VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

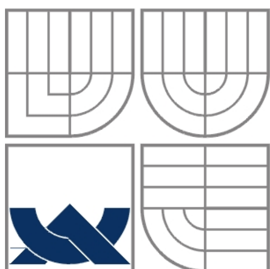
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

ANALÝZA A TRANSFORMACE KÓDŮ

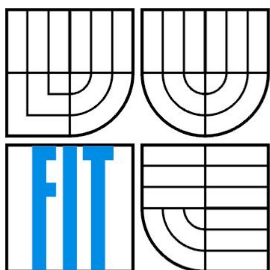
BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

AUTOR PRÁCE JAKUB KŘOUSTEK
AUTHOR

BRNO 2007



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

ANALÝZA A TRANSFORMACE KÓDŮ

CODE ANALYSIS AND TRANSFORMATION

BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

AUTOR PRÁCE JAKUB KŘOUSTEK
AUTHOR

VEDOUCÍ PRÁCE Prof. RNDr. ALEXANDER MEDUNA, CSc.
SUPERVISOR

BRNO 2007

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav informačních systémů Akademický rok 2006/2007

Zadání bakalářské práce

Řešitel: **Křoustek Jakub**
Obor: Informační technologie
Téma: **Analýza a transformace kódů**
Kategorie: Překladače

Pokyny:

1. Seznamte se s metodami analýzy a transformace kódů.
2. Navrhněte modifikace, které urychlí některé známé metody analýzy a transformace kódů.
3. Studujte užití metod navržených v předchozím bodě. Zaměřte se na využití v reverzním překladu z binární formy do jazyka symbolických instrukcí.
4. Zhodnoťte dosažené výsledky a diskutujte další možný vývoj projektu.

Literatura:

- Meduna, A.: Automata and Languages, Springer, London, 2000

Při obhajobě semestrální části projektu je požadováno:

- Body 1 a 2.

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním paměťovém médiu (disketa, CD-ROM), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Meduna Alexander, prof. RNDr., CSc.**, UIFS FIT VUT
Datum zadání: 1. listopadu 2006
Datum odevzdání: 15. května 2007

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav informačních systémů
612 66 Brno, Božetěchova 2

L.S.

doc. Ing. Jaroslav Zendulka, CSc.
vedoucí ústavu

LICENČNÍ SMLOUVA
POSKYTOVANÁ K VÝKONU PRÁVA UŽÍT ŠKOLNÍ DÍLO

uzavřená mezi smluvními stranami

1. Pan

Jméno a příjmení: **Jakub Křoustek**
Id studenta: 84274
Bytem: Hany Kvapilové 4396/37, 586 01 Jihlava
Narozen: 11. 07. 1984, Jihlava
(dále jen "autor")

a

2. Vysoké učení technické v Brně

Fakulta informačních technologií
se sídlem Božetěchova 2/1, 612 66 Brno, IČO 00216305
jejímž jménem jedná na základě písemného pověření děkanem fakulty:

....
(dále jen "nabyvatel")

Článek 1
Specifikace školního díla

1. Předmětem této smlouvy je vysokoškolská kvalifikační práce (VŠKP):
   bakalářská práce

Název VŠKP: Analýza a transformace kódů
Vedoucí/školitel VŠKP: Meduna Alexander, prof. RNDr., CSc.
Ústav: Ústav informačních systémů
Datum obhajoby VŠKP: ....

VŠKP odevzdal autor nabyvateli v:

tištěné formě počet exemplářů: 1
elektronické formě počet exemplářů: 2 (1 ve skladu dokumentů, 1 na CD)

2. Autor prohlašuje, že vytvořil samostatnou vlastní tvůrčí činností dílo shora popsané a specifikované. Autor dále prohlašuje, že při zpracovávání díla se sám nedostal do rozporu s autorským zákonem a předpisy souvisejícími a že je dílo dílem původním.
3. Dílo je chráněno jako dílo dle autorského zákona v platném znění.
4. Autor potvrzuje, že listinná a elektronická verze díla je identická.

Článek 2
Udělení licenčního oprávnění

1. Autor touto smlouvou poskytuje nabyvateli oprávnění (licenci) k výkonu práva uvedené dílo nevýdělečně užít, archivovat a zpřístupnit ke studijním, výukovým a výzkumným účelům včetně pořizování výpisů, opisů a rozmnoženin.
2. Licence je poskytována celosvětově, pro celou dobu trvání autorských a majetkových práv k dílu.
3. Autor souhlasí se zveřejněním díla v databázi přístupné v mezinárodní síti:
   ☒ ihned po uzavření této smlouvy
   ☐ 1 rok po uzavření této smlouvy
   ☐ 3 roky po uzavření této smlouvy
   ☐ 5 let po uzavření této smlouvy
   ☐ 10 let po uzavření této smlouvy
   (z důvodu utajení v něm obsažených informací)
4. Nevýdělečné zveřejňování díla nabyvatelem v souladu s ustanovením § 47b zákona č. 111/ 1998 Sb., v platném znění, nevyžaduje licenci a nabyvatel je k němu povinen a oprávněn ze zákona.

Článek 3
Závěrečná ustanovení

1. Smlouva je sepsána ve třech vyhotoveních s platností originálu, přičemž po jednom vyhotovení obdrží autor a nabyvatel, další vyhotovení je vloženo do VŠKP.
2. Vztahy mezi smluvními stranami vzniklé a neupravené touto smlouvou se řídí autorským zákonem, občanským zákoníkem, vysokoškolským zákonem, zákonem o archivnictví, v platném znění a popř. dalšími právními předpisy.
3. Licenční smlouva byla uzavřena na základě svobodné a pravé vůle smluvních stran, s plným porozuměním jejímu textu i důsledkům, nikoliv v tísni a za nápadně nevýhodných podmínek.
4. Licenční smlouva nabývá platnosti a účinnosti dnem jejího podpisu oběma smluvními stranami.

V Brně dne: ....

....	....
Nabyvatel	Autor

Abstrakt

Práce popisuje metody a postupy používané k analýze a transformaci kódů. Obsahuje základní informace o vědním oboru reverzní inženýrství a jeho užití ve výpočetní technice i mimo ni. Hlavním cílem je vytvoření prostředku ke zpětnému překladu z binární formy do jazyka symbolických instrukcí. Tato činnost je silně závislá na konkrétní instrukční sadě a musí být použita pro předem známou architekturu procesorů. Uvedený problém je řešen pomocí šablon, zásuvných modulů a modulárnosti zpětného překladače. Zmíněné vlastnosti dovolí uživatelům rozšiřovat program o nové instrukční sady. Výstupem je textová reprezentace instrukcí, funkčně ekvivalentní vstupu. Práce demonstruje nejenom běžně používané postupy dekódování, ale i nové postupy navržené autorem.

Klíčová slova

Reverzní inženýrství, zpětný překlad, překladač, dekompilátor, disassembler, debugger, assembler.

Abstract

This paper describes methods and procedures used for code analysis and transformation. It contains basic information of a science discipline called reverse engineering and its use in information technologies. The primary objective is a construction of tool that can disassemble from binary form to symbolic machine code. This operation is highly dependent on the concrete instruction set, and it has to be used for a beforehand known processor architecture. This problem is solved with patterns, plug-ins, and modularity of disassembler. These features provide users the ability to add new instruction sets into this disassembler. The output is the text representation of instructions and is functionally equivalent to the in-put. The thesis demonstrates usual methods of disassembly as well as the methods made by the author.

Keywords

Reverse engineering, recompilation, compilator, decompilator, disassembler, debugger, assembler.

