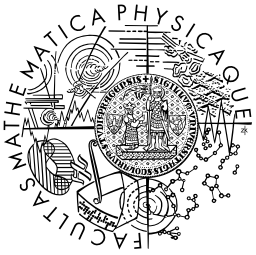


MATEMATICKO-FYZIKÁLNÍ FAKULTA
Univerzita Karlova

BAKALÁŘSKÁ PRÁCE

Jan Kubový

Prostředí pro lifting

Katedra softwaru a výuky informatiky

Vedoucí bakalářské práce: RNDr. Josef Pelikán

Studijní program: Informatika

Studijní obor: IPSS

Praha 2017

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 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: Prostředí pro lifting

Autor: Jan Kubový

Katedra: Katedra softwaru a výuky informatiky

Vedoucí bakalářské práce: RNDr. Josef Pelikán, Katedra softwaru a výuky informatiky

Abstrakt: Cílem práce je vytvořit knihovnu, pomocí které bude možno snadno vytvářet výpočetní sítě a dále s nimi experimentovat. Pojmem výpočetní sítě jsou myšleny algoritmy, které je možné rozdělit na jednoduché části (uzly), ze kterých se následně vytvoří větší výpočetní celek. Příkladem takovýchto výpočetních celků jsou šifrovací algoritmy. Důležité jsou výpočetní sítě, u kterých existují inverzní operace, zejména transformace založené na liftingu. Hlavní důraz této práce je kladen na jednoduchost tvorby nových uzlů a následných zapojení. Univerzálnost je další důležitá vlastnost při práci s touto knihovnou. Takto vytvořená knihovna bude sloužit ke snadné implementaci různých výpočetních sítí a následné experimentování s nimi.

Klíčová slova: wavelety lifting šifrování steganografie

Title: Environment for Lifting

Author: Jan Kubový

Department: Department of Software and Computer Science Education

Supervisor: RNDr. Josef Pelikán, Department of Software and Computer Science Education

Abstract: The aim of the thesis is to create a library that will provide ease way to creating and experimenting with computing networks. The concpet of computing netowork can be explained as algorithms whitch can be devided into small simple parts (nodes). From these nodes the computing netowork can be build. Examples of such computational units are cryptographic algorithms. Most important computing network are these where exist inverse operations. Especially lifting-based transformations are important. The main emphasis of this work is on the simplicity of creating new nodes follows by sipmle nodes connecting. Versatility is another important feature in working with this library. This library will be used to easily implement and experiment with the various computing networks.

Keywords: wavelets lifting encryption steganography

Rád bych na tomto místě poděkoval vedoucímu práce, RNDr. Josefovi Pelikánovi, za rady a veškerý čas, který mi věnoval při zpracování této práce. Dále děkuji všem, kteří mě jakkoli podpořili při studiu.

Obsah

1. Úvod

1.1 Zaměření práce

Cílem této práce je vytvořit knihovnu, pomocí které by bylo snadné implementovat různé výpočetní sítě transformující vstupní signál. Tyto sítě se skládají z jednotlivých výpočetních uzlů či segmentů a definovanými propojeními mezi těmito částmi. Samotný výpočet pak probíhá přeposíláním mezivýsledků mezi jednotlivými částmi výpočetní sítě. Důraz byl kladen na jednoduchou tvorbu nových uzlů a snadné pozměňování již existujících zapojení.

Hlavní podmínkou pro tuto knihovnu byla možnost implementovat výpočetní sítě, u kterých existují inverzní operace. Zejména možnost implementování výpočetních sítí, umožňující transformace založené na liftingu.

Dále bylo v plánu vytvořit různé sítě pro demonstraci funkčnosti a všestrannosti knihovny. Jednou z podmínek práce byla univerzalita knihovny, aby byla schopná pracovat s různými typy signálu. Proto vstupním signálem může být jak zvuk, tak i barvy pixelů obrázku.

1.2 Výpočetní síť

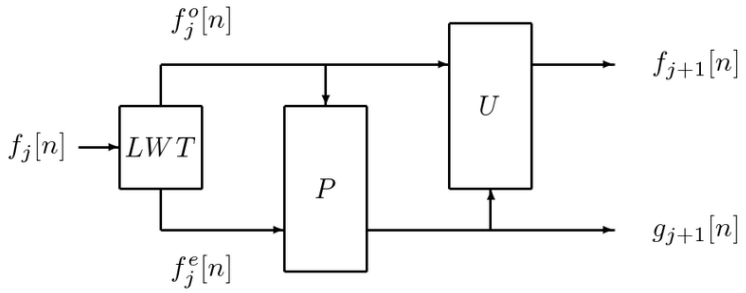
Pro správné porozumění textu je třeba definovat pojem výpočetní síť. Jedná se o algoritmy nebo transformace vstupních dat, které je možné rozdělit na malé části (uzly), ze kterých se následně skládá výsledný výpočetní celek. Typickým příkladem takovéhoto zapojení jsou šifrovací algoritmy. Ty jsou často vytvářeny pomocí malých částí (například jedna část může počítat XOR mezi dvěma signály), které se dále propojují. Tato jednoduchost je u šifrovacích algoritmů z toho důvodu, aby bylo možné vytvořit specializovaný hardware na míru algoritmu.

1.3 Lifting

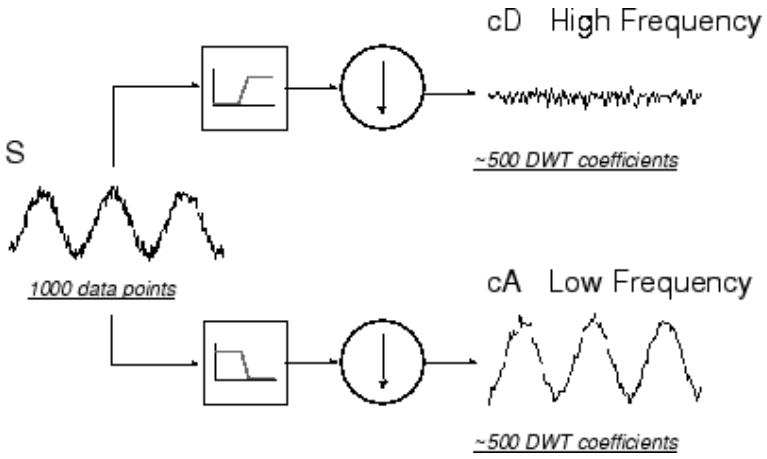
Lifting je výpočetní schéma, které představil Sweldens a kol. (1995), pro práci s diskrétními vlnkovými transformacemi (DWT). Toto schéma se dělí na tři části: rozklad signálu, predikce a aktualizace. V první části se rozděluji signál na dvě části (u diskrétních systémů se často dělí na dvě skupiny se sudým a lichým indexem). Další fází je predikce, kde dochází k první transformaci signálu. Poslední fází je aktualizace, kde dochází k další změně signálu. Tato změna je třeba pro zachování určitých vlastností výstupního signálu po aktualizaci. Toto schéma je znázorněné na obrázku 1.1. Podrobný matematický rozbor tohoto výpočetního schématu je v článku Sweldens (1997). Pro naše účely stačí výše zmíněný obecný popis schématu a podmínka, aby transformace byly invertibilní. To je zajištěno invertibilní operací plus/minus nebo XOR u transformace signálu.

Samotná invertibilita funkce je velice důležitá pro její využití, protože se často používá pro bezeztrátovou transformaci signálu. Příkladem takové transformace je komprimace obrázku standardem JPEG 2000. Jak již bylo zmíněno v úvodu, lifting je schéma zapojení diskrétní vlnkové transformace, které poprvé představil Wim Sweldens. Z této myšlenky vychází obecný lifting autorů Rolón a Salembier

(2007), která je vyobrazena na obrázku 1.1. Příklad rozkladu zvukového signálu pomocí DWT je vidět na obrázku 1.2.



Obrázek 1.1: Obecný lifting - schéma zapojení.



Obrázek 1.2: Příklad rozložení zvukového signálu pomocí DWT.

1.4 Šifra

Šifrování, tedy znemožnění přečtení zprávy pro ty, co neznají tajné heslo, se využívalo již před naším letopočtem. Metody, kterými lze zabezpečit informaci, je možné rozdělit na dvě časové období. První část funguje do první poloviny 20. století, kde se jednalo o nějakou formu transpoziční šifry. To jsou metody, u kterých se zaměňují znaky za jiné, podle předem stanoveného klíče. Zlom nastal v období světových válek, kde se zvyšoval tlak na existenci bezpečné šifry. Pro šifrování se začaly využívat poznatky z matematiky a samotné zašifrování začaly provádět stroje místo lidí. V dnešní době kvůli složitým matematickým operacím není téměř možné provádět šifrování a dešifrování ručně.

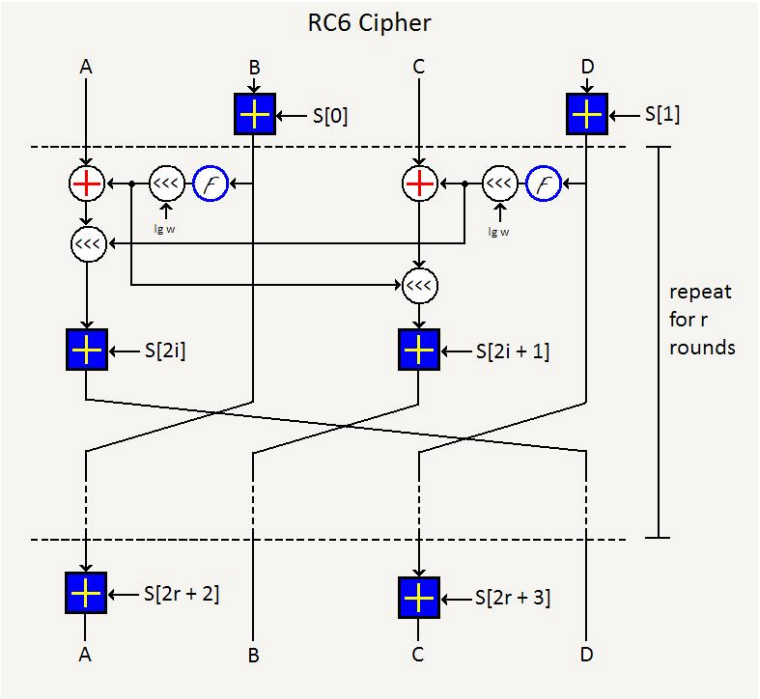
Šifra se skládá ze dvou procesů a to šifrování a dešifrování. Oba tyto procesy lze popsat jako série po sobě jdoucích přesně definovaných kroků. Při šifrování je informace, či signál, transformován pomocí klíče do jiné informace, respektive signálu, u které by mělo být nemožné získat původní data bez znalosti klíče. Dešifrování je inverzní operace k šifrování. Jinými slovy se z již transformované (zašifrované) informace opět získá informace původní. K tomuto procesu je zpravidla zapotřebí původní klíč nebo spárovaný klíč s původním klíčem.

Šifry se dělí na nesymetrické a symetrické, které se dále dělí na blokové a proudové. Existují i další typy, které lze nastudovat v knížce Menezes a kol. (1997). Pro tento projekt stačí znát toto jednoduché rozdělení. Symetrické šifry jsou algoritmy, které pro šifrování a dešifrování používají jeden klíč. U některých algoritmů (např. IDEA) je třeba klíč před dešifrováním upravit, takže se nejedná o naprosto totožný klíč. Na druhou stranu u nesymetrických šifer jsou klíče pro šifrování a dešifrování naprosto odlišné a není možné se znalostí jednoho zkonstruovat druhý. Další dělení symetrických šifer spočívá v tom, jak algoritmy pracují s informacemi. U blokové šifry se pracuje s větším blokem dat pevné délky, které se zpracovávají všechny stejně. Naproti tomu u proudových šifer se počítá s proudem dat, a proto se pracuje s menšími bloky. Transformace se typicky průběžně mění v závislosti na datech nebo pořadí. Obecně jsou proudové šifry jednodušší než blokové.

V této práci se setkáme pouze se šiframi symetrickými, ale bylo by možné pomocí této knihovny vytvořit i asymetrické šifrování.

1.4.1 Blokové šifry

Jak bylo zmíněno výše, blokové šifry pracují nad pevně stanovenou velikostí dat, které transformují. Jde o nejčastěji rozšířené šifry. Jeden z důvodů je fakt, že je možné pomocí blokových šifer vytvořit šifry proudové a tudíž není zapotřebí se soustředit na jiné typy. Pro ilustraci blokové šifry je uveden obrázek 1.3 ukazující zapojení šifrovacího algoritmu RC6. Ten byl publikován v roce 1998 a jeho autory jsou Ron Rivest, Matt Robshaw, Ray Sidney a Yiqun Lisa Yin.



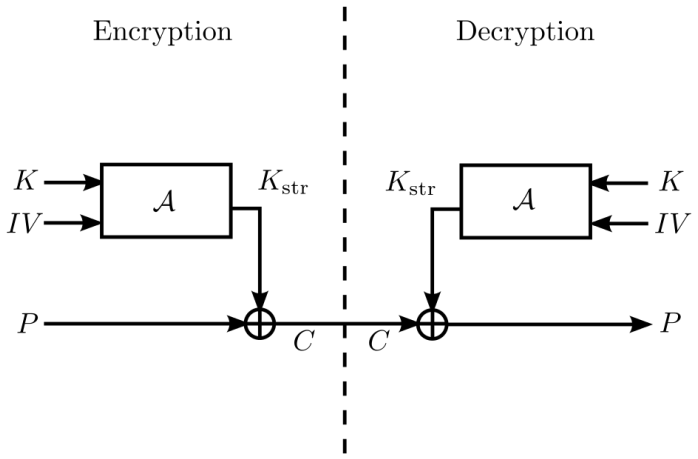
Obrázek 1.3: Schéma šifrovacího algoritmu RC6.

Pomocí obrázku 1.3 lze jednoduše popsat, jak vypadá výpočetní síť. Jednotlivé uzly jsou reprezentovány kruhy a čtverci, kde každý symbol znamená jednoduchou operaci s daty. Čáry mezi těmito objekty ukazují, kudy putují data při výpočtu.

Jak je patrné z obrázku, pro implementování této výpočetní sítě jsou zapotřebí čtyři druhy výpočetních uzlů: sčítání, XOR, levá rotace bitů a uzel pojmenovaný F. Jak pracuje uzel F není pro představu sítě důležité, ale dále v této práci bude popsáno, jak transformuje data.

1.4.2 Proudové šifry

Tento druh šifer je nejčastěji používán u šifrování signálu, u kterého dopředu není známá velikost dat. Ukázka fungování šifrování proudu dat je vyobrazena na obrázku 1.4. Proudové šifry fungují, podobně jako blokové šifry, na pevně stanovené velikosti bloku. Je to z toho důvodu, že je vždy nastavena minimální velikost dat, které se může přenést (například jeden byte). Jak bylo zmíněno v předešlé části, k tvorbě proudové šifry lze využít šifry blokové. Na základě použití blokové šifry je lze rozdělit na tři druhy.



Obrázek 1.4: Ukázka fungování proudové šifry. Vstupní data je transformována operací XOR s proudem dat generovaných pomocí klíče K.

