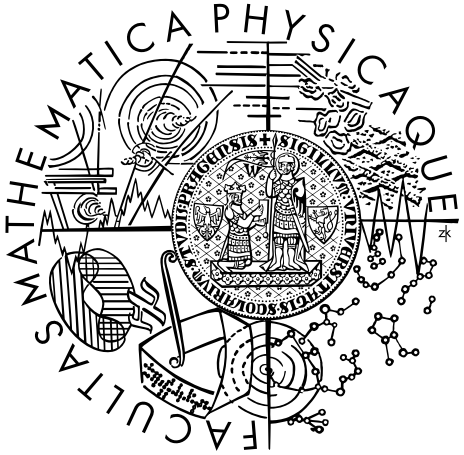
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE



Martin Dvořák

Využití celulárních automatů pro šifrování dat

Katedra teoretické informatiky a matematické logiky

Vedoucí bakalářské práce:	Mgr. Otakar Trunda
Studijní program:	Informatika
Studijní obor:	Obecná informatika

Praha 2016

Prohlašuji, že jsem tuto bakalářskou práci vypracoval(a) samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V ..... dne ..... Podpis autora

Název práce: Využití celulárních automatů pro šifrování dat

Autor: Martin Dvořák

Katedra teoretické informatiky a matematické logiky: Katedra teoretické informatiky a matematické logiky

Vedoucí bakalářské práce: Mgr. Otakar Trunda, Katedra teoretické informatiky a matematické logiky

Abstrakt: Abstrakt.

Klíčová slova: celulární automaty šifrování protahování klíčů

Title: Using Cellular Automata for Data Encryption

Author: Martin Dvořák

Department of Theoretical Computer Science and Mathematical Logic: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: Mgr. Otakar Trunda, Department of Theoretical Computer Science and Mathematical Logic

Abstract: Abstract.

Keywords: cellular automata cryptography key stretching

Děkuji svému vedoucímu za ochotnou pomoc.

Obsah

Úvod **3**
Struktura práce . . . 3

1 Celulární automaty **4**
1.1 Co jsou to celulární automaty . . . 4
1.2 Elementární celulární automaty . . . 4
1.3 Jednorozměrné celulární automaty . . . 5
1.4 Dvourozměrné celulární automaty . . . 8
1.5 Implementace celulárních automatů . . . 9

2 Šifrování **10**
2.1 Základní pojmy . . . 10
2.2 Základní východiska . . . 10
2.3 O co nám jde . . . 10
2.4 Pseudonáhodné generátory . . . 10
2.5 Blokové šifry . . . 11
2.5.1 Operační mody . . . 12
2.6 Útoky na šifry . . . 16
2.6.1 Rozdělení podle povolených prostředků . . . 17
2.6.2 Příklady útoků . . . 17
2.7 Testy . . . 17
2.7.1 Testy na úrovni jednotlivých výstupů . . . 17
2.7.2 Testy na úrovni celého zobrazení . . . 18
2.7.3 Použití testů . . . 19
2.7.4 Grafické znázornění . . . 19

3 Nástin architektury **21**
3.1 Úvod k návrhu . . . 21
3.2 Cellular . . . 21
3.2.1 Abstraktní třída CellularAutomaton . . . 21
3.2.2 Koncept třídní hierarchie . . . 22
3.2.3 Ukládání stavu celulárního automatu . . . 23
3.2.4 Optimalizace výpočtu elementárních automatů . . . 24
3.3 Crypto . . . 24
3.4 Testing . . . 25
3.5 Program . . . 25
3.6 CryptographyUnitTests . . . 25

4 Protahování klíčů pomocí celulárních automatů **26**
4.1 KeyExtenderQuadratic . . . 28
4.2 KeyExtenderSimpleLinear . . . 29
4.3 KeyExtenderInterlaced . . . 33
4.4 KeyExtenderUncertain . . . 38

4.5 KeyExtenderGenetic 42

5 Další možnosti šifrování pomocí celulárních automatů 48
5.1 Řešení 48

Závěr 51

Seznam použité literatury 52

Seznam obrázků 53

Seznam tabulek 55

Seznam použitých zkratek 56

Přílohy 57

Úvod

Toto je implementačně–experimentální práce. Cílem práce je vytvořit šifrovací algoritmus využívající celulární automaty. Úkolem je naprogramovat různé způsoby natahování šifrovacích klíčů a naměřit změřit jejich vlastnosti, za účelem zvolení správného algoritmu pro šifrovací aplikaci.

Uchovávání dokumentů v tajnosti je velmi důležité. Kompromitace údajů, které nesmí být veřejné, může vést k finančním ztrátám, nepříjemným událostem v životě nebo postihům za porušování zákonů. V současné době je známo velké množství šifrovacích algoritmů, které slouží právě k tomu, aby nějaký dokument či zprávu byl schopen číst jen jeho vlastník, případně člověk, pro kterého je zpráva určena.

Není ale zatím moc známo o použití celulárních automatů pro šifrování dat. Ukazuje se, že celulární automaty (například elementární celulární automat číslo 30) dokáží generovat data, která jsou i podle přísných hledisek pseudonáhodná. To nás vede k naději, že by z celulárních automatů šel udělat například nástroj na protahování šifrovacích klíčů. Celulární automaty jsou velmi univerzální model výpočtů a kdyby se je podařilo aplikovat na šifrování dat, mohlo by to vést k mnohým výhodám díky jejich jednoduchosti a paralelizovatelnosti. Domníváme se, že takový algoritmus založený na celulárních automatech by mohl být v budoucnu implementován na velmi rozmanitých podobách hardwaru.

Struktura práce

V 1. kapitole jsou představeny celulární automaty, jak je rozdělujeme, co umí a jaké chování vykazují. Ve 2. kapitole se nachází úvod do šifrování a trocha teorie o testování kvality šifrovacích algoritmů. Ve 3. kapitole je představen vytvořený program v C#, z jakých částí se skládá a pár zajímavých implementačních detailů. Těžištěm práce je 4. kapitola, ve které jsou rozebírány vyzkoumané možnosti natahování klíčů a výsledky na nich naměřené. V 5. kapitole jsou stručně rozebrány další možnosti využití celulárních automatů v kryptografii.

1. Celulární automaty

1.1 Co jsou to celulární automaty

Celulární automat (CA) je diskrétní model, který se skládá z pravidelné mřížky buněk. Buňky se nacházejí v určitých stavech, přičemž množina stavů a pravidla pro přechod mezi stavy jsou společná pro celý automat. CA se vyvíjí diskrétně v čase (celý najednou).

Specialitou CA je to, že i velmi jednoduchá sada pravidel může vést k velmi komplexnímu chování. Celulární automaty našly využití jako modely v biologii, chemii, fyzice, ale také třeba jako nástroj při procedurálním generování terénu pro počítačové hry.

Budeme uvažovat různé varianty CA podle následujících kritérií:

- Počet dimenzí: Jsou všechny buňky v jedné řadě (1D), nebo ve čtvercové mřížce (2D), či krychlově uspořádané (3D)?
- Počet stavů: Pro mnoho aplikací stačí používat binární CA, kde se každá buňka může nacházet ve stavu 0, nebo ve stavu 1. Jindy se ale uvažuje mnohem více stavů (například při zkoumání tzv. self-replicating machines).
- Okolí: Uvnitř CA jsou všechna pravidla lokální, což znamená že přechodová funkce pro buňku závisí jen na stavu jejím a několika buněk v okolí. Toto okolí může mít různý počet buněk a různý tvar.
- Typ pravidla: Pravidlo může každé uspořádané n-tici stavu buněk v okolí přiřadit nový stav buňky. Nebo může mít pravidlo jednodušší podobu, například jen posčítat stavy buněk v okolí a nerozlišovat různé lokalizace.
- Speciální údaje: Existují také varianty CA, které v buňkách ukládají navíc určité speciální údaje. Například by nás mohl zajímat CA, ve kterém jsou některé buňky zmrazeny (a nevyvíjejí se) a jiné se vyvíjejí (přičemž podle přechodového pravidla mohou rozmrazit sousedy či naopak zmrazit samy sebe). Dalším speciálním údajem může být paměť. Kromě současného stavu může být v buňce zapamatován také její předchozí stav a ten může být využit v přechodové funkci. Pokud se jedná o binární CA s pamětí předchozího stavu, lze ho bez újmy na obecnosti odsimulovat na normálním CA bez paměti, který má 4 stavy. A v případě 1D automatu lze ještě tento 4-stavový automat simulovat na binárním automatu, který ale používá větší okolí a více buněk celkově, viz (Wolfram, 2002), strana 655.

1.2 Elementární celulární automaty

Wofram popisuje na počátku své rozsáhlé práce (Wolfram, 2002) nejprve elementární celulární automaty, kterých existuje 256. Elementární CA obsahuje jednoduchou řadu buněk (jednorozměrnou), kde se každá buňka může nacházet ve 2 možných stavech. Stav 0 označme bílou barvou a mluvme o dané buňce jako o mrtvé. Stav 1 označme červnou barvou a mluvme o dané buňce jako o živé. U elementárních CA každý nový stav každé buňky závisí pouze na jejím stavu a stavu přímých sousedů. Pokud přechodovou funkci

nazveme f, čas označíme t a buňky automatu reprezentuje vektor x (například 5. buňka v počátečním stavu je reprezentovaná hodnotou $x_5(0)$), tak můžeme přechod formálně zapsat jako:

$$x_i(t+1) = f(x_{i-1}(t), x_i(t), x_{i+1}(t))$$

, kde

$$f : \{0,1\}^3 \rightarrow \{0,1\}$$

, tudíž existuje $2^{2^3} = 2^8 = 256$ takových funkcí f.

Elementární automaty číslujeme tak, že pravidlo zapsané jako osmice bitů (kde první bit je obraz (1, 1, 1) a poslední bit je obraz (0, 0, 0), řadí se lexikograficky) převedeme do dvojkové soustavy.

Ukážeme si teď několik elementárních automatů, na které se budeme v práci odkazovat. Zakreslíme vždy jejich vývoj v čase, kde čas roste směrem dolů, přičemž počáteční konfigurace se skládá se samých nul (bílých / mrtvých buněk) s jedinou jedničkou (černou / živou buňkou) uprostřed. Lze si všimnout, že budeme využívat jen automaty se sudými čísly. Ty s lichými čísly totiž zobrazují (0, 0, 0) na 1 a hned v prvním kroce zaplní neomezené množství buněk.

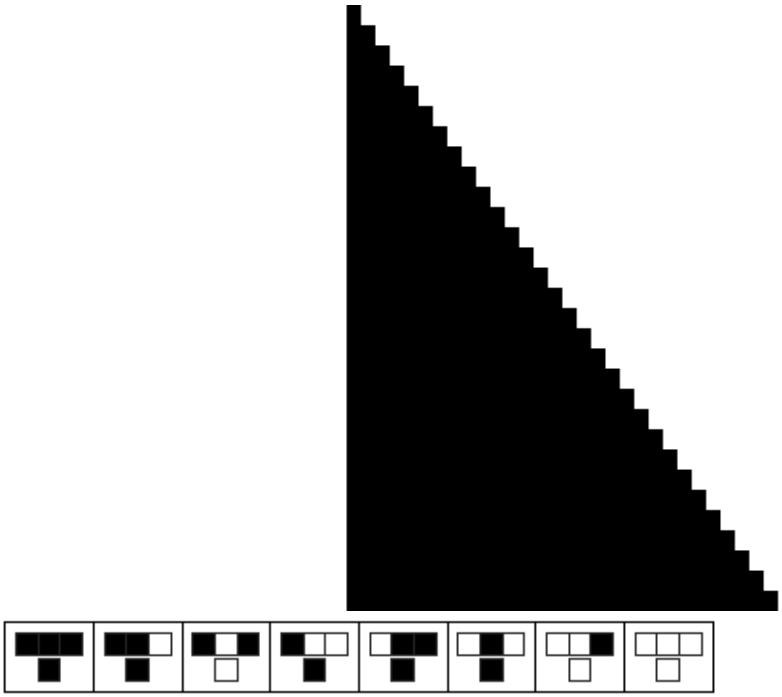
V knize (Schiff, 2008) je na straně 70 popsána klasifikace celulárních automatů do 4 tříd. Příkladem CA s úplně jednotvárným chováním (Class I) je elementární automat číslo 220 (viz 1.1). Podobně jednotvárné chování vykazuje automat číslo 94 (viz 1.2). Zajímavější je automat číslo 90, který opakuje stejné vzory (Class II) a dokonce generuje fraktální tvary (viz 1.3). Jeho pravidlo se dá popsat tak, že ignoruje současný stav buňky a jako nový stav buňka nastaví xor mezi stavy levého a pravého souseda. Automat číslo 30 vykazuje chaotické (pseudonáhodné) chování (Class III, viz 1.4). A nakonec si prohlédneme automat číslo 110 (viz 1.5), který vykazuje určité lokální struktury, které spolu mohou složitě interagovat (Class IV).

1.3 Jednorozměrné celulární automaty

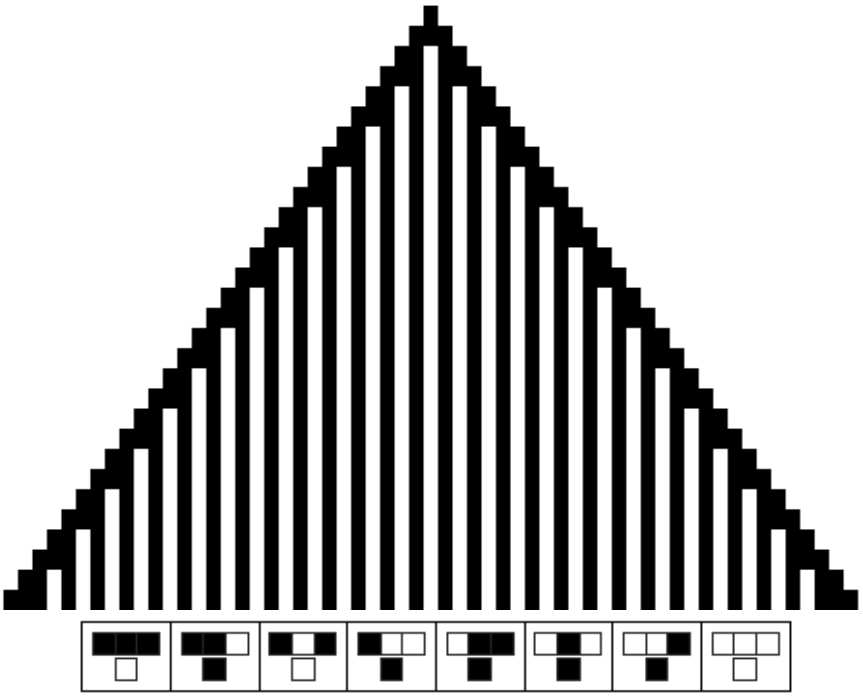
Elementární automaty byly příkladem 1D binárního automatu. Ale lze uvažovat i jiné jednorozměrné automaty. Můžeme ponechat to, že veškerá data se nacházejí v jednorozměrné mřížce buněk, ale učinit následující úpravy:

- Nemusíme se omezovat jen na těsné sousedy.
- Můžeme povolit více než 2 stavy.
- Významným druhem celulárních automatů jsou totalistické automaty. Nehledě na to, jak velké se používá okolí a kolik stavů buňlky nabývají, v totalistických automatech se pouze číselně posčítají hodnoty hodnoty buňky s celým jejím okolím a podle součtu hodnot se přiřadí výsledná hodnota. Přechodová funkce pro totalistický automat s okolím velikosti o a počtem stavů s se tedy dá vyjádřit jako:

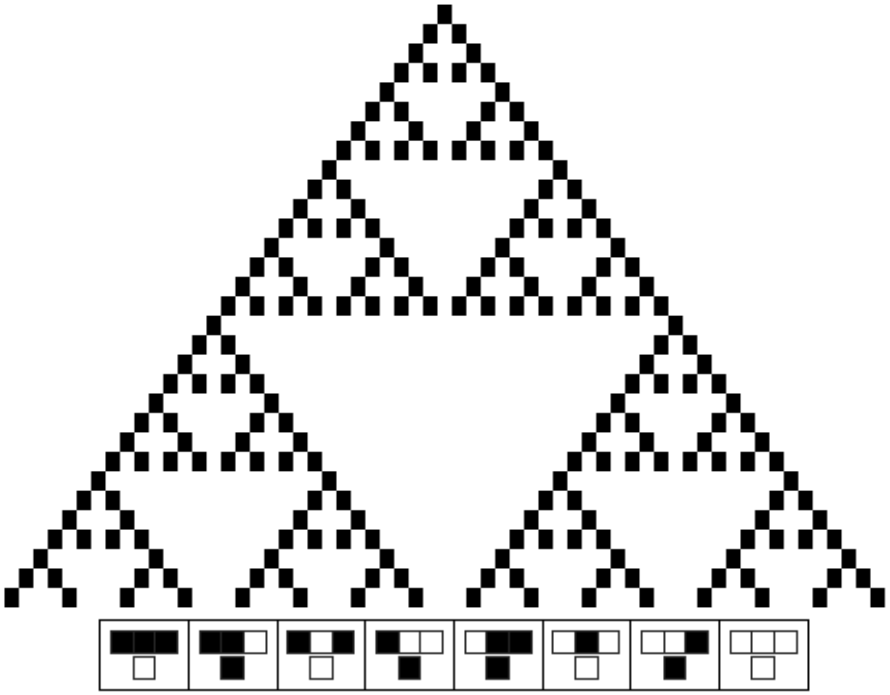
$$f : \{0,..,(s-1)(o+1)\} \rightarrow \{0,..,s-1\}$$



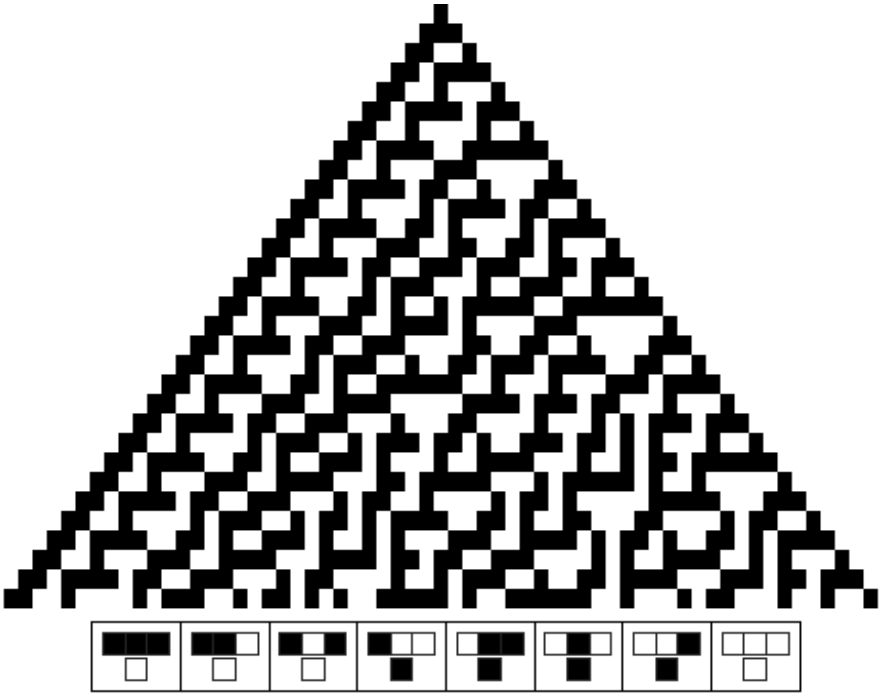
Obrázek 1.1: Elementární automat číslo 220



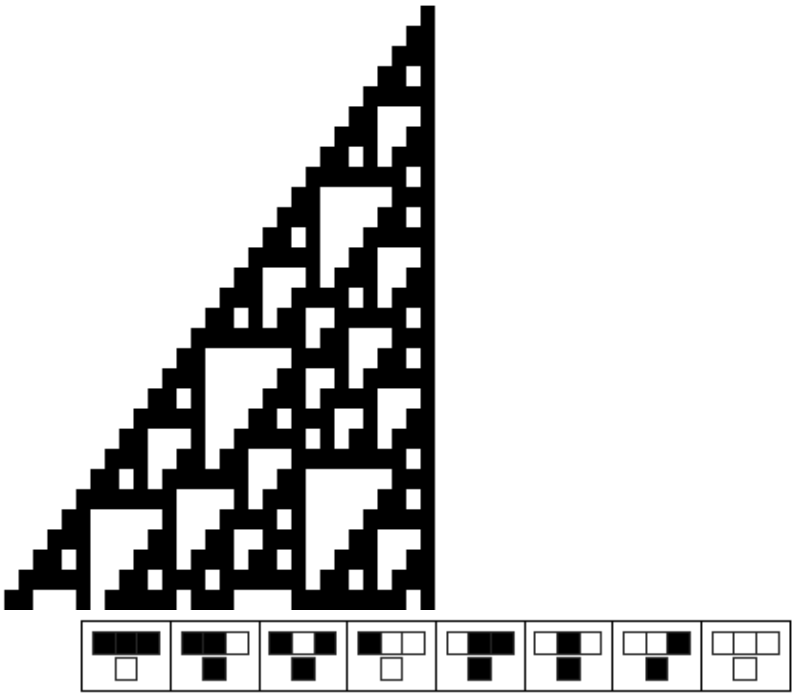
Obrázek 1.2: Elementární automat číslo 94



Obrázek 1.3: Elementární automat číslo 90



Obrázek 1.4: Elementární automat číslo 30



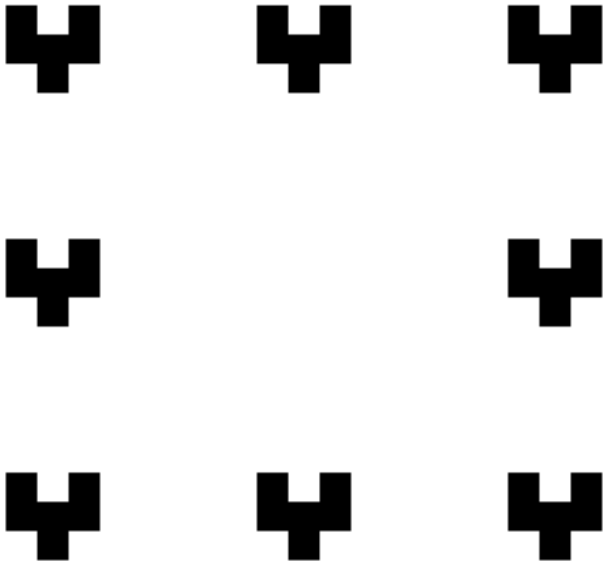
Obrázek 1.5: Elementární automat číslo 110

1.4 Dvourozměrné celulární automaty

Buňky taky můžeme umístit do pravidelné dvojrozměrné mřížky, která vypadá jako čtverečkovaný papír. V oblasti 2D celulárních automatů se používají hlavně totalistické automaty, protože vytváření jiných typů pravidel by bylo příliš složité. Typických příkladem totalistického 2D automatu je Game Of Life. To je dvourozměrný automat nad binární abecedou používající 8-okolí, který se řídí pravidlem, že živá buňka přežívá, pokud má 2 až 3 živé sousedy (jinak umírá), zatímco mrtvá buňka obžije, pokud má právě 3 živé sousedy. Game Of Life se stal zdrojem mnoha různých hříček. Nicméně pro účely šifrování se nezdá být moc užitečný, protože se chová příliš „pravidelně“ a po mnoha provedených krocích mívá velké oblasti bez jakékoliv živé buňky.