Citace

Jakub Křoustek: Analýza a transformace kódů, bakalářská práce, Brno, FIT VUT v Brně, 2007

Analýza a transformace kódů

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Prof. Alexandra Meduny.
Další informace mi poskytli Ing. Roman Lukáš a Ing. Luboš Lorenc.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……………………
Jakub Křoustek
18. 4. 2007

Poděkování

Na tomto místě bych rád poděkoval mému vedoucímu Prof. Alexandru Medunovi za odborné vedení, za poskytnuté rady a konzultace a za vstřícnost při výběru tématu zadání. Dále bych chtěl poděkovat Ing. Luboši Lorencovi za rady z oblasti zpětného překladu a Ing. Romanu Lukášovi za výklad tématu párových gramatik.

© Jakub Křoustek, 2007.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

Obsah 1
Úvod 3
1 Reverzní inženýrství 4
1.1 Reverzní inženýrství mimo informační technologie 4
1.1.1 Genetika 4
1.1.2 Vojenství 5
1.1.3 Architektura 5
1.1.4 Další využití 5
1.2 Reverzní inženýrství v informačních technologiích 6
1.2.1 Kryptografie 7
1.2.2 Architektury integrovaných obvodů 8
1.2.3 Vývoj softwaru 8
1.2.4 Disassembler 10
1.2.5 Dekompilace 11
1.3 Právní hledisko 13
2 Zpětný překlad do jazyka symbolických instrukcí 15
2.1 Architektura procesoru 15
2.2 Instrukční sada 16
2.3 Formát binárních souborů 18
2.4 Transformační algoritmy 19
2.4.1 Lineární průchod 20
2.4.2 Rekurzivní zanořování 21
2.4.3 Hybridní algoritmy 24
2.4.4 Párové gramatiky a párové automaty 24
3 Implementace disassembleru 25
3.1 Návrh 25
3.2 Implementační prostředky 26
3.3 Grafické uživatelské rozhraní 26
3.4 Programová implementace 29
3.5 Zásuvné moduly 31
3.6 Ovládání 32
4 Závěr 34
Literatura 35
Seznam příloh 36

Příloha 1 – CHIP-8 a jeho instrukční sada....37
Příloha 2 – Obsah CD....39

Úvod

Pojem analýza a transformace kódů představuje sbírku metod a algoritmů, která nachází uplatnění v mnoha různorodých oblastech. Tento koncept můžeme vidět např. v chování lidského těla, ve snaze biologů o rozložení složitých proteinových řetězců nebo třeba v úsilí začínajícího programátora o porozumění jednoduchému programu. Vždy se jedná o pochopení dílčích částí a převod do uživateli srozumitelného celku.

Stále častější je však užití těchto procesů ve výpočetní technice. Díky nim můžeme zjišťovat principy programů, analyzovat zdrojové kódy či například zkoumat zhoubně se šířící viry. Obvyklé je též využití při hledání chyb v softwarových produktech, kdy můžeme objevit nejrůznější programátorské prohřešky počínajíce špatnou inicializací proměnných a konče chybou přetečení zásobníku. Způsobů užití je skutečně mnoho, stejně tak je mnoho způsobů naložení se získanými informacemi. I touto otázkou se práce zabývá.

Hlavním cílem tohoto díla je předvést jednotlivé metody analýzy a transformace kódů ve výpočetní technice, a to především v oblasti překladačů. Jako demonstrace praktického využití byl sestrojen program, který je na základě analýzy schopen převést binární zkompilovaný program zpět do jazyka symbolických instrukcí.

Bližší informace o obecných výrazech jsou zapsány v první kapitole. Zde jsou také představeny některé základní pojmy reverzního inženýrství v informačních technologiích, ale i mimo ně. Prezentovány budou pojmy jako zpětný překlad, disassembler či dekompilace. Nastíněno bude rovněž právní hledisko spojené s touto tématikou.

Druhá kapitola má za úkol vysvětlit základní pojmy spojené s tvorbou disassembleru. V ní budou objasněny spojitosti mezi instrukčními sadami a architekturami procesorů. Dále budou ukázány algoritmy používané při tvorbě tohoto nástroje. Postupně jsou představovány jednotlivé metody užívané pro analýzu a transformaci při převodu kódu z binární formy do jazyka symbolických instrukcí. Rozebrány zde jsou jejich klady i zápory a možnosti modifikací.

Kapitola třetí pojednává o vlastní implementaci zpětného překladače. Budou rozebrány jednotlivé etapy vývoje a úskalí, která během této doby nastala. Detailněji je pak popsán samotný návrh, implementace a seznam použitých nástrojů.

V závěrečné kapitole je diskutován budoucí rozvoj tohoto disassembleru, stejně tak jeho možné uplatnění na softwarovém trhu.

1 Reverzní inženýrství

Reverzní inženýrství (někdy též označované jako zpětné inženýrství, z anglického reverse engineering, či zkráceně RE) je proces, jehož cílem je pochopení zkoumaného objektu, jeho vlastností, vnitřních vztahů, architektury a designu. Výsledkem tohoto procesu jsou informace, pomocí kterých můžeme objekt znovu zkonstruovat. Může se jednat například o plán stavby, zdrojový kód programu nebo třeba nákres stroje.

1.1 Reverzní inženýrství mimo informační technologie

Přestože pojem reverzní inženýrství je většinou chápán jako čistě softwarová záležitost, můžeme nalézt mnoho oborů lidské činnosti, kde se používají velmi podobné principy. Uvedeny budou alespoň některé z nich.

1.1.1 Genetika

Genetika je biologická věda, zabývající se geny a dědičností. Tento vědní obor je poměrně mladý, vznikl až během 19. století. Nejdříve byla genetika definována jako *Věda o křížení a šlechtění rostlin* (více viz [1]), později se definice rozšířila na zkoumání dědičnosti všech organismů.

Základní jednotkou dědičnosti je gen, přičemž genem se nazývá úsek DNA (Deoxyribonukleová kyselina), který má schopnost vytvořit svojí identickou kopii, či přenést svoji informaci do další generace buňky. Dříve se uznávala teorie, že gen je úsek DNA, který kóduje protein (bílkovinu). Od tohoto názoru se však již upouští. Zkoumání proteinů se tak stalo vlastní vědní disciplínou (velice zajímavé experimentální projekty zkoumání proteinů, založené na distribuovaných výpočtech, jsou Folding@Home [2] a Rosetta@Home [3]). Sekvence DNA si můžeme představit jako úseky instrukcí programovacího jazyka. Podle kombinací těchto instrukcí dostává buňka různé vlastnosti a schopnosti.

Jako u ostatních vědních oborů i v genetice časem došlo k větvení do různých pod-oborů. Těmi nejznámějšími jsou: imunogenetika, populační genetika, klinická genetika a genetické inženýrství. Z hlediska reverzního inženýrství je pro nás nejzajímavější poslední z nich – genetické inženýrství. Zkoumání dědičnosti se zde neděje klasickou metodou vertikálního přenosu DNA *shora-dolů*, z generace na generaci, ale metodou horizontálního přenosu, v rámci jedné generace. Vědci jsou díky novým technologiím schopni přenášené úseky DNA modifikovat či vytvářet identické kopie. Příkladem může být třeba geneticky upravené obilí, které má díky modifikaci DNA lepší odolnost

vůči škůdcům, nebo větší odolnost proti teplotním výkyvům. Další informace o genetickém inženýrství jsou k nalezení na [4].

Jedná se tedy o analýzu vlastností (genů) organismu, jejich pochopení (zjištění kombinací DNA) a transformaci sekvencí DNA do nové buňky.

1.1.2 Vojenství

Vojenství je oborem, který velmi často reverzní inženýrství zneužívá. Národy se pomocí něho snaží dohnat technologickou vyspělost svého nepřítele. Zařízení, informace a zbraně jsou nejdříve získávány pomocí špionáže nebo vojáky na bitevním poli. Následně pak v laboratořích vědci získané předměty zkoumají a pokoušejí se je sami znovu vytvořit.

Tato technika byla používána nejvíce za druhé světové války, později během studené války, ale děje se tak i dnes. Příkladem může být okopírování německého kanistru na benzín Brity (tzv. *Jerrycan*) během druhé světové války nebo vytvoření ruského bombardéru Tupolev Tu-4 podle amerického vzoru B-29 během studené války.

