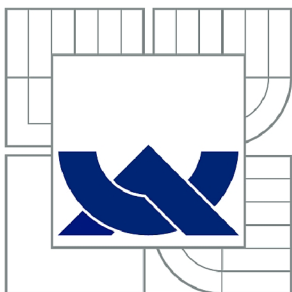
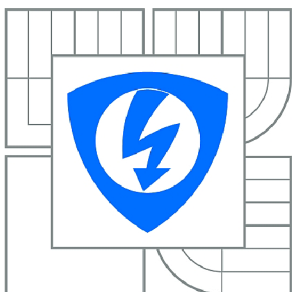


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

ÚSTAV TELEKOMUNIKACÍ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

DEPARTMENT OF TELECOMMUNICATIONS

ZVUKOVÝ KODEK S PODPOROU ZABEZPEČENÍ PRO PBX ASTERISK

SECURED AUDIO CODEC FOR ASTERISK PBX

DIPLOMOVÁ PRÁCE

MASTER'S THESIS

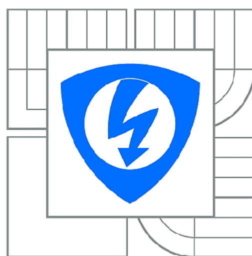
AUTOR PRÁCE Bc. MICHAL JAKUBÍČEK

AUTHOR

VEDOUCÍ PRÁCE Ing. ONDŘEJ KRAJSA, Ph.D.

SUPERVISOR

BRNO 2015



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Fakulta elektrotechniky a komunikačních technologií

Ústav telekomunikací

Diplomová práce

magisterský navazující studijní obor
Telekomunikační a informační technika

Student: Bc. Michal Jakubíček ***ID:*** 125232
Ročník: 2 ***Akademický rok:*** 2014/2015

NÁZEV TÉMATU:

Zvukový kodek s podporou zabezpečení pro PBX Asterisk

POKYNY PRO VYPRACOVÁNÍ:

Navrhněte a realizujte modul audiokodeku pro PBX Asterisk s možností zabezpečení přenášených audio dat. Tento kodek bude parametrizovatelný pomocí konfiguračního souboru. Navržené řešení otestujte.

DOPORUČENÁ LITERATURA:

[1] WOOTTON, Cliff. A practical guide to video and audio compression: from sprockets and rasters to macroblocks. Amsterdam: Focal Press, 2005, xii, 787 s. ISBN 0-240-80630-1
[2]Russell Bryant, Leif Madsen, and Jim Van Meggelen. 2013. Asterisk: The Definitive Guide (4th ed.). O'Reilly Media, Inc, ISBN 978-1449332426

Termín zadání: 9.2.2015 ***Termín odevzdání:*** 26.5.2015

Vedoucí práce: Ing. Ondřej Krajsa, Ph.D.
Konzultanti diplomové práce:

doc. Ing. Jiří Mišurec, CSc.
Předseda oborové rady

UPOZORNĚNÍ:

Autor diplomové práce nesmí při vytváření diplomové práce porušit autorská práva třetích osob, zejména nesmí zasahovat nedovoleným způsobem do cizích autorských práv osobnostních a musí si být plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č.40/2009 Sb.

ABSTRAKT

Tato práce se věnuje návrhu zabezpečení zvukového kodeku pro pobočkové ústředny Asterisk. V první kapitole je osvětleno základní rozdělení pobočkových ústředen na tradiční výrobce a na ústředny s otevřeným kódem. Druhá kapitola se zabývá popisem struktury PBX Asterisk a jejím základním rozdílem oproti tradiční PBX. Asterisk je založen na komponentách zvaných moduly, proto se práce dále zabývá nejstěžejnějšími moduly pro chod ústředny a také jejich dělením z hlediska podpory, či dělením podle typu použití a jejich vlastností. V této kapitole jsou detailněji popsány zvukové kodeky A-law a μ-law. Třetí kapitola obsahuje jednoduchý návod, jak se zorientovat ve stavbě modulu pro PBX Asterisk, a dále je tento návod doplněn jednoduchým příkladem vytvoření modulu, ukázkou postupu jeho překladu, zprovoznění a načtení do Asterisku. Simulace zabezpečení hlasu se nachází ve čtvrté kapitole, která přináší popis navrženého řešení zabezpečení s následnou realizací v programu Simulink. Tato simulace ověřuje funkčnost navrženého řešení zabezpečení telefonního hovoru. V páté kapitole jsou nastíněny používané historické techniky šifrování, především zrcadlení spektra a časové dělení signálu, a jejich srovnání se současnými moderními digitálními technikami. V poslední šesté kapitole se nachází samotná realizace modulu zvukového kodeku se zabezpečením.

KLÍČOVÁ SLOVA

Asterisk, PBX, modul pro Asterisk, zabezpečení zvukového kodeku, šifrování

ABSTRACT

This thesis is focused on the design of secured audio codec for Asterisk PBX. The first chapter is focused on the basic division of traditional PBX producers and the open source PBX. The second chapter explains the structure of Asterisk PBX and its fundamental difference from a traditional PBX. Asterisk is based on components called modules, therefore the work also deals with the most important modules for operation of exchanges and their division of terms of support and dividing by the type of application and their properties. In this chapter there are described in more detail audio codec A-law and μ-law. The third chapter contains simple instructions to get your orientation in the construction of the module for Asterisk PBX and this guide is accompanied by a simple example of creating a module demonstration of his method of translation, commissioning and loaded into Asterisk. Simulation of voice security is in the fourth chapter which provides a description of the proposed security solutions with subsequent implementation in Simulink. This simulation verifies the functionality of the solution proposed security phone call. In the fifth chapter outlines the historical use of encryption techniques primarily mirroring the spectrum and time division signal and comparing them with current modern digital technics. In the last sixth chapter is the actual implementation audio codec module with encryption.

KEYWORDS

Asterisk, PBX, modul for Asterisk, security audio codec, encryption

JAKUBÍČEK, Michal *Zvukový kodek s podporou zabezpečení pro PBX Asterisk*: diplomová práce. Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav telekomunikací, 2015. 51 s. Vedoucí práce byl Ing. Ondřej Krajsa, Ph.D.

PROHLÁŠENÍ

Prohlašuji, že svou diplomovou práci na téma „Zvukový kodek s podporou zabezpečení pro PBX Asterisk“ jsem vypracoval samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor uvedené diplomové práce dále prohlašuji, že v souvislosti s vytvořením této diplomové práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a/nebo majetkových a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů, včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb.

Brno . . . .

podpis autora

PODĚKOVÁNÍ

Rád bych poděkoval vedoucímu diplomové práce panu Ing. Ondřeji Krajsovi, Ph.D. za obětovaný čas, odborné vedení, konzultace, trpělivost a podnětné návrhy k práci. Dále bych rád poděkoval svým spolužákům za vytvoření výborných studijních podmínek a za jejich cenné připomínky a rady k práci. V neposlední řadě také své rodině za podporu během celého studia.

Brno . . . .

podpis autora

Faculty of Electrical Engineering
and Communication
Brno University of Technology
Purkynova 118, CZ-61200 Brno
Czech Republic
http://www.six.feec.vutbr.cz

PODĚKOVÁNÍ

Výzkum popsaný v této diplomové práci byl realizován v laboratořích podpořených z projektu SIX; registrační číslo CZ.1.05/2.1.00/03.0072, operační program Výzkum a vývoj pro inovace.

Brno ....

podpis autora

EVROPSKÁ UNIE
EVROPSKÝ FOND PRO REGIONÁLNÍ ROZVOJ
INVESTICE DO VAŠÍ BUDOUCNOSTI

OP Výzkum a vývoj
pro inovace

OBSAH

Úvod 10

1 PBX - pobočkové ústředny 11

2 Asterisk 12
2.1 Architektura Asterisku . . . 12
2.2 Hardwarová náročnost . . . 14
2.3 Moduly . . . 14
2.4 Základní zvukové kodeky využívané v PBX Asterisk . . . 16
2.4.1 A-law . . . 16
2.4.2 u-law . . . 17

3 Jak psát moduly pro Asterisk 18

4 Simulace zabezpečení hlasu v programu Simulink 23

5 Historie šifrování hlasu 26
5.1 Skramblování hlasu . . . 26
5.1.1 Zrcadlení spektra . . . 26
5.1.2 Časové dělení . . . 27
5.1.3 Delta modulátory . . . 28

6 Zvolená metoda šifrování 29
6.1 Požadavky na klíč . . . 29
6.2 Generátor pseudonáhodné posloupnosti . . . 29
6.3 XOR funkce . . . 30
6.4 Šifrování ve zvukovém kodeku PBX Asterisk . . . 31

7 Závěr 35

Literatura 36

Seznam příloh 37

A Zdrojový kód kodeku A-law s podporou šifrování pro Asterisk 38

B Zdrojový kód kodeku u-law s podporou šifrování pro Asterisk 44

C Obsah přiloženého CD 51

SEZNAM OBRÁZKŮ

2.1 Rozdíl mezi tradiční PBX a Asteriskem [1] 12
2.2 Blokové schéma architektury Asterisku [3] 13
2.3 Kódování A-law [6] 17
3.1 Konzolové okno s výpisem po zadání příkazu sudo make 19
3.2 Konzolové okno s výpisem po zadání příkazu pro načtení modulu . . 20
3.3 Konzolové okno s výpisem po zadání příkazu pro odstranění modulu . 20
3.4 Konzolové okno s výpisem po zadání příkazu pro ověření načteného modulu 20
4.1 Zadávání klíče pro šifrování hlasu 23
4.2 Vstupní bloky modelu simulace 24
4.3 Šifrovací část modelu simulace 25
4.4 Dešifrovací část modelu simulace 25
5.1 Postup zrcadlení spektra 26
5.2 Příklad postupu skramblování pomocí časového dělení [8] 27
6.1 Proudová šifra využívající GPNP 29

ÚVOD

Telefon je již téměř 100 let běžnou součástí firem. Firmy však mají v mnoha případech desítky i stovky telefonů. Kdyby se každý měl připojit do veřejné sítě, byla by to ekonomicky velmi náročná záležitost. Místo toho se hledal způsob řešení, jak toto vše snadněji a jednodušeji obejít, a tak vznikla pobočková ústředna. Ta obecně obsluhuje vnitřní telefonní síť a propojuje ji s vnější sítí pomocí jednoho bodu.

Velmi důležitým bodem je zabezpečení hovoru proti odposlouchávání. Asterisk nabízí šifrování pomocí TLS protokolu, případně SRTP protokolu. Proč to ale nezkusit již na úrovni kodeku?

Tato práce se věnuje návrhu zabezpečení zvukového kodeku pro pobočkové ústředny Asterisk. V první kapitole bude osvětleno základní rozdělení pobočkových ústředen na tradiční výrobce a na ústředny s otevřeným kódem. Druhá kapitola se bude zabývat popisem struktury PBX Asterisk a jejím základním rozdílem oproti tradiční PBX.