Lepší jsou pro nás CA, která se chovají více „chaoticky“. Mezi totalistickými 2D automaty je to například automat jménem Amoeba Universe (Wojtowicz, 2001), v němž živá buňka přežívá, má-li 1, 3, 5 či 8 živých sousedů a mrtvá buňka obžije, pokud má 3, 5, či 7 živých sousedů. Na sledování člověkem vývoj tohoto automatu není nic zajímavého (na rozdíl od Game Of Life), ale statistické vlastnosti se ukázaly být slušné.

Poslední dvourozměrný automat, který lze v programu získat přímo pod určitým jménem, je Replicator Universe. Ten má veselé pravidlo, které zapomíná původní stav buňky a její nový stav je 1, pokud má 1, 3, 5 či 7 živých sousedů, viz (Schiff, 2008), strana 116. V této práci je uveden jen pro zajímavost. Jakákoliv počáteční konfigurace je totiž po určitém počtu kroků zkopírovaná do všech 8 směrů od počátku (jako třeba ta, co je na obrázku 1.6). A proč se tak vůbec chová? Jeho přechodová funkce je vlastně XOR sousedů a je to tak lineární zobrazení nad $\mathbb{Z}_2$. Proto i po jakémkoliv počtu kroků tento CA stále definuje lineární zobrazení mezi počátečním stavem a novým stavem. Díky linearitě můžeme prozkoumat zvlášť, jaké všechny buňky budou ovlivněny jednou živou buňkou.



Obrázek 1.6: Replicator Universe po 8 krocích

Výsledek pak vznikne XORováním těchto výsledků pro všechny živé buňky na vstupu (jako kdyby se každá buňka vstupu vyvíjela zcela samostatně). Při hraní s jedinou živou buňkou na vstupu je už snadnější nahlédnout, proč vždy po provedení 2^n kroků je živá buňka nakopírována do všech 8 stran s velkými mezerami od jiné (všechny ve vzdálenosti n od počátečního místa, ať už „rovně“ či „šikmo“).

1.5 Implementace celulárních automatů

V teorii se typicky uvažují celulární automaty na nekonečném hřišti. V počátečním stavu se však všechny nenulové hodnoty vyskytují jen uvnitř nějaké konečné oblasti buněk. Nenulové hodnoty se pak ale mohou neomezeně rozrůstat do všech stran. Rychlost tohoto rozšiřování lze zhora omezit na základě velikosti okolí aplikovaného pravidla.

V praxi implementujeme celý automat na konečném hřišti. Možnosti jsou dvě. Buď celý automat zacyklíme a jeho vývoj v čase „pokresluje nekonečnou válcovou plochu“, nebo automat na stranách ohraničíme a při aplikaci pravidel na okraji hřiště čteme nulové hodnoty za jeho okrajem.

Výhodou je snadná implementace a nízké paměťové nároky. Nevýhodou je, že se vývoj celého automatu periodicky opakuje, pokud provedeme dostatečný počet kroků. Délku této periody lze bohužel odhadnout pouze shora.

2. Šifrování

2.1 Základní pojmy

Úkolem šifrování je uchovat a předat tajnou zprávu tak, aby ji mohl přečíst ten, pro koho je určena, ale už nikdo jiný. Dále se někdy za cíl dává ověřitelnost autora zprávy. Věda zabývající se šifrováním se nazývá *kryptografie*. Lámáním šifer se zase zabývá *kryptoanalýza*.

Původní čitelná zpráva se nazývá *plaintext*. Data po zašifrování se nazývají *ciphertext*. Pro převod plaintextu na ciphertext je potřeba použít šifrovací algoritmus. Ten by měl zároveň být schopen převézt ciphertext zpět na plaintext (tzv. dešifrování). Protože fungování šifrovacího algoritmu se velmi snadno vyzradí, zásadní roli hraje *šifrovací klíč*. Pokud se jedná o *symetrickou kryptografii*, tak stejný klíč slouží i k dešifrování. V případě *asymetrické kryptografie* se používají dva různé klíče. Šifrovací resp. dešifrovací klíče v asymetrické kryptografii se s ohledem na jejich použití nazývají jako *veřejný* resp. *tajný* klíč.

2.2 Základní východiska

Před zašifrováním zprávy se často provádí její *komprese*. To má za následek nejen úsporu přenosového pásma a množství práce (času) šifrovacího algoritmu, ale velkou výhodou je dosažení výrazně rovnoměrnějšího pravděpodobnostního rozdělení na prostoru plaintextů, což výrazně komplikuje kryptoanalýzu.

Jedinou zcela neprolomitelnou šifrovací metodou je Vernamova šifra. V binární podobě tato šifra vypadá tak, že se použije one-time pad, což je sekvence náhodných bitů, kterou vlastní obě strany. Odesílatel provede operaci plaintext XOR one-time pad a příjemce provede operaci ciphertext XOR one-time pad, čímž dostane zpět plaintext. Problém je, že one-time pad musí být stejně dlouhý jako plaintext a nelze ho použít opakovaně. Kvůli tomu se Vernamova šifra používá jen pro kritické aplikace, před kterými se mohou komunikující strany fyzicky setkat a předat si one-time pad.

2.3 O co nám jde

V této práci se budeme věnovat vytvoření algoritmu na protahování klíčů (anglicky key stretching). Cílem je z krátkého klíče (který lze například v krátkém čase přenést pomocí RSA) vygenerovat dlouhý klíč (one-time pad, kterým lze přeXORovat celý soubor). Je to tedy podobný úkol jako naprogramovat *Keystream Generator*, akorát my budeme dopředu vědět cílenou délku výsledného klíče.

2.4 Pseudonáhodné generátory

Jako velmi jednoduchý Keystream Generator by se dal použít nějaký z generátorů pseudonáhodných čísel. Narozdíl od „opravdových“ generátorů náhodných čísel, které

používají nějaký fyzikální zdroj náhody (třeba radioaktivní rozpad), jsou hodnoty vytvářené generátorem pseudonáhodných čísel deterministicky určeny počáteční hodnotou (tzv. seed), která může být zvolena na základě krátkého klíče.

Typickým příkladem může být lineární kongruenční generátor pseudonáhodných čísel (LCG - Linear Congruential Generator). Ten je charakterizován rozsahem hodnot m, koeficientem a, inkrementem c a počáteční hodnotou X_0. LCG provádí krok podle vzorce:

$$X_{n+1} = (aX_n + c) \mod m$$

Pro to, aby byl generátor pseudonáhodných čísel kvalitní, je potřeba, aby výstupy neměly zjevné lineární vztahy mezi sebou a měly dlouhou periodu. Podmínky pro to, aby LCG měl periodu délky m (tj. vystřídal všechny možné hodnoty), jsou následující (viz (Knuth, 1981)):

- c a m jsou nesoudělná čísla
- $a - 1$ je dělitelné všemi prvočiniteli m
- pokud je m dělitelné 4, je i $a - 1$ dělitelné 4

Třetí pravidlo je velice důležité, protože s ohledem na výkon programu na reálných procesorech se obvykle volí m v hodnotě mocniny dvou[1].

Jako pseudonáhodnou posloupnost není dobré použít přímo hodnoty X_i. Problém je zejména vidět tehdy, když $m = 2^n$. Pak se lze podívat na posledních d bitů čísel X_i, označme je $Y_i = X_i \mod 2^d$. Ty tvoří posloupnost s kratší periodou. Její předpis lze napsat stejným způsobem, jako předpis pro LCG:

$$Y_{n+1} = (qY_n + r) \mod 2^d$$

, kde $q = a \mod 2^d$, $r = c \mod 2^d$.

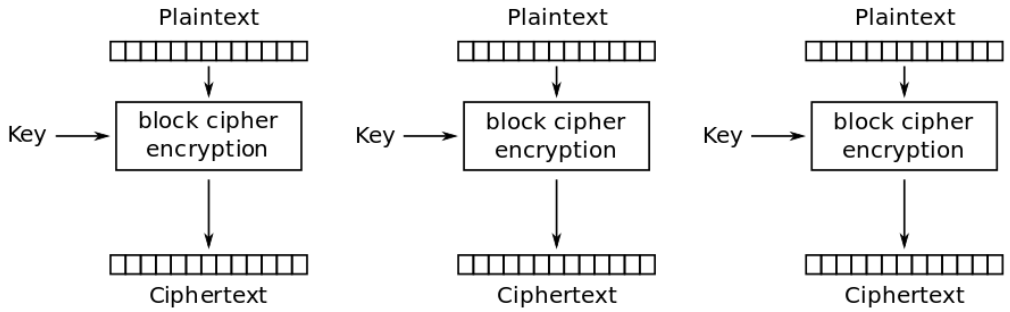
Řešením je použití jen několika počátečních bitů z každého X_i. Pokud by se na výstup posílala celá hodnota X_i, byla by část bitů snadno předvídatelná. Například každý m-tý bit výstupu by pravidelně střídal 0 a 1.

Existují i jiné generátory pseudonáhodných čísel, které jsou vhodnější pro kryptografické účely. Ve skutečnosti není skoro žádný rozdíl mezi Keystream Generator a genetárorem pseudonáhodných čísel. Odlišuje je jen to, že generátory pseudonáhodných čísel používají seed menší velikosti, kdežto Keystream Generator lze inicializovat klíčem větší velikosti a taky udržuje „složitejší“ vnitřní stav.

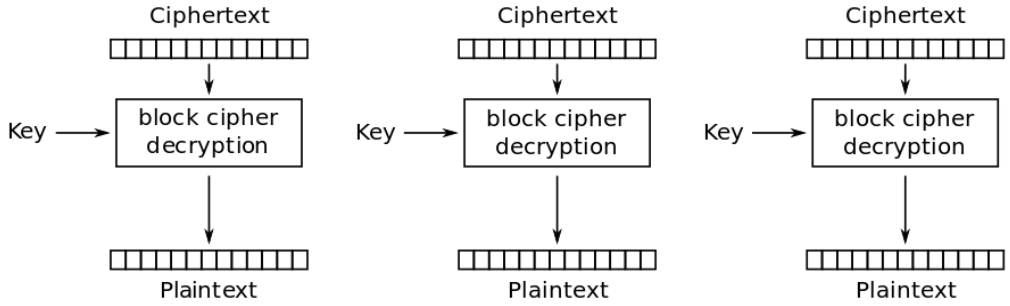
2.5 Blokové šifry

Blokové šifry jsou klasickým způsobem symetrické kryptografie, které se dnes používají ve většině kryptosystémů. Společným znakem blokových šifer je, že s využitím šifrovacího klíče transformují vstup pevné velikosti na výstup stejné velikosti (jde o bloky bitů, jejichž velikost je určena již při návrhu šifry). Aby bylo možné ze ciphertextu opět

[1] Běžně se používá nastavení, kde rozsah je mocnina dvou ($m = 2^n$, kde n je číslo od 24 do 64), $c = 1$, a libovolné takové, že $1 \equiv a \mod 4$



Obrázek 2.1: ECB encryption



Obrázek 2.2: ECB decryption

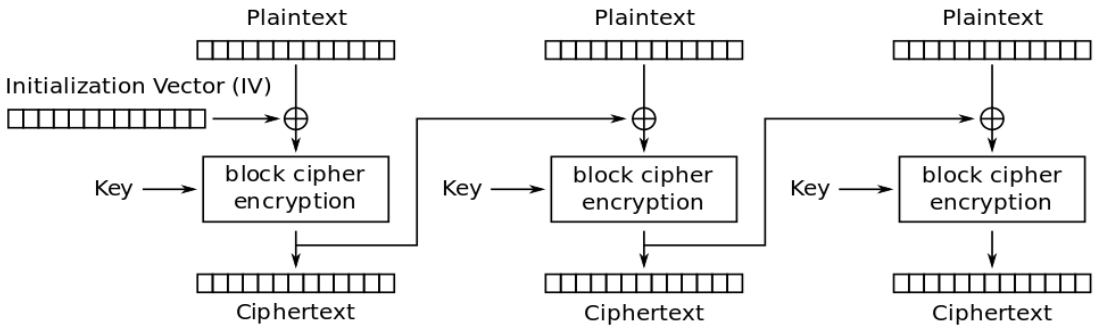
získat plaintext (bloková šifra definuje, jak se stejným klíčem provést obě zobrazení), musí být pro každou hodnotu klíče toto zobrazení bijekce mezi vstupy a výstupy. Plaintext se rozdělí na bloky dané velikosti, každý z nich se zašifruje a bloky ciphertextu se zřetězí za sebou (bloky ciphertextu jsou opět stejně dlouhé, tudíž není potřeba jejich explicitní oddělování). Velikost bloku může být například 128 bitů. Klíč může být větší i menší.

2.5.1 Operační mody

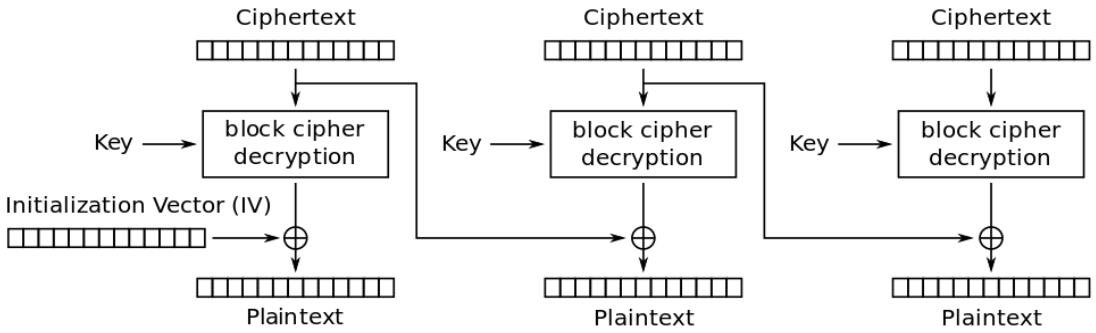
Existuje několik operačních modu blokových šifer. Ty popisují, jakým způsobem je bloková šifra zapojená při šifrování celé zprávy.

Nejjednodušším modem je ECB (Electronic CodeBook). Ten šifruje každý blok plaintextu přímou aplikací blokové šifry bez jakékoliv návaznosti na ostatní bloky (zašifrování je na obrázku 2.1, rozšifrování je na obrázku 2.2). Bezpečnost tohoto modu je dost slabá, protože kdykoliv obsahuje plaintext dvojici stejných bloků, jsou jim odpovídající části ciphertextu také shodné (takže po zašifrování je například možné lidským okem rozpoznávat tvary v bitmapových obrázcích, které obsahují jednobarevné plochy).

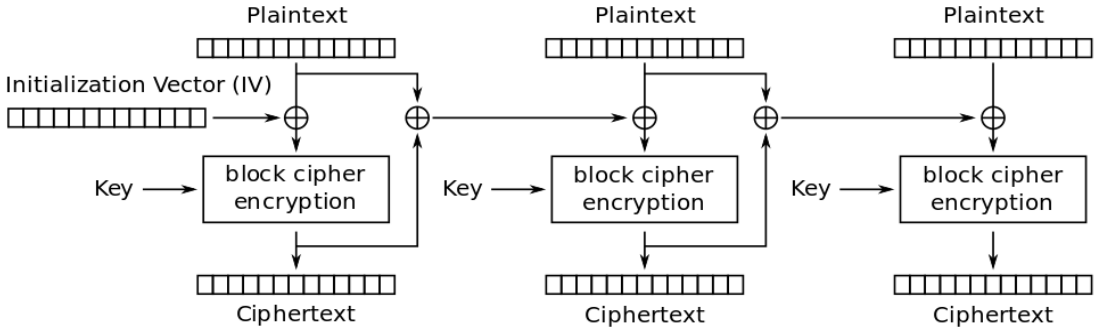
Lepším modem je CBC (Cipher Block Chaining). Zde je každý blok plaintextu před zašifrováním XORován s předchozím blokem ciphertextu (zašifrování je na obrázku 2.3, rozšifrování je na obrázku 2.4). První plaintext je XORován s inicalizačním vektorem. Při ponechání stejného plaintextu a stejného klíče se změnou inicializačního vektoru mění celý ciphertext, což útočníkovi komplikuje rozpoznání, kdy jsou odeslány 2 stejné zprávy. CBC



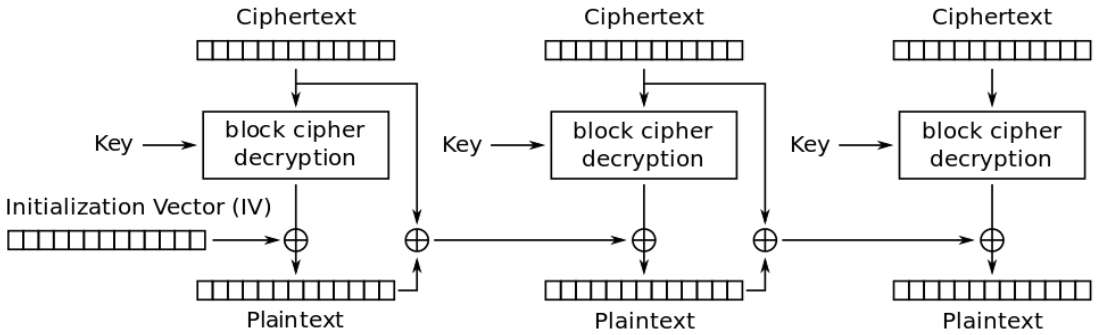
Obrázek 2.3: CBC encryption



Obrázek 2.4: CBC decryption



Obrázek 2.5: PCBC encryption



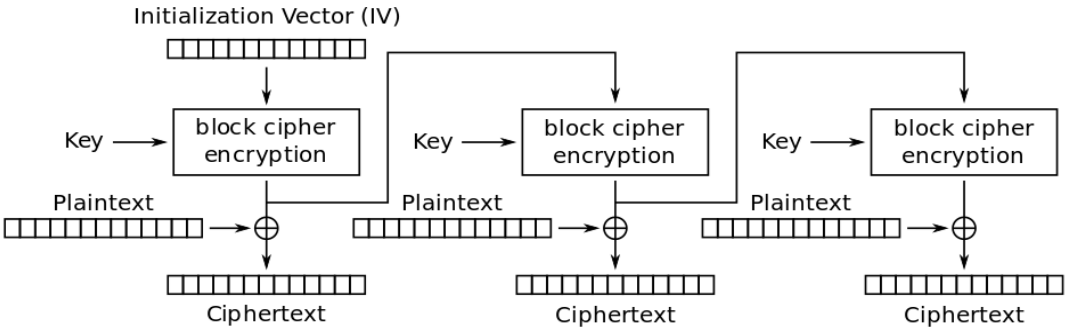
Obrázek 2.6: PCBC decryption

má výrazně lepší vlastnosti než ECB, ale to neznamená, že bychom mohli použit libovolnou blokovou šifru. Kdybychom jako blokovou šifru zvolili pouhý XOR mezi plaintextem a šifrovacím klíčem, tak by při CBC zapojení došlo k tomu, že každý druhý blok zprávy by vůbec nebyl ovlivněn klíčem (byl by to jen plaintext XOR inicializační vektor). Ale při využití vhodné blokové šifry (mající vysokou difuzi a konfuzi) jsou výsledky velmi dobré. Nevýhodou této metody je, že se zašifrování nedá paralelizovat (ale rozšifrování lze).