Jako ochrana proti těmto praktikám se používá implementace fatálních chyb do návrhů nových technologií. Do výrobních plánů se tak zakreslují chybné části, které při sestrojení výrobku někým nepovolaným způsobí minimálně nefunkčnost.

1.1.3 Architektura

Další oblastí uplatnění jsou CAD (*Computer-aided design*), CAM (*Computer-aided manufacturing*) a CAE (*Computer-aided engineering*) systémy, které lze najít například v architektuře. Jedná se o převod vlastností fyzického objektu do trojrozměrného (3D) modelu reprezentovaného v počítači. Měřením fyzikálních vlastností objektů, jako jsou rozměry, hustota nebo váha, získáme data, která postačují k rekonstrukci do 3D modelu. Tento model je pak možné různě modifikovat a pomocí speciálních 3D tiskáren opět transformovat do fyzické podoby.

Pro měření objektů se používá například 3D skenování, tomografie nebo laserový skener. Naměřené hodnoty se pak musí přenést do počítače a zde vhodnou formou zobrazit. Nejpoužívanějšími metodami zobrazení jsou polygony, NURBS křivky či CAD modely.

Následujícím stupněm jsou trojrozměrné tiskárny, které dokáží počítačový model přetvořit na model hmatatelný. To se děje postupným nanášením velmi tenkých vrstev plastu. Využití nalezneme například v lékařství při vytváření protéz či implantátů nebo při designování vzhledu nových zařízení.

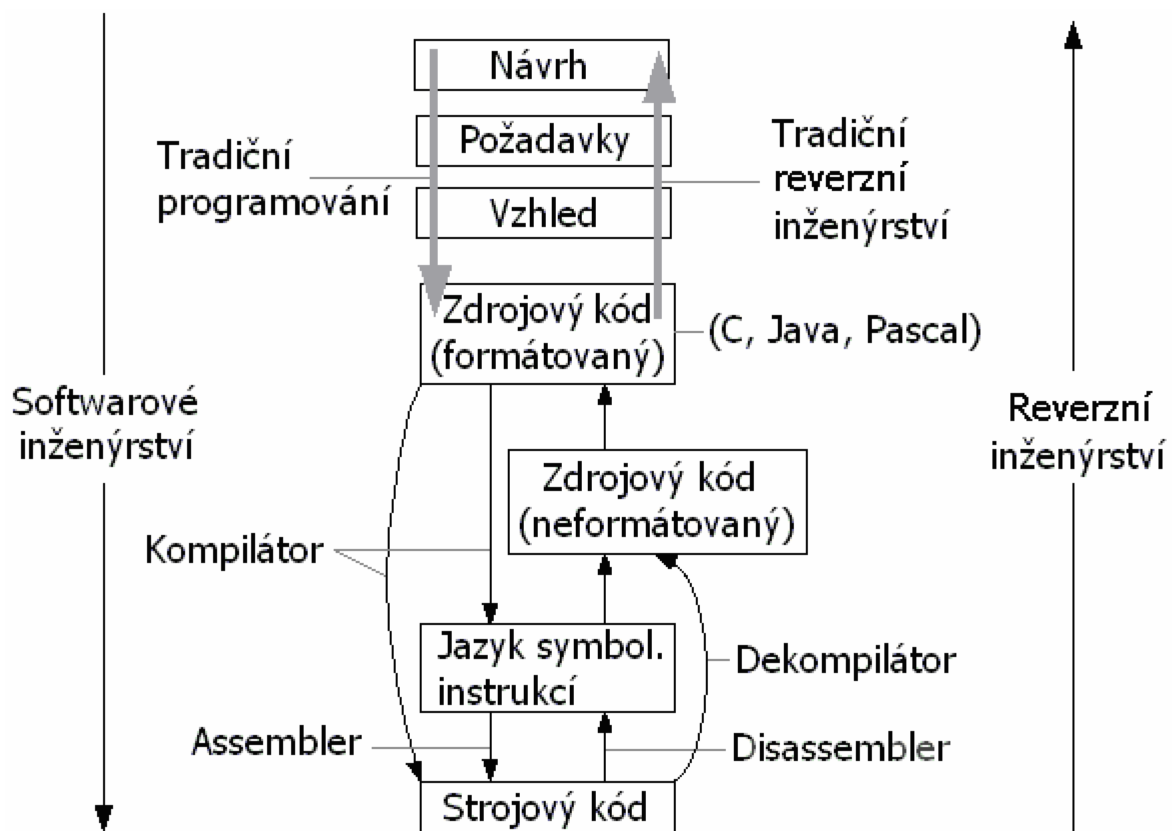
1.1.4 Další využití

Mezi další obory patří takové, kde je nutné testovat vlastnosti finálního výrobku. Své uplatnění nalézá reverzní inženýrství i v okamžiku kdy potřebujeme zjistit postup výroby díla, které jsme kdysi

vytvořili, ale již jsme ztratili výrobní postup. Totéž se může hodit i ve firmě, která ztratí klíčového pracovníka, jenž nezanechal podklady ke své práci.

1.2 Reverzní inženýrství v informačních technologiích

Snaha o zkoumání a poznávání neznámých věcí byla člověku vždy vlastní. Proto není divu, že se lidská zvídavost rozšířila i v moderních vědních oborech výpočetní techniky. Pojem reverzní inženýrství chápeme v informatice jako celou řadu disciplín, které vedou k úplnému pochopení vlastností a vazeb zkoumaného objektu a jejich přetvoření do jiné formy. Tento výklad si můžeme představit jako proces inverzní ke klasickému softwarovému inženýrství. V obou případech nacházíme totožné kroky (návrh, implementace, zdrojový kód, výsledný binární produkt, …), s tím rozdílem, že jsou chronologicky řazeny opačně. Více je patrno z obrázku 1.



1 Jednotlivé metody reverzního inženýrství

Některé z metod reverzního inženýrství jsou vysvětleny detailněji v následujícím textu, další metody jsou popsány v [5].

1.2.1 Kryptografie

Kryptografie je nauka o šifrách a utajování textů zpráv. Název pochází z řeckých slov *kryptós* a *gráfo* neboli *tajně psaný*. Kryptografie zahrnuje také pojmy kryptologie (zkoumání tajů) a kryptoanalýza (někdy též lámání hesel). První zmínky o kryptografii můžeme nalézt v Egyptě 4500 let před naším letopočtem. V průběhu let se kryptografie dále vyvíjela a vznikaly nové a nové metody šifrování.

Zpráva, která je pomocí některé kryptografické metody zašifrována, by měla být čitelná jen za znalosti principu či klíče potřebného k dešifrování. Jednotlivé metody šifrování můžeme rozdělit do několika kategorií:

- *Posuny písmen a aditivní šifry* – Posun písmen ve zprávě používala již velmi stará a jednoduchá metoda římského Caesara. Každý znak je posunut o zvolený počet pozic v abecedě. Na tuto metodu navazuje Vigenérova šifra. Každé písmeno hesla zde udává abecední posun následujícího znaku šifrované zprávy.
- *Substituční šifry* – Jsou založeny na nahrazení určitých znaků nebo skupin znaků znaky jinými. Klíč k dešifrování spočívá ve znalosti substituce – čím jsou které znaky nahrazeny.
- *Transpoziční šifry* – Původní zpráva je zapsána do tabulky o určitém počtu sloupců a řádků. Klíčem šifrování je záměna sloupců a (nebo) řádků v daném pořadí.
- *Šifrovací stroje* – Byly hojně používány začátkem 20. století a jejich vrcholem byl stroj Enigma během 2. světové války. Jednalo se o poměrně složité stroje, které transformovaly vstupní text na výstupní šifrovaný text podle určitého algoritmu.
- *Symetrické šifry* – Moderní pojetí šifrování za pomoci počítačů. Pro šifrování i dešifrování je použit jediný klíč. Jsou výpočetně nenáročné, ale vzniká problém s utajením klíče. Příkladem jsou např. šifry *DES*, *AES* a *Blowfish*.
- *Asymetrické šifry* – Často se tyto metody používají pro digitální podpis či utajení počítačové komunikace. Šifruje se pomocí dvojice klíčů, tajného a veřejného. Oproti symetrickým šifrám jsou výpočetně mnohonásobně pomalejší, ale dosahují vyšší úrovně zabezpečení. Příkladem jsou šifry *ElGamal* či *RSA*.
- *Další šifry* – K nalezení v publikaci [6].