Asterisk je založen na komponentách zvaných moduly, proto se práce dále bude zabývat nejstěžejnějšími moduly pro chod ústředny a také jejich dělením z hlediska podpory, či dělením podle typu použití a jejich vlastností. V této kapitole budou detailněji popsány zvukové kodeky A-law a μ-law, které jsou v současné době občas nahrazovány efektivnějšími kodeky, ale stále patří mezi základní kodeky audio signálu.

Třetí kapitola obsahuje jednoduchý návod, jak se zorientovat ve stavbě modulu pro PBX Asterisk, a dále je tento návod doplněn jednoduchým příkladem vytvoření modulu, ukázkou postupu jeho překladu, zprovoznění a načtení do Asterisku.

Simulace zabezpečení hlasu se nachází ve čtvrté kapitole, která přináší popis navrženého řešení zabezpečení s následnou realizací v programu Simulink. Tato simulace ověří funkčnost navrženého řešení zabezpečení telefonního hovoru.

V páté kapitole budou nastíněny používané historické techniky šifrování, především zrcadlení spektra a časové dělení signálu, a jejich srovnání se současnými moderními digitálními technikami.

V poslední šesté kapitole bude samotná realizace modulu zvukového kodeku se zabezpečením.

1 PBX - POBOČKOVÉ ÚSTŘEDNY

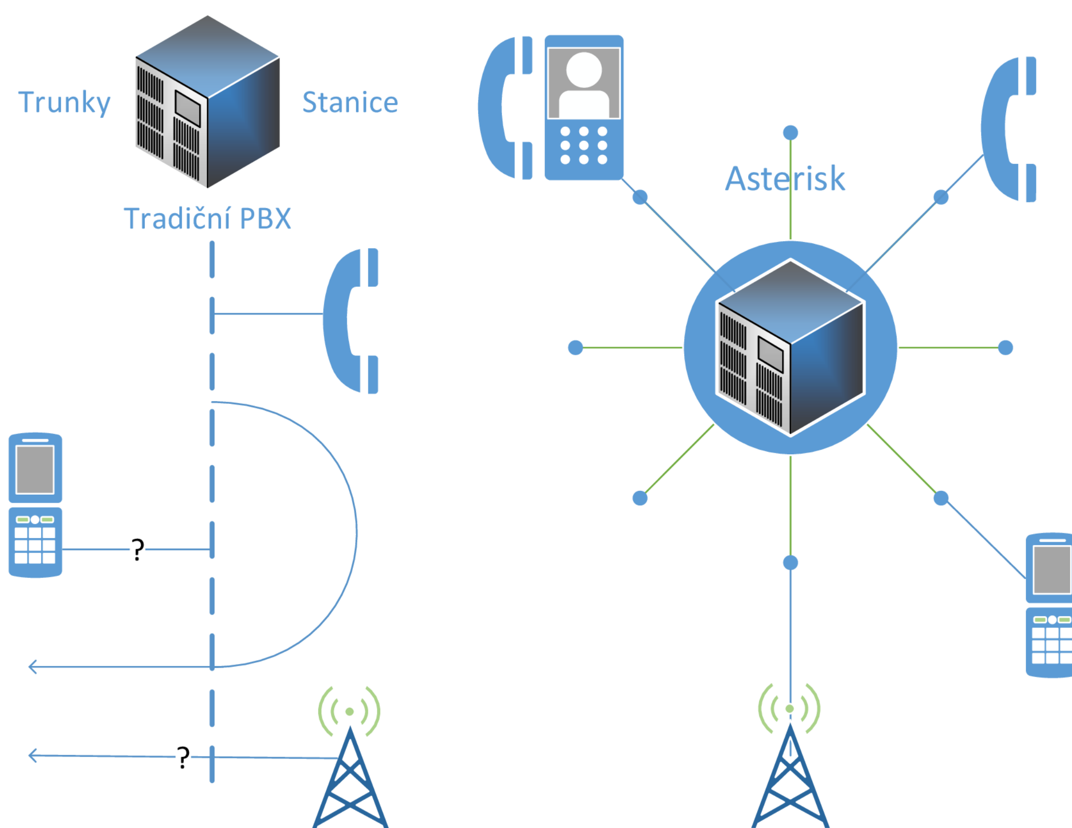
Pobočkové ústředny (PBX – Private Branch Exchange) jsou neveřejné ústředny sloužící pro uzavřený okruh uživatelů. Bývají propojeny do veřejné sítě, avšak mohou být i zcela izolované (domácí ústředny). Ústředny se z hlediska otevřenosti jejich kódu dělí zpravidla na:

1. tradiční výrobce, kteří využívají vlastní hardware (HW) a software (SW)
2. ústředny s otevřeným kódem

Tradiční výrobci využívají proprietární HW a SW, často si musí uživatel zakoupit i licenci. Jejich představitelé jsou především firmy Panasonic, Siemens, Cisco, Alcatel. Oproti tomu ústředny s otevřeným kódem využívají běžný HW, jejich představitelé jsou např. Asterisk, YATE, FreeSwitch [1][2].

2 ASTERISK

Asterisk je kompletní softwarová pobočková ústředna s otevřeným zdrojovým kódem pod licencí GNU/GPL. Z hlediska zacházení s příchozími kanály je odlišná oproti tradičních PBX ústřednen, pracuje se všemi kanály stejným způsobem. V tradiční ústředně je rozdíl mezi telefonními stanicemi a trunky, kterými se spojujeme s veřejnou sítí. To například znamená, že nemůžeme připojit bránu na port stanice a směrovat externí volání bez vytočení čísla dané linky [1].



Obr. 2.1: Rozdíl mezi tradiční PBX a Asteriskem [1]

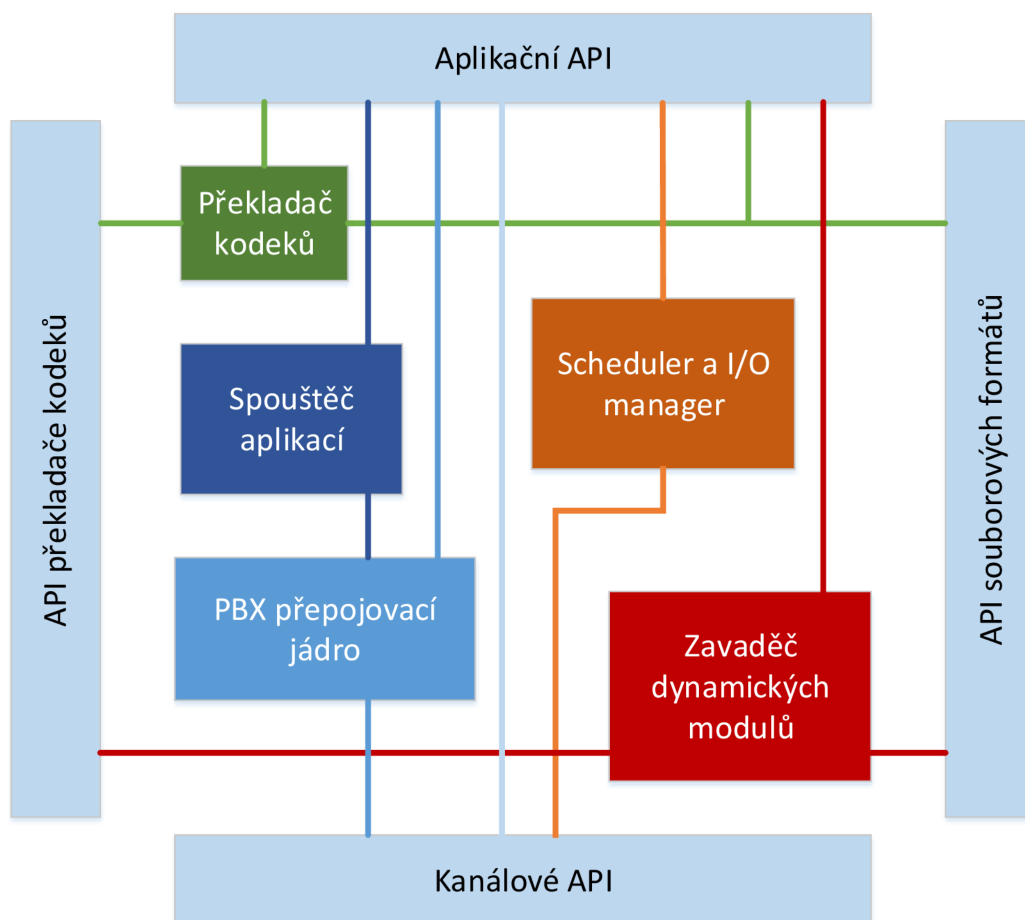
2.1 Architektura Asterisku

Asterisk je složen z centrálního jádra a specificky definovaných aplikačních rozhraní API. Jádro ovládá vnitřní propojení PBX, tj. ovládá specifické protokoly, kodeky, rozhraní HW telefonních aplikací. Tímto Asterisk použije vhodnou technologii a vykoná propojení HW s aplikací. Jádro řídí tyto položky [1]:

- **PBX spojování (PBX Switching)** – hlavním cílem Asterisku je především propojování v PBX, spojování mezi různými uživateli a automatizovanými

úlohami. Přepojovací jádro transparentně spojuje příchozí volání na různých HW a SW rozhraních.

- **Spouštěč aplikací (Application Launcher)** – spouští aplikace zajišťující služby, jako jsou například hlasová pošta, přehrání souboru a výpis adresáře.
- **Překladač kodeků (Codec Translator)** – používá moduly kodeků pro kódování a dekódování různých zvukových kompresních formátů používaných v telefonním prostředí. Množství dostupných kodeků je vhodné pro různorodé potřeby a docílení stavu rovnováhy mezi zvukovou kvalitou a použitou šířkou pásma.
- **Scheduler a I/O manažer (Schedule and I/O manager)** – slouží k ovládání nízkoúrovňových úloh a systémového řízení pro optimální výkon podle stavu zatížení.



Obr. 2.2: Blokové schéma architektury Asterisku [3]

2.2 Hardwarová náročnost

Asterisk lze jednoduše přizpůsobit z hlediska potřebné výkonnosti hardwaru tím, že vypustíme nepotřebné a nedůležité součásti. Tímto můžeme jednotky i desítky uživatelů propojit na počítači i s relativně malým výkonem, např. oblíbené Raspbery PI zvládne obsloužit najednou i 10 hovorů [4]. Při náročnějších aplikacích je již třeba výkonnějšího počítače, zvláště pokud chceme provádět konverzi mezi kodeky, používat digitálního recepčního IVR, přehrávat hudbu při čekání na spojení, nahrávání hovorů apod.

2.3 Moduly

Asterisk je založen na komponentách zvaných moduly. Jednotlivé moduly mají na starost různé funkce ústředny. Ty jsou zaváděny na základě konfiguračního souboru `/etc/asterisk/modules.conf`. Důležité jsou např. moduly: app_dial, app_stack, app_voicemail, codec_alaw, codec_dahdi, codec_ulaw, res_rtp_asterisk. V podstatě je možné Asterisk spustit bez jakéhokoli načteného modulu, avšak v takovémto případě neplní ústředna žádnou funkci. Je to ale výhodné v případě, kdy chceme výkonově systém optimalizovat a odstranit vše, co nepotřebujeme. Pro výpis všech aktuálně zavedených modulů použijeme příkaz

`module show`.