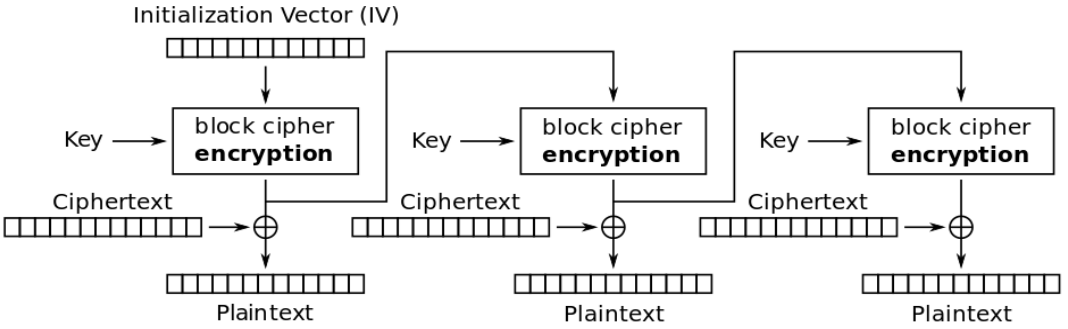
Podobným modem je PCBC (Propagating Cipher Block Chaining). Zde se do výpočtu následující bloku neposílá jen ciphertext, ale i plaintext (zašifrování je na obrázku 2.5, rozšifrování je na obrázku 2.6). Nejsou mi známy žádné výhody této metody ve srovnání s CBC.

A teď se dostáváme ke dvěma metodám, které umožňují použít blokovou šifru jako Keystream Generator, protože se XOR s plaintextem aplikuje zvlášť a je možné ho provést až po všech vyvolání blokové šifry.

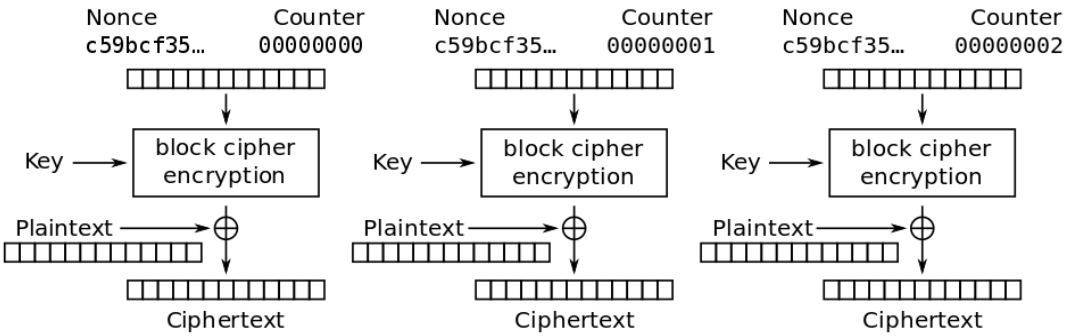
První z nich je OFB (Output Feedback). OFB nejprve použije blokovou šifru k tomu, aby zašifrovala inicializační vektor pomocí klíče. Výsledek je jednak poslán ven a dále je poslán na vstup blokové šifry. Tento blok je zase zašifrován pomocí stejného klíče a výsledek je poslán opět jak na výstup, tak do další blokové šifry. Postup si lze prohlédnout na obrázku 2.7. Dá se říci, že OFB posílá na výstup Keystream, který se skládá z řady $f(x), f(f(x)), f(f(f(x))), f(f(f(f(x)))), f(f(f(f(f(x))))), ...$, kde x je inicializační vektor a funkce f je již zobrazení s konkrétním klíčem. Rozšifrování je triviální obdobou



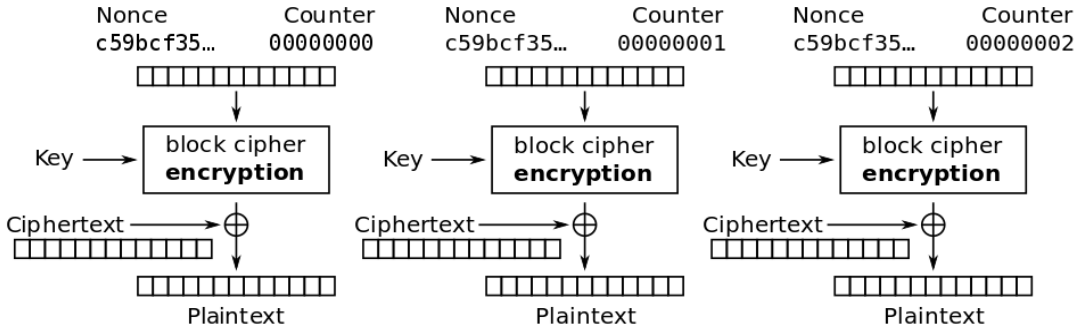
Obrázek 2.7: OFB encryption



Obrázek 2.8: OFB decryption



Obrázek 2.9: CTR encryption



Obrázek 2.10: CTR decryption

tohoto postupu. Není k němu potřeba žádné inverzní zobrazení. Výstup blokové šifry se XORuje s ciphertextem a vznikne plaintext (viz obrázek 2.8). Je zřejmé, že zašifrování ani rozšifrování nelze při OFB paralelizovat.

Jako poslední si ukážeme mod CTR (Counter). Na vstup blokové šifry jdou data, která se pokaždé o 1 změní. Výstup blokové šifry (keystream) je pak XORován s plaintextem resp. ciphertextem (viz obrázky 2.9 a 2.10). CTR se může podobat ECB v tom ohledu, že lze každý blok plaintextu je šifrován samostatně a proto je opět možné paralelizovat jak zašifrování, tak rozšifrování. Nicméně zde jsou jednotlivé bloky plaintextu zašifrovány jinak, i kdyby byly všechny bloky plaintextu stejné. Bezpečností aplikace se proto CTR blíží spíše OFB. Counter posílá na výstup keystream, který se skládá poustupně z řady $f(x \oplus n), f(x \oplus (n+1)), f(x \oplus (n+2)), ...$, kde n je „nonce“. To je číslo zvolené pro šifrování jedné konkrétní zprávy, které musí být přiloženo k ciphertextu pro umožnění rozšifrování zprávy. Cílem je neopakovat znovu stejný keystream při ponechání stejného klíče (hraje tedy stejnou roli jako inicializační vektor u předchozích modů).

2.6 Útoky na šifry

Kvalita šifrování musí být posuzována v kontextu možných útoků. Pokaždé budeme předpokládat, že útočník nezná tajný klíč (jinak by dokázal vše), ale že důkladně rozumí šifrovacímu algoritmu (viz Kerkhoffův princip).

2.6.1 Rozdělení podle povolených prostředků

V knize (Delfs a Knebl, 2002) jsou útoky na šifry rozděleny následujícím způsobem:

Nejslabším typem útoku je *ciphertext-only attack*. Při něm má útočník možnost vidět pouze ciphertexty. Pokud nějaký šifrovací algoritmus snad dává možnost rozluštit plaintext pouze při znalosti cipthertextu, je to zcela nepoužitelný algoritmus.

O mnoho silnější útok se nazývá *known-plaintext attack*. Při něm má útočník k dispozici dvojice ciphertext-plaintext. Na základě nasbíraných informací se snaží rozluštit jiný ciphertext. Musíme počítat, že dvojici ciphertext-plaintext může v praxi útočník poměrně snadno zístat. Například může útočník uhodnout, jaká zpráva je odesílána, protože je to zjevné z délky zprávy a širšího kontextu. Nebo může odesílatel zbytečně použít šifrovací mechanismus i v případě přenosu zprávy, která ve skutečnosti není tajná. Útočník v danou chvíli může třeba vidět odesílateli danou zprávu přes rameno a hned získá dvojici ciphertext-plaintext. A nakonec občas i znalosti části plaintextu můžu být prostředkem k útoku. Známou částí plaintextu může být například hlavička nějakého standardního souboru, padding, nebo automaticky vygenerovaný podpis na konci emailu.

Ještě silnějším útokem je *chosen-plaintext attack*. Při něm si může útočník sám určit, jaká zpráva má být zašifrována a obdržet k ní odpovídající ciphertext. Jeho cílem je zase buď přímo odvodit šifrovací klíč, nebo jiným způsobem získat možnost rozluštit ostatní ciphertexty.

Posledním zesílením prostředků k útoku je *adaptivaly-chosen-plaintext attack*. Při něm může útočník nejen analyzovat dvojice plaintext-ciphertext, ale na základě jejich znalosti volit nové zprávy k zašifrování a získávat další dvojice plaintext-ciphertext, dokud nebude mít dostatek informací.

2.6.2 Příklady útoků

Ukážeme si několik jednodušších příkladů kryptoanalytických útoků.

2.7 Testy

Je příliš obtížné ukázat o šifrovacím algoritmu, že je doopravdy kvalitní. Jako záruka kvality se proto v praxi používá spíše jeho zveřejnění na několik let, aby ho měli šanci oponovat nejlepší odborníci. Pokud se ani po několika letech neukáže jeho slabina, je šifrovací algoritmus považován za dost dobrý. Naštěstí alespoň ty velmi špatné šifrovací algoritmy je možné rychle rozpoznat statistickými testy. Na to se zaměříme v této práci.

2.7.1 Testy na úrovni jednotlivých výstupů

Pro kryptografii je potřeba, aby dlouhé klíče měly vlastnosti pseudonáhodných posloupností. Pochopitelně zde není možné dosáhnout, aby všechny dlouhé klíče byly stejně pravděpodobné a dosáhli bychom tak „pravé náhodnosti“, ale můžeme alespoň otestovat konkrétní výstup, jestli se podobá náhodné posloupnosti.

Třída Crypto.RandomnessTesting obsahuje následující metody.

- EntropyTest(BitArray b, byte lengthLimit) : Testuje entropii bloků o velikosti od 1 po lengthLimit. Jde o sledování frekvence jednotlivých bloků. V náhodné posloupnosti by měly být všechny bloky stejné délky přibližně stejně časté. Vzorec pro výpočet entropie bloků jedné konkrétní délky, kde p_i označuje pravděpodobnostii (zde relativní četnosti) výskytu jednotlivých bloků, zní:

$$\sum_{i=1}^{n} p_i \cdot \log_2 \frac{1}{p_i} = -\sum_{i=1}^{n} p_i \cdot \log_2 p_i$$

  Pro krátké posloupnosti (zde pod 10 tisíc bitů) samozřejmě není možné, aby se všechny bloky (zde délky 10) vyskytly, takže jsou všechny výsledky porovnány s maximálním možným výsledkem. Výstupem je vážený průměr, kde mají testy entropií všech délek výsledky v rozsahu 0 až 1. Optimální hodnota je 1.

- CompressionTest(BitArray b) : Zkusí data zkomprimovat pomocí programu gzip s optimální úrovní komprese a vrátí poměr mezi novou a původní velikostí. O posloupnostech, které lze zkomprimovat, je známo, že nejsou dokonale náhodné. Konkrétně velikost dat po kompresi je horním odhadem na Kolmogorovskou složitost. Optimální hodnota je 1.

2.7.2 Testy na úrovni celého zobrazení

Skutečnost, že výstupem algoritmu je pseudonáhodná posloupnost čísel, je jistě dobrá. Ale co když algoritmus všem vstupům přiřadí stejnou pseudonáhodnou posloupnost? Nebo jeden konkrétní bit na vstupu neovlivní výsledek? Takovou nekvalitu musí objevit druhá skupina testů.

Budeme zde zkoumat vlastnosti algoritmů jako vlastnosti celého zobrazení z krátkých klíčů do dlouhých klíčů. Protože si budeme často klást otázky typu „Jak moc se liší výstup A od výstupu B?“, tak by bylo vhodné zavést nějaké hodnocení, ideálně s vlastností metriky. Porovnávat budeme vždy jen výstupy stejné délky. Oblíbenými metrikami pro řetězce jsou *Hammingova vzdálenost* a *Levenshteinova vzdálenost*. Hamming měří počet pozic, na kterých se řetězce liší. Levenshtein měří minimální počet potřebných změn k tomu, abychom z jednoho řetězce dostali ten druhý. Jako změnu je možné provést záměnu znaku, smazání znaku, nebo dopsání znaku na libovolné místo.

Hammingova vzdálenost je triviálně horním odhadem na Levenshteinovu vzdálenost. V případě náhodných binárních řetězců je jejich hodnota občas stejná, ale někdy může být Levenshtein výrazně nižší. Například když zrotujeme řetězec o jednu pozici, tak Levenshtein dává vzdálenost 2, zatímco Hamming může dát hodně vysoké číslo (až délka řetězce). Významná pro nás bude rychlost výpočtu. Hammingovu vzdálenost lze triviálně určit v lineárním čase, ale výpočet Levenshteinovy vzdálenosti zabere čas kvadratický. Že to rychleji nejde, se nelze divit, protože Levenshtein vlastně spouští prohledávání prostoru editací. Díky dynamickému programování to lze provést alespoň v tom kvadratickém čase.

Program ještě obě vzdálenosti vydělí délkou řetězce, aby výsledky měly hodnotu v rozmezí 0 (shodné řetězce) až 1 (například řetězec samých nul porovnán se řetězcem samých jedniček). Střední hodnota pro dvojici náhodných binárních posloupností je u Hamminga triviálně 0,5. Nesrovnatelně těžší je odhadnout střední hodnotu Levenshteinovy vzdálenosti. Posloupnost středních hodnot Levenshteina se vzrůstající délkou vstupu roste jako

subaditivní posloupnost [2]. Relativní hodnota proto může pro delší vstupy pouze klesat. Limitní hodnotu se zdá být příliš těžké odhadnout, ale orientační experimenty i diskuze na internetu naznačují, že můžeme počítat s hodnotou kolem 0,29 (Beenakker, 2013).

Třída Crypto.FunctionTesting obsahuje následující metody. Podle nastavení v konstruktoru mohou všechny testy používat buď Hammingovu, nebo Levensteinovu vzdálenost. Dále uváděno podle Hamminga.

- TestAverageDistance(IKeyExtender algorithm, int ratio) : Testuje průměrnou vzdálenost výstupů příslušející dvěma různým náhodně zvoleným vstupům. Optimální hodnota je 0,5.

- TestBitChange(IKeyExtender algorithm, int ratio) : Testuje, jak velká část bitů výstupu se změní při změně jednoho bitu vstupu. Metoda sampluje náhodné vstupy a pro každý z nich zkouší změnit zvlášť všechny bity. Optimální hodnota je 0,5.

  TestBitChange je z praktického hlediska přísnější než TestAverageDistance. Přestože pro naše algoritmy lze TestAverageDistance chápat jako horní odhad výsledku TestBitChange, lze teoreticky vymyslet i zobrazení, které v TestBitChange dopadne dobře, ale v TestAverageDistance selže. Například když všem vstupům se sudou paritou přiřadíme výsledek ze samých nul (000...0) a všechny vstupy s lichou paritou zobrazíme na pravidelné střídání hodnot nula a jedna (010101...01), tak TestBitChange dosáhne optimálního výsledku 0,5, zatímco TestAverageDistance vykáže velmi podezřelou hodnotu 0,25.

- TestLargestBallExactly(IKeyExtender algorithm) : Testuje, jaká největší koule se dá vměstnat do prostoru výstupů tak, aby neobsahovala žádný vygenerovatelný dlouhý klíč. Zkouší úplně všechny vstupy na malém prostoru a ty natahuje na dvojnásobek. Motivací je, že pokud zobrazení nazaplňuje prostor dostatečně rovnoměrně, pak to rozpoznáme tak, že se do prostoru vejde velká koule.

- TestLargestBallApprox(IKeyExtender algorithm) : Testuje to samé, ale používá delší vstupy, které už nezvládá vyzkoušet všechny, takže je sampluje náhodně.

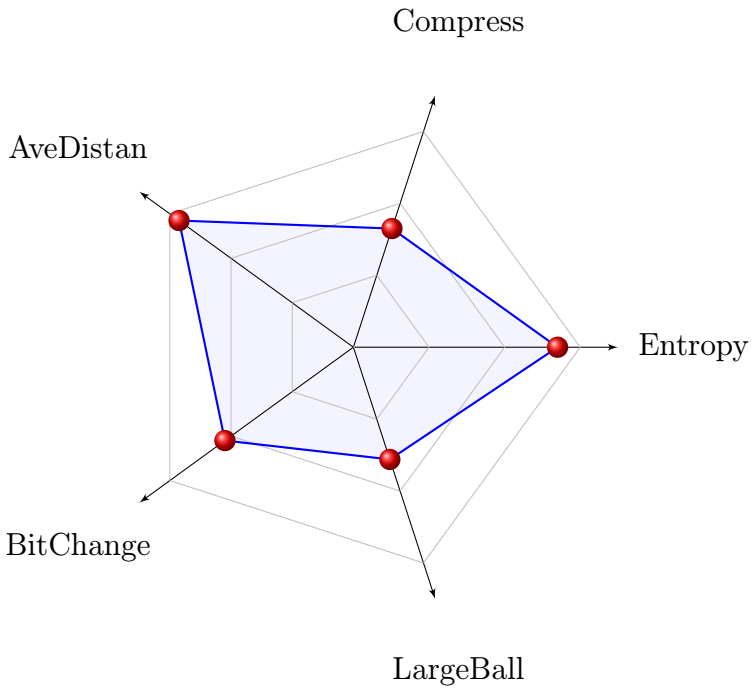
2.7.3 Použití testů

Všechny doposud zmíněné testy jsou obaleny ve třídě Crypto.FunctionTestsForThesis, kde jsou výsledky těchto testů transformovány způsobem, který zaručí, že vyšší výsledek je lepší. S výjimkou testů maximálních koulí, které jsou realizovány v této třídě trochu jinak, platí, že nejhorší hodnota je 0 a nejlepší hodnota je 1.

2.7.4 Grafické znázornění

Výsledky jednotlivých algoritmů ve výše uvedených testech budeme znázorňovat na diagramech jako je ten na obrázku 2.11.

[2]Posloupnost x_i se nazývá subaditivní, pokud $\forall n,m \in \mathbb{N} : x_{n+m} \leq x_n + x_m$



Obrázek 2.11: Ukázkový radar chart

Pro diagramy jsou hodnoty přeškálovány. S výjimkou testů největších koulí, kde neznáme optimální hodnotu, je škálování takové, aby optimální hodnota byla 1. Tedy například když TestBitChange vrátí hodnotu x, pak je do diagramu zobrazeno

$$\min\{2x, 2(1-x)\}$$

, aby optimum (zanesené jako hodnota 1) byl výsledek 0,5 a odchylky na obě strany „stejně vážné“.

3. Nástin architektury

Tato kapitola je určitým doplňkem k vývojové dokumentaci, která byla vytvořena z dokumentačních komentářů. Tato kapitola se nesnaží nahradit čtení vývojové dokumentace ani přečtení ostatních kapitol této práce, ve kterých jsou podrobněji vysvětleny klíčové části programu.