- EBC mode: Jedná se o nejjednodušší přístup. K proudovému šifrování se využívá již existující bloková šifra. Nedochází k žádným změnám na základě dat. Díky tomu je možné i po ztrátě nějaké šifrované části dešifrovat zbytek signálu.
- CBC mode: Tento druh je podobný předchozímu typu s tím rozdílem, že se před šifrováním provede XOR operace s předchozím zašifrovaným blokem. Na začátku se použije takzvaný inicializační vektor, což je blok náhodně vygenerovaných dat. Tento vektor je třeba znát spolu s heslem pro dešifrování. Tato operace způsobí, že po výpadku jednoho bloku není možné dešifrovat zbytek dat, ale na druhou stranu ztěžuje prolomení šifry.
- CTR mode: Poslední zmiňovaný typ nešifruje data přímo. Místo toho šifruje pořadí dat doplněný nulami na velikost bloku. Následuje operace XOR s daty. Tento postup má výhodu při použití, kde po výpadku bloku dat je možné dešifrovat zbytek signálu, pokud víme kolik bloků nedorazilo.

Příkladem použití proudové šifry je komunikace pomocí technologie Wifi používající staré zabezpečení Wired Equivalent Privace (WEP), která využívá šifru RC4. Nevhodnost použití tohoto zabezpečení je uveden v článku Lazar Stošic (2012). Tuto šifru vytvořil Ron Rivest roku 1987 a jelikož není postavená na žádné blokové šifře, nepatří do žádné z výše popsaných skupin. Při šifrování se v závislosti na šifrovaných datech mění klíč, což způsobí, že výpadek nějaké části způsobí nemožnost dešifrovat zbytek.

1.5 Komprese dat

Komprese je proces, při kterém se zmenší objem dat potřebný pro uložení dané informace. Rozlišujeme komprese podle možnosti rekonstrukce původních dat na ztrátové a bezeztrátové. Pojem kompresní poměr značí kolikrát se zmenšila velikost dat pro transformaci. Jak vyplývá z názvu, u ztrátové komprese se vytrácí část dat a komprimovaná data jsou pouze přibližná originálu. Tento druh komprese se nejčastěji využívá u multimediálních dat jako je obraz (video) nebo audio. V případě bezeztrátových kompresí je kompresní poměr horší než u ztrátových kompresí, ale umožňují zrekonstruovat původní data.

1.6 Komprimace obrázků

Komprimace obrázku je proces, při kterém se zmenší objem dat reprezentující daný obrázek. Jelikož jde o kompresi specifických dat, existují také dva druhy komprimace a to ztrátová a bezeztrátová. Při komprimaci obrázků se nejčastěji používá ztrátová komprese, pro větší kompresní poměr. Jedna z nejjednodušších metod komprimace obrázku je zredukování velikosti prostoru barev. Tím se získá obrázek menší velikosti, ale s lehce pozměněnými barvami. Příkladem takovéto komprimace je obrázek 1.5. Nevýhodou je znatelná změna obrázku, proto se spíše používají metody, které se snaží změnit vzhled co nejméně.

My se zaměříme na bezeztrátovou kompresi založenou na diskrétní vlnkové transformaci (DWT), popsané v kapitole Lifting. Konkrétně jde o 2D DWT, protože se transformace použije dvakrát – jednou na řádky a podruhé na sloupce pixelů obrázku. Tímto postupem se zmenší komprimovaný obrázek na čtvrtinu. Při této transformaci se oddělí nízké frekvence (komprimovaný obraz) a vysoké frekvence (detaily, o které komprimovaný obraz přišel).

1.7 Mallatův rozklad

Reprezentace signálu, nejčastěji používaná pro efektivní analyzování obrázku. Jedná se o transformaci signálu, při které se rozdělí na čtyři stejně velké části. Jedna z částí je aproximace původního signálu se čtvrtinovým rozlišením. Ostatní části v sobě zachovávají informace, o které aproximace přišla. Celková velikost obrázku se nezmění, pouze se rozdělí na čtyři části (viz obrázek 1.6) Původně se při tomto rozkladu použila vlnková transformace, ovšem princip rozkladu se dá použít na jakékoli transformace, ke kterým existuje inverzní operace, a která rozdělují vstupní signál na dva výstupní. Počet úrovní rozkladu je určen podle počtu

Obrázek 1.5: Převedení barev obrázku ze tří barevných kanálů na jeden. Tím lze obrázek teoreticky zmenšit na třetinu.

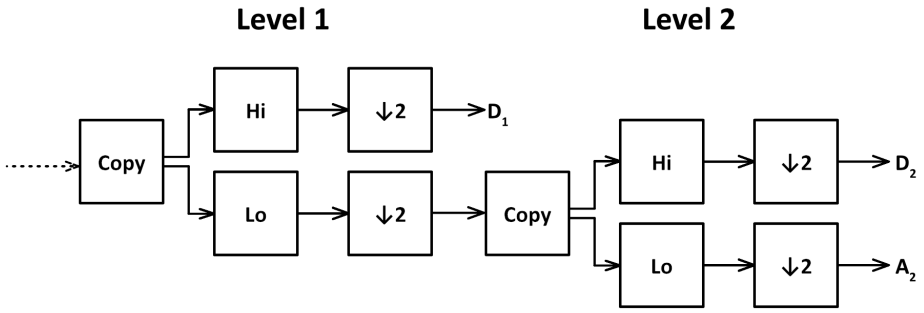
opakování rozkladu na již aproximovaném (zmenšeném) obrázku. Matematický rozbor tohoto rozložení je uveden v článku Mallat (1989).

1.8 Použití knihovny

Práce s knihovnou je rozdělena na dvě části. K tomuto rozdělení bylo přistoupeno k jednoduššímu experimentování s výpočetními sítěmi. První částí tohoto rozdělení jsou zdrojové kódy samotné knihovny Lifting, ve kterých se definují jednotlivé výpočetní a technické uzly. Dále je možné v těchto kódech definovat určité výpočetní části. Druhá část se skládá z definování výpočetních sítí v separátních skriptech, kde se využívají uzly a části definované v knihovně.

Kódy knihovny Lifting jsou psané v jazyce C#. Po jakékoli změně či přidání nového uzlu je třeba knihovnu opět přeložit, čímž vznikne knihovna `Lifting.dll`. Již v základu jsou dostupné různé uzly pro práci s obrázky, soubory a dalšími typy. Dále jsou dostupné výpočetní uzly (sčítání, násobení a další) a technické uzly (například uzel, který duplikuje data do více výstupů).

U druhé části se používá skriptovací jazyk IronPython. Tento programovací



Obrázek 1.6: Dvě úrovně Mallatova rozkladu obrázku. Zkratky Hi (Lo) reprezentuje filtr vysokofrekvenční (nízkofrekvenční), D detaily a A aproximace.

jazyk poskytuje jednoduchou syntaxi při definování výpočetních sítí a možnost pracovat s .NET objekty. V tomto případě umožňuje pracovat s uzly, implementované v knihovně Lifting, na kterou se ve skriptu odkazuje. Toto separátní definování výpočetních sítí umožňuje jejich jednoduchou a rychlou modifikaci (bez nutnosti překladu knihovny). Vytvoření nové sítě spočívá v napsání nového skriptu.

1.9 Programovací jazyk

Jeden z hlavních důvodů při výběru programovacího jazyka byla jeho jednoduchost a rozšířenost. Záměrem bylo co nejvíce usnadnit budoucím uživatelům práci s knihovnou a vytváření vlastních výpočetních uzlů. Proto jsme vybrali programovací jazyk C#, který splňuje stanovené požadavky. Dále tento jazyk umožňuje snadnou práci s grafickým rozhraním, čehož se může využít při vylepšování této knihovny.

Pro vytváření jednotlivých zapojení byl vybrán skriptovací jazyk IronPython, což je programovací jazyk Python, implementovaný v jazyku C#. IronPython je cílen na práci nad platformou .NET a umožňuje snadnou práci s jeho objekty. Ideou bylo vytvořit jednotlivé uzly v knihovně napsané pomocí C# a následné vytváření a experimentování s výpočetními sítěmi, které bude probíhat právě v jazyce IronPython. Tento způsob implementace má výhodu v tom, že po přidání všech uzlů a segmentů stačí knihovnu přeložit pouze jednou. Následné experimentování bude probíhat mimo zdrojový kód knihovny – nebude třeba knihovnu překládat kvůli změně v zapojení.

Následuje ukázka vytvoření stejné výpočetní sítě v jazyce C# a IronPython.
C#:

```
string key = "Key";

var byter = new ConsoleByter();
var rc4 = new RC4(key);
var writer = new ConsoleWriter();

rc4.SetInput(byter.GetOutputBuffer(0), 0);
writer.SetInput(rc4.GetOutputBuffer(0), 0);
Collection.CheckConnections();
```

```
writer.Process();
```

IronPython:

```
keyPhrase = "Key"

byter = ConsoleByter()
rc4 = RC4(keyPhrase)
writer = ConsoleWriter()

rc4.SetInput(byter.GetOutputBuffer(0), 0)
writer.SetInput(rc4.GetOutputBuffer(0), 0)
Collection.CheckConnections()

writer.Process()
```

Jak je vidět, rozdíly v zápisu jsou minimální. Protože jde o programovací jazyk Python, tak není třeba deklarovat typ proměnných nebo používat klíčové slovo „new“ při volání konstruktoru třídy.

2. Popis knihovny

2.1 Popis návrhu

Jak jsme si již psali v sekci 1.1, cílem bylo vytvořit prostředí, ve kterém bude možné vytvářet výpočetní celky. Pro tento účel vznikla následující práce, umožňující tvorbu výpočetních uzlů, které se pak skládají do sítí.

Celý projekt se skládá ze dvou částí – knihovny psané v jazyce C# a definice zapojení uzlů v IronPython. Výpočetní síť je složena z jednotlivých uzlů, které mohou být výpočetní nebo technické, a výpočetních segmentů. Každý uzel obsahuje libovolný počet výstupních bufferů a taktéž libovolný počet ukazatelů na vstupní buffery. Dále každý uzel obsahuje metodu `Process()`, která by měla při jednom zavolání provést výpočet a vložit příslušná data do výstupních bufferů. Slovo měla je z toho důvodu, že je nutné, aby se nějaká data přidala po konečném počtu zavolání. Tato metoda je volána z uzlů, které leží ve výpočetní síti dál a potřebují data z předchozích uzlů. Celý výpočet se pouští z posledního uzlu, který většinou volá metodu `Process()` na předchozích uzlech. Tímto je zajištěno, že se spouštějí výpočty pouze na těch uzlech, které jsou potřeba pro výpočet výsledku. Má-li výpočetní síť více výstupních uzlů, a my potřebujeme pouze jeden výstup, tak se nemusí počítat ty části, které nejsou v cestě výpočtu.

Mezi jednotlivými uzly se pomocí bufferů přeposílají instance třídy `Data`. Tato třída má v sobě interně pole `double[]`, které reprezentuje posílaný signál a `Dictionary<string, object>`, což je slovník, do které je možné ukládat dodatečné informace. Typ dat `double` byl zvolen pro korespondenci se signálem. Je-li to potřeba, tak lze posílat přes `double` menší hodnoty typu `int` nebo `byte`. Veškerá přetypování jsou v kompetenci uživatele knihovny, který vytváří jednotlivé uzly. Tento způsob se využívá u šifrovacích sítí, kde se pracuje s typem `byte`.

Budování výpočetních sítí probíhá, jak již bylo zmíněno, v jazyce IronPython v separátních skriptech. Tento způsob byl zvolen pro jednodušší zápis a také proto, že při změně výpočetní sítě (což se bude dít často při experimentování) není třeba opakovaně překládat knihovnu.

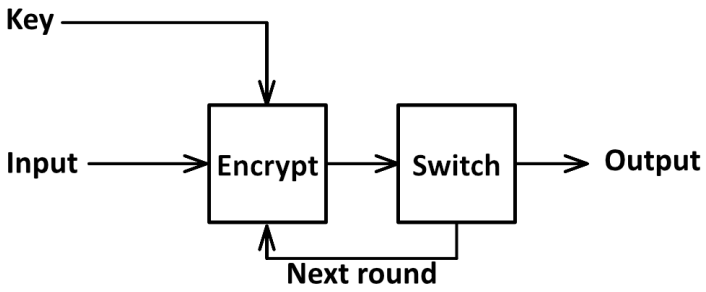
2.2 Data

Veškerá komunikace mezi jednotlivými uzly probíhá přes instance třídy `Data`. Uvnitř se skládá ze dvou druhů dat:

- `double[]`: Pole hodnot typu `double`. Jde o přeposílaný signál, se kterým se pracuje.
- `Dictionary<string, object>`: Slovník, do kterého je možné ukládat dodatečné informace o datech.

Příklad užitečnosti dodatečných informací může být demonstrováno při čtení jednotlivých pixelů obrázku. Pomocí slovníku je možné přeposílat spolu se signálem informaci, kolik procent již bylo odesláno. Tato informace není důležitá z pohledu signálu a výpočtu, ale může se hodit pro informování uživatele, jak velká část již

byla zpracována. Dalším příkladem je informace, kolikrát prošel daný signál kolem šifrování (ilustrace 2.1). U zapojení šifrovacího algoritmu RC6 probíhají data určitým výpočetním úsekem s předem stanoveným počtem kol.



Obrázek 2.1: Ilustrace kola šifrování. Uzel Encrypt pracuje se samotnými daty a Switch bude využívat dodatečné informace o počtu již absolvovaných kol.

V knihovně jsou již napsány operace pro sčítání a odečítání dvou instancí třídy Data a dělení a násobení dat reálným číslem. Tyto metody jsou připraveny pro snazší práce s knihovnou. V případě potřeby dalších operací jsou vytvořeny příslušné uzly, které pracují s daty. Například sčítání v omezeném prostoru hodnot, kde je třeba použít operaci modulo.

Jelikož je třeba nějak zakončit posílání dat, existuje speciální typ dat, který má proměnou se signálem nastavený na `NULL`. Tato data nazýváme prázdná. Pro testování je předem připravena metoda `IsEmpty()`, pomocí které je možné zjistit, jestli se jedná o platná data nebo "zarážku". Prázdná data jsou posílána v případě, že následující uzel žádá o data, ale všechna platná data již byla poslána. Další situací, kde vznikají prázdná data, jsou neplatné operace. Například pokus o sečtení dat s daty prázdnými.

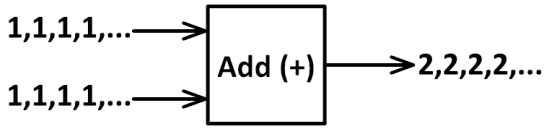
2.3 Buffer

Buffer je hlavní element, přes který se přeposílají data mezi jednotlivými uzly. Interně buffer funguje jako fronta a jednotlivá data se do něj přidávají z uzlu, do kterého daný buffer patří. Pokud nějaký uzel chce získat další instanci třídy `Data`, zavolá metodu `Pop()` na svém vstupním bufferu. Pokud je fronta neprázdná, buffer funguje stejně jako známá datová struktura fronta a vrátí data, která byla přidána nejdříve. Jestliže data nejsou k dispozici, využije se informace, že buffer ví, do kterého uzlu patří, a opakovaně na něm volá metodu `Process()` do té doby, než jsou nějaká data ve frontě k dispozici. S tímto systémem je třeba počítat a při tvorbě nových uzlů vytvořit metodu `Process()` tak, aby po konečném počtu zavolání přidal nějaká data do bufferu. Pokud není další výpočet možný, do bufferu se přidají data prázdná, aby proces výpočtu mohl pokračovat.