Asterisk podporuje následující kodeky [5]:

1. Audio kodeky:
    - G.711 ulaw (USA) – (64 Kbps).
    - G.711 alaw (Evropa) – (64 Kbps).
    - G.722 (širokopásmový kodek 7 kHz) – (64 Kbps).
    - G.723.1 – pouze pass-through režim
    - G.726 – (16/24/32/40kbps)
    - GSM – (12–13 Kbps)
    - iLBC – (15 Kbps)
    - LPC10 – (2.5 Kbps)
    - Speex – (2.15–44.2 Kbps)
    - G.729 – nutná licence (8Kbps)
    - SILK – nutná licence (superwideband – 12 kHz)
    - Siren7 – nutná licence, G.722.1 , 7 kHz
    - Siren14 – nutná licence, G.722.1 AnnexC, 14 kHz
2. Video kodeky:
    - H.263

- H.264

Asterisk podporuje následující protokoly:

- SIP
- H.323
- IAX2
- MGCP
- SCCP (Cisco Skinny)
- Nortel unistim
- Jingle

V Asterisku existují rozdílné druhy podpory, podle které se dělí moduly na: hlavní (core), rozšířené (extended) a zastaralé (deprecated).

1. **Hlavní**
   Základní moduly dostávají největší stupeň podpory. Většina modulů spadá do této kategorie. Pokud nalezneme problém s těmito moduly, můžeme ho nahlásit do sledování problému (Asterisk issue tracker), kde bude problém evidován a umístěn do fronty k řešení. Největší část prostředků sponzora Digium je věnována právě tomuto druhu podpory.

2. **Rozšířené**
   Rozšířené moduly jsou podporovány komunitou Asterisku, mohou, ale i nemusí mít aktivní vývojáře. Obecně lze říci, že mají nižší úroveň podpory než předešlé, a proto ohlášené problémy nebudou řešeny tak rychle a nebude jim věnována velká pozornost.

3. **Zastaralé**
   Zastaralé moduly mají nulovou úroveň podpory. Moduly spadající do této kategorie mají obvykle své alternativy v hlavních nebo rozšířených modulech.

Dále můžeme moduly rozdělit do skupin podle jejich typu na [1]:

- **Aplikace**
  Rozvrh aplikací je daný v souboru extensions.conf a definuje rozdílné chování, které je následně použito pro hovor. Například aplikace `Dial()` je zodpovědná za provedení odchozího spojení, což je jedna z nejdůležitějších aplikací.

- **Přemostění (Bridging)**
  Tyto moduly provádějí přemostění kanálu do nové aplikace. Každý modul nabízí různé funkce v závislosti na tom, kterou daný most aktuálně potřebuje.

- **CDR – Call Detail Recording**
  Tyto moduly jsou vytvořeny pro zachycení různých detailů hovoru. Následně se mohou využít např. ke vzdálené autentizaci uživatele (RADIUS).

- **CEL – Channel Event Logging**
  Modul CEL poskytuje mnohem silnější kontrolu nad výpisy aktivity volání. Z tohoto důvodu však vyžaduje pečlivé plánování svého dialplánu.

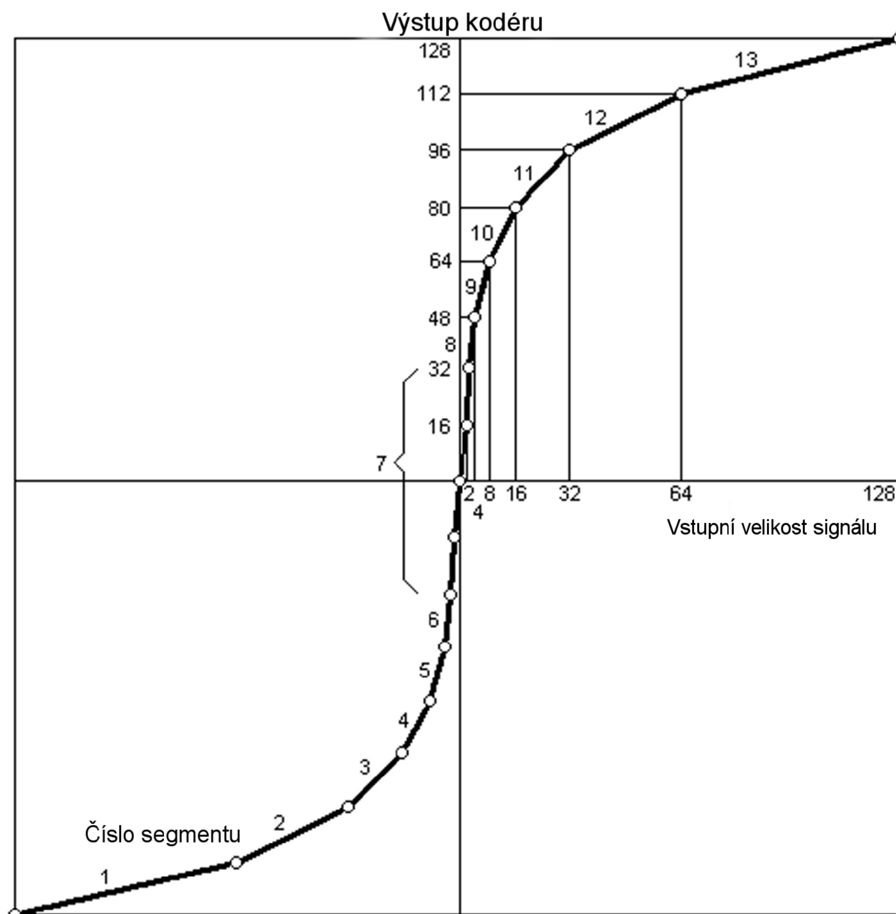
- **Channel driver**
  Bez tohoto ovladače by Asterisk nebyl schopen provádět hovory. Každý ovladač je specifický pro protokol nebo typ kanálu, který podporuje (např. SIP, ISDN a jiné). Jedná se o typ modulu, který dokáže přistupovat k jádru Asterisku.

- **Překladač kodeků**
  Překladač kodeků umožňuje Asterisku konverzi mezi různými typy zvukových formátů. Tato konverze však u některých typů zvukových formátů dosti vytěžuje procesor. Proto se pro větší ústředny používá speciální hardware od firmy Sagoma nebo Digium pro kódování/dekódování např. kodeku G.729.

2.4 Základní zvukové kodeky využívané v PBX Asterisk

2.4.1 A-law

A-law využívá pro modulaci Pulzní kódovou modulaci PCM (Pulse Code Modulation). Spadá společně s kodekem μ-law pod standard G.711. V tomto algoritmu je nejprve provedena lineární 13bitová modulace signálu a následně logaritmická komprese do 8 bitů. Důvod, proč je provedena tato komprese, vyplývá z citlivosti lidského ucha na zvuk, která je taktéž v logaritmickém měřítku – většina lidí si ani nevšimne provedené komprese. Vzorkování signálu je provedeno s frekvencí 8 kHz. Tento kodek tedy požaduje přenosový kanál o rychlosti 64 kb/s. Po dekódování tohoto signálu je zpětně převeden na 13bitový. V kombinaci s antialiasingovým filtrem dekódujeme audiosignál o standardním telefonním rozpětí frekvencí přibližně 300 Hz–3400 Hz. Mezi výhody daného kodeku patří jednoduchost a velmi malé zpoždění – každých 0,125 ms je vyroben vzorek popisující akustickou vlnu. Kodek A-law se používá v zemích Evropy, také v Austrálii a v mnoha dalších zemí světa.



Obr. 2.3: Kódování A-law [6]

2.4.2 u-law

Kodek μ-law pracuje na stejném principu jako A-law. Rozdíl je pouze v parametrech logaritmické funkce. Příchozí audio signál je navzorkován s rozlišením 14 bitů a zpětně je také převeden na 14 bitů. Tímto se teoreticky dosáhne o něco lepší kvality zvuku, než u A-law. Tento kodek je využíván primárně v Severní Americe a v Japonsku.

3 JAK PSÁT MODULY PRO ASTERISK

Stavba modulu pro Asterisk je poměrně jednoduchá. Můžeme si to ukázat na příkladu „Hello world“. Pro potřeby diplomové práce byl využit virtuální stroj s Asteriskem 1.8 používaném v předmětu MTIS, který je možný stáhnout na adrese <`http://anca.utko.feec.vutbr.cz/vyuka/mtis`>.

Vytvoříme si soubor `res_helloworld` ve složce
`home/Install/asterisk-1.8.2.3/res`.
V první řadě musíme v každém modulu zahrnout hlavní hlavičkový soubor Asterisku:

```
#include "asterisk.h"
```

Dále zahrneme makro

```
ASTERISK_FILE_VERSION(__FILE__, "$Revision: $"),
```

kde doplníme číslo revize. To zjistíme z konzolového okna Asterisku po zadání

```
core show file version like res_helloworld.c.
```

Zahrneme také hlavičkový soubor modulů Asterisku, který obsahuje definice nezbytné pro implementaci našeho modulu:

```
#include "asterisk/module.h".
```

Poslední hlavička, kterou budeme používat, se nazývá `logger` a slouží k tomu, abychom mohli používat rozhraní logování Asterisku. To nám umožní vypisovat zprávy do konzole Asterisku,

```
#include "asterisk/logger.h".
```

Asterisk Logger umožňuje vypisovat zprávy podle jejich povahy či důležitosti:

- notice,
- warning,
- error,
- debug,
- verbose,
- dtmf.

Nastavení vypisování či nevypisování těchto jednotlivých typů zpráv lze nastavit v konfiguračním souboru `logger.conf` umístěném v `/etc/asterisk/`.

Základní dvě funkce, které musí každý modul bezpodmínečně obsahovat, jsou `load_module()` a `unload_module()`. Tyto funkce jsou volány, když je modul nahrán a nebo uvolněn z Asterisku. V nadsázce je úkolem funkce `load_module()` říci

Asterisku: „Ahoj Asterisku, já jsem modul a mohu poskytnout nějaké funkce Asterisku, zde je to, jak je lze použít". Opačnou práci vykonává `unload_module()` který Asterisku říká: „Ahoj Asterisku, chystám se zmizet, proto prosím nepoužívej dále žádnou z těchto mých funkcí, které jsem ti poskytoval".