3.1 Úvod k návrhu

Celý zdrojový kód se nachází v repozitáři na adrese:
https://madv.visualstudio.com/DefaultCollection/_git/Cellular%20Cryptography#_a=contents

V rámci jedné Solution ve Visual Studiu byly vytvořeny tři projekty. Projekt se jménem MartinDvorak, který měl původně být jedinou částí aplikace a měl jasně identifikovat tuto práci při jejím elektronickém odevzdávání, obsahuje veškerou logiku popisovanou v textu práci. Pomocí tohoto projektu byly prováděny veškeré experimenty. Obsahuje 3 jmenné prostory: Cellular, Crypto a Testing.

Jako druhý vznikl malý projekt se jménem Program. Ten využívá nástroje vytvořené v prvním projektu a kompiluje se na WinForms aplikaci, kterou mohou uživatelé použít k zašifrování svých souborů. Třetí je testovací projekt se jménem CryptographyUnitTests. Ten testuje, že šifrování funguje správně (rozšifrování je inverzním zobrazením).

Práce neobsahuje žádné závislosti na externích knihovnách. Ke spuštění stačí mít nainstalován .NET framework verze 4.5 (či vyšší), který je automatickou součástí operačního systému Windows 8 (či novější), ale může se nacházet i na starších verzích (já ho mám na Windows 7).

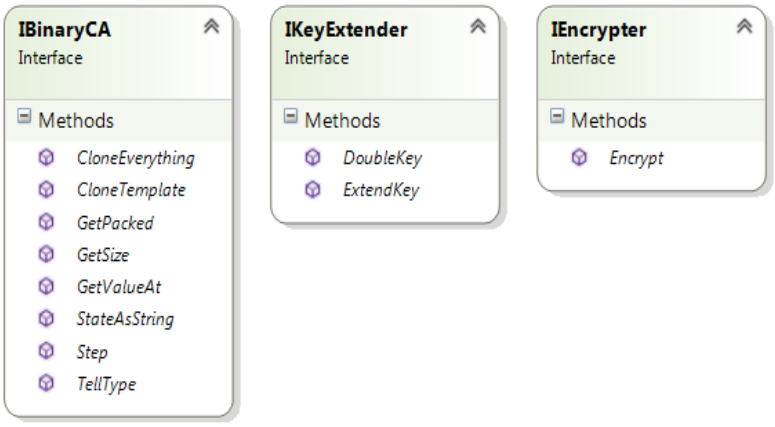
Při vývoji byl kladen důraz na znovupoužitelnost kódu. To bylo důležité, protože v průběhu programování ještě nebylo jasné, jakým způsobem se bude využívat který automat či algoritmus. Znovupoužitelnost je dosahována pomocí kvalitního objektového návrhu, většího množství variant konstruktorů a propojování objektů prostřednictvím rozhraní. Stěžejní trojice rozhraní je uvedena na obrázku 3.1 Dále byla věnována pozornost vysokému výkonu.

Důležitá je také přehlednost zdrojového kódu. Proto byly ke všem třídám a ke všem jejím věřejným metodám (a občas i těm ostatním) psány XML komentáře podle zvyklostí programátorů v C# a práce ve Visual Studiu. Z nich byla také vygenerována vývojářská dokumentace v HTML.

3.2 Cellular

3.2.1 Abstraktní třída CellularAutomaton

Vršek hierarchie všech celulárních automatů. Je tu jediná společná datová položka pro všechny druhy automatů – time. Je to diskrétní čas začínající na 0, který udává, kolik kroků výpočtu již proběhlo. Tento údaj oznamuje metoda GetTime. Dále je tu abstraktní metoda Step, která záleží na konkrétním typu automatu. Kromě zavolání Step bez parametru je možné zavolat Step s uvedením počtu, které tolikrát zavolá metodu Step.



Obrázek 3.1: Hlavní rozhraní v naší práci

Položku time musí updatovat metoda Step sama o sobě. Dále jsou tu abstraktní metody Clone a TellType.

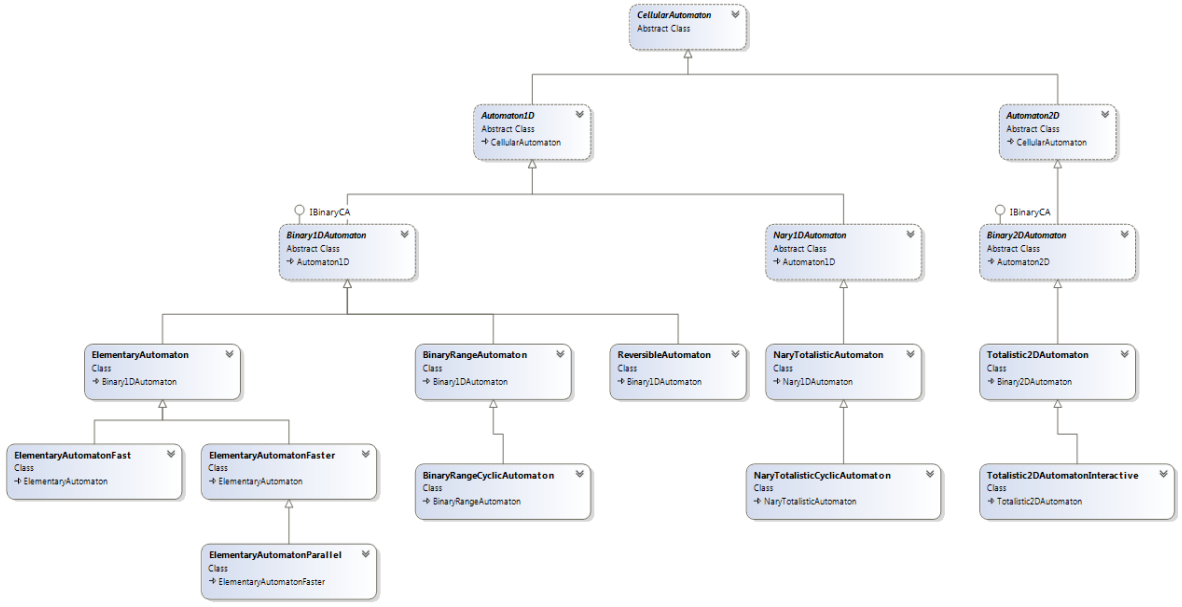
CellularAutomaton má dva přímé následníky: abstraktní třídy Automaton1D a Automaton2D (viz obrázek 3.2). Protože ještě nevíme nic o konkrétním typu automatu, není tu žádná nová položka kromě velikosti. Konkrétně, Automaton1D má jedinou položku size, Automaton2D má dvojici položek width, height. Viditelnost mají protected, tudíž k nim nikdo nemůžu přistupovat zvenčí přímo.

3.2.2 Koncept třídní hierarchie

Balíček Cellular je navržet tak, aby sloužil nejen ke konkrétnímu šifrovacímu algoritmu, ale aby mohl přijít jiný člověk a hrát si s celulárními automaty (ať už za účelem poznávání, nebo za účelem generování dat pro další aplikace). Proto byla při implementaci snaha, aby kromě výstupu pro počítač existovaly i „grafické výstupy“ pro debugging a sledování chování automatů. Pro snadnější práci je občas naimplementován i takový typ CA, který se speciálním případem nějakého obecnějšího typu CA. Příkladem může být imlementace elementární 256 automatů, které stačí zadat jejich číslem (podle Stephena Wolframa). Snaha o rozšiřitelnost se projevila různými „mezitřídami“ mezi úplně abstraktním automatem (který nic nespecifikuje) a konkrétní implementací konkrétního typu automatů.

Těžištěm práce jsou binární automaty, protože jsou jejich výstupy snadno zpracovatelné a protože je jejich chování dostatečně komplexní a rozmanité, aby dokázaly nasimulovat jakýkoliv druh automatu. Nicméně byly ve velmi omezené míře implementovány i jiné než binární automaty.

Abstraktní OOP přístup přirozeně vede k „diamond inheritance problem“ například mezi čtveřicí abstraktních tříd: úplně obecný CA, obecný jednorozměrný CA, jakýkoliv binární CA a jednorozměrný binární CA. Protože C# nepodporuje vícenásobnou dědičnost, bylo vytvořeno rozhraní IBinaryCA, které implementují všechny binární celulární automaty. Takový počin má jisté výhody i nevýhody. Výhodou je, že IBinaryCA je už ze sémantiky rozhraním. I kdyby byl abstraktní třídou, nemá vlastní datové položky ani vlastní implementaci metod, takže je použití interfacu vhodné. Nevýhodou interfacu



Obrázek 3.2: Diagram hierarchie celulárních automatů

je to, že není možné vynutit, aby rozhraní IBinaryCA implementovaly pouze podtřídy CellularAutomaton.

3.2.3 Ukládání stavu celulárního automatu

Původní implementace jednorozměrných binárních CA ukládala stav jako bool[ ] a pro dvojrozměrné byl datovým typem bool[,]. Již brzy bylo jasné, že to byla dost špatná volba a na výběr byly dvě lepší alternativy:

- BitArray – buňka zabírá jen jeden bit, zatímco při použití System.Boolean buňka zabírala celý byte. Přístup k datům je stejný – přes index. Akorát u dvojrozměrných automatů je použit typ BitArray[ ], ke kterému se přistupuje jako state[i][j] místo původního state[i,j] a je potřeba zavolat větší množství konstruktorů.
- byte[ ] obsahující jen hodnoty 0 a 1 - stejná velikost jako při použití bool[ ], ale není potřeba používat podmínky na hodnotu pro dosazení do aritmetických operací (zisk indexu do pole popisujícího přechodovou funkci či pouhé sčítání hodnot sousedních buněk u totalistických pravidel).

Nakonec byla zvolena první varianta, protože při použití CA na prodlužování šifrovacích klíčů může být velikost automatu v posledních krocích výpočtu opravdu velká. A nechceme pro šifrování souboru o velikosti 1 GB spotřebovat přes 8 GB paměti.

Pro N-ární automaty byl zvolen poněkud nehospodárný int[]. To může být do budoucna změněno, pokud se bude intenzivněji pracovat i s jinými než binárními automaty.

Datové položky ukládající stav byly navržené jako immutable. Takže například není problém dávat do konstruktorů pole s iniciálním stavem (předávané vždy referencí) bez kopírování. Kdykoliv se provádí krok CA, je vytvořen nový vnitřní stav a po proběhnutí výpočtu je nahrazen ukazatel na tento stav.

3.2.4 Optimalizace výpočtu elementárních automatů

Při výpočtu nového stavu ve třídě ElementaryAutomaton se musí vždy podle stavu 3 buněk rozhodnout stav 1 nové buňky (a nový stav se získává z trojrozměrného pole o velikost 2x2x2). To je neefektivní, protože počet čtení stavu je zde třikrát větší než počet vytvářených stavů. Kdykoliv se vyskytne ve stavu elementárního automatu stejný blok, nový stav těch buněk v dalším kroku, kromě buňky úplně vlevo a buňky úplně vpravo, je vždy stejný. Proto by se mohl uspořit čas výpočtu (hlavně snížením počtu podmíněných skoků při vykonávání), kdyby se celé bloky zobrazovaly na celé bloky podle nějaké předgenerované lookup tabulky. Tato optimalizace byla naimplementována v třídě ElementaryAutomatonFast, kde bylo zvoleno mapování vždy 10 buněk na nových 8 buněk. Na konci práce na ročníkovém projektu byla knihovna v tomto stavu: 256 elementární CA mohly simulovat tři různé třídy, přičemž ElementaryAutomatonFast to dělal nejrychleji, ElementaryAutomaton středně rychle a BinaryRangeAutomaton nejpomaleji.

Během programování bakalářské práce byla většina času věnována ostatním jmenným prostorům, ale v posledních měsících došlo i na optimalizaci výpočtu celulárních automatů. Protože BinaryRangeAutomaton prováděl k určení stavu každé nové buňky $1 + 2r$ čtení předchozích stavů, kde r je počet používaných buněk na každou stranu, byla zvolena jiná implementace. Nyní jsou právě přečtené buňky uloženy v číselné proměnné (kterou lze indexovat do pole popisujícího pravidlo) a při přesunu na novou buňku je číselná hodnota pouze zdvojnásobena (s oříznutím na správnou velikost, aby se zahodila buňka úplně vlevo) a přičten stav buňky úplně vpravo. Nová implementace BinaryRangeAutomaton byla natolik rychlá, že dokonce porazila ElementaryAutomatonFast při simulaci elementárních automatů. Tím jsme se poučili pro vytvoření ještě lepší implementace elementárních CA, která také čte každou buňku jen jednou, průběžný stav uchovává v číselné proměnné a indexuje do jednorozměrného pole popisujícího pravidlo (rychlost nové implementace BinaryRangeAutomaton totiž ukázala, že je to rychlejší než lookup tabulka). Tato implementace by měla kombinovat výhodu nové implementace BinaryRangeAutomaton s výhodou specifické implementace na předem známou velikost pravidla.

Byla tak vytvořena třída ElementaryAutomatonFaster a pak ještě její paralelní implementace ve třídě ElementaryAutomatonParallel. Při měření výkonu jsme zjistili, že ve srovnání s výpočtem pomocí třídy ElementaryAutomaton dosáhla implementace ElementaryAutomatonFast 86% času, ElementaryAutomatonFaster dosáhl pouhých 48% času, ElementaryAutomatonParallel to měl za 75% času s BinaryRangeAutomaton za 51% času. Neefektivita paralelní implementace byla zklamáním. Pro malé velikosti automatu se jistě nevyplatí kvůli režii spouštění vláken a pro velké velikosti automatu je pravděpodobně problém s paralelním čtením velkého množství dat z paměti (a současným zápisem).

3.3 Crypto

Tento namespace jednak obsahuje natahovače klíčů (popsané v následující kapitole) a potom testy (popsané v předchozí kapitole). Také se tu nachází statická třída Factory, která nahrazuje možnost deserializace CA a větších objektů, které je obsahují.

Natahovače klíčů implementují rozhraní IKeyExtender. Třída EncrypterStreamCA umí použít libovolnou implementaci IKeyExtender k šifrování pole bitů. Nad tou je ještě

vybudovaná abstrakce v podobě rozhraní IEncrypter, které obecně poskytuje přístup k libovolnému symetrického kryptografickému algoritmu. Rozhraní IEncrypter ještě implementuje třída EncrypterReversibleCA, která realizuje šifrování alternativním způsobem (viz poslední kapitola). Nad implementacemi rozhraní IEncrypter je ještě vybudována společná obalová vrstva v podobě třídy EncryptionProvider. Ta umožňuje šifrovat přímo Streamy dat a zároveň umožňuje kromě šifrovacího klíče použít kombinaci uživatelského hesla a náhodného saltu, což bude použito v praktické aplikaci.

Statická třída Export poskytuje přístup k věcem, co mají být použitelné i z jiné assembly a provádí veškerou instanciaci interních tříd, které již není možné ovládat přímo z vnějšku. Například poskytuje jednu konkrétní implementaci IKeyExtender, do níž je vložen elementární automat číslo 30 (ve zrychlené variantě). Z vnějšku není možné vynutit jinou variantu. Dále poskytuje přístup ke třídě EncryptionProvider (jsou tu dvě varianty pro dva různé vložené šifrovací algoritmy).

3.4 Testing

Nejedná se o žádné Unit testy, ale o poměrně chaotickou hromadu metod, které umožňují zkoušet různé části programu. Lze si hrát s vizualizací nějakých automatů.

3.5 Program

Okenní aplikace, umožňuje šifrovat a dešifrovat. V podstatě realizuje jen volbu vstupního a výstupního souboru. O zbytek se stará projekt MartinDvorak.

Největší možná velikost šifrovaného souboru je 128MB. Důvodem je, že pro větší velikost by již bylo potřeba použít BitArray o velikosti 256MB a protože se velikost zadává v počtu bitů a argument je typu int, bylo by potřeba vložit hodnotu 2^{31}, ale nejvyšší hodnota intu je $2^{31} - 1$. I kdyby nebylo nutné použít konstruktor se zadáním přesného počtu bitů (ale bral by třeba počet bytů), tak tím by int mohl sloužit ještě k indexaci bitů v poli o velikosti přesně 256MB, ale ne většího.

Limit by mohl být zvýšen, pokud bychom vytvořili vlastní implementaci BitArray, která by pro určení velikosti a indexaci používala typ uint, nebo ještě lépe long. Nicméně by to asi nebylo velmi užitečné rozšíření, protože už šifrování souboru o velikosti 256MB trvá několik minut.

3.6 CryptographyUnitTests

Testuje šifrovací algoritmy skrze veřejnou třídu EncryptionProvider poskytnutou ve dvou verzích třídou Export.

4. Protahování klíčů pomocí celulárních automatů

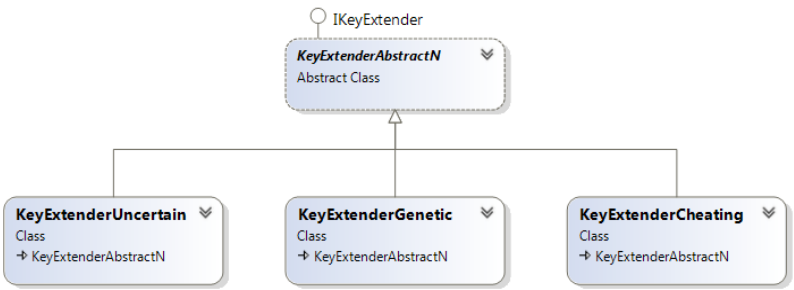
V rámci práce na ročníkovém projektu byla vytvořena řada různých celulárních automatů (potomci abstraktní třídy CellularAutomaton), viz výše. Většina z nich je binárních (tj. každá buňka může nabývat jen dvou různých stavů) a ty zároveň implementují rozhraní IBinaryCA, které vynucuje většinu pro nás užitečných metod.

V rámci bakalářské práce byly vytvořeny algoritmy na protahování klíčů, které binární celulární automaty využívají. Všechny tyto algoritmy implementují rozhraní IKeyExtender. Toto rozhraní obsahuje metodu DoubleKey pro vytvoření klíče s přesně dvojnásobnou délkou a metodu ExtendKey pro natažení klíče na libovolnou zadanou délku. Vstupy i výstupy musí být typu BitArray, což je pole logických hodnot, které však v 1 bytu ukládá 8 hodnot (narozdíl od bool[ ]).

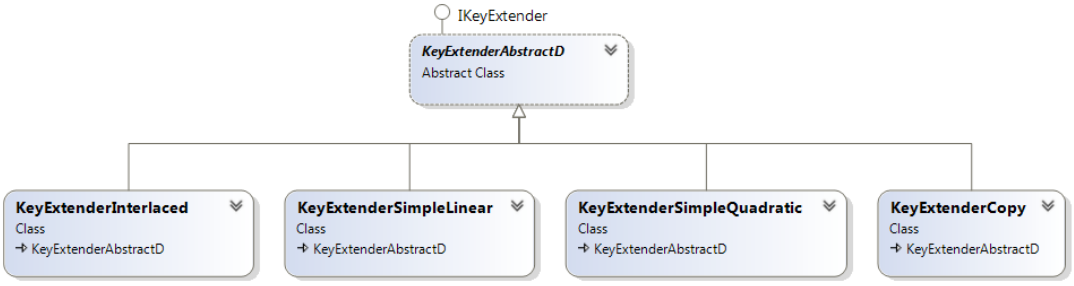
Při vytváření instancí tříd implementující IKeyExtender se skrze konstruktor vkládá dovnitř libovolná implementace IBinaryCA. To umožňuje zvolit si zvlášť druh automatu, který určuje fungování přechodové funkce, a zvlášť protahovací algoritmus, který určuje způsob čtení hodnot z automatu a jejich využití. Jedná se tedy o techniku „Inversion of control“. To se hodí, abychom mohli snadno zkoušet všechny možné způsoby protahování klíčů. Pokud je dovnitř algoritmu vložen 2D automat, pak se stejně indexuje jednorozměrně (při čtení hodnot ze stavu je opravdu jedno, jak funguje automat).

Některé z algoritmů přímo generují dlouhý klíč zadané délky – ty dědí od abstraktní třídy KeyExtenderAbstractN (viz 4.1). Tato třída překládá volání DoubleKey na volání ExtendKey a potomci této třídy implementují pouze ExtendKey. Jiné algoritmy vždy prodlouží klíč na dvojnásobek a obecné natažení realizují iterací tohoto postupu – ty jsou odvozené od abstraktní třídy KeyExtenderAbstractD (viz 4.2). Ta realizuje metodu ExtendKey pomocí logaritmického počtu volání DoubleKey a ořezává výsledek na správnou velikost. Potomci této třídy implementují již pouze DoubleKey.