Reverzní inženýrství zde nachází uplatnění především v kryptoanalýze. Kryptoanalytici, se zabývají převážně úrovní bezpečnosti jednotlivých šifer. Pomocí matematických metod se šifry snaží prolomit (anglicky se kryptografii někdy též říká *codebreaking*) a tím demonstrovat jejich zastaralost. Takto znehodnocené šifry by se měly co nejdříve nahradit novými silnějšími šiframi. To je důležité především v bankovnictví a ve vládním využití. Příkladem této činnosti může být nedávné (rok 2006) prolomení hashovací funkce *MD5* (*Message Digest 5*) skupinou kryptoanalytiků, v popředí s českým vědcem Vlastimilem Klímou [7].

1.2.2 Architektury integrovaných obvodů

I v hardwarovém odvětví výpočetní techniky jsou jasně patrné vlivy reverzního inženýrství. Výrobci různých integrovaných obvodů využívají výhod analýzy a transformace při tvorbě nových zařízení. Stává se, že jeden výrobce zkoumá vlastnosti cizího produktu a zjištěné výsledky použije při tvorbě produktu vlastního. Právní stránka této problematiky bude zkoumána později. Pro uživatele a programátory je pozitivní fakt, že výrobky jsou navzájem kompatibilní. Jelikož jsou postaveny na velmi podobné technologii, bude způsob zacházení s nimi obdobný.

Jako příklad nám může posloužit konkurenční boj mezi výrobci procesorů pro stolní počítače. Tento trh začala ovládat firma Intel. Ostatní menší firmy za ní pomalu začaly technologicky zaostávat. Proto se při tvorbě svých procesorů uchýlily k použití technologií podobných těm, které vyvinula společnost Intel. Především se jednalo o formát instrukční sady 32bitového procesoru Pentium. Vývoj těchto technologií trval firmě Intel poměrně dlouhou dobu a konkurence si díky jejímu použití tento cyklus značně zkrátila. Po letech se však situace obrátila a byla to právě firma Intel, která využila již existující instrukční sadu konkurenčního 64bitového procesoru Opteron firmy AMD (Advanced Micro Devices) k tvorbě vlastního 64bitového procesoru. Více o tomto příkladu je k nalezení v [8].

1.2.3 Vývoj softwaru

Reverzní inženýrství může být neuvěřitelně mocný nástroj pro softwarové vývojáře. Mohou pomocí něj například zjišťovat jak využívat nedokumentovaný, případně jen částečně dokumentovaný software ve spojení s jejich vlastním. Dalším příkladem může být zkoumání kvalit kódů třetích stran, jako jsou moduly, knihovny či celé operační systémy. Stejně jako výše u návrhu hardwaru, i zde existuje mezi vývojáři softwaru snaha získat technologie použité v cizích produktech. Také v samotném životním cyklu vytvořeného softwaru je několik etap, kdy se reverzní inženýrství uplatňuje.

Při odlaďování a testování programu se používá speciální nástroj zvaný *debugger*. Programy jsou příliš složité na to, aby člověk pouze pohledem do zdrojového kódu dokázal nalézt chybu. Debugger je program, který umožňuje programátorovi přímo vidět, co jeho aplikace vykonává. Debugger dokáže zobrazovat stav programu přímo za běhu pomocí hodnot a stavů proměnných, registrů procesoru, zásobníku a případně dalších. Děje se tak pomocí dvou základních metod – *krokování* a *breakpointu* (česky též někdy jako zarážka). Krokování dává programátorovi možnost pohybovat se programem instrukci za instrukcí a vidět přímo tok programu. Instrukce mohou být buďto přímo příkazy programovacího jazyka, ve kterém je program psán (např.: Pascal, C, …) nebo instrukce na nejnižší úrovni interpretačního prostředku (např.: *bytecode* v jazyce Java nebo assembler pro procesory u kompilovaných programů). Breakpoint je funkce, která při splnění určitých podmínek zastaví na zvoleném místě běh programu. Podmínkou může být například zpracovávání označené

instrukce či hodnota některé proměnné v určitém místě programu. Debugger může být přímo implementován v programovacím prostředí (např.: *Microsoft Visual Studio 2005*) nebo existovat jako separátní aplikace (*OllyDbg*, *Compuware DriverStudio*, případně starší *NuMega SoftICE*).

Mezi nejznámější debuggery patří:

- *GDB a DDD* – GNU debugger a jeho grafická nadstavba. GDB samotné pracuje pouze v příkazové řádce, DDD přidává vstřícnější uživatelské rozhraní. Otcem projektu je Richard Stallman. GDB pracuje s mnoha programovacími jazyky, např. C, C++, Ada, Fortran. Podporuje přes dvě desítky architektur procesorů. Jako první debugger podporoval vzdálené ladění (*remote debugging*). Na počátku roku 2007 se začalo pracovat na funkci zpětného ladění. Jedná se o možnost zpětného zobrazení již vykonaných instrukcí.
- *Microsoft Visual Studio Debugger* – Tento debugger je součástí každé verze vývojového prostředí *Visual Studio .NET*. Funkčně vychází z aplikace *CodeView*, což byl textový debugger v prostředí *Microsoft Visual C++*. Je stabilní, rychlý a jednoduchý na použití. Záporem je neschopnost ladit jaderné funkce operačního systému.
- *Turbo Debugger* – Vyvinut firmou Borland pro ladění aplikací v prostředí MS-DOS. První verze tohoto produktu je z roku 1989, kdy byl součástí programového balíčku společně s programem *Turbo Assembler*. Později se používal ve vývojových prostředích *Borland Turbo C* a *Borland C++*. V současnosti se tento debugger již příliš nevyužívá.
- *WinDbg* – Debugger od firmy Microsoft. Je určen pouze pro Microsoft Windows. Jedná se o víceúčelový nástroj, který je schopen mimo jiné i ladit ovladače či jaderné služby operačního systému. Využívá se také při zkoumání pádů operačního systému (tzv. *Blue Screen of Death*).
- *SoftICE* – Jde o velice mocný debugger, který pracuje na nejnižší vrstvě operačního systému. V roce 1987 vytvořila společnost Numera první verzi tohoto debuggeru, vyvíjen byl bezmála 20 let až do roku 2006, kdy byl jeho další vývoj pozastaven. Jako jeho následovníci vznikly debuggery *Syser* a *Rasta Ring 0 Debugger*. SoftICE pracuje pod operačním systémem Microsoft Windows (dříve také pod Microsoft MS-DOS). Tento debugger je určen především pro ladění ovladačů hardwarových zařízení, ale díky svým vlastnostem je používán i pro ladění ostatních programů či pro tzv. *software cracking*.
- *OllyDbg* – Velice zdařilý freeware debugger pro operační systém Microsoft Windows, jehož autorem je Oleh Yuschuk. Mnohdy se používá jako alternativa k debuggeru *SoftICE*. Díky obrovské uživatelské základně pro tento program existuje nepřeberné množství rozšíření a vylepšení. V průběhu roku 2007 má být dokončena druhá verze tohoto programu, která má přinést mnoho revolučních metod do tohoto oboru reverzního inženýrství.