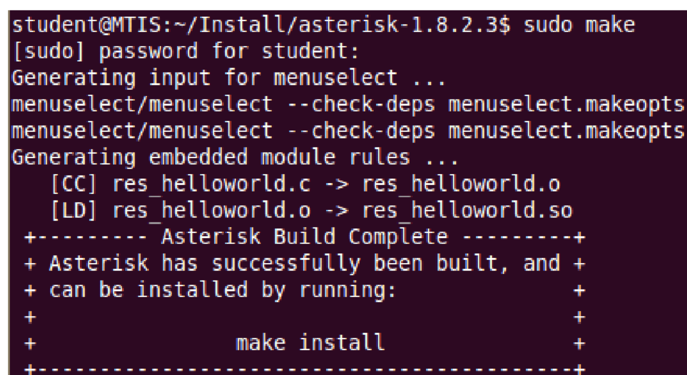
```
static int load_module(void)
{
ast_log(LOG_NOTICE, "Hello World!n");
return AST_MODULE_LOAD_SUCCESS;
}

static int unload_module(void)
{
ast_log(LOG_NOTICE, "Goodbye World!n");
return 0;
}
```

Každý modul Asterisku musí obsahovat instanci jednoho z `AST_MODULE_INFO` maker. Toto makro obsahuje potřebný kód modulu pro jeho správnou registraci s jádrem Asterisku,

```
AST_MODULE_INFO_STANDARD(ASTERISK_GPL_KEY, "Hello World");.
```

Nyní již máme testovací modul napsaný a stačí se přesunout v příkazovém řádku do složky Asterisku (`cd home/Install/asterisk-1.8.2.3`) a zadat příkaz `sudo make` (heslo pro zmíněný virtuální stroj je student*10).



Obr. 3.1: Konzolové okno s výpisem po zadání příkazu sudo make

Právě vytvořený přeložený soubor `helloworld.so` je dále třeba zkopírovat do adresáře pro moduly (`usr/lib/asterisk/modules`) a případně restartovat Asterisk

```
MTIS*CLI> module load res_helloworld.so
Loaded res_helloworld.so
[May  3 14:57:49] NOTICE[5579]: res_helloworld.c:10 load_module: Hello World!n Loaded res_helloworld.so => (Hello World)
MTIS*CLI>
```

Obr. 3.2: Konzolové okno s výpisem po zadání příkazu pro načtení modulu

příkazem `core restart now`. Následně příkazem `module load res_helloworld.so` načteme modul do Asterisku a vypíše se nám hlášení o zavedení modulu.

Obdobně můžeme modul odstranit příkazem `module unload res_helloworld.so`.

```
MTIS*CLI> module unload res_helloworld.so
Unloaded res_helloworld.so
[May  3 15:00:14] NOTICE[5579]: res_helloworld.c:16 unload_module: Goodbye World!
```

Obr. 3.3: Konzolové okno s výpisem po zadání příkazu pro odstranění modulu

Případně pro ověření, že je modul zaveden, můžeme využít příkazu `module show like res_helloworld`.

```
MTIS*CLI> module show like res_helloworld.so
Module                          Description                         Use Count
res_helloworld.so               Hello World                         0
1 modules loaded
```

Obr. 3.4: Konzolové okno s výpisem po zadání příkazu pro ověření načteného modulu

Dále si zběžně ukážeme, jak implementovat příkazy Asterisk CLI, které jsou velmi užitečné pro zobrazení konfigurace, statistiky a jiné účely (např. debugování). Nejdříve je potřeba doplnit hlavičkový soubor, který definuje rozhraní příkazového řádku (CLI).

`#include "asterisk/cli.h"`.

Napíšeme si funkci pro příkazový řádek „echo“, která vypíše zpět první argument příkazu příkazového řádku. První řádek funkce definuje vstupní proměnné funkce. Argument `ast_cli_entry *e` obsahuje informace o vykonávaném příkazu. Argument `cmd` je nastaven, pokud je příkaz volán z jiného prostředí než z příkazové řádky. Poslední argument `ast_cli_args` obsahuje specifické informace pro vykonání příkazu.

```
static char *handle_cli_echo(struct ast_cli_entry *e, int cmd, struct
                    ast_cli_args *a)
```

Dále napíšeme příkaz `switch`, kterým se zpracuje argument `cmd`. Přepínání se bude volit mezi dvěma možnostmi:

- CLI_INIT – vykoná zobrazení dokumentace k funkci echo a další možné použití funkce,
- CLI_GENERATE – vykoná zobrazení možných příkazů po zmáčknutí tabulátoru, zde je třeba přidat požadovaný kód.

```
switch (cmd) {
case CLI_INIT:
    e->command = "echo";
    e->usage =
        "Usage: echo <stuff>n"
        "       Print back the first argument.n"
        "Examples:n"
        "       echo foon"
        "       echo "multiple words"n"
        "";
    return NULL;
case CLI_GENERATE:
    return NULL;
}
```

Další část kódu zjišťuje, že byl poskytnut alespoň jeden argument. Argument `ast_cli_entry` zde určuje, kolik argumentů bylo s příkazem vloženo, a argument `ast_cli_args` nám říká, kolik argumentů je právě specifikováno v příkazovém řádku. Pokud jsou si tyto argumenty rovny, funkce echo byla vykonána.

```
if (a->argc == e->args) {
ast_cli(a->fd, "You did not provide an argument to echonn");
return CLI_SHOWUSAGE;
}
```

Nakonec vytiskneme do příkazového řádku jeden argument a funkce navrátí hodnotu, že provedení příkazu v příkazové řádce bylo úspěšné.

```
ast_cli(a->fd, "%sn", a->argv[1]);
return CLI_SUCCESS;
```

Dále musíme přidat tabulku příkazů, která je součástí tohoto modulu. Použijeme `AST_CLI_DEFINE ()` pro přidání záznamu do této tabulky. `AST_CLI_DEFINE` zahrnuje ukazatel na funkci `handle_cli_echo` a také stručný přehled o tom, co příkaz vykonává.

```
static struct ast_cli_entry cli_helloworld[] = {
AST_CLI_DEFINE(handle_cli_echo, "Echo to the CLI"),
};
```

Nakonec napíšeme funkci pro odstranění modulu z Asterisku:
`ast_cli_unregister_multiple(cli_helloworld, ARRAY_LEN(cli_helloworld));`,
a také pro načtení modulu:
`ast_cli_register_multiple(cli_helloworld, ARRAY_LEN(cli_helloworld));`.

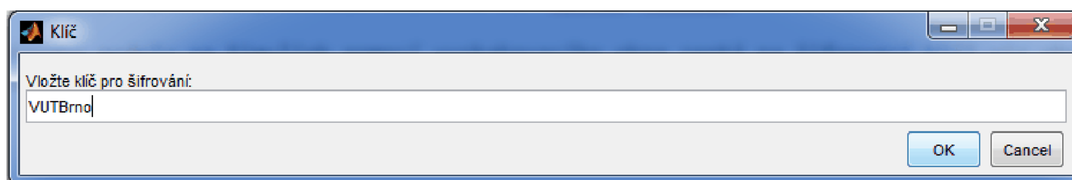
Nyní si můžeme vyzkoušet zadat do příkazového řádku např.: `help echo`, `echo hello world` či `"echo hello world"`. V prvním případě nám příkaz vyvolá nápovědu, ve druhém vypíše pouze první slovo (argument `hello`) a v třetím případě navrátí všechna slova (`hello world`) [7].

4 SIMULACE ZABEZPEČENÍ HLASU V PROGRAMU SIMULINK

Z hlediska realizace šifrování audio dat je vhodné si návrh před samotnou implementací odsimulovat. Pro tuto simulaci byl zvolen program Simulink, který je součástí programu MATLAB. MATLAB je integrované prostředí pro vědeckotechnické výpočty, modelování, návrhy algoritmů, simulace, analýzu a prezentaci dat, paralelní výpočty, měření a zpracování signálů, návrhy řídicích a komunikačních systémů. MATLAB je nástroj jak pro pohodlnou interaktivní práci, tak pro vývoj širokého spektra aplikací. Simulink je potom nadstavba MATLABu pro simulaci a modelování dynamických systémů, který využívá algoritmy MATLABu. Pro svou simulaci jsem využíval MATLAB R2014a (verze 8.3).

Simulink lze spustit napsáním příkazu `simulink` na příkazovou řádku v okně Command Window, po jejímž potvrzení se nám následně zobrazí okno Simulink Library Browser, kde pomocí karty `File -> New -> Model` vytvoříme nový model. Jiná možnost spuštění Simulinku a nového modelu se nám nabízí pomocí kliknutí na ikonku Simulinku nebo výběru v kartě `Home -> New -> Simulink model`. Do něj lze přetáhnout jednotlivé bloky z knihoven Simulinku. Více informací pro práci s MATLABem a Simulinkem lze nalézt v nápovědě či na webové adrese <`http://www.mathworks.com/`>. Model následující simulace je k dispozici zde C. Pro simulaci kódování a dekódování hlasu či zvuku byla použita především knihovna DSP System Toolbox.

Před spuštěním vytvořeného modelu se Simulink pomocí vyskakovacího okna zeptá na šifrovací klíč viz obr 4.1.



Obr. 4.1: Zadávání klíče pro šifrování hlasu

Nastavení tohoto okna se nachází v horní liště File -> Model Properties -> Model Properties. Zde v kartě Callbacks -> InitFcn. Zde se nachází následující kód:

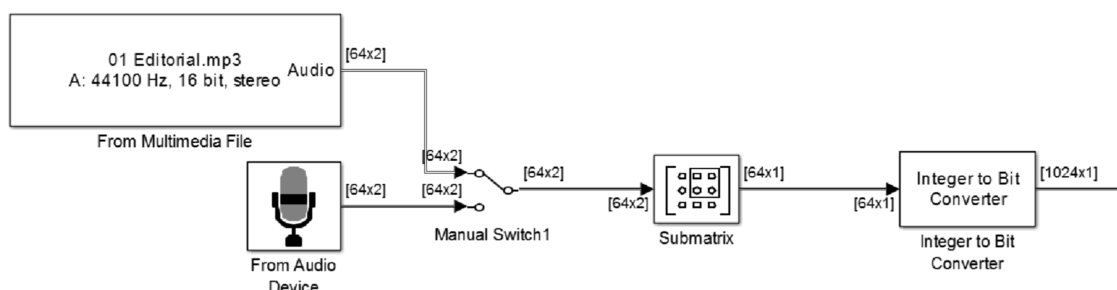
```
x = inputdlg('Vložte klíč pro šifrování:',...
'Klíč', [1 120]);
k = dec2bin((x{:}),8);
key = k(:)'-'0';
```

který vytvoří ono vyskakující okno. Vzor pro tento kód byl převzat ze stránek Mathworks Documentation [9].

Vstup dat realizovaného modelu je možné provést dvěma způsoby:

1. použitím bloku `From Audio Device`, tzn. mikrofonem, který je připojený k počítači,
2. použitím bloku `From Multimedia File`, v nastavení tohoto bloku si nastavíme cestu k audio souboru (Simulink podporuje velkou škálu formátů, např. soubory s příponou .wav, .mp3, .wma, .m4a, .ogg a další),

a mezi těmito vstupy si lze přepínat pomocí přepínače `Manual Switch 1`.