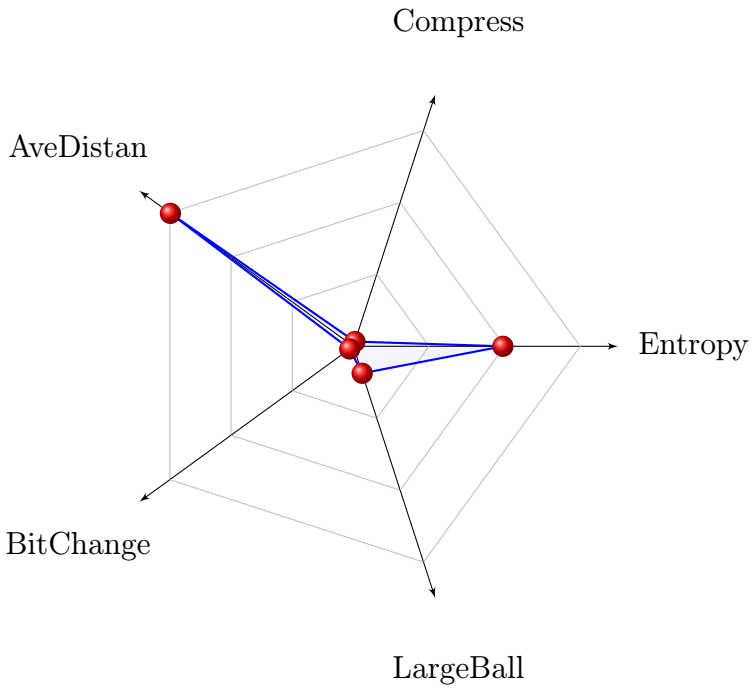
Kromě opravdových algoritmů byly vytvořeny ještě dva falešné algoritmy pro účely demonstrace, nakolik jsou kritické naše testovací metody. KeyExtenderCopy jen kopíruje kratší klíč dokola. Jeho výsledek znázorňuje obrázek 4.3. KeyExtenderCheating vylosuje pseudonáhodnout posloupnost bez ohledu na vstup. Jak hezky vypadá výsledek podvodného generátoru, si můžete prohlédnout na obrázku 4.4.



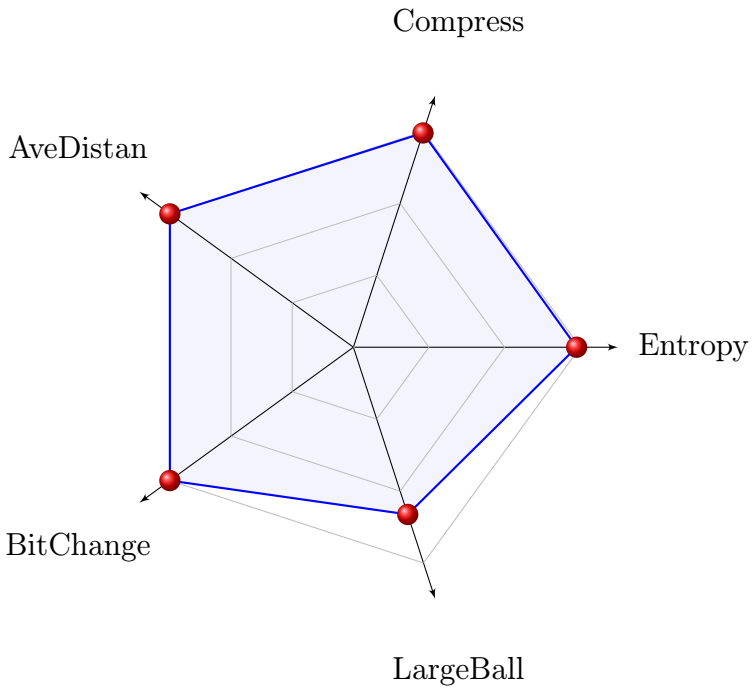
Obrázek 4.1: KeyExtenderAbstractN



Obrázek 4.2: KeyExtenderAbstractD



Obrázek 4.3: KeyExtenderCopy



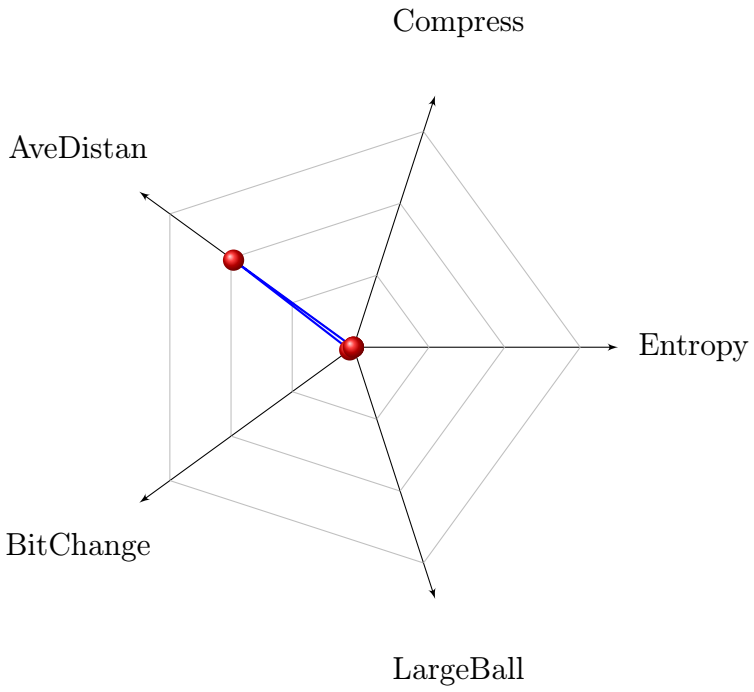
Obrázek 4.4: KeyExtenderCheating

4.1 KeyExtenderQuadratic

Stephen Wolfram popisuje (viz (Wolfram, 2002), strana 30), že když se použije elementární automat číslo 30 na pole obsahující jednu 1 uprostřed (jinak samé 0) a sleduje se, jak se mění prostřední buňka v čase (podobně jako na obrázku 1.4), tak její vývoj je perfektní pseudonáhodnou posloupností (splňující všechny testy pseudonáhodnosti, které vyzkoušel). To nás vede k otázce, jestli by z jiných počátečních stavů vznikly jiné kvalitní pseudonáhodné posloupnosti. Wolfram dále ukazuje ukazuje (viz (Wolfram, 2002), strana 251), že když na pseudonáhodném vstupu (počátečním stavu) automatu 30 změní jediný bit, tak se změna propaguje dolů a doprava, ale doleva se téměř nepropaguje. To zřejmě nebude platit zcela obecně, protože třeba pro změnu samých 0 na jednu 1 dojde ke změně, která se šíří do všech stran maximální rychlostí. Jeví se proto jako pravděpodobné, že vývoj prostřední buňky automatu 30 je ovlivněn postupně všemi buňkami nalevo a minimálně některými buňkami napravo.

A teď už k vlastnímu algoritmu: Nejprve je vytvořen celulární automat, který je dva a půl krát širší, než délka krátkého klíče. Ten krátký klíč uloží do jeho prostředních buněk. Tedy klíč délky n se uloží do stavu automatu s $2{,}5n$ buňkami a to od $0{,}75n$ po $1{,}75n$. Pak automat udělá $2n$ kroků. Výstup se čte z prostřední buňky (1 bit po každém kroku automatu). Původní návrh používal nekonečnou plochu, ale protože nám stačí natažení na $2n$, tak plocha o šířce $2{,}5n$ funguje stejně, jako kdyby se automat mohl rozpínat do nekonečna (efekt okraje se už nestačí promítnout do stavu prostřední buňky).

Je jisté, že výsledky testů budou silně záviset na použitém celulárních automatu. Například pokud podkladový automat vykazuje větší četnost 1 než 0 (či naopak), tak tím pravděpodobně budou negativně ovlivněny výsledky všech testů a o většině z testů (Entropy, Compress, AveDistan, BitChange) se také dá snadno ukázat, že se ani optimální



Obrázek 4.5: KeyExtenderSimpleQuadratic na automat 220

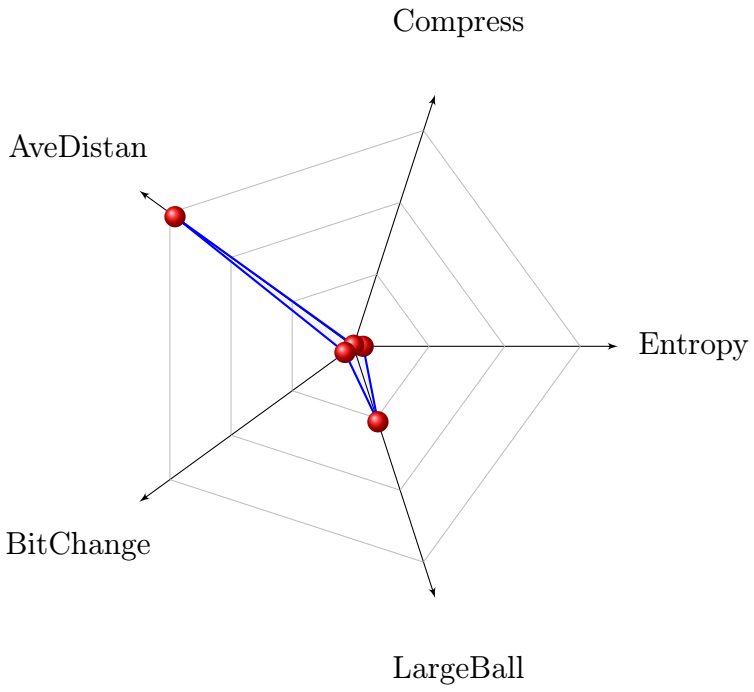
hodnotě při nevyrovnaných četnostech 0 a 1 nemohou blížit.

Vidíme, že použití automatů s monotónním chováním vede k příšerně špatným výsledkům (například viz 4.5). Automaty s fraktálním chováním na tom také nejsou dobře (třeba viz 4.7), ale automat číslo 30, který vykazuje pseudonáhodné chování, dává opravdu vynikající výsledek (viz 4.8). Použití náhodného (i když jinak kvalitně zvoleného) automatu s 2-okolím vede jen k průměrnému výsledku (jako třeba na obrázku 4.10). Celkově můžeme dospět k závěru, že chování algoritmu pro správně zvolené automaty (například ten 30) je velmi uspokojivé, ale jeho kvadratická časová složitost ho činí nepoužitelným pro šifrování delších souborů (aby vznikl dostatečný one-time pad pro zašifrování 1MB velkého souboru, tak bychom museli provést 8 milionů kroků automatu, což představuje $1{,}6 \cdot 10^{14}$ vyvolání přechodové funkce, což by na dnešních procesorech trvalo řádově jeden den).

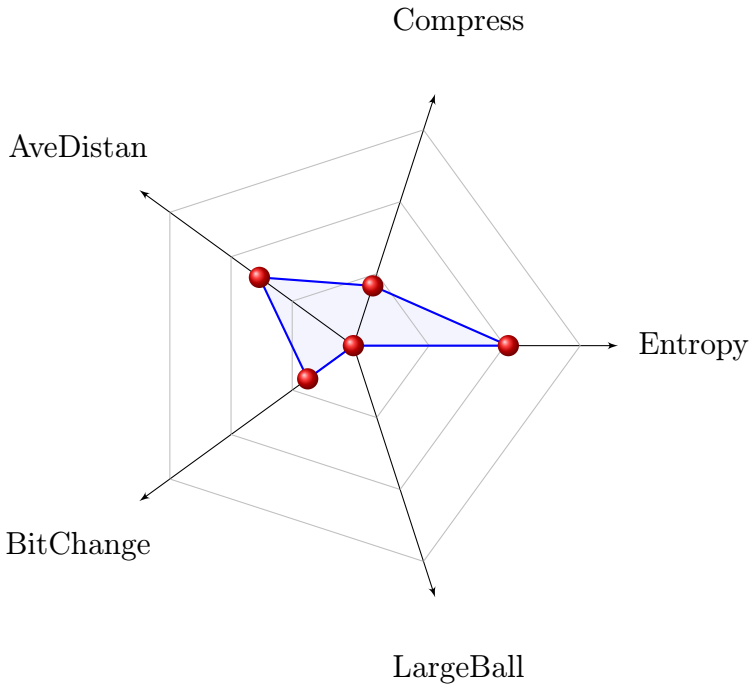
4.2 KeyExtenderSimpleLinear

Nejjednodušší způsob, jak natáhnout klíč na dvojnásobek. Tento algoritmus použije vstup jako počáteční konfiguraci celulárního automatu. Pak udělá krok a uloží jeho stav do první poloviny dlouhého klíče (postupně ze všech buněk). Pak udělá druhý krok a načte druhou polovinu dlouhého klíče. Při použití elementárního automatu číslo 204 jeho chování degeneruje na kopírování hodnot (stejný výstup jako KeyExtenderCopy), ale tím se zabývat nebudeme. Při použití automatu číslo 51 je druhá polovina výstupu přesnou negací první poloviny výstupu, což je stejně tak nezajímavé.

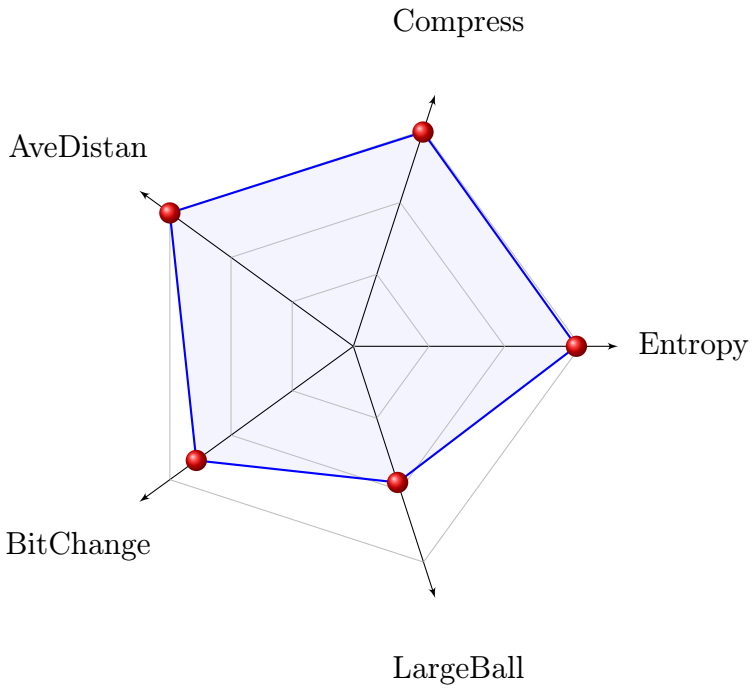
Od tohoto algoritmu nemůžeme čekat moc krásné chování. Například při jeho aplikaci s elementárním automatem za účelem natažením klíče na dvojnásobek dochází k tomu, že



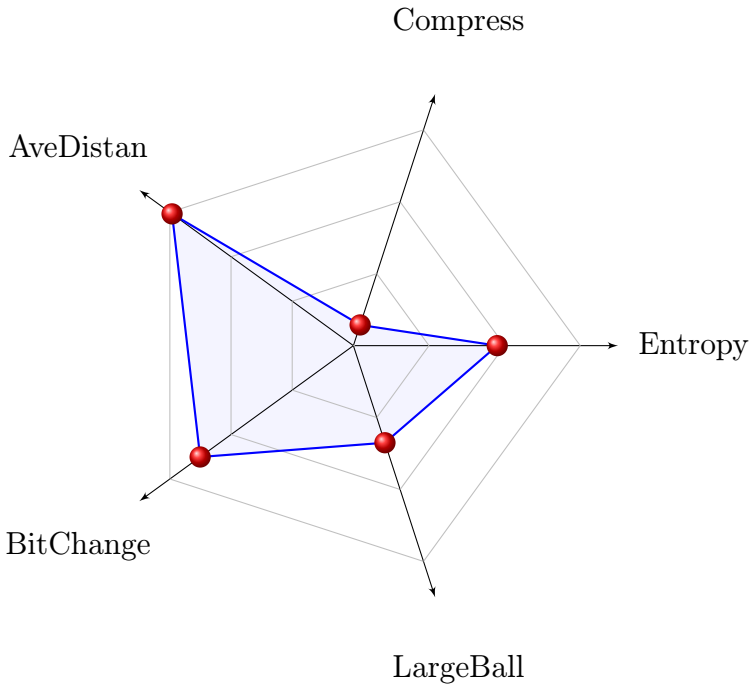
Obrázek 4.6: KeyExtenderSimpleQuadratic na automat 94



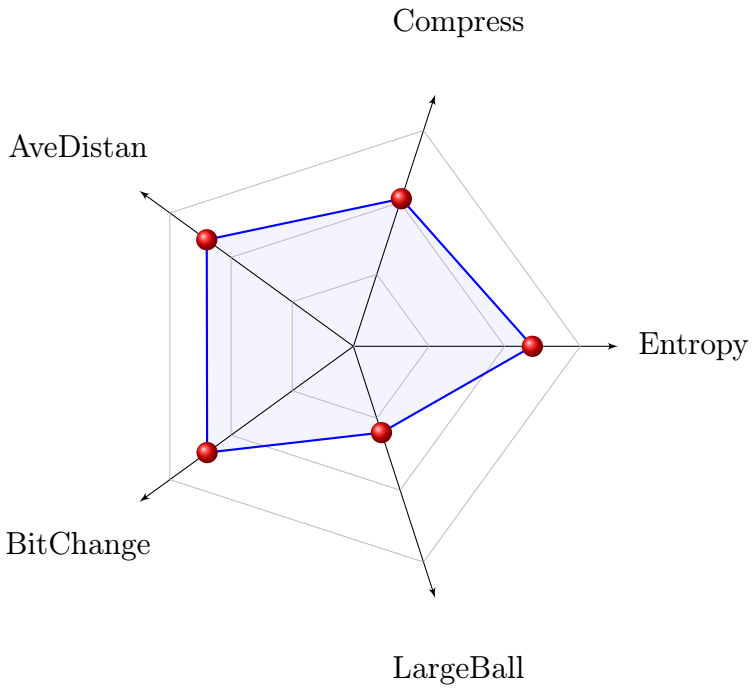
Obrázek 4.7: KeyExtenderSimpleQuadratic na automat 90



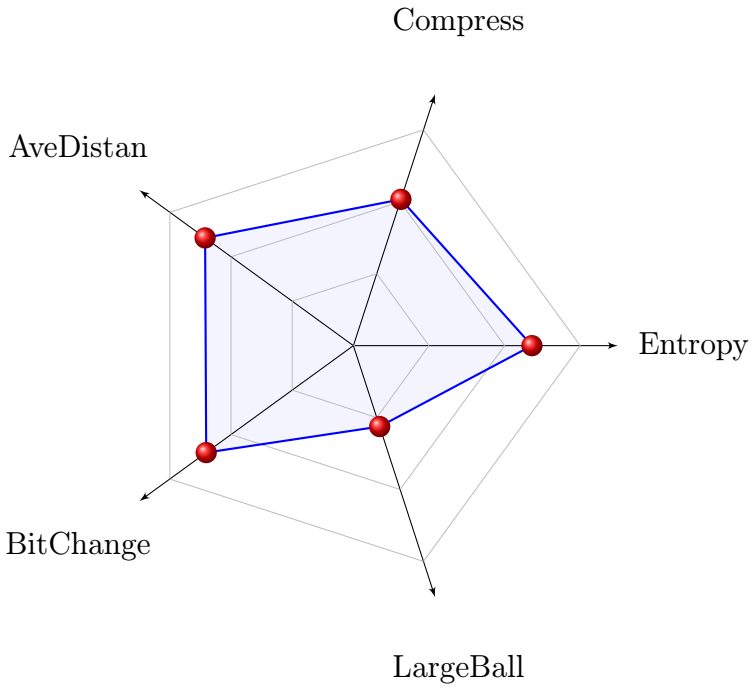
Obrázek 4.8: KeyExtenderSimpleQuadratic na automat 30