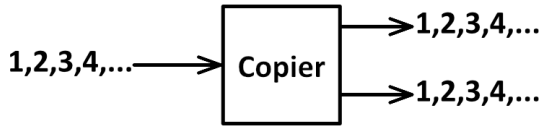
2.4 Uzel

Uzel je hlavní a zároveň nejmenší částí jakékoli výpočetní sítě. Dělíme je do dvou kategorií, podle jejich účelu:

- Výpočetní uzel: Jak vyplývá z názvu, tyto uzly počítají nové hodnoty na základě přijatých dat. Příkladem tohoto typu může být sčítací uzel (2.2). Přijme data ze dvou vstupů, signál sečte a data předá do výstupního bufferu.
- Technický uzel: S tímto typem se setkáme u práce s proudem dat. V případech, kde je tok dat složitější nebo se nějak větví. Příkladem takové situace může být rozdvojení proudu dat, kde je třeba data duplikovat a posílat po dvou různých cestách (2.3). Tento uzel zasahuje do samotného signálu minimálně nebo vůbec, ale umožňuje tvorbu složitějších sítí.



Obrázek 2.2: Příklad výpočetního uzlu, který sčítá data.



Obrázek 2.3: Příklad technického uzlu, který duplikuje data.

Při vytváření nového uzlu je možné stanovit libovolný počet vstupních a výstupních bufferů. Výstupní buffery jsou inicializovány při inicializaci uzlu. Na druhou stranu buffery vstupní jsou přidávány až po inicializaci jako ukazatele na výstupní buffery z jiných uzlů. U uzlu, který čte jednotlivé pixely z obrázku, a posílá je dál, je třeba nastavit 0 vstupních bufferů a 1 výstupních. Čtení pixelů bude probíhat postupně voláním metody `Process()` a při každém zavolání se přidají nově přečtená data do výstupního bufferu.

Každý uzel je potomkem abstraktní třídy `NodeBase`, která obsahuje metodu `Process()`. Tato metoda je reprezentuje jeden výpočet uzlu a měla by po jednom zavolání přidat data do výstupního bufferu. Další metodou je `ResetNode()`. Její zavolání způsobí vymazání dat ve všech výstupních bufferech a dále zavolá metodu `ResetVariables()`. Pomocí této metody by se měly resetovat veškeré proměnné potřebné pro výpočet. Promazávání uzlů se používá, pokud uživatel

chce pomocí stejného zapojení provést více nezávislých výpočtů. Mezi jednotlivými vstupy je třeba obnovit výpočetní síť do původního stavu, aby se jednotlivé vstupy nijak neovlivňovaly.

2.5 Propojení uzlů

Propojování jednotlivých uzlů probíhá tak, že se přidávají ukazatele na výstupní buffery k následujícím uzlům. Jinak řečeno se uzlu přidá ukazatel na buffer, ze kterého bude požadovat data pro svůj výpočet.

```
// Vytvoření jednotlivých uzlů
byter = ConsoleByter()
rc4 = RC4(keyPhrase)

// Zapojení 1. výstupního bufferu v˜uzlu byter
// na 1. vstupní pozici uzlu rc4
rc4.SetInput(byter.GetOutputBuffer(0), 0)
```

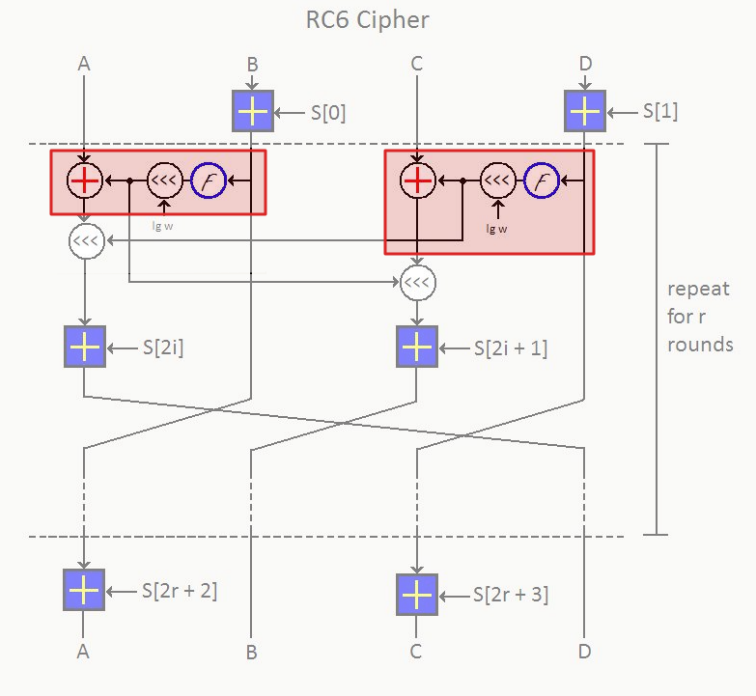
Při propojování se dále musí specifikovat, kolikátý výstupní buffer se má použít, a na jakou pozici se má daný ukazatel uložit. Pomocí statické třídy `Collection` je možné zkontrolovat, jestli jsou všechny ukazatele na vstupní buffery ve všech inicializovaných uzlech validní. Tuto kontrolu je třeba spustit manuálně zavoláním `CheckConnections()`. V některých případech nevadí, jestliže nejsou všechny vstupy uzlů validní, protože je možné spouštět pouze část výpočtu. Jestliže je nějaký ukazatel prázdný (`NULL`), vypíše se výstraha:

```
// {0} je ID uzlu, který nemá validní ukazatel na vstupní buffer
// {1} reprezentuje pořadí vstupního bufferu
"Node {0} input buffer {1}: NULL"
```

2.6 Výpočetní segment

V této části práce si popíšeme, co je myšleno pojmem výpočetní segment. Jedná se o část výpočetní sítě, která má stejné rozhraní jako jednoduchý uzel (je taktéž potomkem třídy `NodeBase`). Rozdíl je v tom, že se pro výpočet použijí již existující uzly. Vytvoří se část výpočetní sítě, se kterou se pracuje úplně stejně jako s uzlem.

Pomocí výpočetních segmentů je možné zjednodušit stavbu výpočetních sítí a odstranit z nich duplicitní části. Příkladem může být šifrovací algoritmus RC6, který jsme si představili v úvodní části. V tomto zapojení můžeme vytvořit jeden výpočetní segment, který se opakuje (viz Obrázek 2.4). Tímto způsobem je možné pokračovat dále a vytvořit další výpočetní segment, který bude korespondovat s jedním šifrovacím kolem algoritmu (viz Obrázek 2.5). Takto vytvořené výpočetní segmenty ulehčí výrobu výpočetní sítě.



Obrázek 2.4: RC6 šifrovací algoritmus se zvýrazněnými částmi možných výpočetních segmentů (červené obdélníky).

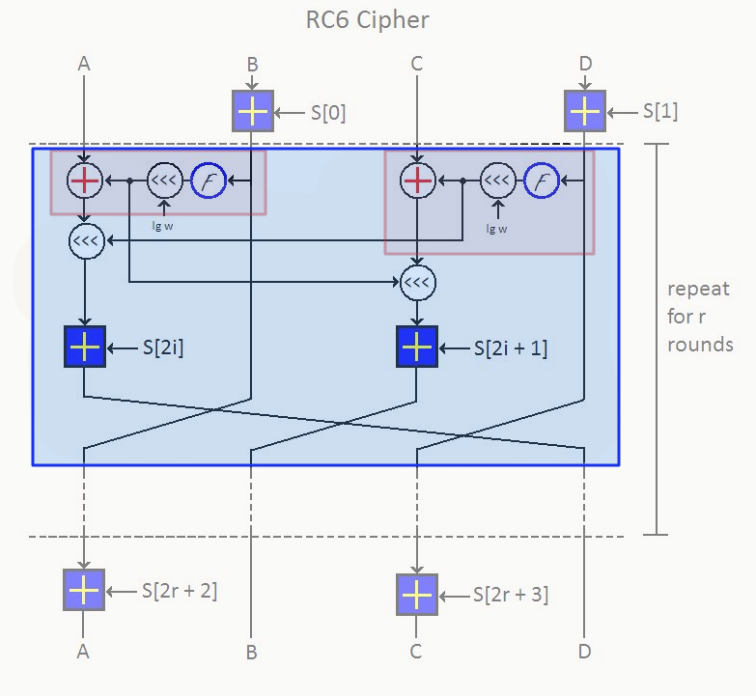
2.7 Výpočetní síť

Výpočetní síť se nazývá finální zapojení jednotlivých uzlů a výpočetních segmentů do jednoho celku. Jedná se o zapojení, které provádí všechny kroky výpočtu. Pro ukázku je na obrázku 2.6 ilustrováno zapojení šifrovacího algoritmu RC4, které čte data z konzole a výsledek opět vypisuje na konzoli. Klíč je předán jako argument programu při spuštění. Tvorba sítí se liší od ostatní práce s knihovnou v tom, že se zapojení definuje v samostatných skriptech v jazyce IronPython, jak bylo nastíněno v kapitole 1.9.

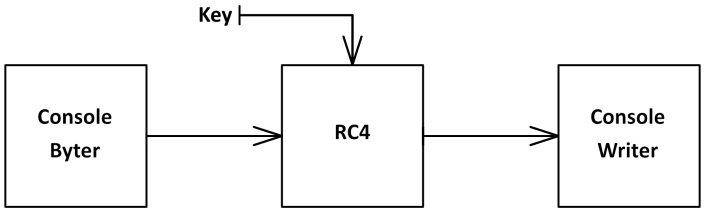
2.8 Třída NodeBase

`NodeBase` je abstraktní třída, která představuje jednu část výpočetní sítě. Tento způsob — dědičnost — jsme zvolili, aby samotné psaní nového uzlu bylo co nejsnazší a bylo potřeba doplnit minimum nového kódu. Názorné ukázky si ukážeme v další kapitole. Třída obsahuje dva kontejnery `inputBuffers` a `outputBuffers`, které v sobě případně drží ukazatele na jednotlivé vstupní a výstupní buffery. Dále má v sobě metody na práci s uzlem.

- `public NodeBase(int inputBufferCount, int outputBufferCount)`: Konstruktor, který vytvoří kontejnery `inputBuffers` a `outputBuffers`. Dále inicializuje všechny výstupní buffery.
- `public Buffer GetOutputBuffer(int index)`: Funkce, která vrátí odkaz na výstupní buffer s požadovaným indexem. Je používána při propojování jednotlivých uzlů.



Obrázek 2.5: RC6 šifrovací algoritmus se zvýrazněným jedním kolem šifrování (modrý obdélník).



Obrázek 2.6: Výpočetní síť šifrovacího algoritmu RC4.

- `public void SetInput(Buffer buffer, int inputBufferIndex)`: Metoda přiřazující zadaný buffer mezi vstupní buffery uzlu.
- `public void ResetNode()`: Metoda vymaže obsah všech výstupních bufferů a zavolá metodu `ResetVariables()`.
- `internal void ResetVariables()`: Metoda s prázdným tělem, která se používá k vymazání obsahu, případně opětovnému inicializování potřebných proměnných v uzlu.
- `public abstract void Process()`: Abstraktní metoda, kterou je třeba doplnit při tvorbě nového uzlu. Jedná se o samotný výpočet uzlu.
- `public void CheckConnections()`: Metoda, která kontroluje, jestli ukazatele na všechny vstupní buffery nejsou `NULL`.

Z výše uvedených metod vidíme, že jediné, co je nutně potřeba dopsat při

tvorbě nového uzlu, je metoda `Process()` a specifikovat konstruktor. U jednoduchých uzlů je obvyklé, že se specifikují pouze tyto dvě části.

3. Práce s knihovnou

V této kapitole si ukážeme, jak se pracuje s knihovnou. Popíšeme si postup od vytvoření nového uzlu až po vytvoření konkrétní výpočetní sítě. Ukážeme si, co všechno knihovna poskytuje pro uživatele, na co je dobré myslet a dát si pozor. Veškeré části výpočetní sítě (myšleno uzly nebo výpočetní segmenty) musejí být potomkem abstraktní třídy `NodeBase`, aby je bylo možné propojit.

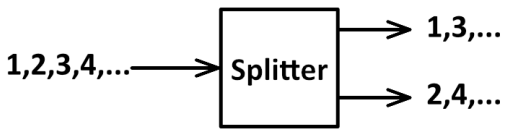
3.1 Uzel

V této části si popíšeme dva konkrétní příklady uzlů, které jsou již napsané v knihovně. Ukážeme si na těchto dvou příkladech, jak vypadá tvorba jednoduchého výpočetního uzlu.

```
public class Splitter : NodeBase
{
    public Splitter(int outputBufferCount) :
         base(1, outputBufferCount) { }

    public override void Process()
    {
        for (int bufferIndex = 0;
                  bufferIndex < outputBuffers.Length;
                    bufferIndex++)
        {
            outputBuffers[bufferIndex].Push(inputBuffers[0].Pop());
        }
    }
}
```

Jedná se o uzel `Splitter`, který posílá data, která přijímá postupně do zadaného počtu výstupů. Pokud bychom zvolili dva výstupy, tak data s lichým pořadím poputují do prvního výstupního bufferu a sudé do druhého (3.1).



Obrázek 3.1: Technický uzel, který rozděluje vstupní data do dvou výstupů.

- Každý uzel musí být potomkem abstraktní třídy `NodeBase`

- Je třeba dopsat vlastní konstruktor. Implicitní konstruktor neví, s jakými parametry se má zavolat konstruktor v abstraktní třídy `NodeBase`. Zde se specifikuje, kolik bude mít uzel vstupních a výstupních bufferů.
- V poslední řadě je třeba přepsat metodu `Process()`, která se stará o samotný výpočet uzlu. V tomto případě se jedná o jednoduchý cyklus, který do každého výstupního bufferu pošle data z bufferu vstupního.