Obr. 4.2: Vstupní bloky modelu simulace

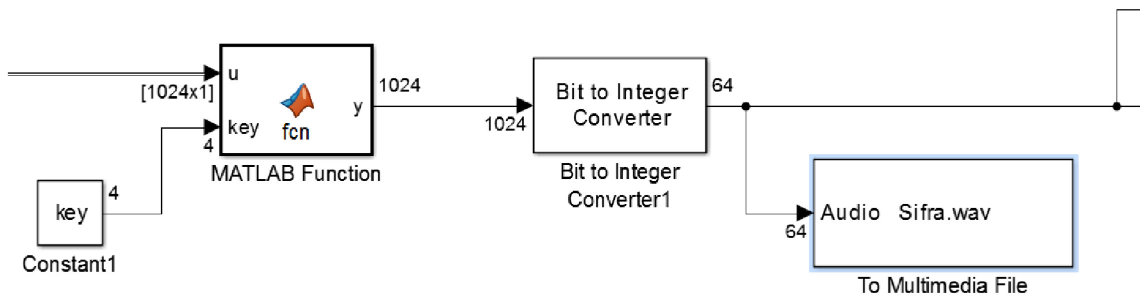
Následujícím blokem se u stereo signálu ořeže jeden kanál a dál vstupuje signál pouze jednokanálově (což se pro hlasové hovory používá výhradně). Blok `Integer to Bit Converter` převádí vektor integerů na vektor bitů. Tato úprava je nutná, jelikož operace XOR je bitová operace.

Následuje konečně šifrovací blok, který je realizovaný pomocí MATLABovské funkce zvané M-funkce. Obsahuje vstupní argumenty, vektor bitů u a zadaný klíč key, a dále výstup funkce y, tedy vektor zašifrovaných bitů. Uvnitř funkce se nejprve vytvoří šifrovací klíč c_key z klíče key tak, aby byl stejně dlouhý jako vektor dat u. Poté se provede pomocí operace XOR šifrování a uložení do výstupu funkce y:

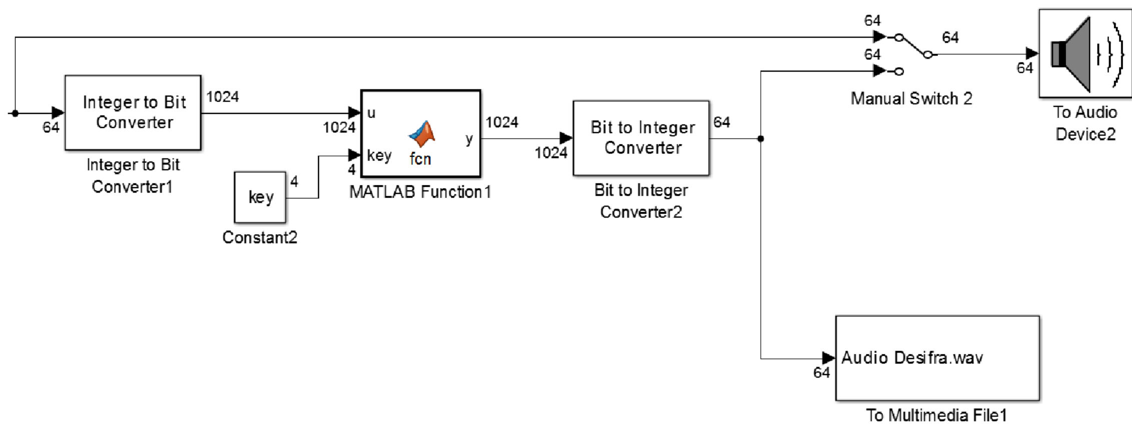
```
y = xor(u,c_key);
```

Poslední část šifrovacího bloku je převod vektoru bitů na vektor integerů a následné uložení do souboru `Sifra.wav` nebo přehrání pomocí reproduktorů.

Dešifrovací blok viz obr 4.4 je velmi podobný šifrovacímu (což vychází z podstaty symetrického šifrování). V první řadě se převede vektor z integerů na bitový vektor, dále se využije stejná M-funkce jako při šifrování. Opět se vytvoří c_key, musí být stejný jako při šifrování, a provede se opět operace XOR vektoru u a c_key. Výstupní vektor bitů se převede do vektoru integerů a uloží se do souboru `Desifra.wav`, dále může být přehrán v počítači a porovnán se zašifrovaným signálem přepnutím přepínače `Manual Switch 2`.



Obr. 4.3: Šifrovací část modelu simulace



Obr. 4.4: Dešifrovací část modelu simulace

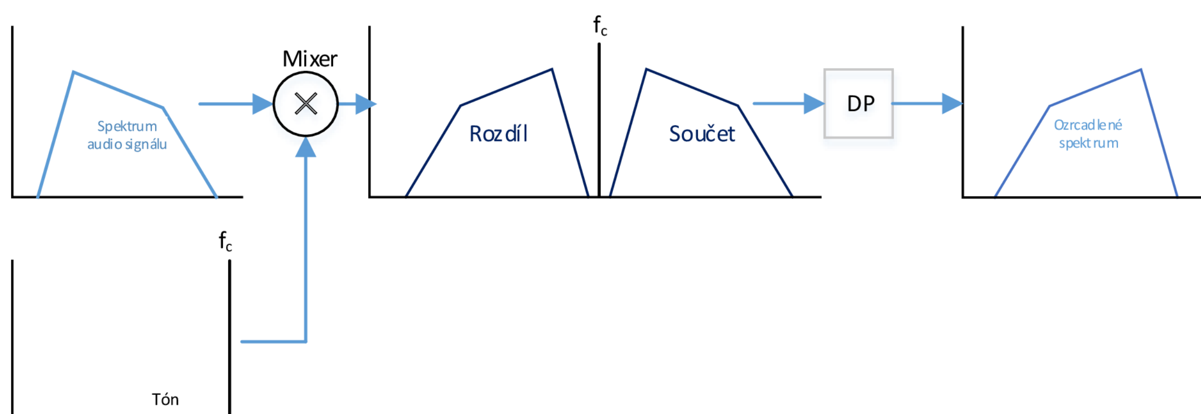
5 HISTORIE ŠIFROVÁNÍ HLASU

Z historického hlediska nastal největší „boom“ v prolomení různých dnes již jednodušších algoritmů (např. šifra A-3) především během druhé světové války. Následně byly vytvářeny nové odolnější šifrovací algoritmy a šifrovací jednotky (specializovaná zařízení), které se používaly pro šifrování hovoru. Dnes již drtivá většina hovorových šifrovacích koncových zařízení používá pro šifrování data v digitální podobě [8].

5.1 Skramblování hlasu

5.1.1 Zrcadlení spektra

Jedná se o techniku, která předcházela digitálnímu zpracování a šifrování, a byla hojně užívána od 70. do 90. let minulého století v mnoha zemích, především policií. Její princip spočívá ve frekvenční inverzi – ozrcadlení zvukového frekvenčního spektra kolem dané středové frekvence, případně spektra rozděleného do více frekvenčních pásem. Audio spektrum se tedy smísí s nosnou frekvencí *fc*, čehož výsledkem jsou dvě spektra: jedno vyšší spektrum, které je dáno součtem nosné frekvence a audio spektra, a druhé nižší, které je dáno jejich rozdílem. Dále je dolní propustí (DP) vyfiltrováno vrchní spektrum a zůstane jen nižší rozdílové spektrum, které se s výhodou nachází v audio frekvenčním pásmu (oproti jiným šifrovacím technikám, které zvětšují šířku pásma přenášeného signálu). Na straně příjemce se spektrum ozrcadlí ještě jednou, a poté je zvuk opět srozumitelný.



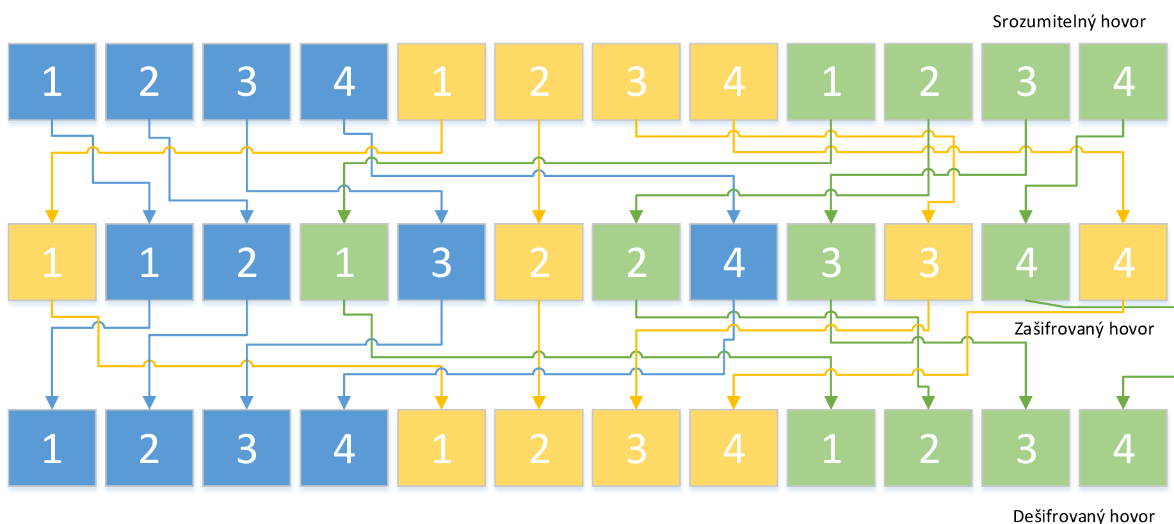
Obr. 5.1: Postup zrcadlení spektra

Pro větší bezpečnost je možné audio spektrum rozdělit na více menších pásem a každé toto pásmo je zrcadlené zvlášť s jinou nosnou frekvencí. Průběžnou změnou těchto parametrů (teď již myšleno spíše pomocí číslicového zpracování signálu) se

stává dekódování nežádoucí osobou mnohem složitější. Dnes jsou však tyto skramblery považovány ze své podstaty za nespolehlivou metodu, protože je možné je dešifrovat i za pomocí poměrně jednoduché elektroniky.

5.1.2 Časové dělení

Druhou technikou skramblování je pomocí časového dělení (Time-division Speech Scrambler), která je o něco málo bezpečnější než předchozí metoda, ale stále mnohem méně bezpečná než moderní digitální šifrátory. Podstata se nachází v tom, že se hovor rozdělí na krátké úseky, a ty si pak navzájem mění své pořadí podle měnícího se pořádku. Toto pořadí je dáno pseudonáhodnou posloupností, která je odvozena z inicializačního semene (v některé literatuře uváděné jako semínko, anglicky seed) vloženého uživatelem. Tento systém je však také náchylný ke kryptografickým útokům, při zkoumání signálu na osciloskopu jsme schopni rekonstruovat původní signál, a tedy i odvodit původní klíč. Z podstaty časového dělení hovoru je jasné, že další nevýhodou bude i velké zpoždění dané skramblováním a deskramblováním, které činí přibližně 500 ms, což je pro kvalitní hovor již za hranicí a koncoví uživatelé nemusejí být s tak vysokou prodlevou, a tudíž kvalitou hovoru, spokojeni (hraniční hodnoty pro přijatelné zpoždění hovoru jsou 150–400 ms).



Obr. 5.2: Příklad postupu skramblování pomocí časového dělení [8]

U mnoha šifrovacích zařízení probíhá šifrování následovně. V první řadě je řeč u účastníka A převedena analogově-digitálním (A/D) převodníkem na datový tok, který se pomocí operace XOR převede společně s datovou posloupností z generátoru pseudonáhodné posloupnosti, jež je závislá na vstupním klíči, na tok výstupních zašifrovaných dat. Tato data jsou přenesena druhému účastníkovi B, který provede

opět operaci XOR s přijatými daty a stejnou pseudonáhodnou posloupností, jakou byla data zašifrována. Výsledkem jsou původní nešifrovaná data, která po projití digitálně-analogovým (D/A) převodníkem jsou již člověku srozumitelná.