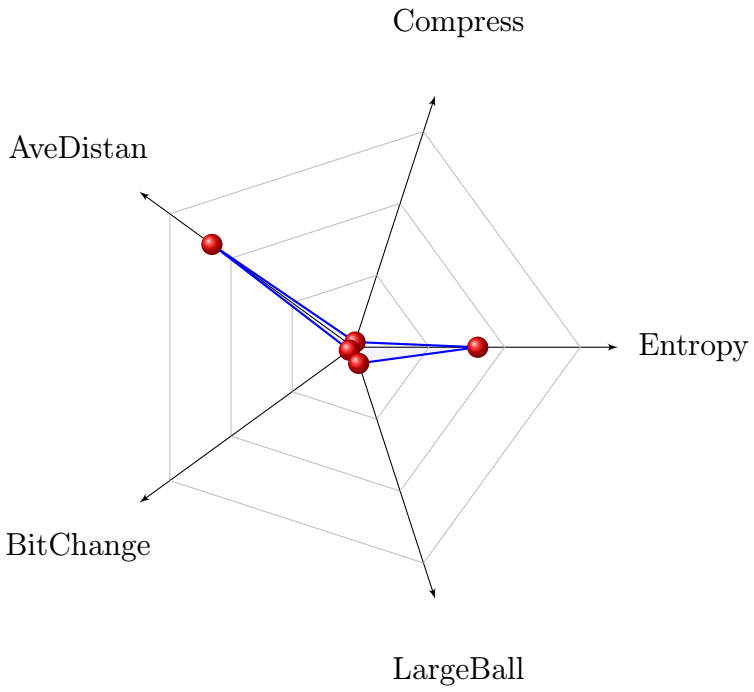
Obrázek 4.9: KeyExtenderSimpleQuadratic na automat 110



Obrázek 4.10: KeyExtenderSimpleQuadratic na omezený automat s 2-okolím



Obrázek 4.11: KeyExtenderSimpleQuadratic na cyklický automat s 2-okolím



Obrázek 4.12: KeyExtenderSimpleLinear na automat 220

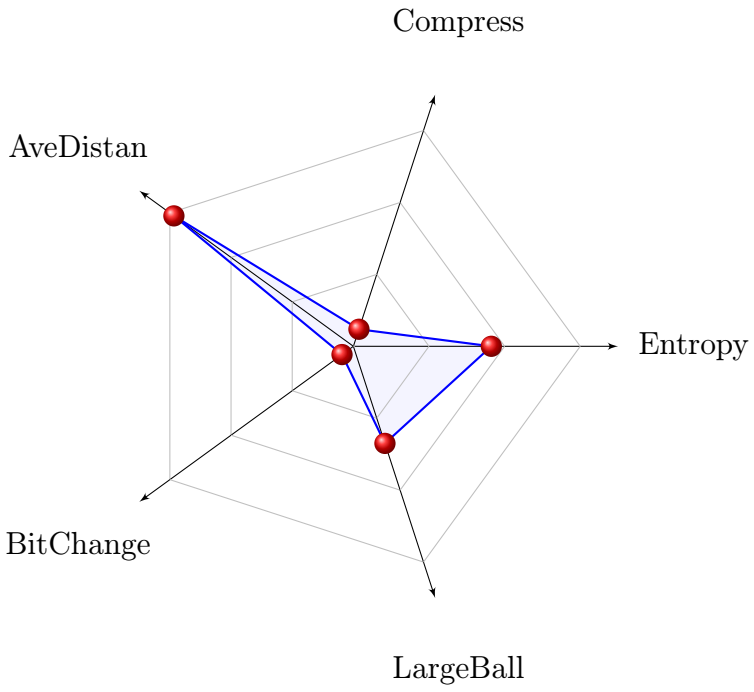
když změníme 1 bit na vstupu, nemůže se změnit více než 8 bitů na výstupu (a konkrétní podoba této změny je určena jen 9 sousedními buňkami v původním stavu). Pro 2D automaty nebo automaty využívající v přechodové funkci větší okolí budou sice tato čísla vyšší, ale stále to budou nějaké konstanty, které nezávisí na velikosti klíče.

Kombinace jednoduchého algoritmu a automatu s jednoduchým chováním vede pochopitelně ke špatným výsledkům (viz obrázek 4.12). Ovšem výsledky automatu, který na vstupu s jedinou jedničkou generuje fraktály, není tak špatný (viz obrázek 4.14). Příjemným překvapením je ovšem výsledek 2D automatu Amoeba Universe (viz 4.19), který v testu komprese dosahuje výsledku přes 0,8 (zatímco 1D automaty nepřesáhly hodnotu 0,2).

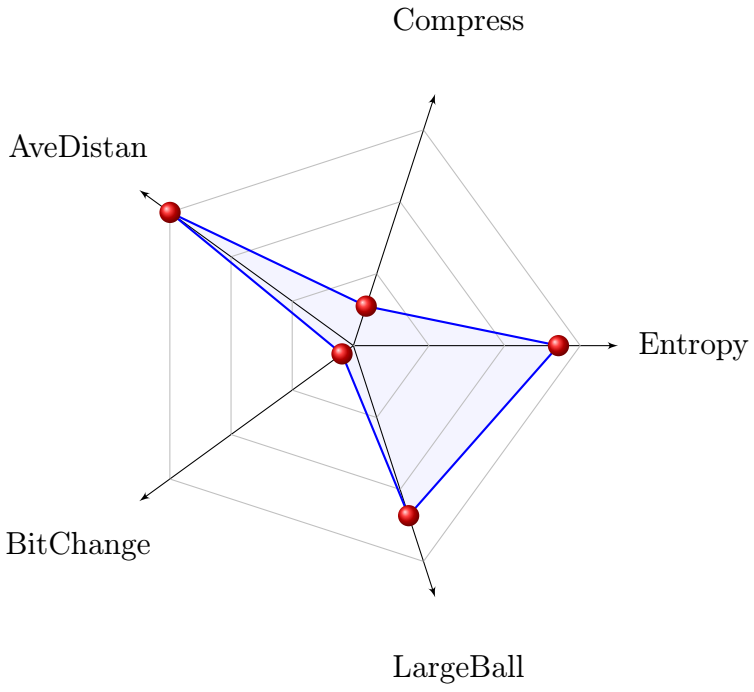
4.3 KeyExtenderInterlaced

KeyExtenderInterlaced generuje výstupní klíč z obecně většího počtu stavů. Tento algoritmus se tváří jako kompromis mezi předchozími dvěma algoritmy, ale blíží se spíše variantě KeyExtenderSimpleLinear. Algoritmus je parametrizován počtem řad p, ze kterých má dlouhý klíč generovat, a údajem q, kolik kroků navíc má automat vždy provést mezi generováním využívaných stavů. Pokud se spustí s parametry $p = 2$, $q = 0$, potom generuje totožný klíč jako lineární algoritmus. Jeho časová složitost je lineární ve velikosti vstupu a ještě lineární v součinu $p(q + 1)$.

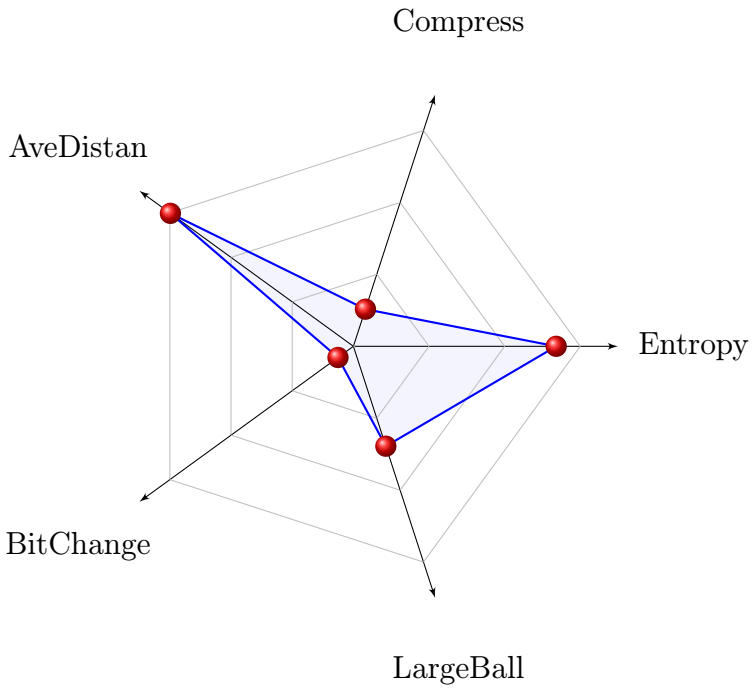
Jaké máme očekávání od tohoto algoritmu? Vzdálenost výstupů při změně jednoho bitu vstupu se zvýší oproti jednoduché lineární variantě a toto zlepšení bude tím vyšší, čím větší budou hodnoty p, q. U velmi špatných automatů, kde přechodová funkce neustále snižuje původní náhodnost vstupu, způsobí problémy vyšší počet provedených kroků a



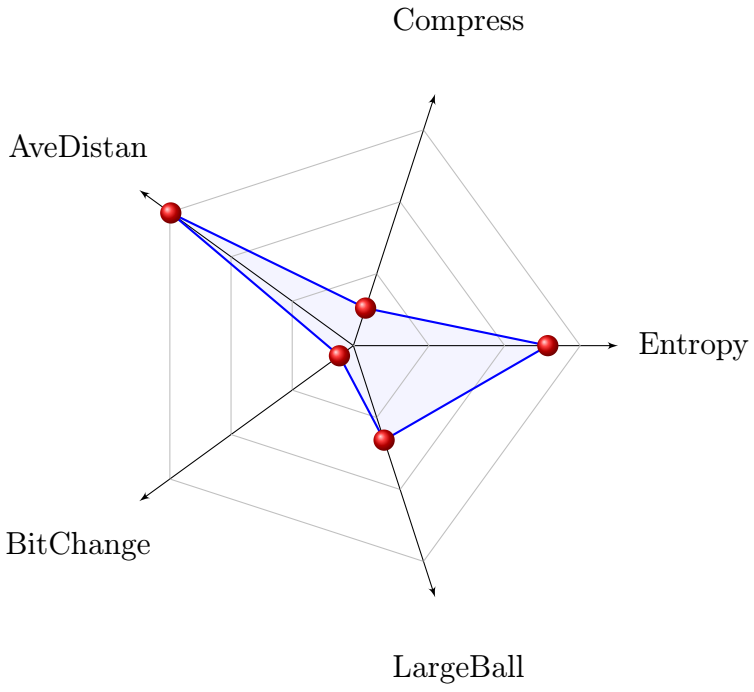
Obrázek 4.13: KeyExtenderSimpleLinear na automat 94



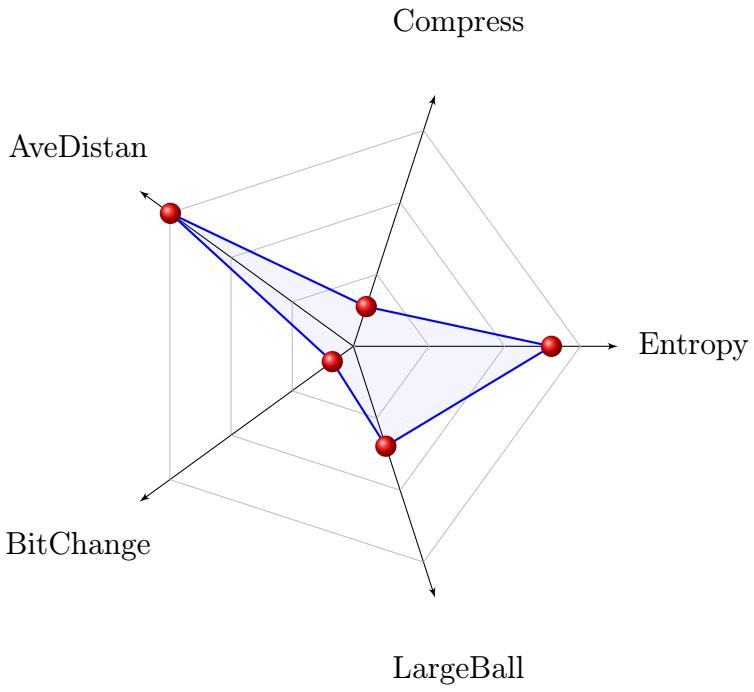
Obrázek 4.14: KeyExtenderSimpleLinear na automat 90



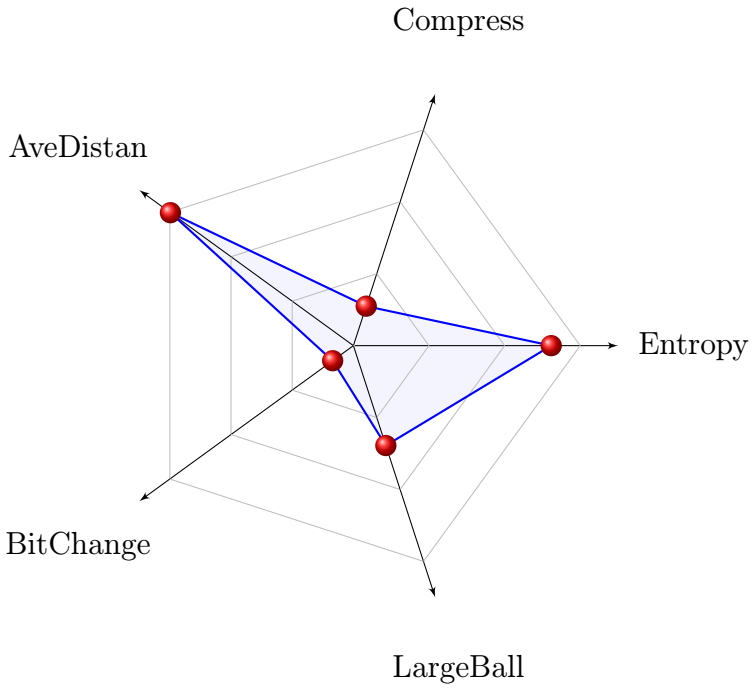
Obrázek 4.15: KeyExtenderSimpleLinear na automat 30



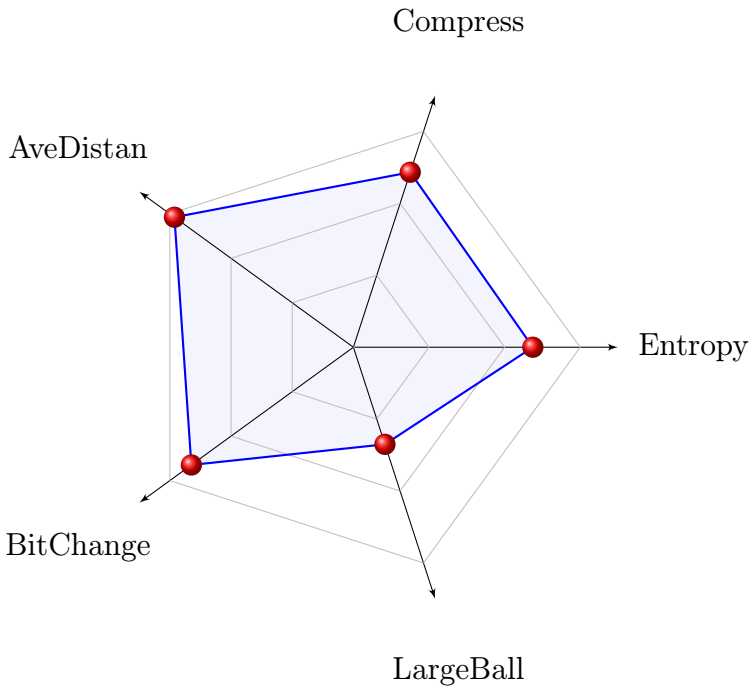
Obrázek 4.16: KeyExtenderSimpleLinear na automat 110



Obrázek 4.17: KeyExtenderSimpleLinear na omezený automat s 2-okolím



Obrázek 4.18: KeyExtenderSimpleLinear na cyklický automat s 2-okolím

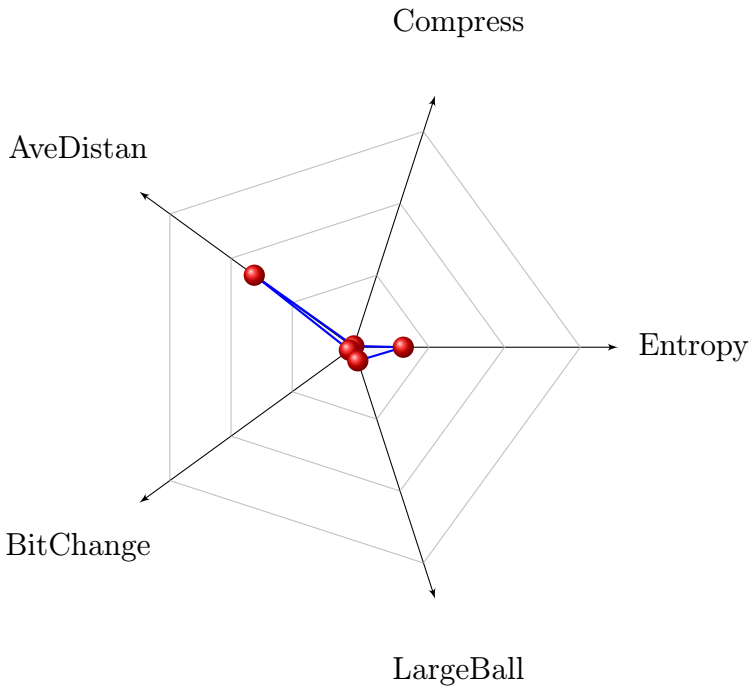


Obrázek 4.19: KeyExtenderSimpleLinear na Amoeba Universe

dojde k výraznému snížení všech 5 měřených veličin.

Zaměřme se ještě na volení hodnot argumentů p, q. Velké hodnoty znamenají hodně kroků automatu. Všeobecně se zdá být lepší zvyšovat p, aby se na výstupu podílelo více různých stavů automatu. Nejmenší možná hodnota $p = 2$ není vhodná ani při vysokém q, protože vede ke čtení souvislého bloku buněk automatu a tudíž lze ze znalosti první poloviny klíče snadno vypočítat druhou polovinu klíče. Tím má smysl se znepokojovat! Pokud by útočník zachytil ciphertext a uhodl první polovinu zprávy, tak ví i první polovinu klíče, z ní dokáže spočítat druhou polovinu klíče a tím rozluští neznámou druhou polovinu zprávy [1]. Tato neřest by se mohla projevit v testu komprese, protože pokud je kompresní algoritmus dost chytrý, tak by mohl zmenšit výstup až na polovinu. Stejná výtka platí pro KeyExtenderSimpleLinear. Když ovšem použijeme z jednoho řádku méně hodnot, tak se dopočítání celého stavu silně komplikuje, přestože budeme mít údaje o celkově vyšším počtu řádků. Jinými slovy, pro $p > 3$ je šířka použitého automatu větší než velikost výstupu, takže počet logických proměnných, které by musel útočník dopočítat, je ostře vyšší než počet „rovnic“ (vztahů výrokové logiky), které může využít. To samozřejmě není žádný důkaz, protože pro některé automaty bude možné i z malého počtu bitů, které jsou nějak nasamplované z různých stavů, dopočítat celý stav automatu. Zvyšování hodnoty q tuto výhodu neposkytuje, ale vztahy mezi jednotlivými použitými stavy

[1] Z tohoto důvodu také provádíme jak u algoritmu KeyExtenderSimpleLinear i u algoritmu KeyExtenderInterlaced jeden krok automatu ještě před prvním čtením hodnot. Kdybychom nejprve četli stav a pak až provedli krok a volali bychom DoubleKey třeba 2x za sebou, abychom klíč natáhli na čtyřnásobek (typicky to bude ještě mnohem větší počet!), tak už by útočníkovi stačilo uhodnout jen první čtvrtinu zprávy, aby byl schopnen rozluštit celou zprávu. Pokud ale nejprve provedeme krok automatu, tak útočník sice z první čtvrtiny klíče může spočítat třetí čtvrtinu klíče, ale pro dopočítání druhé čtvrtiny klíče by musel první krok druhého volání DoubleKey invertovat, což je málo kdy možné.



Obrázek 4.20: KeyExtenderInterlaced(10, 0) na automat 220

jsou alespoň o dost komplikovanější.