U druhého příkladu se zaměříme na složitější uzel, který bude číst barvy jednotlivých pixelů zadaného obrázku. Z nich vytvoří instance třídy `Data`, které bude posílat dál ke zpracování. Z důvodu, jak fungují jednotlivé uzly, je třeba zajistit, aby vstupní uzly při konečném počtu zavolání metody `Process()` poslali nějaká data. Pro tuto potřebu existují takzvaná prázdná data, které fungují jako zarážka (viz sekce Data).

```
public class ImagePixelator : NodeBase
{
    private const int ZERO_SIGNAL_COUNT = 4;
    Bitmap image;
    int x, y;
    bool finished;

    public ImagePixelator() : base(0, 1)
    {
        finished = true;
    }
    public void SetProperties(Bitmap image)
    {
        this.image = image;
        x = y = 0;
        finished = false;
    }
    public override void Process()
    {
        if (finished)
        {
            outputBuffers[0].Push(new Data(null));
        }
        else
        {
            double[] signal = GetSignal(x, y);
            outputBuffers[0].Push(new Data(signal));
            NextPixel();

            if (finished)
            { ... }
        }
    }
    private void NextPixel()
    { ... }
    private double[] GetSignal(int x, int y)
```

```
    {
        Color pixelColor = image.GetPixel(x, y);
        return new double[] { pixelColor.R,
                              pixelColor.G,
                              pixelColor.B };
    }
    internal new void ResetVariables()
    {
        image = null;
        x = y = 0;
        finished = true;
    }
}
```

V první řadě si musíme popsat jednotlivé proměnné používané v popisované třídě `ImagePixelator`. Dále si nastíníme, co dělají jednotlivé metody.

- `ZERO_SIGNAL_COUNT`: Počet nulových signálu poslaných po konci obrázku. Používá se pro dopočítání konce výpočtu, kde se pracuje s posledními pixely.
- `image`: Obrázek, u kterého čteme jednotlivé pixely.
- `x, y`: Souřadnice čteného pixelu.
- `finished`: Pravdivostní hodnota udávající, jestli se přečetly všechny pixely obrázku, nebo jestli ještě nebyl přidán žádný obrázek. Tím se vyhneme čtení pixelu z neinicializovaného obrázku, protože je po vytvoření uzlu tato hodnota nastavena na `True`.

V tomto příkladu je použita proměnná `ZERO_SIGNAL_COUNT`, která reprezentuje počet nulového (nikoli prázdného) signálu. Při různých transformacích obrázku se počítá s více pixely (například průměr dvou po sobě jdoucích barev). V tomto případě potřebujeme ještě nějaká neprázdná data po přečtení všech pixelů, aby bylo možné dopočítat konec. Tímto se simuluje práce s analogovým signálem, kde začátek i konec je na úrovni 0.

- `public void ImagePixelator()`: Konstruktor, který nastaví hodnotu proměnné `finished`. Dále jsou zde uvedeny parametry, se kterými se zavolá konstruktor ze třídy `NodeBase`. Typicky nemá vstupní uzel žádný vstupní buffer, ale má jeden výstupní buffer.
- `public void SetProperties(Bitmap image)`: Přiřazení ukazatele na instanci obrázku, se kterým se bude pracovat.
- `pubic override void Process()`: V případě `False` hodnoty v proměnné `finished` plní výstupní buffer prázdnými daty. V opačném případě vytvoří data z aktuálního pixelu, které odešle. Pokud se jedná o poslední pixel, odešle se zadaný počet nulových signálů.
- `private void NextPixel()`: Nastavení souřadnic `x` a `y` na další pixel. V případě konce obrázku nastaví `finished = True`.

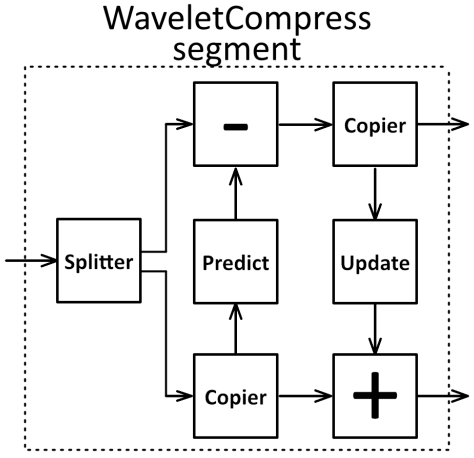
- `private double[] GetSignal(int x, int y)`: Vytvoří nové pole s hodnotami, které korespondují s barvou pixelu. Poté toto pole vrátí.
- `internal new void ResetVariables()`: Tato metoda slouží k vymazání proměnných uzlu a tím ho připraví na další možný vstup.

U metody `Process()` je třeba zdůraznit její chování po odeslání veškerého signálu. Při každém zavolání přidá prázdná data do výstupního bufferu, ze kterého je může použít další uzel v pořadí. Tím je zajištěno, že se výpočet na nějakém uzlu „nezasekne“ při čekání na data. Na toto chování je třeba myslet při vytváření vstupních uzlů.

3.2 Výpočetní segment

Zde si popíšeme, jak vytvořit výpočetní segment z již existujících uzlů. Postup je podobný jako u nového uzlu, ale jsou zde jisté rozdíly. Jeden z nich je udávaný počet vstupních bufferů, který musí být vždy 0. Je to způsobeno tím, že uzly používané v segmentu již mají vlastní vstupní buffery, které je třeba použít. Tím jsme se dostali k druhému rozdílu, kdy je třeba zajistit, aby se po zavolání metody `SetInput(Buffer buffer, int inputBufferIndex)` správně přiřadily ukazatele na buffery. V jednoduchém uzlu je o to postaráno již napsanou metodou v abstraktní třídě `BaseNode`. V tomto případě potřebujeme, aby se buffery přiřazovaly k uzlům v segmentu.

Popisovaný segment slouží k diskrétní vlnkové transformaci, která se používá u komprimace obrázků. Schéma zapojení je vyobrazeno na obrázku 3.2. Konkrétně je využíván u Mallatova rozkladu, kde je tento segment použit třikrát.



Obrázek 3.2: Schéma zapojení výpočetního segmentu reprezentující základní diskrétní vlnkovou transformaci.

```
public class WaveletCompress : NodeBase
{
    private Buffer outputBuffer1;
```

```
private Buffer outputBuffer2;
private NodeBase inputNode;
private List<NodeBase> nodes;
public WaveletCompress() : base(0, 2)
{
    BuildConnections();
}
private void BuildConnections()
{
    var splitter = new Splitter(2);
    var substract = new Substract();
    var p = new P();
    ...
    var add = new Add(2);

    nodes = new List<NodeBase>();
    nodes.Add(splitter);
    ...
    nodes.Add(add);

    outputBuffer1 = copier2.GetOutputBuffer(1);
    outputBuffer2 = add.GetOutputBuffer(0);
    inputNode = splitter;

    substract.SetInput(splitter.GetOutputBuffer(1), 0);
    substract.SetInput(p.GetOutputBuffer(0), 1);
    ...
    add.SetInput(copier1.GetOutputBuffer(1), 1);
}
public override void Process()
{
    outputBuffers[0].Push(outputBuffer1.Pop());
    outputBuffers[1].Push(outputBuffer2.Pop());
}
public new void SetInput(Buffer buffer, int inputBufferIndex)
{
    if (inputBufferIndex == 0)
    {
        inputNode.SetInput(buffer, 0);
    }
    else
    {
        Logger.Error("Node {0} can't set input number {1}",
                     ID, inputBufferIndex);
    }
}
public new void ResetNode()
{
    for (int i = 0; i < nodes.Count; i++)
    {
        nodes[i].ResetNode();
```

```
        }
    }
}
```

- `outputBuffer1-2`: Jedná se o ukazatele na výstupní buffery posledních uzlů v zapojení. Jejich hlavní použití je v metodě `Process()`.
- `inputNode`: Ukazatel na vstupní uzel. Používá se při přiřazování vstupního bufferu do segmentu.
- `nodes`: Kolekce všech uzlů v segmentu. Je třeba si pamatovat všechny uzly, v případě potřeby resetovat segment.

Veškeré výše popsané proměnné jsou zde z toho důvodu, aby se se segmentem pracovalo stejně jako s jednoduchým uzlem. Veškeré chování by mělo být transparentní pro uživatele a nemělo by se nijak lišit, jestli se pracuje s jednoduchým uzlem nebo s velkým výpočetním segmentem.

- `public WaveletCompress()`: Konstruktor třídy, u kterého se volá metoda `BuildConnections()`. Můžeme si všimnout, že specifikovaný nulový počet vstupních bufferů.
- `private void BuildConnections()`: Inicializování potřebných uzlů a vytvoření zapojení segmentu. Kromě toho jsou všechny použité uzly přidávány do kontejneru `nodes` a jsou inicializovány proměnné `outputBuffer1-2` spolu s `inputNode`.
- `public override void Process()`: Tato metoda je velice jednoduchá. Jediné, co dělá, je předání instancí třídy `Data` z předem uloženého bufferu posledních uzlů do výstupních bufferů segmentu.
- `public new void SetInput(...)`: Metoda přiřazující vstupní buffer do vstupního uzlu.
- `public new void ResetNode()`: Jedná se o resetování všech uzlů v segmentu. Zde se využije kontejner se všemi uzly.

Z výše uvedených metod jsou patrné tři rozdíly oproti tvoření nového uzlu. V první řadě je konstruktor neprázdný a je třeba v něm inicializovat používané uzly a vytvořit konečné zapojení celého segmentu. Dále je důležité se postarat o vstupní a výstupní buffery v tom smyslu, aby navenek vše fungovalo stejně jako u uzlu. V poslední řadě se musí přepsat metody `SetInput(...)` a `ResetNode()`, protože vykonávají jinou činnost, než je uvedeno v abstraktní třídě `NodeBase`.

3.3 Výpočetní síť

V této části práce se zaměříme na vytváření finální výpočetní sítě a popíšeme si rozdíly oproti předcházejícím příkladům z tvorby uzlů a segmentů. Jak již bylo napsáno v úvodu, vytváření výpočetních sítí probíhá ve skriptovacím jazyce IronPython. Proto je třeba již mít napsané veškeré uzly a případné výpočetní

segmenty ve zdrojových kódech knihovně, která poté byla přeložena. Tím získáme soubor knihovny, který budeme ve skriptech využívat. Jednotlivé části si popíšeme na příkladu. Rozebereme si proudovou šifru RC4, o které jsme psali v úvodu. Schéma zapojení uzlů je vidět na obrázku 2.6.

```
from LiftingReferences import *
import sys
```

Důležitý je první řádek odkazující na již vytvořený skript, který slouží k naimportování popisované knihovny. To je třeba z důvodu zpřístupnění jednotlivých uzlů a segmentů z tohoto skriptu.

```
Logger.Info("Starting RC4")
if (len(sys.argv) <= 1):
    Logger.Error("No key phrase given")
    sys.exit()
keyPhrase = sys.argv[1]
```

Zde je patrná první práce s knihovnou. Voláním metody `Info(...)` ze statické třídy `Logger`, která je součástí popisované knihovny. Dále pokračuje kontrola počtu parametrů a případný konec programu v případě nepředání dostatečného počtu parametrů. Do proměnné `keyPhrase` je přiřazen první argument programu reprezentující klíč, který se má použít v RC4 šifře.

```
byter = ConsoleByter()
rc4 = RC4(keyPhrase)
writer = ConsoleWriter()
rc4.SetInput(byter.GetOutputBuffer(0), 0)
writer.SetInput(rc4.GetOutputBuffer(0), 0)
Collection.CheckConnections()
```

První tři řádky inicializují instance třech různých uzlů, které jsou následně přiřazeny do proměnných. Ve spodní části je napojení vstupních bufferů uzlů `rc4` a `writer`. Na posledním řádku je spuštěna kontrola zapojení, jestli není nějaký vstupní buffer jakéhokoli uzlu prázdný. Tato kontrola není nutná, ale rozhodně doporučovaná.

```
writer.Process()
Logger.Success("RC4 finished successfuly")
```

Zavoláním metody `Process()` na posledním uzlu `writer` dojde ke spuštění samotného výpočtu. Konec programu je pouze vypsání hlášky o ukončení výpočtu. Tyto čtyři popsané části tvoří celé zapojení výpočetní sítě. Upravuje se pouze skript popisující danou síť.

4. Implementované výpočetní sítě

V následující kapitole si popíšeme konkrétní příklady již naimplementovaných výpočetních sítí. Ukážeme si zapojení pro transformaci obrázku, konkrétně Mallatův rozklad pomocí diskrétní vlnkové transformace, a dvě šifrovací zapojení. Zaměříme se na popsání algoritmů, použitých uzlů a důležitých částí zapojení. Dále uvedeme konkrétní příklady použití, respektive výstupy výpočtů. U šifrovacích algoritmů je třeba ověřit správné implementování zapojení pomocí testovacích vstupů, u kterých je předem znám výsledek výpočtu. V případě shody předpokládaného výsledku se vzorem lze považovat danou implementaci za funkční. Při transformaci obrázku se pracuje s desetinnými čísly, tudíž se využije celá hodnota typu `double`. Jelikož jsou data pevně nastavena na tento typ, tak se v některých případech ořezávají na menší typy. Například u šifrování se pracuje typem `byte`, ale data zůstávají stejná. Pouze se v každém uzlu přetypují.

4.1 Mallatův rozklad

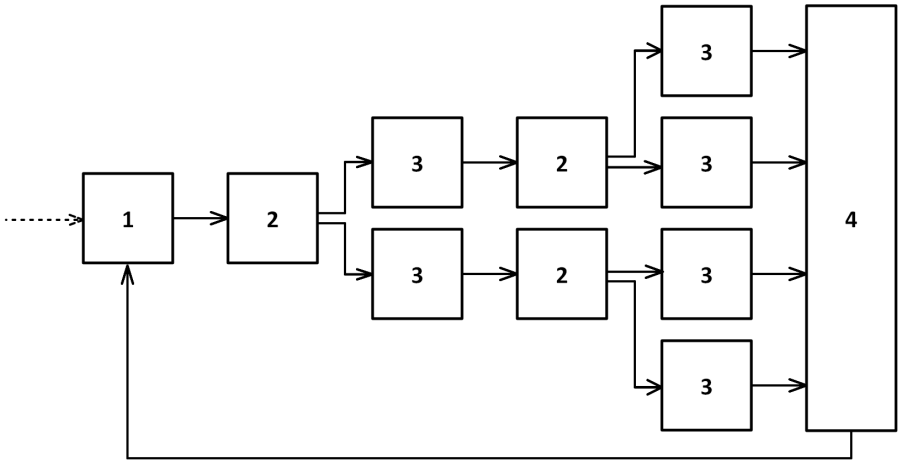
Pro popis fungování Mallatova rozkladu musíme v první řadě popsat uzly, které se v tomto zapojení používají, a jak pracují. Samotný Mallatův rozklad není závislý na konkrétním algoritmu transformace obrázku. Princip spočívá v použití této transformace určitým způsobem, který vytvoří výsledný obrázek. V našem případě je použita diskrétní vlnková transformace, která byla představena v úvodu této práce. Hlavní podmínkou, kterou musí takováto transformace splňovat, je rozdělení jednoho vstupu na dva výstupy a existence inverzní operace. Je možné použít například Haarovu vlnkovou transformaci.