- *Valgrind* – Valgrind je zaměřen především na odhalování chyb spojených se špatnou prací s pamětí. Je stvořen pro operační systém Linux na architektuře Intel x86. Prostřednictvím dynamické analýzy debugger zjišťuje nežádoucí jevy typu špatná alokace či uvolnění paměti, práce s nedefinovanou hodnotou proměnné či paměťové úniky. Nezřídka se používá i pro optimalizaci finálních verzí programů.

Při auditu vytvořeného programu se zjišťuje mimo jiné i úroveň bezpečnosti. Pomocí řady monitorovacích aplikací se zkoumá jeho komunikace s okolním světem. Programy, které nedávají příliš najevo své vnitřní stavy a chyby, jsou hůře prolomitelné. Pomocí programů typu *Ethereal* či *Wireshark* se dá pozorovat i způsob síťové komunikace a simulovat případné útoky při absenci šifrování přenosu.

1.2.4 Disassembler

Tímto tématem se bude zabývat převážně následující kapitola. Na tomto místě budou vysvětleny základní vlastnosti a charakteristiky disassembleru, nutné k pochopení dalšího textu.

Disassembler je počítačový program, který překládá strojový kód do jazyka symbolických instrukcí. Tento nástroj je používaný pro tzv. zpětný překlad a pracuje opačným způsobem než assembler. Výstupem bývá většinou lidsky čitelný kód, někdy též nazývaný mrtvý kód. Vyplývá to z jeho povahy, jelikož se po vytvoření již nijak nemění. Disassembler je také základní částí debuggeru (viz výše).

Při překladu do strojového kódu překladače téměř vždy odstraňují názvy proměnných, uživatelské komentáře a jména návěští. Je to dáno snahou optimalizovat výsledný kód a také faktem, že procesor nepotřebuje znát tyto údaje pro vykonávání programu. Disassemblerem vytvořený kód je tak častokrát velice strohý a lidsky těžko čitelný. Některé disassemblery však tento nedostatek eliminují vytvářením vlastních názvu proměnných, které si uživatel zapamatuje lépe než třeba číslo paměťové adresy.

Proces převodu ze strojového kódu do jazyka symbolických instrukcí je silně závislý na architektuře procesoru a na typu instrukční sady. Ty se od sebe dosti liší a je značně obtížné vytvořit disassembler, který by umožňoval práci s instrukčními sadami více procesorů.

Samotný převod není triviální. Při jeho provádění nastává řada problémů, které se řeší pomocí různých metod a algoritmů. Více bude vysvětleno v další kapitole.

Komerčních i volně šiřitelných disassemblerů je celá řada. Ty nejznámější zde budou uvedeny. Mnoho dalších lze nalézt na [9].

- *IDA Pro* – Patrně nejpoužívanější disassembler, jehož autorem je Ilfak Guilfanov. Obsahuje celou řadu instrukčních sad a funkcí (automatické komentování, vestavěný debugger, rozšiřitelnost pomocí zásuvných modulů, atd.). Velkým kladem je

multi-platformnost, tedy dostupnost programu na většině operačních systémů. Projekt je komerční.

- *BDASM* – Poměrně nová alternativa k projektu IDA Pro, kterou vytvořil Manuel Jiménez. Stějně jako IDA Pro je i BDASM komerčně prodávaný produkt. Jednou z jeho výhod je schopnost pracovat s instrukčními sadami některých herních konzolí.
- *BORG* – Freeware disassembler vytvořený Paulem Youngem (přezdívaný jako Kronos či Caesum). Disassembler je jednoduchý, ale velice rychlý a efektivní. Je zaměřen pouze na 32bitovou architekturu Intel x86.
- *Lida (Linux Interactive DisAssembler)* – Projekt určený pro operační systém Linux. Mimo obvyklé vlastnosti vyniká v kryptoanalýze některých chráněných souborů (typu ELF). Volně šiřitelný včetně zdrojových kódů.
- *XDASM* – Komerční disassembler schopný pracovat s instrukčními sadami mnoha 8 či 16bitových procesorů. Obsahuje rovněž rozhraní pro přidávání nových sad instrukcí.
- *Bastard* – Volně širitelný, rychlý a výkonný. Je určený pro platformy Windows, Linux a FreeBSD. Zatím pracuje pouze s architekturou Intel x86, v dalších verzích by měl ale být rozšířen o další architektury.
- *obj2asm* – Program napsaný Robinem Hilliardem a šířený pod licencí GNU. Pracuje s formátem OBJ, který je někdy používán pro šíření knihoven.
- *WDASM* – Legenda v reverzním inženýrství pro platformu Windows. Zavedla některé funkce, které se používají až dodnes. První verze tohoto programu byla již pro operační systém Windows 3.11. Jejím autorem je Eric Grass. Projekt byl komerční, ale posledních několik let se již nevyvíjí.

1.2.5 Dekompilace

Dekompilace je programová transformace, která má za cíl obnovit zdrojový kód, napsaný ve vyšším programovacím jazyku, ze zkompilovaného binárního souboru, který je většinou reprezentován strojovým či virtuálním kódem.

Neformálně se dá tato činnost přirovnat k získání stromu ze štosu papírů. Strom zde figuruje jako program sémanticky zapsaný pomocí vyššího programovacího jazyka. Drť, z níž se vyrábí papír, vykonává funkci prostředníka mezi stromem a papírem a reprezentuje assembler. Papír je finálním produktem a symbolizuje strojový kód. Postup výroby papíru ze stromů je všeobecně známý a praktikuje se již tisíciletí. Postup obrácený, tedy získání zdroje z produktu, však znám není a my nejsme schopni náš hypotetický strom zrekonstruovat. Stejně tak se často děje na poli výpočetní techniky ve snaze získat zpět zdrojové soubory z vytvořených programů. Výjimky však existují, a když se proces dekompilace podaří, pak je velikým přínosem.

Na tomto místě je dobré vysvětlit, proč se dekompilace vlastně využívá a proč je velmi žádaným nástrojem. Někdo by si mohl myslet, že pokud jsme schopni sestrojit disassembler pro určitou instrukční sadu a architekturu procesoru, tak již dekompilátor není potřeba. Opak je pravdou. Dekompilátor nám dává kód v jazyce, ve kterém byl původně napsán, tedy tak jak to zamýšlel jeho autor. Při použití disassembleru dostáváme pouze kód na úrovni jazyka symbolických instrukcí, který každou instrukci vyššího jazyka prezentuje jako několik (desítek) instrukcí symbolických. To celý program hodně zatemňuje a zvláště začínající programátoři s tím mají velké problémy. Dalším argumentem proti disassembleru je nutnost naučit se soubor symbolických instrukcí a jejich význam. Tento problém se dále umocňuje s přenositelností programu na více architektur, kdy každá z nich má vlastní instrukční sadu a programátor by se je muset naučit všechny.

Proces dekompilace je složen z několika stěžejních fází. Podrobnější přehled lze nalézt na [9].

- *Analýza formátu binárního souboru* – Především se jedná o zjištění identifikační hlavičky souboru či jen jeho názvu.
- *Převod ze strojového kódu do jazyka symbolických instrukcí* – Proces popsaný v minulé podkapitole.
- *Sémantická analýza symbolických instrukcí* – Pomocí pokročilých metod se zjišťují závislosti a vztahy mezi jednotlivými symbolickými instrukcemi.
- *Transformace do instrukcí vyššího jazyka* – Symbolické instrukce se shlukují do instrukcí vyššího jazyka. Při transformaci se využívají diagramy toku dat.
- *Kontrola celistvosti kódu* – Závěrečná fáze transformace, která slouží převážně jako kontrola vygenerovaného kódu.