5.1.3 Delta modulátory

Během 70. let 20. století se používalo mnoho systémů, které využívaly pro A/D převod delta modulátory s plynule měnícím se sklonem. Tato technika však již potřebuje velmi velkou šířku pásma, využívá pásma velmi krátkých vln VKV (VHF) a ultra krátkých vln UKV (UHF), a proto ji v 80. letech vytlačila praktičtější úzkopásmová zařízení.

6 ZVOLENÁ METODA ŠIFROVÁNÍ

6.1 Požadavky na klíč

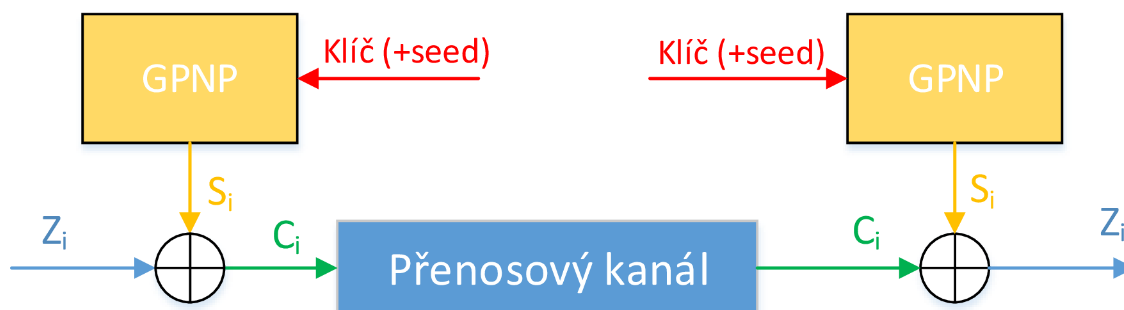
Aby byla zaručena nemožnost rozšifrovat odposlechnutou komunikaci, a tím se dostat do maximálního stupně utajení, je nutné dodržet následující podmínky pro klíč (plynoucí z dokonalé – Vernamovy šifry):

- klíč musí být zcela náhodný (generovaný např. z tepelného šumu polovodiče),
- klíč musí být použit jen jednou a následně se musí zlikvidovat (při použití stejného klíče vícekrát je útočník schopen odvodit klíč, kterým se šifrovalo),
- klíč musí být stejně dlouhý jako zpráva.

Tyto podmínky jsou poměrně nepraktické, protože velikost klíče vychází velmi velká a klíč se těžko přenáší bezpečnou cestou oběma komunikujícím stranám. Používá se proto jen pro přenos velmi důležitých a přísně utajovaných zpráv (příklad „horké linky“ Bílý dům–Kreml během studené války). Pro naši aplikaci se smíříme s nižším, ale pro běžné aplikace zcela vyhovujícím, stupněm zabezpečení. Použijeme generátor pseudonáhodné posloupnosti, který nám na základě klíče relativně malé velikosti vygeneruje posloupnost v podstatě libovolné velikosti, se kterou následně budeme šifrovat audio signál.

6.2 Generátor pseudonáhodné posloupnosti

Pro zabezpečený přenos audio signálu se často využívá proudových šifer, jelikož jsou velmi rychlé a ne příliš náročné na hardware. Tato šifra pracuje přímo s bity audio dat, tzn. šifrují bit po bitu pseudonáhodnou posloupnost s těmito daty, jak lze vidět na obrázku 6.1. V našem případě využíváme jako generátor pseudonáhodné posloupnosti (GPNP) již implementovanou funkci v jazyce C – `rand()`.



Obr. 6.1: Proudová šifra využívající GPNP

Z_i představuje na obrázku 6.1 i-tý bit zprávy, dále i-tý bit pseudonáhodné posloupnosti, která je určena klíčem a případně semenem, představuje veličina S_i. Z těchto dvou vstupů operátor XOR vyprodukuje C_i, což je zašifrovaný bit kryptogramu. Tento bit se přenese přes přenosový kanál a společně s odpovídajícím bitem pseudonáhodné posloupnosti se stanou vstupy druhé funkce XOR. Výstupem se stává opět původní zpráva Z_i.

Při šifrování ve zvukovém kodeku PBX Asterisk budeme využívat jak stálý tajný klíč zadaný uživatelem do konfiguračního souboru Asterisku `key.conf`, tak i tzv. semeno (anglicky seed) přidávané do každého rámce audio dat, a to z toho důvodu, aby se pseudonáhodná šifrovací posloupnost stále měnila. Tím je zajištěna možnost, jak se vypořádat s výpadkem rámce po trase, a dále nemožnost rozšifrovat hovor jinak než hrubou silou (vyzkoušením všech možností klíčů).

Rozdíl mezi náhodnou a pseudonáhodnou posloupností je v jejím vytvoření. Náhodná posloupnost má zcela náhodný charakter a je založena na základě náhodných jevů (např. radioaktivní rozpad atomů či teplotní šum). Oproti tomu pseudonáhodná posloupnost je generována deterministickým způsobem (počítačem), avšak na základě statistických testů se jeví jako zcela náhodná [10]. Generátor pseudonáhodné posloupnosti (GPNP) je tedy generátorem, který tvoří onu pseudonáhodnou posloupnost.

6.3 XOR funkce

Funkce XOR, někdy označovaná jako exkluzivní OR nebo exkluzivní disjunkce, je logická operace, jejíž výstup je obecně pravda, pokud každý vstup nabývá jiné hodnoty v porovnání s ostatními vstupy. Hojně se tato operace používá v kryptografii např. v šifře AES, DES, dále také v hešovací funkci. Operuje se na bitové úrovni, máme tedy dva vstupy a jeden výstup, což vidíme z pravdivostní tabulky 6.1. Tato operace bývá obvykle integrována přímo do procesoru, a tudíž je vykonávána velice efektivně.

Tab. 6.1: Pravdivostní tabulka operace XOR

$$\begin{array}{ccccc} 0 & \oplus & 0 & = & 0 \\ 0 & \oplus & 1 & = & 1 \\ 1 & \oplus & 0 & = & 1 \\ 1 & \oplus & 1 & = & 0 \end{array}$$

6.4 Šifrování ve zvukovém kodeku PBX Asterisk

V této podkapitole bude vysvětlen kód kodeku A-law a kodeku μ-law, které byly upraveny pro podporu šifrování hlasových dat. Realizace šifrovaného hovoru probíhá následujícím způsobem: Strana A používá kodek A-law, strana B kodek u-law. To proto, aby docházelo k překladu daných kodeků do formátu slin, který je ještě před odesláním protistraně šifrován. Protistrana obdobně ještě před překladem do svého kodeku data ve formátu slin rozšifruje. Důležitá je poznámka, že všechny kodeky, kromě již zmíněných A-law a u-law, se nebudou načítat. Toho docílíme úpravou konfiguračního souboru `module.conf`, podle kterého se načítají či nenačítají jednotlivé moduly při startu Asterisku. Např. nakopírováním následujícího kódu do souboru `module.conf`:

```
;nechci loadovat kodeky kromě A-law a u-law
noload => codec_adpcm.so
noload => codec_alaw_test.so
noload => codec_dahdi.so
noload => codec_g722.so
noload => codec_g726.so
noload => codec_lpc10.so
noload => codec_a_mu.so
noload => codec_gsm.so
```

V první řadě si zahrneme do kódu obou kodeků hlavičkový soubor logger.h:

`#include "asterisk/logger.h"`,

protože bez něj bychom nemohli provádět kontrolní výpisky.

První funkce `SeedRandomNumGenerator()` slouží k načtení pseudonáhodného čísla. V Linuxu je generátor implementován uvnitř jádra a náhodné bajty se načítají ze speciálního souboru `/dev/urandom`. Existuje i varianta `/dev/random`, kde generátor, pro větší kryptografickou bezpečnost, udržuje odhad entropie, což je samozřejmě výhodnější. Avšak když je zásoba entropie prázdná, čtení z `/dev/random` bude blokováno, dokud nebude shromážděno dostatečné množství šumu. Tato možnost je pro náš účel zcela nevýhodná, protože pracujeme v reálném čase a jakékoliv větší zdržení by znamenalo zhoršení kvality hovoru, případně i zhavarování celé komunikace. Proto využíváme první variantu a spokojíme se i někdy s trochu menší entropií, protože pro účel semene (seed), který bude podrobněji popsán níže, tato skutečnost nemá kritický význam. Každý audio rámec má své vlastní semeno, tudíž toto číslo není používané dlouhodobě a není třeba dlouhodobé kryptografické ochrany, kterou poskytuje jen `/dev/random`.

V této funkci tedy otevřeme soubor `/dev/urandom` příkazem `fopen`, příkazem `fread` je ze souboru načítán obsah, tedy náhodná čísla, a plní se jím proměnná `seed` datového typu `unsigned int`. Tato čísla následně vložíme jako semeno pro generátor pseudonáhodné posloupnosti `rand()` implementovanou v jazyce C pomocí příkazu `srand(c_key);`

```
void SeedRandomNumGenerator()
{
    unsigned int seed;
    FILE *fd;

    fd = fopen("/dev/urandom", "r");
    if( fd )
    {
        fread(&seed, sizeof(seed), 1, fd);
        fclose(fd);
        srandom( seed );
    }
}
```

Následující funkce se jmenuje `ReadKey()` a vykonává načítání tajného klíče z konfiguračního souboru `key.conf` uloženém v aresáři `/etc/asterisk`. Velikost klíče `key` je 32 bitů. Pokud klíč není nalezen, je vypsáno hlášení s důležitostí „ERROR“ (viz kapitola 3) a komunikace nelze provést (toto je ošetřeno dále).

```
uint32_t ReadKey()
{
uint32_t key;
FILE *fd;

fd = fopen("/etc/asterisk/key.conf", "r");
if( fd ){
         fread(&key, sizeof(key), 1, fd);
         fclose(fd);
         return key;
}
ast_log(LOG_ERROR, "Nenalezen soubor s klíčem\n");
return 0;
}
```

Nyní si osvětlíme funkci `alawtolin_framein` v kodeku `codec_alaw.c`, která provádí převod z formátu A-law do slin a do které bylo implementováno navržené

zabezpečení zvukových dat. Délku dat jsme prodloužili o 4 bajty (velikost 2 vzorků), protože na začátku každého rámce posíláme inicializační vektor.

```
pvt->datalen += i * 2 + 4;
```

Následně voláme funkci `ReadKey()`, která nám načte klíč ze souboru a uloží ho do proměnné `key`. Pokud není klíč v souboru uložen, vypíše se hlášení o nenalezení klíče a hovor se neuskuteční, jak lze vidět z následujícího kódu.

```
if (key == 0){
ast_log(LOG_ERROR, "Nenalezen platný klíč\n");
i -= 4;
while (i--){
*dst++ = 0;
}
return 0;
}
```

Poté voláme funkci `SeedRandomNumGenerator();`, jež byla popsána výše. Její význam je ve správném fungování pseudonáhodného generátoru `rand()` tak, že náhodným číslem (semenem) naplníme funkci `srand()`.
Dále si vytvoříme proměnnou `random` typu `uint32_t`, kterou nám vyplní náhodnou hodnotou funkce `rand()`. Tuto hodnotu rozdělíme mezi proměnné `dst1` a `dst2` datového typu `uint16_t` následujícím způsobem:

```
uint32_t random = rand();

uint16_t dst1 = random;
*dst++ = dst1;

uint16_t dst2 = (random >> 16);
*dst++ = dst2;
```

Vytvoříme si proměnnou `c_key`, do které uložíme hodnotu `key` XOR `random` a vložíme ji jako semeno generátoru pseudonáhodné posloupnosti (do funkce `srand`). Tímto bude šifrovací posloupnost vždy jiná.

```
uint32_t c_key = key ^ random;
srand(c_key);
```

Proměnnou `tmp` potřebujeme na dočasné uložení vzorků po převodu do formátu slin. Následně jsou vzorky zašifrovány pomocí operace XOR a odeslány příjemci.

```
int16_t tmp;
while (i--){
tmp = AST_ALAW(*src++);
tmp = tmp ^ rand();
*dst++ = tmp;
}
```

Stejný postup je proveden i ve funkci `ulawtolin_framein` v kodeku `codec_ulaw.c`.