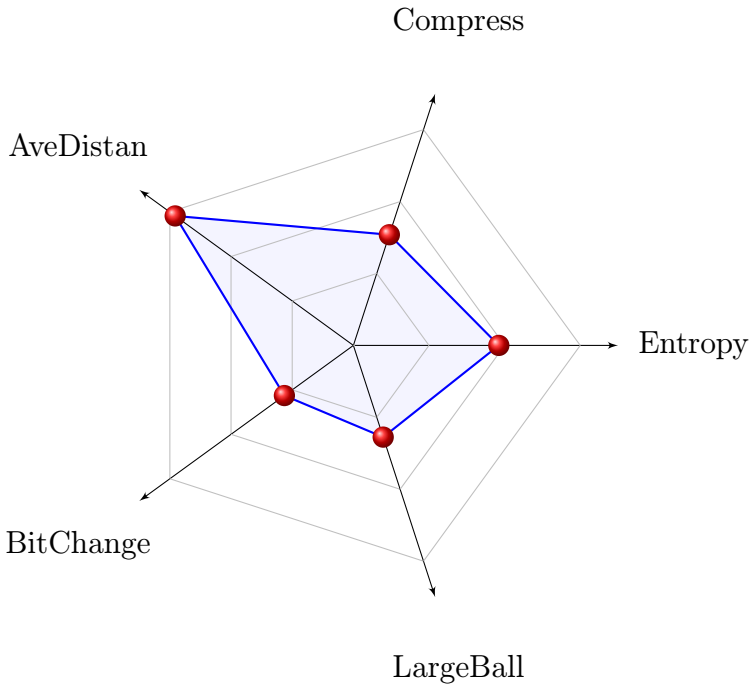
Pro generování výsledků jsme zvolili hodnoty $p = 10, q = 0$. Ve srovnání s KeyExtenderSimpleLinear vede prokládání u mnoha automatů ke znatelnému zlepšení chování (týká se to všech či většiny elementárních CA a nejvíce je to vidět na příkladu pravidla 110, viz 4.24), u jiných k výraznému zhoršení chování (zde viz 4.27).

4.4 KeyExtenderUncertain

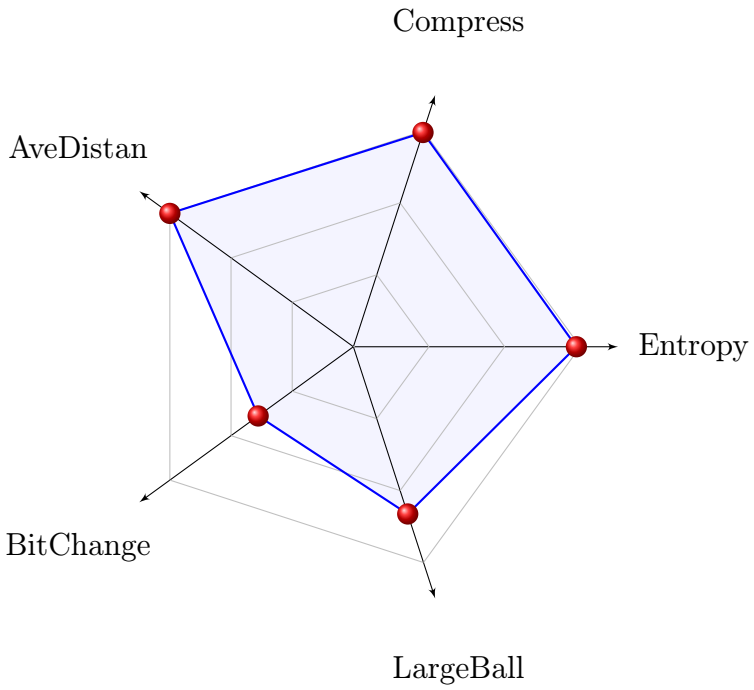
Tento algoritmus řeší častou neřest (prohřešek proti pseudonáhodnosti), kterou vykazují celulární automaty – různý počet 0 a 1. Algoritmus zase využívá celý stav automatu. Jsou čteny vždy dvojice bitů, přičemž dvojice 00 a dvojice 11 jsou zahazovány. Vždy, když algoritmus narazí na dvojici 01, tak pošle na výstup 0. A za každou dvojici 10 pošle na výstup 1. Jedná se o známý trik, který se běžně používá třeba u hardwarových generátorů náhodných čísel, u nichž sice nedochází k periodickému opakování stejných sekvencí, ale občas bývá problém (třeba v důsledku stárnutí senzorů) s rozdílnou četností bitů.

Počet kroků automatu, který bude algoritmus muset provést, není předem známý. Pokud se automat zasekne ve stavu, ze kterého není úniku (například samé nuly u mnoha druhů automatů, nebo také střídání 001100110011..001100 u elementárního automatu, který používá jako přechodovou funkci majorantu buňky a jejich těsných sousedů), je vyhozena výjimka. Případy, kdy se generování dlouhého klíče nepodařilo dokončit, jsou zařazeny do výsledků s hodnout 0 u všech testů.

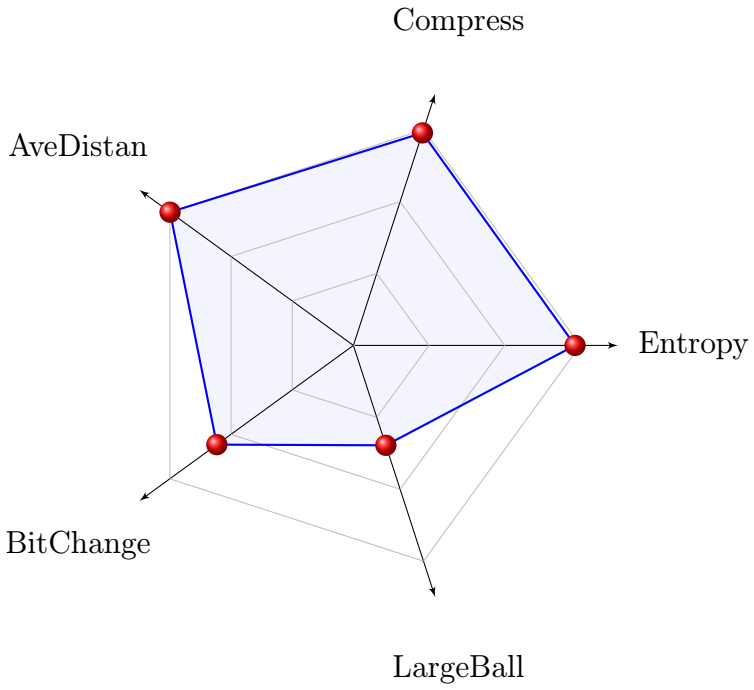
Očekávali jsme, že tento postup zvýší entropii výstupů, protože více vyrovná četnost 0 a 1. Samozřejmě to nemůže fungovat vždy, protože například elementární automat číslo 90 může dojít do stavu, který obsahuje dlouhý úsek pravidelného střídání 010101..0101,



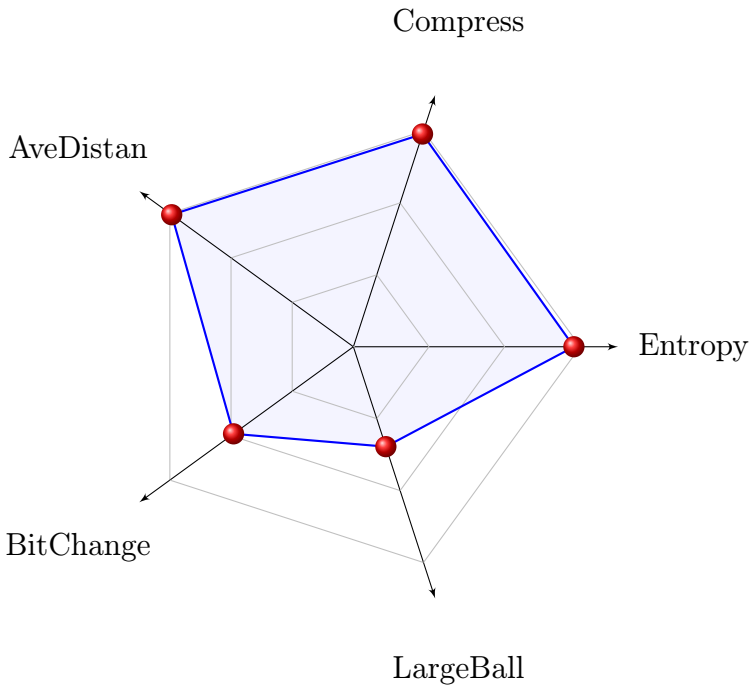
Obrázek 4.21: KeyExtenderInterlaced(10, 0) na automat 94



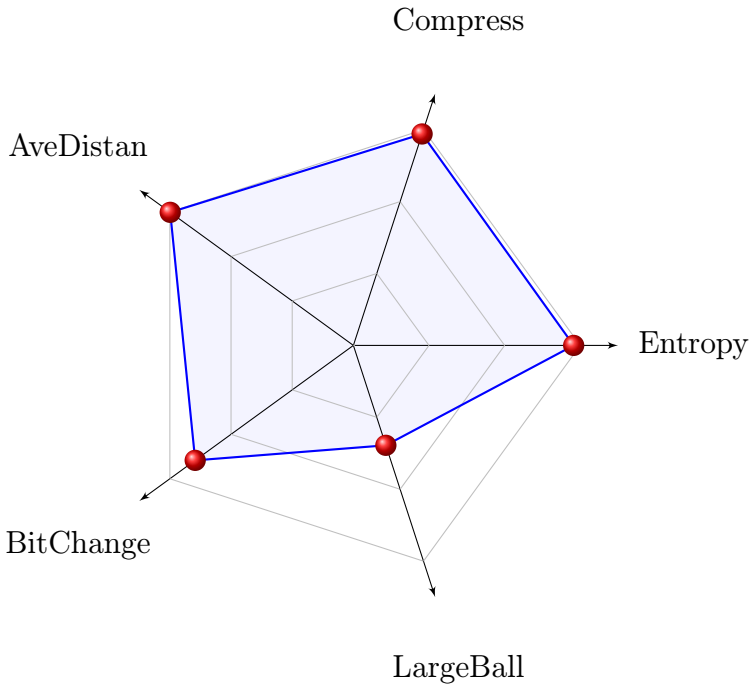
Obrázek 4.22: KeyExtenderInterlaced(10, 0) na automat 90



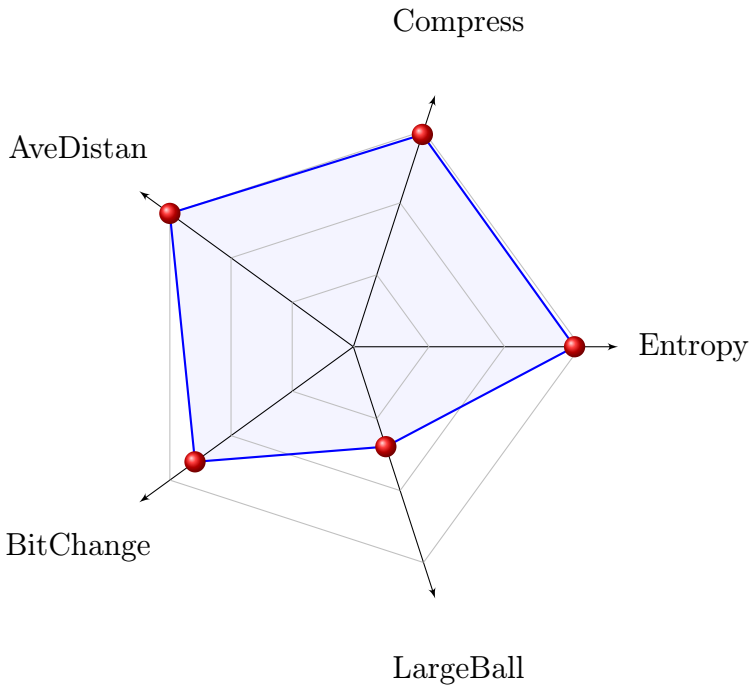
Obrázek 4.23: KeyExtenderInterlaced(10, 0) na automat 30



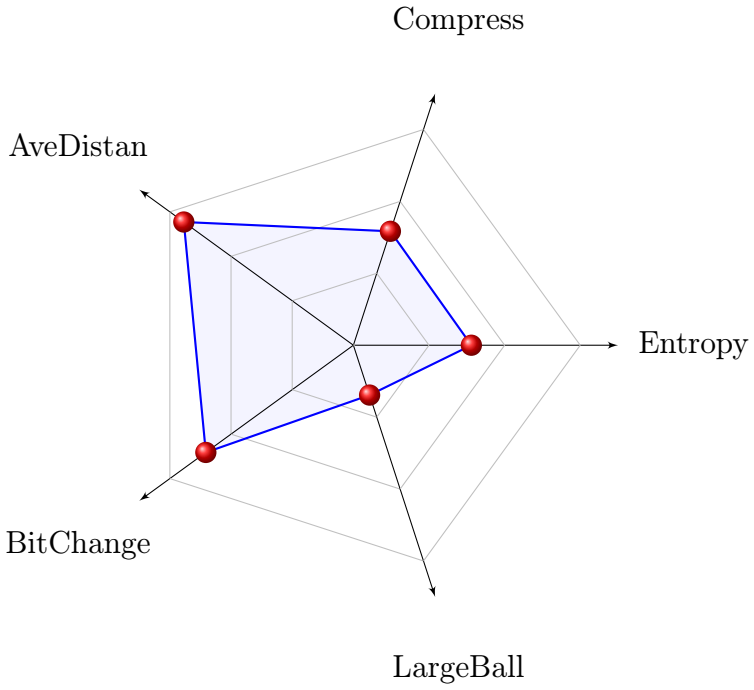
Obrázek 4.24: KeyExtenderInterlaced(10, 0) na automat 110



Obrázek 4.25: KeyExtenderInterlaced(10, 0) na omezený automat s 2-okolím



Obrázek 4.26: KeyExtenderInterlaced(10, 0) na cyklický automat s 2-okolím



Obrázek 4.27: KeyExtenderInterlaced(10, 0) na Amoeba Universe

který se přetvoří na souvislý úsek samých nul a entropie je tím ještě nižší. A taky jsme se trochu obávali, že se program „zasekne“, když bude příliš malá část automatu „živá“ pro generování bitů na výstup, ale bude dlouho trvat, než automat dospěje do konečného neměnného stavu.

Z výsledků se dozvídáme, že tento netradiční způsob čtení hodnot vedl v pár případech k dobrému chování (třeba na obrázku 4.30). Většinou ale špatně dopadal test entropie (hůře než u KeyExtenderSimpleLinear).

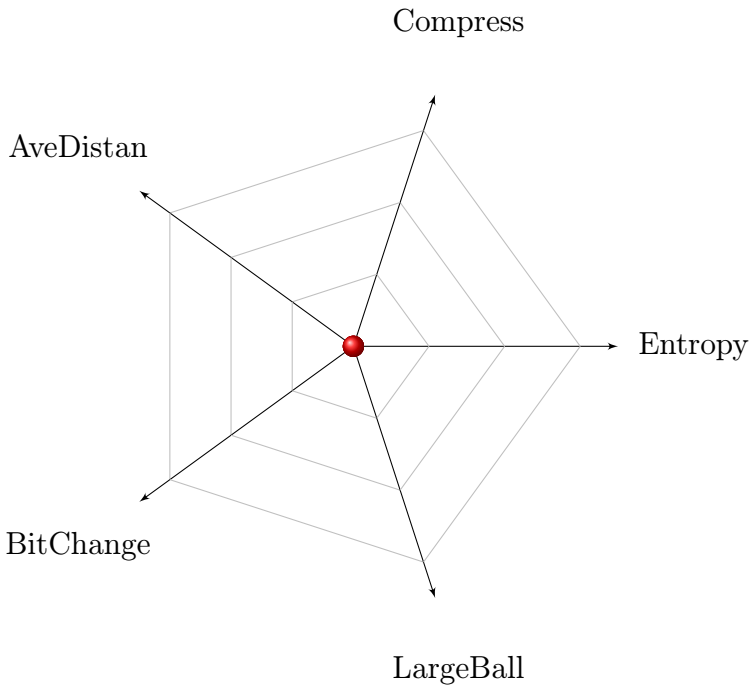
4.5 KeyExtenderGenetic

Nejkomplikovanější algoritmus, který tato práce obsahuje. Jako u mnoha jiných algoritmů i zde se klíč postupně natahuje na dvojnásobek, než dosáhne dostatečné délky, ale každý krok může používat jiný druh automatu a jiný dílčí algoritmus.

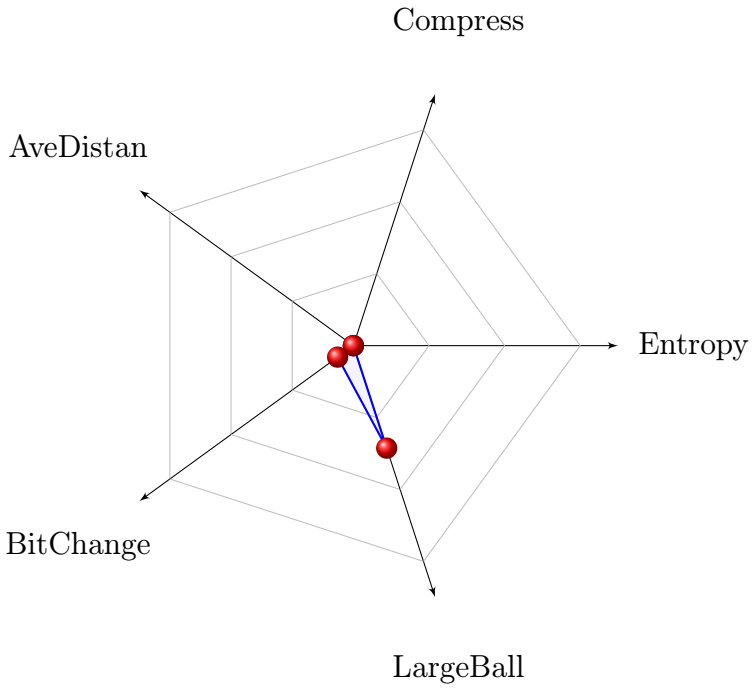
Ke zjištění správné posloupnosti těchto natahovačů je použit genetický algoritmus. Ten tuto posloupnost optimalizuje vždy pro jeden konkrétní vstup. Fitness funkcí je aritmetický průměr z testů entropie a komprese.

Na začátku je vygenerována náhodná populace, kde jedinci jsou sekvence natahovačů (prvky sekvence jsou instance KeyExtenderAbstractD s již přidělenými binárními automaty a celá sekvence je dlouhá $\lceil \log_2 \frac{novaDelka}{puvodniDelka} \rceil$), iniciálně vybrána náhodně. Pak je provedena řada iterací, která se skládá z turnajové selekce a genetických operátorů. Na konci iterace je celá původní populace nahrazena novou populací. Nejlepší jedinec z průběhu běhu celého algoritmu je zapamatován a lze k němu přistoupit i po dokončení generování dlouhé sekvence.