Jak je z jednotlivých fází patrno, disassembler je jednou ze stěžejních částí dekompilátoru. Pokud uvážíme problémy spojené s tvorbou disassembleru, pak je tvorba stejně kvalitního dekompilátoru přibližně kvadraticky složitější. To je hlavní důvod proč je dekompilátorů tak málo. Pokud jsou již vytvořeny, tak pouze pro nejrozšířenější architektury a programovací jazyky. Tuto situaci trochu zmírňují nové postupy kompilací, které vytvářejí tzv. virtuální strojový kód (anglicky *bytecode*).

Dekompilátorů uvedených jako příklad zde nebude tolik jako disassemblerů, a to z výše popsaných důvodů.

- *JDEC* – Jeden z dekompilátorů pro jazyk Java. Java používá vnitřní virtuální kód. To se ukázalo být velmi výhodné z hlediska dekompilace. Překladače Javy jsou velmi „šetrné“ na původní zdrojový kód a dekompilátory tak nemají problém ani obnovit původní názvy proměnných.
- *Lutz Roeder's .NET Reflector* – Jak je již z názvu patrné, jedná se o dekompilátor pro prostředí Microsoft .NET. I zde je použít *bytecode*. Umožňuje snadné obnovení a dokonce modifikace původního kódu.

- *Boomerang* – Velmi ambiciózní projekt, určený pro architektury Intel x86 a Sparc-Solaris. Má za cíl dekompilaci binárních souborů, vytvořených jakýmkoliv procedurálním jazykem. Projekt započal v roce 2002 a již nyní má velice dobré výsledky především s jazykem C. Je vyvíjen s důrazem na vysokou modularitu a rozšiřitelnost. Je volně šiřitelný včetně zdrojových souborů.
- *P32Dasm* – Česko-slovenský freeware dekompilátor. Je určen pouze pro programy napsané v jazyku Visual Basic a zkompilované jako PCode, což je jakási náhrada nativního kódu.
- *DCC* – Dekompilátor jazyka C určený pro operační systém MS-DOS. Vytvořen Cristinou Cifuentes, jako její disertační práce na Queensland University of Technology v Austrálii během let 1991–1994. Dnes se již příliš nevyužívá.
- *Ferma T* – Průmyslově používaný dekompilátor určený pro migraci programů z IBM 370 Assembler na jazyky C a Cobol.
- *Desquirr* – Výsledek magisterské práce Davida Erikssona. Desquirr má spíše demonstrační charakter a je realizován jako zásuvný modul do disassembleru IDA Pro. Ukazuje vzájemnou provázanost mezi disassemblery a dekompilátrory.

1.3 Právní hledisko

Negativní příklad použití reverzního inženýrství je plagiátorství. To je možno nalézt v mnoha oborech lidské činnosti. Jedná se o snahu napodobit dílo někoho jiného a vydávat ho za svůj vlastní výtvor. Jméno pravého autora je zatajeno. Původní dílo je často, zvláště jedná-li se o komerční produkt, vydáváno bez postupu k jeho vytvoření (např. počítačový program bez zdrojových kódů, chemický přípravek, spotřební elektronika). Plagiátor tedy musí použít některé z technik reverzního inženýrství k tomu, aby získal znalosti potřebné k modifikaci a opětovnému vytvoření díla.

Dalším příkladem nekorektního chování je prolamování počítačových ochran (tzv. *cracking*) a počítačové pirátství obecně. Nejčastějšími druhy ochran jsou ochrany proti kopírování, ochrana sériovým číslem či hardwarový klíč. Cracking tyto ochrany odstraňuje, zneškodňuje či obchází. Tato aktivita je ve většině zemí také nelegální.

Plagiátorství a prolamování ochran se snaží zabránit autorský zákon a patentování. Autorský zákon má ve své legislativě ustanoveno mnoho států, přičemž některé státy ho definují přísněji než jiné (kupř. USA oproti Švédsku). Vždy se ale jedná o ochranu autorova díla před jeho odcizením a modifikováním. V české verzi tohoto zákona (zákon č. 121/2000 Sb.) je dílo definováno jako vnímatelné dílo, které je výsledkem jedinečné tvůrčí činnosti autora a také některé výjimky (počítačové programy nebo třeba fotografie). V zákoně jsou dále uvedeny případy, kdy se jedná o porušení autorských práv a kdy nikoliv.

Patent je prostředek pro zákonnou ochranu vynálezů. Vlastníkovi patentu zaručuje výhradní právo k průmyslovému využití vynálezu. Patenty udělují patentové úřady, které se nacházejí v mnoha státech. Oproti autorskému dílu se patentem nestávají autorské počiny automaticky, ale jen po schválení patentovou komisí. Ta za patent prohlásí pouze inovátorské vynálezy využitelné v průmyslu. Pro zajímavost, patentem se nemohou stát počítačové programy, ale technologie v programech použité ano. Po úspěšném zaregistrování se patent na určitou dobu, maximálně 20 let, stává chráněným.

2 Zpětný překlad do jazyka symbolických instrukcí

V této kapitola jsou vysvětleny stěžejní informace, nutné ke konstrukci disassembleru. Od čistě hardwarových témat až po algoritmické pojmy. Další užitečné údaje o tomto tématu jsou k nalezení v [5].

2.1 Architektura procesoru

Pochopení architektury procesoru nám poskytuje znalosti o jeho vnitřní stavbě, o jeho funkci a uplatnění. Díky těmto poznatkům dokážeme procesor využít k vykonání našeho programu. Z pohledu reverzního inženýrství zjišťujeme, jaké má procesor prostředky pro realizaci programu.

Procesor (anglicky *CPU – Central Processing Unit*) je integrovaný obvod, který funguje jako hlavní výpočetní prvek celého počítače. Jeho funkcí je načítání instrukcí z paměti, jejich dekódování a provádění. Každý typ procesoru používá svůj vlastní jazyk, který se nazývá strojový kód. Tento jazyk se skládá z elementárních instrukcí. Strojový kód je pro procesory mnohonásobně snazší na pochopení než jakýkoliv vyšší programovací jazyk. Pro převod mezi vyšším jazykem a strojovým kódem se používá překladač.

První verze procesorů vznikly počátkem 50. let minulého století, kdy byly realizovány jako elektromechanické součástky. Od té doby se rychlost a výkonnost procesorů neustále zvyšuje, podle Moorova zákona dvojnásobně každý rok. Stejně tak se zvyšuje využití procesorů a mikroprocesorů. Ty už se dnes nenacházejí pouze v počítačích, ale lze je nalézt v drtivé většině spotřební elektroniky.

Procesor se skládá z několika základních částí. Pomyslným mozkem procesoru je řadič. Ten se stará o načítání, dekódování a zpracování instrukcí. Dále procesor obsahuje sadu registrů, které slouží pro uchovávání operandů a průběžných výsledků operací. Aritmeticko-logická jednotka (*ALU*) je další částí. Její využití spočívá v provádění matematických výpočtů. Většina novějších procesorů obsahuje i takzvané koprocesory, které jsou požívány například pro operace v plovoucí řádové čárce (*FPU*). S rostoucí rychlostí procesorů roste i jejich vestavěná paměť, nazývaná *cache*.

Rychlost a výkonnost procesorů se měří především podle jejich pracovní frekvence délky cyklů. Pro porovnávání výkonností jednotlivých procesorů se používají sady nezávislých testů (např. MIPS). Více o měření výkonnosti lze nalézt v [10].

Základní členění architektur je podle rozdělení paměti:

- *Harvardská architektura* – Tento design je charakterizován oddělením adresového prostoru pro instrukce a data. Osobní počítače tuto architekturu již nepoužívají, mikroprocesory však ano. U vestavěných systémů to splňuje záměr, kdy program je za běhu konstantní a

mění se pouze data. To souvisí i s typem pamětí u těchto architektur (ROM pouze pro čtení a RAM pro čtení i zápis). Typickým představitelem je mikroprocesor 8051 firmy Intel.