Dále si tedy vysvětlíme příjem a rozšifrování přijatých dat v kodeku `codec_ulaw.c`. Opět načteme klíč z konfiguračního souboru pomocí funkce `ReadKey()`, následně si uložíme první dva vzorky do speciálně připravených proměnných `src1` a `src2` datového typu `uint16_t`. Vytvoříme si proměnnou `random` datového typu `uint32_t` a spojíme do ní data z proměnných `src1` a `src2` tak, aby vznikl původní odeslaný inicializační vektor. Nyní do proměnné `c_key` uložíme XOR hodnot `key` a `random` a ten vložíme jako semeno do funkce `srand()`, podle následujícího kódu.

```
uint32_t key = ReadKey();
uint16_t src1 = *src++;
uint16_t src2 = *src++;
uint32_t random = src2;
random = random << 16;
random = src1 | random;
uint32_t c_key = key ^ random;
srand(c_key);
```

Do pomocné proměnné `tmp` uložíme příchozí vzorky a provedeme dešifrující operaci XOR s šifrovací posloupností a příchozímy vzorky. Poté vložíme do funkce pro převod do formátu μ-law a odešleme příjemci.

```
int16_t tmp;
while (i--){
tmp = *src++;
tmp = tmp ^ rand();
*dst++ = AST_LIN2MU(tmp);
}
```

Obdobným způsobem se dešifruje i ve funkci `lintoalaw_framein` v kodeku `codec_alaw.c`

Tímto byl hovor na straně odesílatele zašifrován a na straně příjemce dešifrován a tím byla provedena zabezpečená komunikace obou stran na úrovni kodeku pobočkové ústředny Asterisk.

7 ZÁVĚR

Cílem práce byl návrh a realizace modulu audiokodeku pro PBX Asterisk s podporou zabezpečení přenášených audio dat. Návrh šifrování byl realizován v programu Simulink, kde byl i následně otestován. Šifrování se zde vykonalo pomocí operace XOR audio dat s šifrovacím klíčem s tím, že klíč byl zadáván při startu simulace do dialogového okna. Vstup audio dat byl realizován načítáním zvuku ze souboru uloženém v počítači, nebo z mikrofonu v reálném čase. Mezi touto volbou se přepínalo přepínačem. Dále byla data upravena tak, aby šla jednoduše zašifrovat. Simulace ukládala zašifrovaná i následně dešifrovaná zvuková data do počítače, nebo je bylo možno ihned přehrát na počítači. Tato simulace ověřila funkčnost navrženého řešení zabezpečení telefonního hovoru.

Realizace a testování modulu audiokodeku se zabezpečením bylo provedeno na kodecích A-law a μ-law. Tajný klíč byl načítán z konfiguračního souboru Asterisku `/etc/asterisk/key.conf`. Oba zdrojové kódy kodeků obsahují dvě nové funkce: `SeedRandomNumGenerator()`, která slouží k načtení pseudonáhodného čísla z generátoru implementovaného uvnitř jádra Linuxu, a toto číslo je dále použito jako semeno (seed) pseudonáhodné posloupnosti pro generování inicializačního vektoru. Druhá funkce je `ReadKey()`, která vykonává načítání tajného klíče z konfiguračního souboru `key.conf`. Inicializační vektor je posílán unikátní v každém rámci audio dat v místě prvních dvou vzorků zvuku. Druhá strana si tyto první dva vzorky odebere a poskládá z nich opět inicializační vektor, který ve spojení s tajným klíčem tvoří semeno pro pseudonáhodnou posloupnost vytvořenou generátorem pseudonáhodné posloupnosti rand() pro šifrování a dešifrování audio dat pomocí operace XOR.

Výsledkem praktických experimentů ve formě dvou komunikujících stran prostřednictvím telefonního hovoru při použití zabezpečeného audiokodeku bylo dokázáno, že hovor lze efektivně šifrovat i na úrovni kodeků.

LITERATURA

[1] RUSSELL BRYANT, Leif Madsen. *Asterisk: the definitive guide.* 4th ed. Farnham: O'Reilly, 2013. ISBN 978-144-9332-426. Dostupné z: <http://www.jelia.us/learn/telephony/Asterisk_The_Definitive_Guide_4th_Edition.pdf>.

[2] ŠILHAVÝ, P. *Pobočkové telefonní ústředny s otevřeným kódem - Open Source.* [přednáška]. Brno: VUT, 2013. Dostupné z: <http://anca.utko.feec.vutbr.cz/vyuka_data/mtis/P2_MTIS.pdf>.

[3] ŠILHAVÝ, P. *Pobočkové telefonní ústředny PBX* [přednáška]. Brno: VUT, 2013. Dostupné z: <http://anca.utko.feec.vutbr.cz/vyuka_data/mtis/P3_MTIS.pdf>.

[4] Asterisk for Raspberry PI. *Asterisk for Raspberry PI* [online]. [cit. 2014-12-2]. Dostupné z: <http://www.raspberry-asterisk.org/faq/#performance>.

[5] DIGIUM, Inc. *Asterisk Project* [online]. 2012 [cit. 2014-12-01]. Dostupné z: <https://wiki.asterisk.org/wiki/display/AST/Home>.

[6] Voip Think - Codec - G.711 a law and µ or u law - PCM. *Voip Think-Voice over IP - Asterisk and SER - SIP IAX and H323* [online]. 2015 [cit. 2015-05-14]. Dostupné z: <Dostupnéz:http://www.en.voipforo.com/codec/codecs-g711-alaw.php>.

[7] How-to: Write an Asterisk Module: Part 1 – 3. *Russell Bryant / Open Source Software Engineering* [online]. 2015 [cit. 2015-04-22]. Dostupné z: <http://blog.russellbryant.net/2008/06/19/how-to-write-an-asterisk-module-part-1/>.

[8] Voice Encryption Units. *Crypto Museum* [online]. 2015 [cit. 2015-04-15]. Dostupné z: <http://www.cryptomuseum.com/crypto/voice.htm>.

[9] Create dialog box that gathers user input. *The MathWorks, Inc* [online]. 2015 [cit. 2015-05-13]. Dostupné z: <www.mathworks.com/help/matlab/ref/inputdlg.html>.

[10] BURDA, K. *Symetrické kryptosystémy* [přednáška]. Brno: VUT, 2014. Dostupné z: <https://www.vutbr.cz/elearning/mod/resource/view.php?id=288566>.