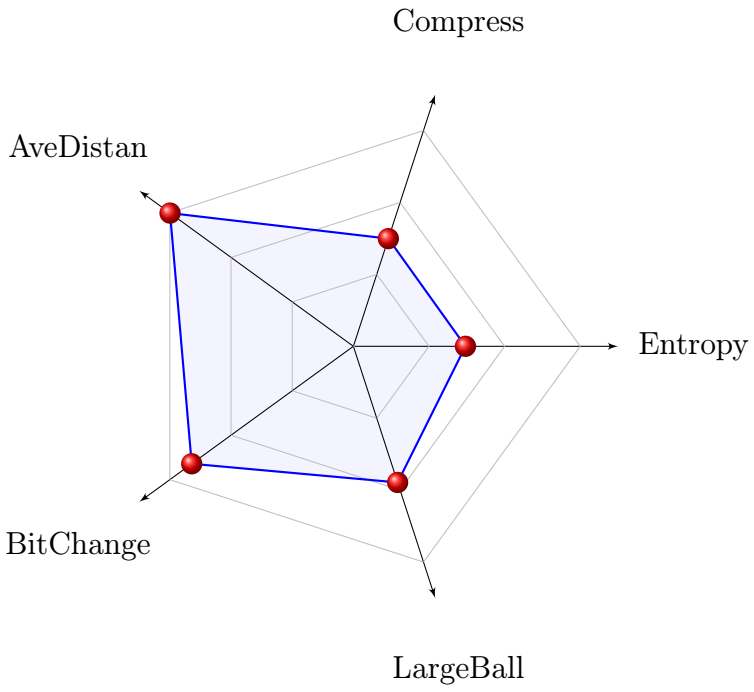
Jsou použity dva genetické operátory. Prvním je jednoduché křížení (One-point cros-



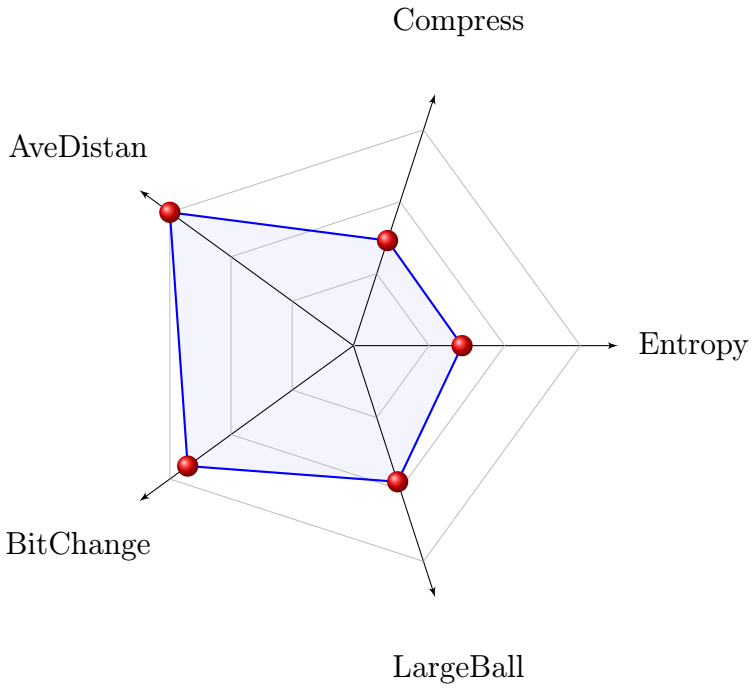
Obrázek 4.28: KeyExtenderUncertain na automat 220



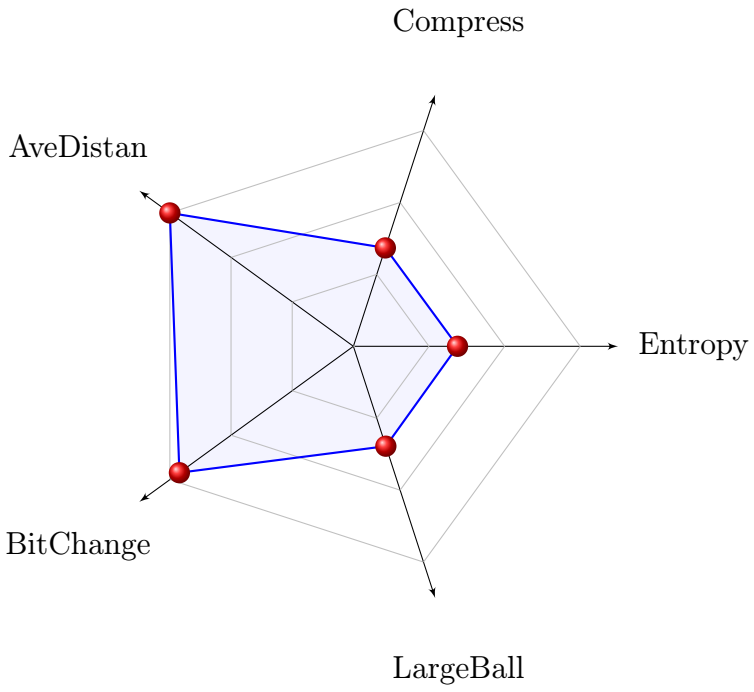
Obrázek 4.29: KeyExtenderUncertain na automat 94



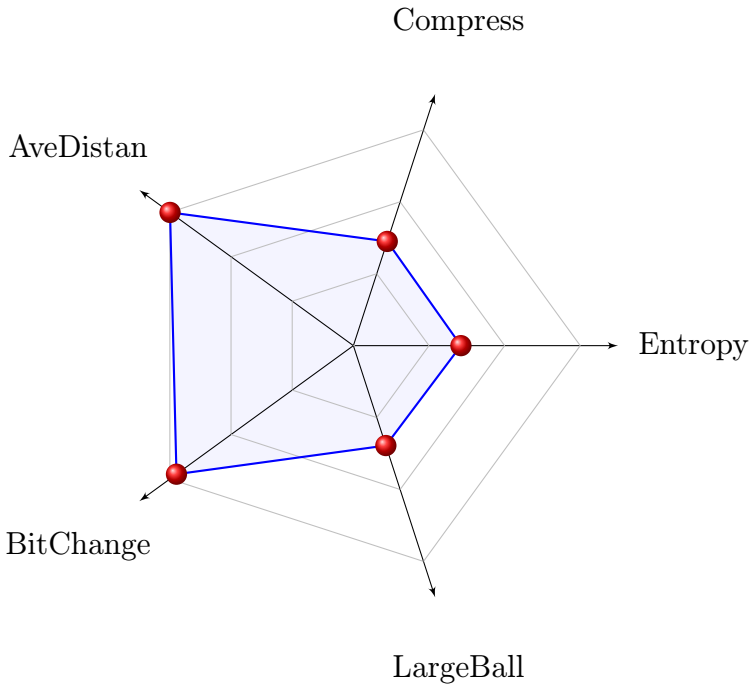
Obrázek 4.30: KeyExtenderUncertain na automat 90



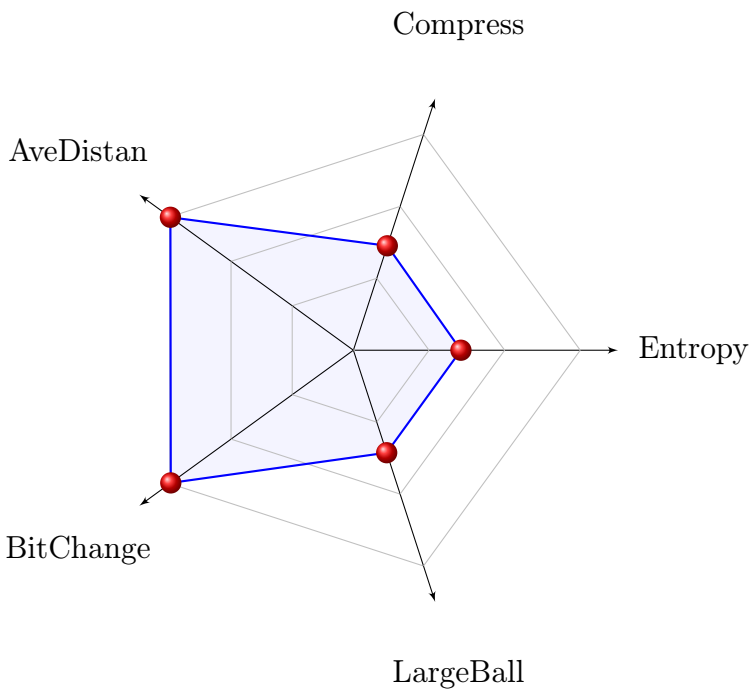
Obrázek 4.31: KeyExtenderUncertain na automat 30



Obrázek 4.32: KeyExtenderUncertain na automat 110



Obrázek 4.33: KeyExtenderUncertain na omezený automat s 2-okolím



Obrázek 4.34: KeyExtenderUncertain na cyklický automat s 2-okolím

sover) a druhým je náhodná mutace (výměna právě jednoho natahovače v sekvenci za náhodný). Pravděpodobnost křížení a pravděpodobnost mutace je dána konstantou v kódu, kterou se klidně ciťte volní změnit. Stejně tak lze změnit velikost populace, počet iterací a selekční tlak.

Ve třídě KeyExtenderGenetic se používá samostatný generátor pseudonáhodných čísel (jehož seed je daný) k provádění všech operací, díky čemuž je pro stejný vstup možné vždy znovu vygenerovat stejný výstup na straně odesílatele i příjemce zprávy. To je velmi výhodné, protože tak není potřeba použitou sekvenci automatů kódovat na začátek zašifrovaného souboru. To šetří práci programátora, přenosové pásmo i informaci, která by mohla potenciálně vyzradit něco o šifrovacím klíči.

Zbývá určit, co má být tím náhodným natahahovačem… Program obsahuje dvě varianty tohoto: vnitřní třídy Primitives a GoodPrimitives, které obě implementují vnitřní rozhraní IPrimitives. Třída Primitives je implementována jako singleton. Při prvním vytvoření se nejprve vygeneruje seznam binárních CA, které mohou být použity. Do seznamu je zařazeno všech 256 elementárních CA, dále 3 druhy 2D totalistických CA (Game of Life, Amoeba Universe a Replicator Universe) a k tomu 200 kusů náhodně vygenerovaných 1D automatů s přechodovou funkcí využívající buňku, její sousedy a sousedy sousedů (z toho 100 je na omezeném hřišti a dalších 100 na zacykleném hřišti). Dílčí algoritmy pro jednotlivé kroky jsou vybírány jen z těch, které mají lineární časovou složitost. S každým z výše uvedených binárních CA je vytvořen 1 KeyExtenderSimpleLinear a 6 různých KeyExtenderInterlaced.

Třída GoodPrimitives funguje jinak. Nejprve jsou vygenerována data o úspěšných natahovačích a použitých automatech během četných běhů genetického algoritmu s využitím třídy Primitives. Za tímto účelem byl mnohokrát spuštěn genetický algoritmus na náhodné klíče o velikosti 100, které natahoval na velikost 25000. Z každého běhu gene-

tického algoritmu byla uložena ta vítězná sekvence do souboru. Celkem ten experiment běžel přes 100 hodin. Teď je možné vytvořit instanci třídy GoodPrimitives, která načte tyto natahovače s jejich automaty z určené složky a pak poskytuje právě takto získané natahovače.

Od genetického algoritmu jsme si slibovali, že najde jednoznačně nejlepší způsob natažení klíče ze všech algoritmů. Pokud by to totiž bylo výhodné, tak by i genetický algoritmus nalezl sekvenci, kde každé zdvojení klíče provede stejný automat stejným způsobem. Zdá se být, že nejdůležitější pro kvalitu výstupu je poslední zdvojení klíče v řadě. Pro hledání nejvhodnějšího algoritmu na zakončení sekvence je důležitá mutace, křížení zde nepomáhá.

Nakonec jsme zjistili, že genetický algoritmus sice vytvoří nejlepší výstup ze všech, ale potřebuje na to velkou populaci a mnoho generací, takže časová složitost je značně vysoká. Kvůli ní taky nebylo možné otestovat všechny vlastnosti algoritmů, které jsme testovali předtím, například průměrnou změnu výstupu při změně jednoho bitu na vstupu. Byly prováděny experimenty, během kterých byly měněny pravděpodobnosti jednotlivých operátorů i selekční tlak, ale na výsledek to téměř nemělo vliv.

5. Další možnosti šifrování pomocí celulárních automatů

Kromě protahování klíčů by šlo využít celulární automaty v kryptografii i jinak. Třeba posadit plaintext jako počáteční stav nějakého reverzibilního CA. Šifrovacím klíčem by zde byla samotná přechodová funkce toho automatu. Ciphertext by byl stav automatu po pár krocích. Rozšifrování by bylo snadné, ale bez znalosti přechodové funkce by se to nedalo tipnout.

Kromě toho, že mezi elementárními celulárními automaty se nachází několik dvojic, které dělají navzájem inverzní zobrazení, lze reverzibilní CA generovat i více systematicky, viz (Schiff, 2008), strana 68.

A možná by tohle šlo dělat i jako asymetrická kryptografie. Chtělo by to nějakou třídu reverzibilních CA (s jiným pravidlem tam a jiným pravidlem zpět) takovou, že by se snadno rychle vygenerovat dvojice (pravidlo, obrácené pravidlo), ale k zadanému pravidlu by se těžko generovalo to obrácené pravidlo. Zde by jedno přechodové pravidlo bylo veřejným klíčem a druhé tajným klíčem. Asi by to bylo hodně náchylné na known-plaintext attack, ale stejně by to mohlo být hezké ...

5.1 Řešení

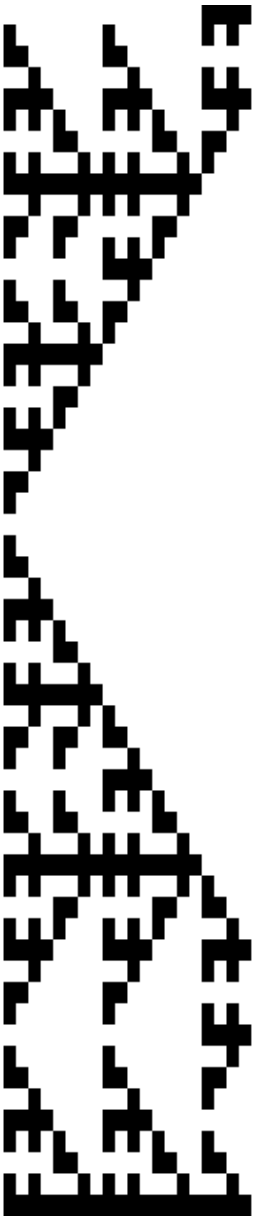
Uvedená symetrická varianta šifrování s využitím reverzibilních celulárních automatů byla implementována i v této práci. Reverzibilní CA je naprogramován ve třídě Cellular.ReversibleAutomaton a jedná se o jednorozměrný binární CA s pravidlem zahrnujícím r sousedů na každou stranu a zacykleným okrajem. Tento automat si kromě současného stavu pamatuje i předchozí stav a svůj krok počítá takto:

$$x_i(t+1) = f(x_{i-r}(t), ..., x_{i-1}(t), x_i(t), x_{i+1}(t), ..., x_{i+r}(t)) \oplus x_i(t-1)$$

Šifrovací algoritmus na něm založený je pak ve třídě Crypto.EncrypterReversibleCA. Šifrování probíhá tak, že algoritmus vytvoří nový reverzibilní CA, jehož přechodová funkce je určena šifrovacím klíčem, který musí mít délku 2^{2r+1} (takže například 32, 128, 512, 2048, ...).

První polovina plaintextu určuje předchozí stav CA a druhá polovina současný stav CA. Pak je proveden daný počet kroků (příklad vývoje v čase z jednoduchého zadání je na obrázku 5.1). Koncový stav je použit jako první polovina ciphertextu a předchozí stav jako druhá polovina ciphertextu. Tak je způsobena symetrie. Když je teď vytvořen nový automat se stejnými pravidly, tak umístěním první poloviny ciphertextu jako předchozí stav a druhé poloviny ciphertextu jako aktuální stav vzniká automat, který jde „pozpátku v čase“ (ale prochází zcela identickými stavy) jako automat, který byl použit na zašifrování (vývoj obráceného automatu odpovídajícímu předchozímu automatu je na obrázku 5.2). Pro provedení daného počtu kroků je možné opět přečíst plaintext z posledních dvou stavů.

Obrázek 5.1: Vývoj automatu R60



Obrázek 5.2: Vývoj automatu R60 zpátky

Závěr

Nejlépe dopadl algoritmus, který jsme uvažovali hned jako první možnost: KeyExtenderSimpleQuadratic using rule No. 30: TestEntropy : 0,983 TestCompression : 0,993 TestAverageDistance: 1,000 TestBitChange : 0,856 TestLargestBall : 0,631 Jeho nevýhodou ale je kvadratická časová složitost.

Z algoritmů s lineární časovou složitostí dopadl nejlépe: KeyExtenderInterlaced(10, 0) using Cyclic Binary automaton with rule 00010010111111001100100010011110: TestEntropy : 0,975 TestCompression : 0,984 TestAverageDistance: 1,000 TestBitChange : 0,863 TestLargestBall : 0,463

Závěrem je, že žádný z našich šifrovacích algoritmů není natolik kvalitní, abychom ho mohli používat pro kritické bezpečnostní aplikace.

Seznam použité literatury

Beenakker, C. (2013). Expected edit distance. URL `http://mathoverflow.net/questions/128903/expected-edit-distance?rq=1`.

Delfs, H. a Knebl, H. (2002). *Introduction to Cryptography.* First edition. Georg-Simon-Ohm University of Applied Sciences Nürnberg, Department of Computer Science, Kesslerplatz 12, 90489 Nürnberg, Germany. ISBN 3-540-42278-1.

Knuth, D. E. (1981). *The Art of Computer Programming, Volume 2: Seminumerical Algorithms.* 2nd edition. Addison-Wesley Pub (Sd), 75 Arlington Street, Suite 300, Boston, Massachusetts, USA. ISBN 978-0201038224.

Schiff, J. L. (2008). *Cellular Automata.* First edition. John Wiley and Sons, Inc., Hoboken, New Jersey, USA, Corporate Headquarters, 111 River Street, Hoboken, NJ 07030-5774. ISBN 978-0-470-16879-0.

Vlastimil Klíma, R. (2003). Blokové šifry. *MFF výukový materiál*, **0**(0), 0.

Wojtowicz, M. (2001). Cellular automata rules lexicon. URL `http://psoup.math.wisc.edu/mcell/rullex_life.html`.

Wolfram, S. (2002). *A New Kind of Science.* First edition. Third printing. Wolfram Media, Inc., 100 Trade Center Drive, Champaign, IL 61820, USA. ISBN I-57955-008-8.

Seznam obrázků

1.1 Elementární automat číslo 220 . . . 6
1.2 Elementární automat číslo 94 . . . 6
1.3 Elementární automat číslo 90 . . . 7
1.4 Elementární automat číslo 30 . . . 7
1.5 Elementární automat číslo 110 . . . 8
1.6 Replicator Universe po 8 krocích . . . 9

2.1 ECB encryption . . . 12
2.2 ECB decryption . . . 12
2.3 CBC encryption . . . 13
2.4 CBC decryption . . . 13
2.5 PCBC encryption . . . 14
2.6 PCBC decryption . . . 14
2.7 OFB encryption . . . 15
2.8 OFB decryption . . . 15
2.9 CTR encryption . . . 16
2.10 CTR decryption . . . 16
2.11 Ukázkový radar chart . . . 20

3.1 Hlavní rozhraní v naší práci . . . 22
3.2 Diagram hierarchie celulárních automatů . . . 23

4.1 KeyExtenderAbstractN . . . 26
4.2 KeyExtenderAbstractD . . . 27
4.3 KeyExtenderCopy . . . 27
4.4 KeyExtenderCheating . . . 28
4.5 KeyExtenderSimpleQuadratic na automat 220 . . . 29
4.6 KeyExtenderSimpleQuadratic na automat 94 . . . 30
4.7 KeyExtenderSimpleQuadratic na automat 90 . . . 30
4.8 KeyExtenderSimpleQuadratic na automat 30 . . . 31
4.9 KeyExtenderSimpleQuadratic na automat 110 . . . 31
4.10 KeyExtenderSimpleQuadratic na omezený automat s 2-okolím . . . 32
4.11 KeyExtenderSimpleQuadratic na cyklický automat s 2-okolím . . . 32
4.12 KeyExtenderSimpleLinear na automat 220 . . . 33
4.13 KeyExtenderSimpleLinear na automat 94 . . . 34
4.14 KeyExtenderSimpleLinear na automat 90 . . . 34
4.15 KeyExtenderSimpleLinear na automat 30 . . . 35
4.16 KeyExtenderSimpleLinear na automat 110 . . . 35
4.17 KeyExtenderSimpleLinear na omezený automat s 2-okolím . . . 36
4.18 KeyExtenderSimpleLinear na cyklický automat s 2-okolím . . . 36
4.19 KeyExtenderSimpleLinear na Amoeba Universe . . . 37
4.20 KeyExtenderInterlaced(10, 0) na automat 220 . . . 38
4.21 KeyExtenderInterlaced(10, 0) na automat 94 . . . 39

4.22 KeyExtenderInterlaced(10, 0) na automat 90 39
4.23 KeyExtenderInterlaced(10, 0) na automat 30 40
4.24 KeyExtenderInterlaced(10, 0) na automat 110 40
4.25 KeyExtenderInterlaced(10, 0) na omezený automat s 2-okolím 41
4.26 KeyExtenderInterlaced(10, 0) na cyklický automat s 2-okolím 41
4.27 KeyExtenderInterlaced(10, 0) na Amoeba Universe 42
4.28 KeyExtenderUncertain na automat 220 43
4.29 KeyExtenderUncertain na automat 94 43
4.30 KeyExtenderUncertain na automat 90 44
4.31 KeyExtenderUncertain na automat 30 44
4.32 KeyExtenderUncertain na automat 110 45
4.33 KeyExtenderUncertain na omezený automat s 2-okolím 45
4.34 KeyExtenderUncertain na cyklický automat s 2-okolím 46

5.1 Vývoj automatu R60 49
5.2 Vývoj automatu R60 zpátky 50

Seznam tabulek

Seznam použitých zkratek

CA	Cellular Automaton (Celulární Automat)
XOR	eXclusive OR (vylučovací disjunkce)
RNG	Random Number Generator (generátor náhodných čísel)
LCG	Linear Congruential Generator (lineární kongruenční generátor pseudonáhodných čísel)
RSA	Rivest-Shamir-Adleman (asymetrický šifrovací algoritmus)
XML	eXtensible Markup Language (značkovací jazyk)
HTML	HyperText Markup Language (jazyk pro popis obsahu webové stránky)

Přílohy