4.1.1 Popis částí

- `JoinerSequential`: Jedná se o uzel, který má více vstupů, ze kterých přijímá data. Začíná číst z prvního vstupního bufferu a po zavolání metody `NextInput()` dojde k přepnutí k dalšímu vstupnímu bufferu.
- `WaveletCompress`: Reprezentuje diskrétní vlnkovou transformaci obrázku.
- `ImageDiagonalFlipper`: Tento uzel si ukládá přijímaná data do 2D pole. Je třeba v uzlu nastavit, jakou velikost má toto pole nabývat. Po přijetí všech dat k zaplnění pole (plní se po řádcích) dojde k odeslání všech dat čtených po sloupcích. Výsledek je stejný, jako by se obrázek převrátil podél diagonály a odeslaly se data opět po řádcích. Jelikož se tento uzel používá před vlnkovou transformací, opět se přistoupí k poslání určitého počtu nulových dat pro dopočítání konce (viz uzel ImagePixelator).
- `MallatDecompositionCollector`: Uzel, který se nachází jako poslední v zapojení. V sobě má interně uložená veškerá data obrázku, řídí výpočet sítě a případně posílá data opět do sítě při počítání dalšího kola transformace.

Průběh výpočtu je řízen z metody `Process()` nacházejícím se ve výpočetním segmentu `MallatDecomposition`. V této metodě se nastavují správné velikosti

polí v instancích třídy `ImageDiagonalFlipper` a přepínání vstupu, ze kterého se čtou data u uzlu `JoinerSequential`. Po úspěšném dokončení výpočtu požadovaného počtu kol jsou zavolány metody na uložení výsledku do souborů.



Obrázek 4.1: Výpočetní segment MallatDecomposition.

Na obrázku 4.1 je ukázané schéma zapojení jednotlivých uzlů uvnitř výpočetního segmentu `MallatDecomposition`.

1 `JoinerSequential`

2 `WaveletCompress`

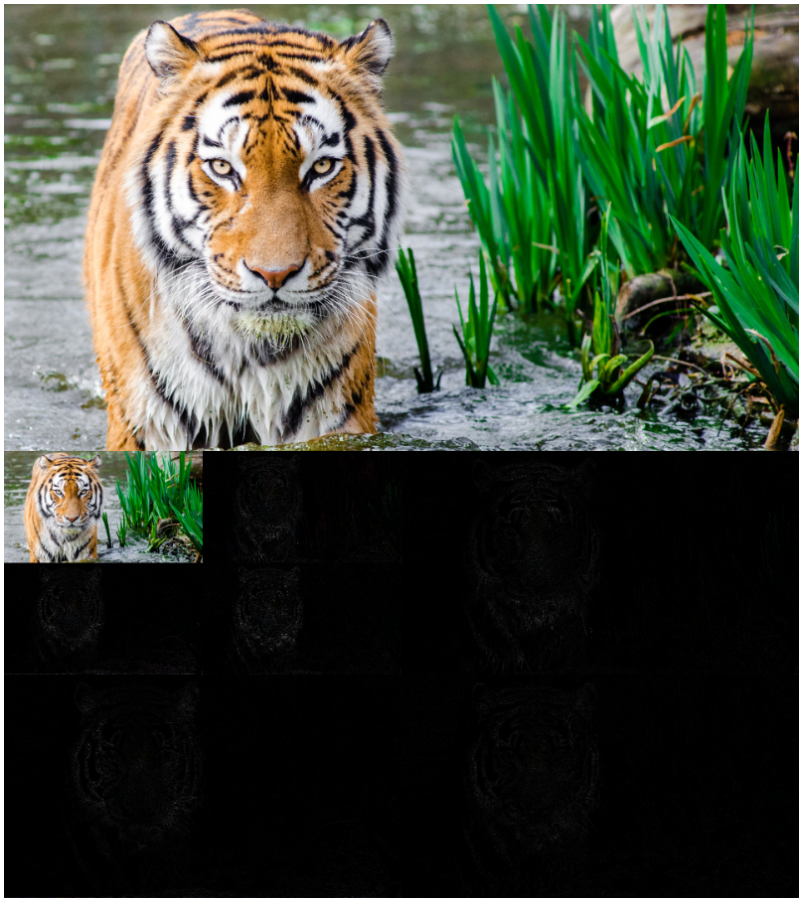
3 `ImageDiagonalFlipper`

4 `MallatDecompositionCollector`

Je zde patrné propojení posledního uzlu (4) s prvním (1). To je používané při více kolech transformace, kde se odesílají nasbíraná data opět k výpočtu.

4.1.2 Ukázka výstupu

Je třeba upozornit na jednu zvláštnost při práci s tímto zapojením. Výsledek je totiž ukládán dvěma způsoby. První z nich je ukládání do obrázku ve formátu PNG, který prošel ořezáním barev. Je to způsobeno tím, že není možné uložit záporné hodnoty jednotlivých barev, které mohou výpočtem vzniknout. Dále veškeré hodnoty nejsou celá čísla. Proto se přistoupilo ke zmíněnému ořezání, kde se použije pouze dolní celá část čísla a dále jsou nastaveny hranice velikosti jednoho barevného kanálu na 0 až 255. Bylo možné použít operaci modulo na jednotlivé barvy, ale tím by se nevyřešily desetinné hodnoty, a hlavně by vznikly barevné artefakty. Tento výstup je pro ukázku fungování algoritmu. Protože by nebylo možné opět zrekonstruovat původní obrázek z takto upraveného výstupu, je vytvořen ještě textový soubor, ve kterém jsou uloženy nezměněné vypočítané



Obrázek 4.2: Příklad mallatova rozkladu obrázku se dvěma úrovněmi pomocí diskrétní vlnkové transformace

hodnoty a na posledních dvou řádcích je přidáno původní rozlišení obrázku. Pomocí tohoto textového souboru je možné spustit inverzní operaci a opět získat původní obrázek. Tím se dokáže existence inverzní transformace.

Pro představu je přiložen obrázek 4.2 ukazující v horní polovině původní obrázek a ve spodní části výsledek Mallatova rozkladu druhé úrovně. Obrázek se člení na čtyři části. V levé horní části se nachází komprimovaný obrázek. V levé spodní a pravé horní části jsou (vysoké frekvence) detaily obrázku v horizontálním a vertikálním směru. Nakonec v pravém dolním rohu jsou uloženy zbylé detaily. Druhá úroveň transformace je patrná tím, že levá horní část je opět členěna, jak bylo popsáno výše. Můžeme nahlédnout na zmíněných detailech, že se jedná převážně o černo-bílé barvy, případně velice nevýrazné barvy. Z tohoto pozorování lze usoudit, že barvy hrají hlavní roli u uložení obrázku a případné detaily jsou pouze informace v šedých odstínech, či nevýrazných barvách.

4.1.3 Spojovací uzel

Pro ilustraci si ukážeme, jak vypadá uzel `JoinerSequential`, který se stará o odeslání dat obrázku v převrácené podobě podél diagonály.

```
public class JoinerSequential : NodeBase
{
    int inputBufferIndex;
```

```
    bool finished;
    public JoinerSequential(int inputBufferCount) :
            base(inputBufferCount, 1)
    {
        inputBufferIndex = 0;
        finished = false;
    }

    public override void Process()
    {
        if (finished)
        {
            outputBuffers[0].Push(new Data(null));
            return;
        }
        Data result = inputBuffers[inputBufferIndex].Pop();
        if (result.IsEmpty() &&
            inputBufferIndex == inputBuffers.Length)
        {
            finished = true;
        }
        outputBuffers[0].Push(result);
    }
    protected new void ResetVariables()
    {
        inputBufferIndex = 0;
        finished = false;
    }
    public void NextInput()
    {
        inputBufferIndex++;
    }
}
```

Opět se můžeme přesvědčit, že tvorba takto jednoduchých uzlů není nijak složitá. Metody `NextInput()` a `ResetVariables()` netřeba více popisovat. Konstruktor počítá s libovolným počtem vstupních bufferů a není omezen na přesný počet. Jediná delší metoda je `Process()`, ve kterém probíhá kontrola, jestli se již nepřečetla všechna data. Při výpočtu se nedělá nic jiného, než že se zkontroluje, jestli se již v minulosti nedošlo na konec. V tom případě dál odešle prázdná data. V případě opačném zkontroluje, jestli přijatá data nejsou prázdná. Pokud opravdu jde o data prázdná, a byla načtena z posledního vstupního bufferu, označí se, že se došlo ke konci. V závěru se odešlou přijatá data.

4.1.4 Skript zapojení Mallatova rozkladu

Nyní si představíme skript definující toto zapojení. U popisování výpočetní sítě se opět zaměříme pouze na důležité části kódu. Není třeba popisovat každý řádek zvlášť. Proto budou vynechány některé kontroly argumentů, které pro popis výpočetní sítě nehrají roli. Jestliže rozlišení obrázku nedovoluje rozdělit obrázek

na požadovaný počet úrovní, je přidán potřebný počet řádků nebo sloupců. Tím se obrázek zvětší, ale po inverzní operaci je opět zmenšen na původní velikost. O této změně je uživatel informován.

```
from LiftingReferences import *
from TextfileAxuiliaryMethods import *
from ImageAxuiliaryMethods import *
clr.AddReference('System.Drawing')
from System.Drawing import Bitmap
import sys

...

imageName = sys.argv[1]
image = Bitmap(imageName + ".png")
outputFileName = imageName + "_mallat"
originalWidth = image.Width
originalHeight = image.Height
levels = int(sys.argv[2])

...

(newWidth, newHeight) = GetProperResolution(originalWidth,
     originalHeight, levels)
if (originalWidth != newWidth or originalHeight != newHeight):
        Logger.Warning("Image resolution doesn't allow {0} level
             decomposition\nImage size will be expanded to
             {1}x{2}".format(levels, newWidth, newHeight))
        image = Resize(image, newWidth, newHeight)

...

pixelator = ImagePixelator()
mallat = MallatDecomposition()
mallat.SetInput(pixelator.GetOutputBuffer(0), 0)
Collection.CheckConnections()
pixelator.SetProperties(image)
mallat.SetProperties(outputFileName, levels, image.Width, image.Height)

mallat.Process()

AppendTextfileFromList(outputFileName, [originalWidth, originalHeight])
```

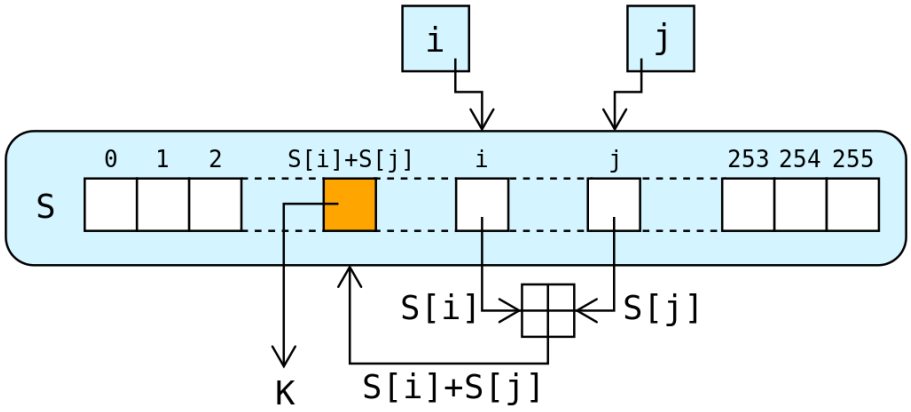
Ukázkový kód jsme rozdělili prázdnými řádky na několik části, které si popíšeme. Jelikož se jedná pouze o ukázku, některé řádky byly vynechány a jsou nahrazeny třemi tečkami. V první části se přidává reference na knihovnu, jiné skripty, které v sobě obsahují dodatečné metody, a elementy, se kterými se dále pracuje. Druhá část je o připravení proměnných, se kterými se bude dál pracovat. Konkrétně instance zpracovávaného obrázku, rozlišení a počet úrovní rozkladu. Další část se zabývá počítáním, jestli rozlišení obrázku dovoluje rozklad na daný

počet úrovní. Pokud tomu tak není, je obrázek rozšířen a uživatel je informován o této akci. Nyní se dostáváme k samotné výpočetní síti. Konkrétně jsme se dostali k inicializaci jednoho uzlu, který tvoří ze čteného obrázku jednotlivá data, a dále k výpočetnímu segmentu, který byl popsán v kapitole výše. Následuje specifikace zapojení a kontrola zapojení vstupních bufferů. Po tomto kroku se předávají parametry jednotlivým uzlům, jako je instance obrázku nebo počet úrovní dekompozice. Řádek `mallat.Process()` spustí požadovaný výpočet. Na posledním řádku je zavolána metoda, která připíše k vytvořenému textovému souboru dva řádky s rozlišením původního obrázku. Tato informace je třeba při inverzní operaci.

4.2 RC4

Nyní se přesuneme od transformací obrázků k šifrovacím algoritmům, které mají podobné rozložení operací. Šifrovací algoritmy jsou navrhovány jako posloupnost jednoduchých transformací vstupních dat. Jinými slovy se jedná o výpočetní sítě, pro které je navrhovaná knihovna uzpůsobená. Takovéto rozložení algoritmů je způsobeno požadavkem pro možnou hardwarovou implementaci šifrování, kterou jsme schopni simulovat vytvořením výpočetní sítě. Častou operací při transformaci dat je XOR nebo sčítání s modulem, čímž se umožní invertibilita šifrování.

Následující šifra patří mezi šifry proudové, která není vytvořena z šifry blokové. Jelikož se jedná o velmi jednoduchý algoritmus, k šifrování je zapotřebí pouze jeden výpočetní uzel. jeho popis je dostupný v knize Schneier (1996). V tomto zmiňovaném uzlu dojde k operaci XOR mezi přijatými daty a jednou hodnotou v poli subklíčů (viz obrázek 4.3)



Obrázek 4.3: Část šifrovacího algoritmu RC4 zobrazující výběr subklíče K z pole S pomocí dvou indexů i a j.

4.2.1 Inicializace subklíčů

Inicializace pole se subklíči je rozdělena na dvě části. Nejprve se vytvoří pole s 256 po sobě jdoucími hodnotami typu `byte` začínající od 0. Poté se provede 256 záměn hodnot v předem připraveném poli na základě zadaného hesla a hodnot v poli.

```
private void SwapSubKeys(int index1, int index2)
{
        byte value1 = subKeysArray[index1];
        subKeysArray[index1] = subKeysArray[index2];
        subKeysArray[index2] = value1;
}
private void InitVector(string key)
{
        int keyLength = key.Length;
        subKeysArray = new byte[256];
        for (int i = 0; i < 256; i++)
        {
                subKeysArray[i] = (byte)i;
        }
        int j = 0;
        for (int i = 0; i < 256; i++)
        {
                j = (j + subKeysArray[i] + key[i % keyLength]) % 256;
                SwapSubKeys(i, j);
        }
}
```

První metoda slouží pouze k prohození hodnot v poli `subKeysArray` se zadanými indexy. Druhá metoda pojmenovaná `InitVector(string key)` slouží k samotné inicializaci pole, které se používá při šifrování dat. Můžeme si všimnout, že se při počítání indexu (4. řádek od konce) používá ASCII hodnota jednotlivých znaků ze zadaného hesla.