- *Von Neumannovská architektura* – Program i data se zde nacházejí ve stejném adresovém prostoru. Používá se především u univerzálních procesorů, jako je PC. Instrukce i data jsou u tohoto přístupu proměnná a mohou být za běhu modifikována. Programátor (či operační systém) musí sám určovat, kam bude ukládat kód a kam data. Reprezentantem této skupiny jsou např. procesor i386 firmy Intel či mikroprocesor HC08 firmy Motorola.

2.2 Instrukční sada

Instrukční sada je seznam všech instrukcí a jejich variant, které dokáže procesor identifikovat a vykonat. Instrukční sady dvou procesorů se mohou někdy částečně či plně překrývat, potom se hovoří o míře kompatibility. Z důvodů optimalizace některé procesory mají instrukční sadu interní i externí (např. Pentium Pro vytvořený firmou Intel).

Instrukce jsou tvořeny sekvencemi bitů a často se zapisují jako hexadecimální čísla. Vždy se skládají z jednoznačného identifikátoru instrukce, kterému se říká *operační kód* (zkráceně *opkód*, či anglicky opcode) a většinou také z jednoho či více *operandů*.

Podle operačního kódu procesor zjišťuje, o jakou instrukci se jedná a jak dlouhá instrukce je (platí pro instrukční sady typu *CISC*, viz dále). Opkód také určuje, do jaké kategorie instrukce patří, kupř. aritmetické, logické, instrukce pro přesun, vstupně-výstupní, atd.

Operand funguje jako parametr instrukce. Operandem může být hodnota, registr, adresa paměti, případně konstanta. Ve většině instrukčních sad instrukce nepoužívají více jak čtyři operandy. Existují i instrukce, které nemají operand žádný.

Programování ve strojovém kódu je prakticky nemožné. Instrukcí a jejich variant jsou pro každý procesor stovky až tisíce a pamatovat si jejich číselné hodnoty není reálné. Proto se používá jazyk symbolických instrukcí, kde se každé číselné hodnotě přiřadí lidsky čitelné slovo či zkratka (v angličtině *mnemonic*). Tato slova se pak překládají pomocí assembleru zpět do strojového kódu. Zkratky jsou pro každou instrukční sadu jiné, ale ty nejobvyklejší zůstávají stejné, např. *add* (sčítání), *sub* (odčítání), *mov* (přesun) či *jump* (nepodmíněný skok).

S instrukčními sadami souvisí i způsob ukládání dat v paměti (tzv. *Little Endian* a *Big Endian*) a adresové režimy. Ve studijní opoře [11] jsou adresové módy přehledně vypsány. Těmi základními jsou:

- *Vlastní (anglicky inherent)* – Operandy jsou dány přímo operačním kódem instrukce. Procesor se tedy dozví umístění zdrojových dat a cílovou adresu přímo z opkódu.
- *Přímý operand (anglicky immediate)* – Operand instrukce obsahuje konstantu, se kterou bude provedena stávající instrukce. Vhodné například pro aritmetické operace.

- *Indexované (anglicky indexed)* – Jedním operandem je bázová adresa a druhým offset (index). Výsledná adresa je dána součtem báze a indexu. Používá se pro instrukce přesunu a skoku.
- *Relativní (anglicky relative)* – Operandem je adresa určitého návěští. Adresa je určovaná relativně, s ohledem na její umístění. Tento režim adresování se využívá především pro instrukce skoku či pro přenos celých datových bloků do jiných programových segmentů.

Během vývoje počítačových architektur bylo vytvořeno několik různých druhů instrukčních sad, z nichž každá byla vytvořena pro určitý cíl. Pro zvýšení výkonu a rozšíření kompatibility se v posledních letech tyto druhy instrukčních sad kombinují. O tomto obsáhlém tématu pojednává publikace [12].

- *CISC (Complex Instruction Set Computer)* – Tyto instrukční sady jsou známé rozsáhlým rejstříkem obsažených instrukcí. Jedná se o původní koncept pro návrh instrukčních sad. Svůj název toto pojetí získalo až po vytvoření instrukčních sad typu *RISC* (bude vysvětleno později). Instrukcí je zde mnoho a každá z nich má různou délku. Je to výhodné z programátorského hlediska, jelikož programátor má širokou nabídku instrukcí. Díky různorodosti se CISC také někdy nazývá jako ortogonální instrukční sada. Představiteli CISC jsou například procesory řady x86 firmy Intel. Nevýhodou je nižší rychlost vykonávání instrukcí z těchto sad a poměrně vysoká složitost tvorby překladačů, která je zapříčiněna vysokým počtem instrukcí. Výhodou je nízká cena těchto architektur.
- *RISC (Reduced Instruction Set Computer)* – Tento návrh vznikl počátkem 70. let minulého století ve snaze o optimalizaci architektury *CISC*. Pomocí statistických testů bylo zjištěno, že 80% výpočtů provádí pouze 20% ze všech instrukcí (některé zdroje uvádějí 90% a 10%). Tyto vytěžované instrukce je dobré zoptimalizovat na co nejvyšší úroveň. Dalšími úpravami u tohoto konceptu jsou např. zjednodušení instrukcí, snížení jejich počtu, jednotný čas na provedení každé instrukce, délka instrukcí je konstantní a začíná se používat zřetězení (anglicky *pipelining*). Příkladem této techniky je procesor PowerPC firmy IBM. Instrukční sady *RISC* jsou velmi rychlé, avšak poměrně finančně nákladné.
- *VLIW (Very Long Instruction Word)* – Jedná se o instrukční sady určené pro paralelní výpočty. Jejich vývoj započal v roce 1980. Masivně se používá techniky zřetězení (*pipelining*). Prvky této superskalární architektury můžeme spatřit například v procesoru Itanium firmy Intel.
- *MISC (Minimal Instruction Set Computer)* – Architektura s minimálním počtem instrukcí. Místo registrů tyto procesory používají zásobníky. To celou architekturu značně zjednodušuje a urychluje. Nevýhodou je malý seznam instrukcí. Tyto procesory jsou

používány ve spojení s virtuálním strojem při kompilaci programů v jazyce Java. *MISC* není příliš rozšířen.

2.3 Formát binárních souborů

Formát souborů představuje znalost, kterou je potřeba ovládat při snaze o pochopení obsahu souboru. Nezáleží na tom, zda se jedná o textový dokument či binární soubor, vždy je nutné mít vědomosti o struktuře obsahu tohoto souboru.

Data jsou na počítačových úložištích ukládána binárně, tj. pomocí nul a jedniček. My však potřebujeme mít uložené znalosti v uživatelsky vlídnějším formátu. Proto se data shlukují do větších částí, kterým říkáme soubory. Jedná se o vysokou abstrakci, která má za cíl poskytnout uživateli jednoduchý přístup k jeho datům.

Souborem může být například obrázek z dovolené, oblíbená písnička či program pro jejich přehrávání na PC. Vždy však tento soubor musí být uložen podle nějakého standardu, který poskytuje informace o typu obsažených dat. Tyto standardy mohou být vytvořené například tvůrcem operačního systému, organizací, která si formát patentovala nebo jakýmkoliv tvůrcem nového formátu.

Z hlediska zpětného inženýrství a především tvorby disassembleru je nejdůležitější formát binárních souborů. Jde o programy, které byly vytvořeny překladači ze zdrojových kódů do spustitelné formy. Je nutné si uvědomit, že formát těchto souborů není jednotný, ba právě naopak, liší se často, a dost podstatně. Rozdílný formát mají zkompilované soubory pro různé procesory na různých architekturách, např. program pro procesor i386 je úplně odlišný od programu vytvořeném na architektuře PowerPC. Různou strukturu mají i programy vytvořené na stejné architektuře, ale jiným operačním systémem. Příkladem je formát *PE (Portable Executable)* pro operační systém *Windows* a formát *ELF (Executable and Linking Format)* pro systém *Unix*. Dokonce různé verze stejného operačního systému mají různé formáty spustitelných souborů. Operační systém *Windows* definoval během svého životního cyklu několik struktur binárních souborů – *MZ (podle Marka Zbikowskiho, designera Microsoftu), NE (New Executable), LX (Linear eXecutable)*, *PE (Portable Executable)* a další.