SEZNAM PŘÍLOH

A Zdrojový kód kodeku A-law s podporou šifrování pro Asterisk **38**

B Zdrojový kód kodeku u-law s podporou šifrování pro Asterisk **44**

C Obsah přiloženého CD **51**

A ZDROJOVÝ KÓD KODEKU A-LAW S PODPOROU ŠIFROVÁNÍ PRO ASTERISK

```
/*
 * Asterisk -- An open source telephony toolkit.
 *
 * Copyright (C) 1999 - 2005, Digium, Inc.
 *
 * Mark Spencer <markster@digium.com>
 *
 * See http://www.asterisk.org for more information about
 * the Asterisk project. Please do not directly contact
 * any of the maintainers of this project for assistance;
 * the project provides a web site, mailing lists and IRC
 * channels for your use.
 *
 * This program is free software, distributed under the terms of
 * the GNU General Public License Version 2. See the LICENSE file
 * at the top of the source tree.
 */

/*! \file
 *
 * \brief codec_alaw.c - translate between signed linear and alaw
 *
 * \ingroup codecs
 */

#include "asterisk.h"

ASTERISK_FILE_VERSION(__FILE__, "$Revision: 267492 $")

#include "asterisk/module.h"
#include "asterisk/config.h"
#include "asterisk/translate.h"
#include "asterisk/alaw.h"
#include "asterisk/utils.h"
```

```
#define BUFFER_SAMPLES   8096        /* size for the translation buffers */

/* Sample frame data */
#include "asterisk/slin.h"
#include "ex_alaw.h"
#include "asterisk/logger.h"

void SeedRandomNumGenerator()
{
    unsigned int seed;
    FILE *fd;

    fd = fopen("/dev/urandom", "r");
    if( fd )
    {
        fread(&seed, sizeof(seed), 1, fd);
        fclose(fd);
        srandom( seed );
    }
}

uint32_t ReadKey()
{
        uint32_t key;
        FILE *fd;

        fd = fopen("/etc/asterisk/key.conf", "r");
        if( fd ){
                fread(&key, sizeof(key), 1, fd);
                fclose(fd);
                return key;
        }
        ast_log(LOG_ERROR, "Nenalezen soubor s klíčem\n");
        return 0;
}

/*! \brief decode frame into lin and fill output buffer. */
static int alawtolin_framein(struct ast_trans_pvt *pvt, struct ast_frame *f)
{
```

```
        int i = f->samples;
        unsigned char *src = f->data.ptr;
        int16_t *dst = pvt->outbuf.i16 + pvt->samples;


        pvt->samples += i;
        pvt->datalen += i * 2 + 4;          /* 2 bytes/sample */


        uint32_t key = ReadKey();

        if (key == 0){
                ast_log(LOG_ERROR, "Nenalezen platný klíč\n");
                i -= 4;
                while (i--){
                        *dst++ = 0;
                }
                return 0;
        }
//          key = "Krno"; //zkouška nesouhlasného klíče
        SeedRandomNumGenerator();

        uint32_t random = rand();

        uint16_t dst1 = random;
        *dst++ = dst1;

        uint16_t dst2 = (random >> 16);
        *dst++ = dst2;

        uint32_t c_key = key ^ random;
        srand(c_key);

//          ast_log(LOG_NOTICE, "dst1 alaw to lin  %u \n", dst1);
//          ast_log(LOG_NOTICE, "dst2 alaw to lin  %u \n", dst2);
//          ast_log(LOG_NOTICE, "random alaw to lin  %u \n", random);
//          ast_log(LOG_NOTICE, "key alaw to lin   %u \n", key);
//          ast_log(LOG_NOTICE, "c_key alaw to lin  %u \n", c_key);
```

```
        int16_t tmp;
        while (i--){
                tmp = AST_ALAW(*src++);
                tmp = tmp ^ rand();
                *dst++ = tmp;
        }

        return 0;
}

/*! \brief convert and store input samples in output buffer */
static int lintoalaw_framein(struct ast_trans_pvt *pvt, struct ast_frame *f)
{
        int i = f->samples;
        char *dst = pvt->outbuf.c + pvt->samples;
        int16_t *src = f->data.ptr;

        pvt->samples += i;
        pvt->datalen += i;        /* 1 byte/sample */

        int32_t key = ReadKey();
        uint16_t src1 = *src++;
        uint16_t src2 = *src++;
        uint32_t random = src2;
        random = random << 16;
        random = src1 | random;
        uint32_t c_key = key ^ random;
        srand(c_key);

//        ast_log(LOG_NOTICE, "src1 lin to alaw  %u \n", src1);
//        ast_log(LOG_NOTICE, "src2 lin to alaw  %u \n", src2);
//        ast_log(LOG_NOTICE, "random lin to alaw  %u \n", random);
//        ast_log(LOG_NOTICE, "key lin to alaw  %u \n", key);
//        ast_log(LOG_NOTICE, "c_key lin to alaw  %u \n\n", c_key);

        int16_t tmp;
        while (i--) {

                tmp = *src++;
```

```
                tmp = tmp ^ rand();

                *dst++ = AST_LIN2A(tmp);

        }

        return 0;
}

static struct ast_translator alawtolin = {
        .name = "alawtolin",
        .srcfmt = AST_FORMAT_ALAW,
        .dstfmt = AST_FORMAT_SLINEAR,
        .framein = alawtolin_framein,
        .sample = alaw_sample,
        .buffer_samples = BUFFER_SAMPLES,
        .buf_size = BUFFER_SAMPLES * 2,
};

static struct ast_translator lintoalaw = {
        "lintoalaw",
        .srcfmt = AST_FORMAT_SLINEAR,
        .dstfmt = AST_FORMAT_ALAW,
        .framein = lintoalaw_framein,
        .sample = slin8_sample,
        .buffer_samples = BUFFER_SAMPLES,
        .buf_size = BUFFER_SAMPLES,
};

/*! \brief standard module stuff */

static int reload(void)
{
        return AST_MODULE_LOAD_SUCCESS;
}

static int unload_module(void)
{
```

```
        int res;

        res = ast_unregister_translator(&lintoalaw);
        res |= ast_unregister_translator(&alawtolin);

        return res;
}

static int load_module(void)
{
        int res;

        res = ast_register_translator(&alawtolin);
        if (!res)
                res = ast_register_translator(&lintoalaw);
        else
                ast_unregister_translator(&alawtolin);
        if (res)
                return AST_MODULE_LOAD_FAILURE;
        return AST_MODULE_LOAD_SUCCESS;
}

AST_MODULE_INFO(ASTERISK_GPL_KEY, AST_MODFLAG_DEFAULT, "A-law Coder/Decoder",
                .load = load_module,
                .unload = unload_module,
                .reload = reload,
               );
```

B ZDROJOVÝ KÓD KODEKU U-LAW S PODPOROU ŠIFROVÁNÍ PRO ASTERISK

```
/*
 * Asterisk -- An open source telephony toolkit.
 *
 * Copyright (C) 1999 - 2005, Digium, Inc.
 *
 * Mark Spencer <markster@digium.com>
 *
 * See http://www.asterisk.org for more information about
 * the Asterisk project. Please do not directly contact
 * any of the maintainers of this project for assistance;
 * the project provides a web site, mailing lists and IRC
 * channels for your use.
 *
 * This program is free software, distributed under the terms of
 * the GNU General Public License Version 2. See the LICENSE file
 * at the top of the source tree.
 */

/*! \file
 *
 * \brief codec_ulaw.c - translate between signed linear and ulaw
 *
 * \ingroup codecs
 */

#include "asterisk.h"

ASTERISK_FILE_VERSION(__FILE__, "$Revision: 267492 $")

#include "asterisk/module.h"
#include "asterisk/config.h"
#include "asterisk/translate.h"
#include "asterisk/ulaw.h"
#include "asterisk/utils.h"
```

```
#define BUFFER_SAMPLES   8096        /* size for the translation buffers */

/* Sample frame data */
#include "asterisk/slin.h"
#include "ex_ulaw.h"
#include "asterisk/logger.h"

void SeedRandomNumGenerator()
{
    unsigned int seed;
    FILE *fd;

    fd = fopen("/dev/urandom", "r");
    if( fd )
    {
        fread(&seed, sizeof(seed), 1, fd);
        fclose(fd);
        srandom( seed );
    }
}

uint32_t ReadKey()
{
        uint32_t key;
        FILE *fd;

        fd = fopen("/etc/asterisk/key.conf", "r");
        if( fd ){
                fread(&key, sizeof(key), 1, fd);
                fclose(fd);
                return key;
        }
        ast_log(LOG_ERROR, "Nenalezen soubor s klíčem\n");
        return 0;
}

/*! \brief convert and store samples in outbuf */
static int ulawtolin_framein(struct ast_trans_pvt *pvt, struct ast_frame *f)
{
```

```
        int i = f->samples;
        unsigned char *src = f->data.ptr;
        int16_t *dst = pvt->outbuf.i16 + pvt->samples;

        pvt->samples += i;
        pvt->datalen += i * 2 + 4;          /* 2 bytes/sample */


        int32_t key = ReadKey();
        if (key == 0){
                ast_log(LOG_ERROR, "Nenalezen platný klíč\n");
                i -= 4;
                while (i--){
                        *dst++ = 0;
                }
                return 0;
        }

        SeedRandomNumGenerator();

        uint32_t random = rand();

        uint16_t dst1 = random;
        *dst++ = dst1;

        uint16_t dst2 = (random >> 16);
        *dst++ = dst2;

        uint32_t c_key = key ^ random;
        srand(c_key);

//          ast_log(LOG_NOTICE, "dst1 ulaw to lin %u \n", dst1);
//          ast_log(LOG_NOTICE, "dst2 ulaw to lin %u \n", dst2);
//          ast_log(LOG_NOTICE, "random ulaw to lin %u \n", random);
//          ast_log(LOG_NOTICE, "key ulaw to lin %u \n", key);
//          ast_log(LOG_NOTICE, "c_key ulaw to lin %u \n", c_key);

        int16_t tmp;
        while (i--){
```

```
                tmp = AST_MULAW(*src++);
                tmp = tmp ^ rand();
                *dst++ = tmp;
        }

        return 0;
}

/*! \brief convert and store samples in outbuf */
static int lintoulaw_framein(struct ast_trans_pvt *pvt, struct ast_frame *f)
{
        int i = f->samples;
        char *dst = pvt->outbuf.c + pvt->samples;
        int16_t *src = f->data.ptr;

        pvt->samples += i;
        pvt->datalen += i;        /* 1 byte/sample */

        uint32_t key = ReadKey();
        uint16_t src1 = *src++;
        uint16_t src2 = *src++;
        uint32_t random = src2;
        random = random << 16;
        random = src1 | random;
        uint32_t c_key = key ^ random;
        srand(c_key);

//        ast_log(LOG_NOTICE, "src1 lin to ulaw %u \n", src1);
//        ast_log(LOG_NOTICE, "src2 lin to ulaw %u \n", src2);
//        ast_log(LOG_NOTICE, "random lin to ulaw %u \n", random);
//        ast_log(LOG_NOTICE, "key lin to ulaw %u \n", key);
//        ast_log(LOG_NOTICE, "c_key lin to ulaw %u \n\n", c_key);

        int16_t tmp;
        while (i--){

                tmp = *src++;

                tmp = tmp ^ rand();
```

```
                *dst++ = AST_LIN2MU(tmp);
        }

        return 0;
}

/*!
 * \brief The complete translator for ulawToLin.
 */

static struct ast_translator ulawtolin = {
        .name = "ulawtolin",
        .srcfmt = AST_FORMAT_ULAW,
        .dstfmt = AST_FORMAT_SLINEAR,
        .framein = ulawtolin_framein,
        .sample = ulaw_sample,
        .buffer_samples = BUFFER_SAMPLES,
        .buf_size = BUFFER_SAMPLES * 2,
};

static struct ast_translator testlawtolin = {
        .name = "testlawtolin",
        .srcfmt = AST_FORMAT_TESTLAW,
        .dstfmt = AST_FORMAT_SLINEAR,
        .framein = ulawtolin_framein,
        .sample = ulaw_sample,
        .buffer_samples = BUFFER_SAMPLES,
        .buf_size = BUFFER_SAMPLES * 2,
};

/*!
 * \brief The complete translator for LinToulaw.
 */

static struct ast_translator lintoulaw = {
        .name = "lintoulaw",
        .srcfmt = AST_FORMAT_SLINEAR,
```

```
        .dstfmt = AST_FORMAT_ULAW,
        .framein = lintoulaw_framein,
        .sample = slin8_sample,
        .buf_size = BUFFER_SAMPLES,
        .buffer_samples = BUFFER_SAMPLES,
};

static struct ast_translator lintotestlaw = {
        .name = "lintotestlaw",
        .srcfmt = AST_FORMAT_SLINEAR,
        .dstfmt = AST_FORMAT_TESTLAW,
        .framein = lintoulaw_framein,
        .sample = slin8_sample,
        .buf_size = BUFFER_SAMPLES,
        .buffer_samples = BUFFER_SAMPLES,
};

static int reload(void)
{
        return AST_MODULE_LOAD_SUCCESS;
}

static int unload_module(void)
{
        int res;

        res = ast_unregister_translator(&lintoulaw);
        res |= ast_unregister_translator(&ulawtolin);
        res |= ast_unregister_translator(&testlawtolin);
        res |= ast_unregister_translator(&lintotestlaw);

        return res;
}

static int load_module(void)
{
        int res;

        res = ast_register_translator(&ulawtolin);
```

```
        if (!res) {
                res = ast_register_translator(&lintoulaw);
                res |= ast_register_translator(&lintotestlaw);
                res |= ast_register_translator(&testlawtolin);
        } else
                ast_unregister_translator(&ulawtolin);
        if (res)
                return AST_MODULE_LOAD_FAILURE;
        return AST_MODULE_LOAD_SUCCESS;
}

AST_MODULE_INFO(ASTERISK_GPL_KEY, AST_MODFLAG_DEFAULT, "mu-Law Coder/Decoder",
                .load = load_module,
                .unload = unload_module,
                .reload = reload,
               );
```

C OBSAH PŘILOŽENÉHO CD

- `Diplomová práce.pdf`
- Model simulace kodeku v Simulinku
- Zvukový kodek A-law s podporou zabezpečení `codec_alaw.c`
- Zvukový kodek u-law s podporou zabezpečení `codec_ulaw.c`
- Konfigurační soubor `key.conf`