4.2.2 Šifrování

Celý algoritmus šifrování je napsán v metodě `Process()` v uzlu `RC4`. Při inicializaci tohoto uzlu dojde k předpřipravení potřebného pole hodnot, o kterém je řeč v předchozí sekci.

```
public class RC4 : NodeBase
{
    private byte[] subKeysArray;
    public RC4(string keyphrase) : base(1, 1)
    {
        InitVector(keyphrase);
    }
    private void InitVector(string key)
    { ... }
    private void SwapSubKeys(int index1, int index2)
    { ... }
    private int i, j = 0;
    public override void Process()
    {
        Data newData = inputBuffers[0].Pop();
        if (!newData.IsEmpty())
```

```
        {
            for (int z = 0; z < newData.Signal.Length; z++)
            {
                i = (i + 1) % 256;
                j = (j + subKeysArray[i]) % 256;
                SwapSubKeys(i, j);
                byte k = subKeysArray[(subKeysArray[i] +
                                       subKeysArray[j]) % 256];
                newData[z] = ((byte)(newData[z])) ^ k;
            }
        }
        outputBuffers[0].Push(newData);
        return;
    }
}
```

Z kódu je patrné, že šifrování je zajištěno jednoduchými operacemi. V dvou proměnných se drží hodnoty dvou indexů do pole klíčů, které se v průběhu mění. Při jednom kole šifrování, kde se zpracovává jeden byte, se hodnoty klíčů v poli vymění. Hlavní operace v tomto procesu je následný XOR hodnoty, která se má zašifrovat, a hodnoty v poli klíčů s indexem, který je závislý jak na předem jmenovaných proměnných, tak na zadaném heslu.

4.2.3 Skript zapojení RC4

Následuje přepis skriptu definující výpočetní síť šifrovacího algoritmu RC4.

```
from LiftingReferences import *
import sys

Logger.Info("Starting RC4")
if (len(sys.argv) <= 1):
    Logger.Error("No key phrase given")
    sys.exit()

keyPhrase = sys.argv[1]

byter = ConsoleByter()
rc4 = RC4(keyPhrase)
writer = ConsoleWriter()

rc4.SetInput(byter.GetOutputBuffer(0), 0)
writer.SetInput(rc4.GetOutputBuffer(0), 0)
Collection.CheckConnections()

writer.Process()
Logger.Info("Ending RC4")
```

Jsou zde použity dohromady tři uzly. První `ConsoleByter` čte text ze standardního vstupu a vytváří z nich hodnoty typu `byte` korespondující s ASCII

hodnotami znaků. Další je výše popisovaný uzel `RC4`. Poslední v zapojení je uzel, který přijímá data a vypisuje je na standardní výstup v šestnáctkové soustavě (HEX).

4.2.4 Ověření správnosti

U šifrovacích algoritmů se používají takzvané „test vectors“ pro ověření správnosti implementace. Ty se skládají z dat pro šifrování, hesla nebo klíče a výsledných šifrovaných dat. Pro RC4 jsou dostupné kromě oficiálních dat (viz Strombergson a Josefsson, 2011) i data neoficiální, které vypadají následovně.

Klíč	Text	Zašifrováno
Key	Plaintext	BBF316E8D940AF0AD3
Wiki	pedia	1021BF0420
Secret	Attack at dawn	45A01F645FC35B383552544B9BF5

Tabulka 4.1: Testovací data šifrování RC4

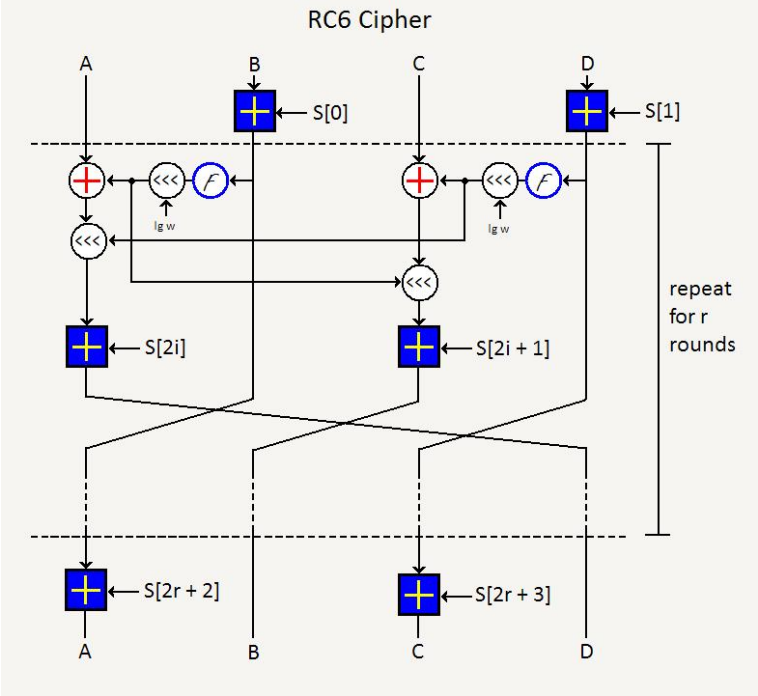
4.3 RC6

Bloková šifra RC6 je podstatně složitější než předchozí proudová šifra. Toto schéma je ukázáno na obrázku 4.4, které je také v úvodu této práce. Pro její zapojení je nutné minimálně 14 výpočetních uzlů. Pro správné fungování algoritmu je potřeba uzlů více. Je to z toho důvodu, že je třeba přidat uzly, které se postarají o čtení a následné uložení informací. Dále jsou třeba uzly, které budou přidávat k datům informaci o počtu kol, kterým data již prošla, a další technické uzly. Všechny si vyjmenujeme a popíšeme dále v kapitole. Dešifrování probíhá opačným průchodem sítě. Opačný průchod dat pomocí naší knihovny není možné spustit, ale pomocí jednoduchého triku není třeba měnit uzel, který přičítá klíč k datům. Trik spočívá v tom, že si při generování klíče vytvoříme opačné hodnoty. Poté používáním sčítání ve skutečnosti klíč odčítáme. Jediný uzel, který je třeba přidat je `ROTR`, který zajistí pravou rotaci bitů. Popsaný šifrovací algoritmus včetně inicializace klíče je v publikaci Rivest a kol. (1998).

4.3.1 Inicializace klíče

Používané pole s klíči se liší od šifry RC4 v tom, že se v průběhu šifrování nijak nemění. To znamená, že se pole s klíči může sdílet mezi uzly bez problému, že by se nějak ovlivňovaly. Pro zajímavost tvorba klíče pro šifru RC6 je stejná jako pro šifru RC5 s tím rozdílem, že je pole kratší o dvě hodnoty.

Algoritmus pro inicializaci klíče je zapsán ve speciálním skriptu pojmenovaném RC6AuxiliaryMethods. Toto rozdělení je z toho důvodu, že se stejná metoda používá i u sítě na dešifrování. V algoritmu se používají dvě magické hodnoty označené `p` a `q`. Ty byly předpočítány a jsou používány pro vhodné vlastnosti při generování klíče. První část je připravení pomocného pole hodnot `s` následujícím způsobem. Délka pole je odvozena od počtu šifrovacích kol. První hodnota v poli



Obrázek 4.4: Schéma šifrovacího algoritmu RC6.

je zmiňovaná magická hodnota `p` a poté se počítají další hodnoty jako výsledek sčítání hodnoty předchozí a `q`.

```
p = 0xB7E15163
q = 0x9E3779B9
s = [0 for x in range(2 * rounds + 4)]
s[0] = p
for i in range(1,2*rounds+4):
        s[i] = (s[i-1] + q) & 0xFFFFFFFF
```

Dále se v kódu objevuje pole `c`, což jsou ASCII hodnoty znaků předaných jako heslo.

```
A = 0
B = 0
j = 0
k = 0
maxValue = 3*max(len(c), 2*rounds+4)
for i in range(maxValue):
        A = s[j] = Rotl((s[j] + A + B) & 0xFFFFFFFF, 3)
        B = c[k] = Rotl((c[k] + A + B) & 0xFFFFFFFF, (A + B) % 32)
        j = (j + 1) % (2 * rounds + 4)
        k = (k + 1) % len(c)
```

Takto se přepočítají hodnoty v poli `s`, což je výsledný klíč, který se používá při šifrování. Příprava klíče ještě pokračuje rozdělením 32bitových hodnot na čtyři 8bitové. To je proto, aby jeden byte byl uložen v poli signálu jako jeden `double`.

4.3.2 Popis technických uzlů

V této části si rozebereme práci technických uzlů použitých v této výpočetní síti. `AddAdditionalValue(string key, object value)` je první uzel, který si v této sekci popíšeme. Ten přenášená data nijak neovlivňuje, pouze přidává dodatečnou informaci k instanci třídy `Data`. V našem případě je přidávána hodnota reprezentující kolika koly šifrování již data prošla. Tato informace se využívá v dalším uzlu `SplitSwitchIntBiggerOrEven(string key, int number)`, který na jejím základě rozhoduje, jestli se data pošlou znovu do šifrovací části nebo do sběrného uzlu. Dalším technickým uzlem je `Sink`. Ten má libovolný počet vstupů, u kterých se nejprve zkouší získat data z 2. a dalšího vstupního bufferu. V případě neúspěchu načítá data z bufferu prvního a posílá je do svého jediného výstupního bufferu. Posledním technickým uzlem, který se v tomto zapojení používá je `Copier`. Jeho účel je kopírování dat z jednoho vstupu do více výstupů. Tento uzel je pěkným příkladem jednoduchosti, kde stačí doplnit pouze pár řádků kódu.

```
public class Copier : NodeBase
{
    public Copier(int outputBufferCount) : base(1, outputBufferCount)
    { }
    public override void Process()
    {
        Data data = inputBuffers[0].Pop();
        for (int bufferIndex = 0; bufferIndex < outputBuffers.Length;
             bufferIndex++)
        {
            outputBuffers[bufferIndex].Push(new Data(data));
        }
    }
}
```

4.3.3 Popis výpočetních uzlů

Nyní se dostáváme k uzlům, které pracují přímo s daty. U všech níže zmíněných uzlů dochází k přetypování signálu na `byte`. První z výpočetních uzlů je `AddKey(string formula)`, který přičítá k přijatému signálu klíč. Ten se získá výpočtem zadané rovnice, čímž se vypočítá index do pole klíčů. Rovnice je uložena jako `string`, kde se slovo „*round*“ substituuje za počet kol, kterými data prošla. Příkladem takové rovnice může být „$2 * round + 1$“. `ROTL` se nazývá uzel, který provádí levou rotaci bitů na datech o počet, který dostane na druhém vstupu. Pro posílání konstantních dat na požádání slouží uzel `Constant`. Tomu se při inicializaci zadají data, která poté odesílá na svůj jediný výstup. Nic neříkající uzel `F`, který je pojmenován stejně, jako je v popisu algoritmu (Rivest a kol., 1998), vypočítává a odesílá výsledek rovnice $X * (2X + 1)$, kde je signál zapsán jako X. Posledním výpočetním uzlem v tomto zapojení je `XOR`, který není třeba nijak speciálně popisovat. Jenom je třeba myslet na to, že jsou data před operací přetypována na `byte`.

4.3.4 Členění zapojení

Při vytváření této sítě se využívá vlastností výpočetních segmentů, protože to výrazně snižuje duplicitu kódu a finální síť se jednodušeji spravuje. První segment si pojmenujeme `TEncrypt`. Bude se jednat o červenou zvýrazněnou část z obrázku 2.4 s přidanými technickými uzly. Ty jsou potřeba například u rozdvojení cest nebo konstantním vstupem do uzlu `ROTL`. Následuje kód metody `BuildConnections()` ze segmentu `TEncrypt`.

```
var copier1 = new Copier(2);
var f = new F();
var rotl = new ROTL();
var constant = new Constant(new double[] { lgw });
var copier2 = new Copier(2);
var xor = new XOR();

nodes = new List<NodeBase>();
nodes.Add(copier1);
nodes.Add(f);
nodes.Add(rotl);
nodes.Add(constant);
nodes.Add(copier2);
nodes.Add(xor);

f.SetInput(copier1.GetOutputBuffer(0), 0);
rotl.SetInput(f.GetOutputBuffer(0), 0);
rotl.SetInput(constant.GetOutputBuffer(0), 1);
copier2.SetInput(rotl.GetOutputBuffer(0), 0);
xor.SetInput(copier2.GetOutputBuffer(0), 1);

outputBuffer1 = xor.GetOutputBuffer(0);
outputBuffer2 = copier2.GetOutputBuffer(1);
outputBuffer3 = copier1.GetOutputBuffer(1);
```

Dalším segmentem použitým v RC6 je `REncrypt`, který v sobě ukrývá dvě instance výše popsaného segmentu `TEncrypt`. Dále se skládá ze dvou uzlů `ROTL` a taktéž ze dvou uzlů `AddKey`. To dává dohromady jedno kolo šifrování. Kol může být proměnlivý počet, proto je třeba vytvořit výpočetní síť tak, aby bylo možné měnit počet průchodů parametrem a nikoli přestavbou celé sítě.

```
var tEncrypt1 = new TEncrypt(lgw);
var tEncrypt2 = new TEncrypt(lgw);
var rotl1 = new ROTL();
var rotl2 = new ROTL();
var add1 = new AddKey("2 * round");
var add2 = new AddKey("2 * round + 1");

nodes = new List<NodeBase>();
nodes.Add(tEncrypt1);
nodes.Add(tEncrypt2);
nodes.Add(rotl1);
```

```
nodes.Add(rotl2);
nodes.Add(add1);
nodes.Add(add2);

rotl1.SetInput(tEncrypt1.GetOutputBuffer(0), 0);
rotl1.SetInput(tEncrypt2.GetOutputBuffer(1), 1);
rotl2.SetInput(tEncrypt2.GetOutputBuffer(0), 0);
rotl2.SetInput(tEncrypt1.GetOutputBuffer(1), 1);
add1.SetInput(rotl1.GetOutputBuffer(0), 0);
add2.SetInput(rotl2.GetOutputBuffer(0), 0);

outputBuffer1 = tEncrypt1.GetOutputBuffer(2);
outputBuffer2 = add2.GetOutputBuffer(0);
outputBuffer3 = tEncrypt2.GetOutputBuffer(2);
outputBuffer4 = add1.GetOutputBuffer(0);
```

Pomocí výše uvedených částí je možné vytvořit finální výpočetní segment `RC6Encrypt`, který bude mít čtyři vstupy i výstupy, jak je uvedeno na ilustraci 1.3. Při vytváření výpočetní sítě se využije tento celý segment, který se napojí na zdrojový uzel, který bude číst data. Pokud by se chtělo nějak více experimentovat se zapojením, tento poslední segment by se vynechal a vše by se napsalo do skriptu definující zapojení. To by vedlo k větší volnosti při experimentování.

4.3.5 Skript zapojení RC6

Při psaní skriptu definující tuto výpočetní síť potřebujeme čtyři druhy uzlů. Prvním je uzel `FileSplitter`, který se stará o čtení souboru a z jednotlivých bytů vytváří instance typu `Data`. Další v pořadí — `Splitter` — rozděluje čtená data do více výstupů. Toto rozdělování je třeba z principu, na jakém pracuje algoritmus RC6. Následuje výpočetní segment `RC6Encrypt`. Poslední v pořadí je uzel, který data opět ukládá do souboru. Je možné vůbec nepoužívat výpočetní segmenty a vše dát dohromady až v tomto skriptu. Nevýhodou by byla podstatně větší složitost a duplicita nějakých spojení.

