC4*. In: *CRYPTO*. Vol. 5157. Lecture Notes in Computer Science. Springer, 2008, pp. 297–316. DOI: 10.1007/978-3-540-85174-5_17.

[13] Alfred Menezes, Paul C. van Oorschot, Scott A. Vanstone: *Handbook of Applied Cryptography*. CRC Press, 1996. ISBN: 0-8493-8523-7.

[14] Leonardo Mendonça de Moura, Nikolaj Bjørner: *Satisfiability modulo theories: introduction and applications*. In: *Commun. ACM* 54.9 (2011), pp. 69–77. DOI: 10.1145/1995376.1995394.

[15] Leonardo Mendonça de Moura, Nikolaj Bjørner: *Z3: An Efficient SMT Solver*. In: *TACAS*. Vol. 4963. Lecture Notes in Computer Science. Springer, 2008, pp. 337–340.

[16] Aina Niemetz, Mathias Preiner, Armin Biere: *Boolector 2.0*. In: *JSAT* 9 (2015), pp. 53–58. URL: https://satassociation.org/jsat/index.php/jsat/article/view/120.

[17] Toshihiro Ohigashi et al.: *How to Recover Any Byte of Plaintext on RC4*. In: *SAC*. Vol. 8282. Lecture Notes in Computer Science. Springer, 2013, pp. 155–173. DOI: 10.1007/978-3-662-43414-7_8.

[18] Goutam Paul, Subhamoy Maitra: *Permutation After RC4 Key Scheduling Reveals the Secret Key*. In: *SAC*. Vol. 4876. Lecture Notes in Computer Science. Springer, 2007, pp. 360–377. DOI: 10.1007/978-3-540-77360-3_23.

[19] Ronald L. Rivest, Jacob C. N. Schuldt: *Spritz—A spongy RC4-like stream cipher and hash function*. Presented at Charles River Crypto Day (2014-10-24). Aug. 19, 2014.

[20] The Sage Developers: *SageMath, the Sage Mathematics Software System (Version 7.0)*. 2016. URL: http://www.sagemath.org.

[21] Kenneth Koon-Ho Wong, Gary Carter, Ed Dawson: *An Analysis of the RC4 Family of Stream Ciphers against Algebraic Attacks*. In: *AISC*. Vol. 105. CRPIT. Australian Computer Society, 2010, pp. 67–74. URL: http://crpit.com/abstracts/CRPITV105Wong.html.

[22] Bartosz Zoltak: *Statistical weakness in Spritz against VMPC-R: in search for the RC4 replacement*. In: *IACR Cryptology ePrint Archive* 2014 (2014), p. 985. URL: http://eprint.iacr.org/2014/985.