4.3.6 Ověření správnosti

Stejně jako u ověřování šifrování RC4 existují data, podle kterých je možné otestovat správnost zapojení šifrovacího algoritmu RC6. Tato testovací data je možné sehnat v repositáři na adrese github.com/das-labor/legacy. Zmiňovaná data byla vytvořena v rámci projektu NESSIE. Je třeba připomenout, že systém Windows ukládá data na disk v pořadí little-endian. Z tohoto důvodu sice nebude výstup totožný s výstupem vzorovým, ale jde o stejná data.

Klíč	80000000000000000000000000000000
Data	00000000000000000000000000000000
Zašifrováno	1AD578A02A08162850A15A1552A17AD4
Klíč	40000000000000000000000000000000
Data	00000000000000000000000000000000
Zašifrováno	912E9CF1473035A8443A82495C0730D3
Klíč	20000000000000000000000000000000
Data	00000000000000000000000000000000
Zašifrováno	3D3E851A80ABAF221761931747473048

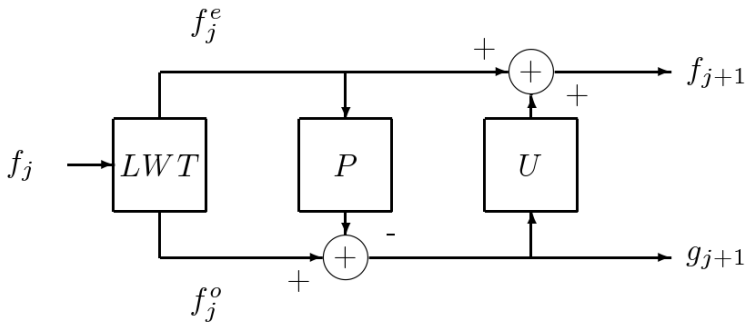
Tabulka 4.2: Testovací data šifrování RC6

5. Ukázka tvorby zapojení

Na následujících dvou příkladech si ukážeme, jak by se vytvořila nová výpočetní síť, která bude provádět Mallatův rozklad pomocí Haarovy vlnkové transformace (viz Sweldens, 1997) a výpočetní segment složitější vlnkové transformace. Budeme využívat již vytvořené části, pomocí kterých postavíme cílený obvod.

5.1 Haarova vlnková transformace

Haarova transformace funguje podobně jako již ukázaná diskrétní vlnková transformace v sekci 3.2. Z tohoto důvodu bude potřeba vytvořit pouze jeden nový uzel, aby bylo možné toto zapojení vytvořit. Rozdíly transformací jsou pouze v částech označených jako Predict (P) a Update (U) na obrázku 5.1. V případě Haarovy transformace je funkce P identita, a tudíž není třeba vytvářet nový uzel. Ten je třeba u funkce U, kde se vypočítává $X \rightarrow X/2$, kde X je posílaný signál.



Obrázek 5.1: Schéma zapojení Haar vlnkové transformace.

U každého výpočetního uzlu, který budeme vytvářet, je třeba dodržet tři body.

- Uzel musí být potomkem abstraktní třídy `NodeBase`
- Je třeba specifikovat konstruktor, zejména parametry, se kterými se bude volat konstruktor napsaný v předkovi
- Musí se přepsat metoda `public override void Process()`

Začneme vytvořením zmiňovaného uzlu. Nejprve je třeba rozmyslet, do jakého souboru se bude přidávat. Jelikož jde o výpočetní uzel, který nějak modifikuje signál, zvolíme soubor ComputeNodes.cs. Vytvoříme novou třídu `UHaar`, která musí být potomkem abstraktní `NodeBase`.

```
public class UHaar : NodeBase { }
```

Dále je třeba specifikovat konstruktor. Jedná se o jednoduchý uzel, který má jeden vstupní a jeden výstupní buffer. Proto vytvoříme konstruktor třídy následovně.

```
public UHaar() : base(1,1) { }
```

Nyní se dostáváme k samotnému výpočtu uzlu, tedy k metodě `Process()`. Ta musí být přepsána. Je třeba přijmout data ze vstupního bufferu, vydělit je dvěma a přidat do bufferu výstupního.

```
public override void Process()
{
    Data data = inputBuffers[0].Pop();
    data /= 2;
    outputBuffers[0].Push(data);
}
```

To je pro tvorbu tohoto jednoduchého výpočetního uzlu vše. Protože již máme všechny potřebné uzly vytvořené, přesuneme se k tvorbě výpočetního segmentu `HaarWaveletCompress`. Tento segment budeme přidávat do souboru Compute-Segments.cs. Opět začneme vytvořením příslušné třídy.

```
public class HaarWaveletCompress : NodeBase { }
```

Konstruktor segmentu bude vypadat následovně.

```
public HaarWaveletCompress() : base(0, 2)
{
    BuildConnections();
}
```

Zopakujeme, že je třeba dodržet nulový počet vstupních bufferů, protože ty budeme mít již připravené ve vstupních uzlech. Metoda `BuildConnections()` vytvoří zapojení segmentu z již napsaných uzlů. Privátní proměnné jsou zde ze stejného důvodu, jak bylo popsáno v sekci 3.2.

```
private Buffer outputBuffer1;
private Buffer outputBuffer2;
private NodeBase inputNode;
private List<NodeBase> nodes;

private void BuildConnections()
{
    var splitter = new Splitter(2);
    var substract = new Substract();
    var copier1 = new Copier(2);
    var copier2 = new Copier(2);
    var u = new UHaar();
    var add = new Add(2);

    nodes = new List<NodeBase>();
    nodes.Add(splitter);
    nodes.Add(substract);
    nodes.Add(copier1);
    nodes.Add(copier2);
```

```
    nodes.Add(u);
    nodes.Add(add);

    outputBuffer1 = copier2.GetOutputBuffer(1);
    outputBuffer2 = add.GetOutputBuffer(0);
    inputNode = splitter;

    substract.SetInput(splitter.GetOutputBuffer(1), 0);
    substract.SetInput(copier1.GetOutputBuffer(0), 1);
    copier1.SetInput(splitter.GetOutputBuffer(0), 0);
    copier2.SetInput(substract.GetOutputBuffer(0), 0);
    u.SetInput(copier2.GetOutputBuffer(0), 0);
    add.SetInput(u.GetOutputBuffer(0), 0);
    add.SetInput(copier1.GetOutputBuffer(1), 1);
}
```

Samotná metoda `Process()`, kterou je třeba dopsat, je velice jednoduchá. Pouze se „přehazují“ instance třídy `Data` z výstupních bufferů posledních uzlů do výstupních bufferů segmentu.

```
public override void Process()
{
    outputBuffers[0].Push(outputBuffer1.Pop());
    outputBuffers[1].Push(outputBuffer2.Pop());
}
```

Nyní nám již zbývá jenom přidat metody, potřebné pro správné fungování výpočetního segmentu. Konkrétně se jedná o metodu přidání vstupního bufferu a resetování segmentu. Přepsání metody na přidávání vstupních bufferů je nutné z toho důvodu, že segment žádné vstupní buffery nemá. Proto je nutné zajistit zapojení do vstupních uzlů v segmentu.

```
public new void SetInput(Buffer buffer, int inputBufferIndex)
{
    if (inputBufferIndex == 0)
    {
        inputNode.SetInput(buffer, 0);
    }
    else
    {
        Logger.Error("Node {0} can't set input number {1}",
                     ID, inputBufferIndex);
    }
}
public new void ResetNode()
{
    for (int i = 0; i < nodes.Count; i++)
    {
        nodes[i].ResetNode();
    }
}
```

Tímto bychom měli hotový výpočetní segment, který reprezentuje vlnkovou Haarovu transformaci. Nyní musíme vytvořit ještě jeden výpočetní segment, který bude pracovat ve dvou kolech. Jinými slovy vytvoříme Mallatův rozklad s pomocí jiného výpočetního segmentu než je DWT. Z tohoto důvodu zde nebudeme uvádět celý zdrojový kód, jelikož je až na čtyři řádky stejný, jako u třídy `MallatDecomposition`. Rozdíly jsou pouze v metodě `BuildConnections()`, kde zaměníme výpočetní segment, a v metodě `Process()` přetypování uzlu.

```
var compress1 = new HaarWaveletCompress();
var compress2 = new HaarWaveletCompress();
var compress3 = new HaarWaveletCompress();

((HaarWaveletCompress)node).ResetNode();
```

Z takto vytvořených částí se můžeme pustit do napsání skriptu definující výpočetní síť. Proto vytvoříme nový skript jazyka IronPython, který bude obsahovat následující kód.

```
from LiftingReferences import *
from TextfileAxuiliaryMethods import *
from ImageAxuiliaryMethods import *
clr.AddReference('System.Drawing')
from System.Drawing import Bitmap
import sys

imageName = sys.argv[1]
image = Bitmap(imageName + ".png")
outputFileName = imageName + "_haar_mallat"
originalWidth = image.Width
originalHeight = image.Height
levels = int(sys.argv[2])

(newWidth, newHeight) = GetProperResolution(originalWidth,
     originalHeight, levels)
if (originalWidth != newWidth or originalHeight != newHeight):
        Logger.Warning("Image resolution doesn't allow {0} level
             decomposition\nImage size will be expanded to
             {1}x{2}".format(levels, newWidth, newHeight))
        image = Resize(image, newWidth, newHeight)

pixelator = ImagePixelator()
mallat = HaarMallatDecomposition()
mallat.SetInput(pixelator.GetOutputBuffer(0), 0)
Collection.CheckConnections()
pixelator.SetProperties(image)
mallat.SetProperties(outputFileName, levels, image.Width, image.Height)

mallat.Process()

AppendTextfileFromList(outputFileName, [originalWidth, originalHeight])
```

Nyní přidáme kontroly parametrů.

```
...

if (len(sys.argv) <= 2):
        Logger.Error("No image or level given")
        sys.exit()

...

if (levels <= 0):
        Logger.Error("Levels can't be less or equal to zero")
        sys.exit()

...
```

Teď nám stačí vysvětlit, co vykonává funkce `GetProperResolution(...)`. Ta počítá, jestli pro dané rozlišení je možný požadovaný počet úrovní rozkladu a vrátí rozlišení, které je třeba. Další použitá funkce je `Resize(...)`. Ta zvětší obrázek na požadované rozlišení a prázdná místa jsou zaplněna černou barvou. Nakonec vrátí instanci takto zvětšeného obrázku. Poslední používaná metoda v předchozím kódu je `AppendTextfileFromList(...)`, která do zadaného textového souboru přidá položky předaného pole. Využívá se k zapsání původního rozlišení obrázku do textového souboru, aby byla možná zpětná transformace. Následuje ukázka funkce `Resize(...)`.

```
def Resize(image, newWidth, newHeight):
    newImage = Bitmap(newWidth, newHeight, image.PixelFormat)
    with Graphics.FromImage(newImage) as g:
        g.FillRectangle(Brushes.Black, 0, 0, newWidth, newHeight)
        g.DrawImage(image, 0, 0, image.Width, image.Height)
    return newImage
```

Po spuštění tohoto rozkladu na stejný obrázek, který jsme použili u rozkladu s diskrétní vlnkovou transformací, dostaneme obrázek 5.2.

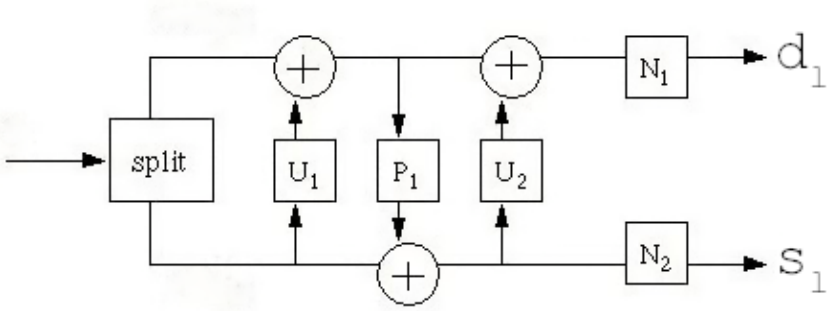
Rozklad vypadá velice podobně. Rozdíly jsou patrné hlavně v částech s detaily, které jsou více zastoupeny. Tímto jsme si ukázali, jak konkrétně postupovat při vytváření nové sítě. Dalším způsobem, jak dosáhnout stejného výpočtu, by mohlo být nevytvoření segmentu `HaarMallatDecomposition`, ale veškerou funkcionalitu zapsat do finálního skriptu. To by ulehčilo následné experimentování se zapojením, ale jelikož se jedná o předem definovaný rozklad, tak se nepředpokládají nějaké budoucí změny.

5.2 Složitější vlnkové transformace

Nyní si ukážeme, jak vytvořit složitější vlnkovou transformaci za použití této knihovny. Haar transformace z předchozí části lze považovat za jednoduchou. My si teď představíme Daubechiesovu vlnkovou transformaci D4, která je popsaná v článku Daubechies a Sweldens (1998, kapitola 7.5). Liftingové schéma zapojení

Obrázek 5.2: Mallatův rozklad s použitím diskrétní Haarovy vlnkové transformace

si lze představit z obrázku 5.3, kde uzly `U P N` odpovídají jednotlivým rovnicím výpočtu. Návod na získání těchto rovnic lze najít ve stejné kapitole nebo ve zprávě Uytterhoeven a kol. (1997).



Obrázek 5.3: Schéma zapojení Daubechiesovi vlnkové transformace D4

$$
\begin{aligned}
d_l^{(1)} &= x_{2l+1} - \sqrt{3}x_{2l} \\
s_l^{(1)} &= x_{2l} + \sqrt{3}/4 d_l^{(1)} + (\sqrt{3}-2)/4 d_{l+1}^{(1)} \\
d_l^{(2)} &= d_l^{(1)} + s_{l-1}^{(1)} \\
s_l &= (\sqrt{3}+1)/\sqrt{2} s_l^{(1)} \\
d_l &= (\sqrt{3}-1)/\sqrt{2} d_l^{(2)}
\end{aligned}
\tag{5.1}
$$

Těmito rovnicemi máme popsanou transformaci a lze z nich vidět, jak by mělo výsledné zapojení vypadat. Ve stejném článku je popsán návod, jak se k těmto rovnicím dostat z jiné reprezentace vlnkových transformací. Signál se rozdělí na dva proudy, lichá a sudá data. Dále je patrné, že je třeba vytvořit výpočetní uzly, které budou korespondovat s koeficienty proměnných. Například pro první rovnici je třeba vytvořit uzel, který vypočítá $\sqrt{3}$ násobek dat. V minulé sekci jsme si ukázali, jak vytvářet jednotlivé výpočetní uzly. Zde si ukážeme jiný přístup, a to použití již existujících uzlů, které nám umožňují definovat výpočet při inicializaci.

Jestliže je třeba nějaký jednodušší výpočetní uzel, tak je možné využít již existující uzel `LambdaCalculationDouble`, který jako argument bere funkci na transformaci položky `double`. Tato funkce se aplikuje nezávisle na všechny položky signálu. Tento uzel se hodí například pro předem popisovanou transformaci vynásobením $\sqrt{3}$. V segmentu se použije následující kód.

```
var p = new LambdaCalculationDouble(x => Math.Sqrt(3) * x);
```

Na druhém řádku si můžeme všimnout, že se počítá s více instancemi dat. K tomuto výpočtu by nám předem jmenovaný uzel nestačil, proto existuje obecnější uzel `LambdaCalculationBuffer`. Ten má přístup přímo ke vstupnímu bufferu, který může ovládat. Následující kód popisuje část výpočtu $\sqrt{3}/4d_l^{(1)} + (\sqrt{3} - 2)/4d_{l+1}^{(1)}$.

```
var u = new LambdaCalculationBuffer(buffer =>
{
    Data data = (Math.Sqrt(3) / 4) * buffer.Pop();
    data += ((Math.Sqrt(3) - 2) / 4) * buffer.Peek();
    return data;
});
```

U třetího řádku výpočtu se setkáváme se situací, kde je třeba předchozí výsledek. U první iterace je proto třeba nějak „přidat“ potřebná data. K tomto účelu slouží uzel `AddFirstNValues`. Ten při první iteraci pošle n-krát předem stanovený signál a poté přeposílá přijímaný signál. Při inverzní transformaci je nutné na stejné místo vložit uzel `SkipFirstNValues`, který naopak zahodí prvních n dat. Výpočetní uzly korespondující se čtvrtou a pátou rovnicí vytvoříme stejně jako uzel první.

Nyní nám nezbývá nic jiného, než si rozmyslet, jak bude vypadat daná výpočetní síť, a napsat požadovaný výpočetní segment. Výpočetní uzly jsme si již představili, ale potřeba jsou také technické uzly, které například duplikují signál. Jelikož jsme si tvorbu výpočetního segmentu popsali v minulé sekci této kapitoly, ukážeme si pouze metodu `BuildConnections()`, která definuje zapojení jednotlivých uzlů.

```
private void BuildConnections()
{
    var splitter = new Splitter(2);
    var p = new LambdaCalculationDouble(x => Math.Sqrt(3) * x);
    var copier1 = new Copier(2);
    var substract = new Substract();
    var add1 = new Add(2);
    var u = new LambdaCalculationBuffer(buffer =>
    {
        Data data = (Math.Sqrt(3) / 4) * buffer.Pop();
        data += ((Math.Sqrt(3) - 2) / 4) * buffer.Peek();
        return data;
    });
    var copier2 = new Copier(2);
```

```
    var copier3 = new Copier(2);
    var delayer = new AddFirstNValues(1, new double[] { 0, 0, 0 });
    var add2 = new Add(2);
    var compute1 = new LambdaCalculationDouble(x =>
         ((Math.Sqrt(3) + 1) / (Math.Sqrt(2))) * x);
    var compute2 = new LambdaCalculationDouble(x =>
         ((Math.Sqrt(3) - 1) / (Math.Sqrt(2))) * x);

    nodes = new List<NodeBase>();
    nodes.Add(splitter);
    nodes.Add(p);
    nodes.Add(copier1);
    nodes.Add(substract);
    nodes.Add(add1);
    nodes.Add(u);
    nodes.Add(copier2);
    nodes.Add(copier3);
    nodes.Add(delayer);
    nodes.Add(add2);
    nodes.Add(compute1);
    nodes.Add(compute2);

    outputBuffer1 = compute1.GetOutputBuffer(0);
    outputBuffer2 = compute2.GetOutputBuffer(0);
    inputNode = splitter;

    copier1.SetInput(splitter.GetOutputBuffer(0), 0);
    p.SetInput(copier1.GetOutputBuffer(0), 0);
    substract.SetInput(splitter.GetOutputBuffer(1), 0);
    substract.SetInput(p.GetOutputBuffer(0), 1);
    copier2.SetInput(substract.GetOutputBuffer(0), 0);
    u.SetInput(copier2.GetOutputBuffer(0), 0);
    add1.SetInput(copier1.GetOutputBuffer(1), 0);
    add1.SetInput(u.GetOutputBuffer(0), 1);
    copier3.SetInput(add1.GetOutputBuffer(0), 0);
    delayer.SetInput(copier3.GetOutputBuffer(0), 0);
    add2.SetInput(copier2.GetOutputBuffer(1), 0);
    add2.SetInput(delayer.GetOutputBuffer(0), 1);
    compute1.SetInput(copier3.GetOutputBuffer(1), 0);
    compute2.SetInput(add2.GetOutputBuffer(0), 0);
}
```

6. Spouštění výpočtů

V této krátké kapitole si ukážeme, jak se spouštějí vytvořené skripty a uvedeme různé příklady výstupů těchto výpočtu. Je třeba upozornit, že pro spuštění skriptů je třeba mít nainstalovaný interpret pro jazyk IronPython. Instalátor nebo zdrojové kódy (pro přeložení) tohoto programu je možné získat z internetové stránky `http://ironpython.net/download/`. Jazyk Python2.7 má sice stejnou syntaxi kódu, ale neumí pracovat s objekty platformy .NET, a proto nelze použít jeho interpret. V ukázkách se používá zkratka `ipy` místo volání celé adresy (např. `C:\Program Files (x86)\IronPython 2.7\ipy.exe`) k programu IronPython. Přesná adresa programu je závislá na místě, kde byl program nainstalován.

6.1 Šifrování

Ukázku začneme jednoduchým šifrovacím obvodem RC4. Tento skript požaduje jeden argument a to heslo, pomocí kterého se bude šifrovat.

```
> echo|set /p="Plaintext" | ipy RC4ConsoleEncrypter.py Key
Starting RC4
BBF316E8D940AF0AD3
RC4 finished successfuly

> echo|set /p="Attack at dawn" | ipy RC4ConsoleEncrypter.py Secret
Starting RC4
45A01F645FC35B383552544B9BF5
RC4 finished successfuly
```

V ukázce se vyskytuje část `set /p=`, která způsobí vypsání pouze textu mezi uvozovkami. Je to z toho důvodu, že příkaz `echo` na konci přidá znak reprezentující nový řádek. Ten by se šifroval také a testovaný výstup by poté byl jiný.

Posuneme se ke složitějšímu šifrovacímu obvodu a to k RC6. Tento skript již nepracuje s textem jako RC4, ale přímo s daty souboru. Z tohoto důvodu jsme vytvořili soubor (rc6_1), který je 16 bytů dlouhý. Tento vstup je používán u testovacích dat. Byly vytvořeny dva druhy skriptů s tímto zapojením. Jeden obecný, který vytváří klíč z předaného hesla, a skript testovací, který má klíč pevně daný. Při spouštění skriptu je požadováno heslo (pouze u obecného skriptu), počet šifrovacích kol a název souboru. Používaný počet kol při normálním průběhu transformace je 20.

```
(HEX souboru rc6_1)
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

> ipy RC6FileEncrypterTESTKEY.py 20 files\rc6_1
Starting RC6Encryption
Rounds: 20
File:   files\rc6_1
Test key:       00000080 00000000 00000000 00000000
RC6TEST finished successfuly
```

```
(HEX nového souboru rc6_1_enc)
A0 78 D5 1A 28 16 08 2A 15 5A A1 50 D4 7A A1 52
```

U tohoto obvodu je potřeba opět zdůraznit, že data jsou uložena v pořadí little-endian. Pokud bychom data pro názornost ručně přepsali, dostaneme toto.

```
Test key:       80000000 00000000 00000000 00000000
(HEX souboru rc6_1_enc BE)
1A D5 78 A0 2A 08 16 28 50 A1 5A 15 52 A1 7A D4
```

Po překontrolování výstupu s testovacími daty můžeme konstatovat, že šifrovací obvod funguje správně. Zpětné dešifrování probíhá následovně.

```
> ipy RC6FileDecrypterTESTKEY.py 20 files\rc6_1_enc
Starting RC6Dencryption
Rounds: 20
File:   files\rc6_1_enc
Test key:       00000080 00000000 00000000 00000000
RC6TEST finished successfuly

(HEX souboru rc6_1_enc_dec)
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

6.2 Mallatův rozklad

Spuštění Mallatova rozkladu používající základní vlnkovou transformaci je stejně snadné jako předchozí příklad. Skript požaduje dva parametry, a to název obrázku, který je ve formátu PNG, a počet úrovní rozkladu.

```
> ipy MallatDecomposition.py files\input 4
Starting MallatDecomposition
File:   files\input.png
Levels: 4
Image resolution doesn't allow 4 level decomposition
Image size will be expanded to 1920x1088
MallatDecomposition finished successfuly
```

Můžeme si všimnou upozornění, že rozlišení neumožňuje 4 úrovně rozkladu. Proto byl obrázek zvětšen o 8 řádků. Po úspěšném dokončení výpočtu se vytvoří dva soubor. Prvním je obrázek (input_mallat.png) a textový soubor (input_mallat.txt). Toto chování již bylo vysvětleno v kapitole 4.1.2. Pro opětovné složení se použije následující příkaz.

```
> ipy MallatRecoveryFromText.py files\input_mallat 4
Starting MallatRecoveryFromText
File:   files\input_mallat.txt
Levels: 4
```

```
Ending MallatRecoveryFromText
Image will be cropped to original resolution 1920x1080
MallatRecoveryFromText finished successfuly
```

V průběhu jsme upozorněni, že výsledný obrázek bude opět oříznut na původní velikost. Na konci je vytvořen obrázek input_mallat_recover.png, který je totožný s původním.

Závěr

V rámci práce byla vytvořena knihovna Lifting, která umožňuje jednoduchou tvorbu výpočetních sítí. Pod tímto pojmem si lze představit algoritmy, které lze rozčlenit na jednoduché transformace vstupního signálu (uzly) a cesty po kterých se přesouvají mezivýsledky. Příkladem takovéto výpočetní sítě jsou šifrovací algoritmy, u kterých jsou přesně definované jednotlivé elementární kroky. Knihovna je připravena na transformaci více-dimenzionálních signálů, příkladem může být zpracování více barevných kanálů při komprimaci obrázku. Hlavní inspirací pro knihovnu byla snadná tvorba vlnkových transformací a následné experimentování s těmito zapojeními.

Dále bylo vytvořeno několik výpočetních sítí pro potvrzení funkčnosti vypracovaného systému. Výše zmiňovaná knihovna Lifting poskytuje kostru těchto sítí spolu s implementovanými základními uzly a výpočetními segmenty. V oblasti komprimace obrázků byly vytvořeny tři výpočetní celky, které byly založeny na základě liftingu (Mallatův rozklad). Konkrétně jde o základní liftingové schéma, o Haar transformaci a o Daubechiesovu vlnkovou transformaci D4. Další vytvořené výpočetní sítě se týkají šifrování. Tím se ukázala univerzalita této knihovny při vytváření transformací, kde je možné pracovat s různými typy signálu. Bylo vytvořeno zapojení proudové šifry RC4, která zpracovává text. Další vytvořenou šifrou je šifra RC6, která je bloková. Ta pracuje přímo s daty souboru nikoli obsahem. Při tvorbě nových uzlů lze postupovat více způsoby. Složitější výpočetní uzly je třeba vytvořit přímo v knihovně, zatímco jednoduché výpočetní uzly není třeba vytvářet přímo a lze místo toho využít již existující uzel `LambdaCalculationDouble`, u kterého se definuje potřebná transformace při inicializaci. K častému přístupu vytvoření nových uzlů jsme se přiklonili z důvodu ukázky možností knihovny a demonstraci dílčích kroků při tvorbě nového uzlu.

Vypracované téma dává příležitost k jistým vylepšením této knihovny. Jedním z nich by mohla být grafická nástavba, která by ukazovala aktuální zapojení. Dalším možným vylepšením by mohlo být vypracování generátoru skriptů výpočetních sítí, které by se definovaly pomocí grafického rozhraní. Přesouvání a propojování různých uzlů by bylo uživatelsky přívětivější oproti psaní zapojení ručně.

Seznam použité literatury

Daubechies, I. a Sweldens, W. (1998). Factoring wavelet transforms into lifting steps. *J. Fourier Anal. Appl*, **4**, 247–269.

Lazar Stošic, M. B. (2012). RC4 stream cipher and possible attacks on WEP. *International Journal of Advanced Computer Science and Applications(IJACSA)*, **3**(3). URL `http://ijacsa.thesai.org/`.

Mallat, G. (1989). A theory for multiresolution signal decomposition : the wavelet representation. *IEEE Transaction on Pattern Analysis and Machine Intelligence*.

Menezes, A. J., Oorschot, P. C. V., Vanstone, S. A. a Rivest, R. L. (1997). Handbook of applied cryptography.

Rivest, R. L., Robshaw, M. J. B., Sidney, R. a Yin, Y. L. (1998). The RC6 Block Cipher. In *in First Advanced Encryption Standard (AES) Conference*, page 16.

Rolón, J. C. a Salembier, P. (2007). Generalized lifting for sparse image representation and coding. In *Picture Coding Symposium*, page 4. Lisbon.

Schneier, B. (1996). *Applied Cryptography*. Wiley John + Sons. ISBN 0471117099. URL `http://www.ebook.de/de/product/3236611/bruce_schneier_applied_cryptography.html`.

Strombergson, J. a Josefsson, S. (2011). Test vectors for the stream cipher rc4. RFC 6229, RFC Editor.

Sweldens, W. (1997). The lifting scheme: A construction of second generation wavelets.

Sweldens, W. a kol. (1995). The lifting scheme: A new philosophy in biorthogonal wavelet constructions. *Wavelet Applications in Signal and Image Processing*, **3**, 68–79.

Uytterhoeven, G., Wulpen, F. V., Jansen, M., Roose, D. a Bultheel, A. (1997). Waili: Wavelets with integer lifting. Technical report, DEPARTMENT OF COMPUTER SCIENCE, KATHOLIEKE UNIVERSITEIT LEUVEN.

Seznam zdrojů obrázků

Odkazy na zdroje použitých obrázků v této práci platné k 18.07.2017.

Obrázek 1.1
https://commons.wikimedia.org/wiki/File:GLScheme.png

Obrázek 1.2
https://www.mathworks.com/help/wavelet/gs/discrete-wavelet-transform.html

Obrázek 1.3 2.4 2.5 4.4
https://commons.wikimedia.org/wiki/File:RC6_Cryptography_Algorithm.JPG

Obrázek 1.4
https://commons.wikimedia.org/wiki/File:Stream_cipher.svg

Obrázek 1.5
https://www.pexels.com/photo/nature-summer-purple-yellow-36753

Obrázek 4.2 5.2
https://www.pexels.com/photo/bengal-tiger-half-soak-body-on-water-during-daytime-145939

Obrázek 4.3
https://commons.wikimedia.org/wiki/File:RC4.svg

Obrázek 5.1
https://commons.wikimedia.org/wiki/File:LiftingScheme.png

Obrázek 5.3
http://www.bearcave.com/misl/misl_tech/wavelets/daubechies

Přílohy

Přílohou této práce je CD obsahující:

- Zdrojové kódy knihovny
- Dokumentace ke zdrojovým kódům vygenerovaná nástrojem `Doxygen`.
- Adresář se skripty definující výpočetní sítě a soubory potřebné pro spuštění výpočtů (obrázky aj.).
- Text práce ve formátu `PDF`.