Chceme-li zpětně přeložit nějaký soubor, potřebujeme jednoznačně určit, o jaký soubor jde, a pro jakou architekturu je vytvořen. Jedině tak budeme schopni soubor převést zpět do jazyka symbolických instrukcí. Pokud se při této analýze zmýlíme, může se stát, že soubor budeme rekonstruovat na instrukce špatného procesoru.

Metod pro identifikaci formátu souborů existuje několik. Různé operační systémy vytvořily různé způsoby jejich detekce. Mezi ty nejpoužívanější patří:

- *Jméno souboru* – Patrně nejprimitivnější způsob rozpoznávání formátů představuje analýza jména souboru. Často se zde používá pojem přípona souboru, což je koncová část jména, která je oddělena tečkou. S tímto přístupem se lze setkat u operačních systémů *Windows*,

MS-DOS či *Mac OS-X*. Metoda je velmi rychlá, není totiž potřeba zkoumat vlastní obsah souboru, ale často může vrátit nekorektní výsledek. Pro analýzu binárních programů není tato metoda vhodná, jelikož kvůli špatné identifikaci může povolit spuštění datového souboru a tím zapříčinit například pád systému. V současnosti se používá spíše jen společně s dalšími metodami.

- *Magické číslo (anglicky Magic Number)* – Způsob detekce formátů souborů v operačním systému *Unix*. Detekce je prováděna na základě několika prvních bajtů na začátku souboru. Tyto *magické bajty* mají často hodnotu zapsanou pomocí *ASCII kódování*, takže je lze přečíst v jakémkoliv textovém editoru. Tato metoda se někdy rozšiřuje na použití tzv. *hlavičky*. Jedná se o delší úsek bajtů, které nesou informace o různých parametrech souboru v jasně definované struktuře a pořadí. Hlavička se velice často užívá u spustitelných binárních souborů a tudíž je pro nás z hlediska tvorby disassembleru velice důležitá. O analýze hlaviček binárních souborů velice detailně pojednává článek [13]. Popsaná metoda má skvělé výsledky úspěšnosti při rozpoznávání formátů, avšak je dosti pomalá kvůli potřebě nahlížet přímo do obsahu souborů.
- *Soubory s metadaty* – Při přenosu souboru mezi různými platformami může dojít ke ztrátě některých informací o souboru (kupř. zkrácení názvu, ztráta času vytvoření nebo jména autora). Tomu se snaží zabránit postup používající externí soubor s metadaty. Princip je poměrně jednoduchý, vytvoří se další soubor, který bude obsahovat data popisující původní soubor. Toto by však vedlo k zacyklení – pro nově vytvořený soubor s metadaty by se musel vytvořit jiný soubor s meta^{2}daty, pro něj soubor s meta^{3}daty, atd. Kvůli tomu se oba soubory (původní a soubor s metadaty) uloží do jednoho společného souboru – archivu. Ten je pomocí svého jména identifikován jako archiv a následně zpracován. Archivem velice často bývá některý z komprimačních formátů typu *ZIP*, *RAR* či *TAR*. Jedná se tedy o kooperaci všech výše zmíněných metod. Zástupcem tohoto návrhu je formát *OTT* používaný programem OpenOffice.

2.4 Transformační algoritmy

Jak již bylo uvedeno v první kapitole, převod z binární formy do jazyka symbolických instrukcí není triviální. Úlohu komplikuje několik faktorů, mezi něž patří především výskyt nepřímých skoků a datových složek v kódovém segmentu (kupř. zarovnání, tabulky skoků, atd.). Disassembler se rovněž musí vypořádat s rozdílnými formami binárního kódu, tak jak je generují různé překladače. Ručně napsaný program v assembleru bude mít jinou strukturu než program kompilovaný překladačem vyššího programovacího jazyka. Dalším problémem je odlišnost počítačových architektur. Především instrukční sady s nejednotnou délkou instrukce (*CISC*) zvyšují náročnost analýzy a transformace kódů. Celá tématika zpětného překladu je včetně příkladu rozebrána v publikaci [14].

Pro převod do jazyka symbolických instrukcí bylo vytvořeno již několik algoritmů. Každý z nich má své klady i zápory. Některé algoritmy si s výše uvedenými problémy dokáží poradit lépe než jiné. V následujících podkapitolách budou uvedeny jejich vlastnosti a navrženy jejich modifikace.

2.4.1 Lineární průchod

Tento algoritmus prochází přímočaře vstupní soubor bajt po bajtu a podle databáze instrukcí vyhodnocuje každý prvek. Jedná se o nejznámější metodu, kterou používá například GNU program *objdump*. Algoritmus je značně rychlý, ale také velmi nepřesný. Chybuje například při snaze o dekódování bajtů využitých jako výplň pro zarovnání. V tomto případě nemusí disassembler zachytit začátek instrukce a bude se snažit dekódovat nesmyslnou posloupnost binárních dat.

Více je patrno z příkladu na obrázku 2.

Adresa	Strojový kód	Disassembler	Správný význam
0x100	0x11	mov r1,	mov r1,
0x101	0x03	0x03	0x03
0x102	0x22	inc r2	inc r2
0x103	0x33	jmp	jmp
0x104	0x01	[0x106]	[0x106]
0x105	0x00	sub r0,	**zarovnání**
0x106	0x55	0x55	add r5,
0x107	0x33	jmp	0x33
0x108	0x77	[0x177]	dec r7
…	…	…	…

operační kód operand

2 Ukázka problémů při využití lineárního algoritmu

Disassembler v tomto příkladě zpracovává jednotlivé bajty a podle operačních kódů určuje typ instrukce a případně typ a počet operandů. Jedná se o hypotetickou architekturu s instrukční sadou typu *CISC*. Algoritmus na počáteční adrese *100h* zpracuje první operační kód a podle tabulky instrukcí zjistí, že se jedná o instrukci přesunu do registru *r1*. Nový obsah tohoto registru je definován operandem na dalším bajtu. Algoritmus takto sekvenčně zpracovává instrukce až do adresy *103h-104h*, kde se nachází nepodmíněný skok na adresu *106h*. Na následující adresu (*105h*) umístil překladač jeden bajt pro zarovnání. Důvodem použití tohoto datového bajtu v kódovém segmentu může být například optimalizace přístupu do paměti. Program se na adresu *105h* díky přemostění zmíněným skokem nikdy nedostane a procesoru je jedno, jaká hodnota se na této adrese nalézá. To však netuší lineární algoritmus, který po dekódování tohoto skoku pokračuje automaticky na další adrese – *105h*. Zde se pokusí v tabulce instrukcí nalézt operační kód odpovídající hodnotě *00h*. To se mu pravděpodobně podaří, jelikož jsou instrukční sady procesorů velice rozsáhlé a téměř každá

číselná kombinace je v nich obsažena. Mylně se tak bude domnívat, že se na tomto místě nachází operace odčítání, která je definována dvěma bajty. Tím dojde k nesprávnému rozpoznání instrukce na adrese *106h*, kde má být správně operační kód, ale disassembler zde bude hledat operand. To povede k dominovému efektu, kdy budou chybně dekódovány instrukce i na dalších adresách. Díky další možné chybě dále v programu může dojít k opětovnému nalezení správného „vlákna", ale to nemusí být pravděpodobné. Při dalším zpracování vytvořeného kódu bychom si mohli všimnout dalšího problému. Již několikrát zmíněný skok na adrese *103h* má za cíl svého skoku adresu *106h*, jenže tato adresa se nachází uprostřed jiné instrukce, což je pochopitelně špatně.

Jak tedy z příkladu vyplývá, lineární algoritmus není vhodný u programů, kde se mísí datové a kódové sekvence. Tomu se však děje u naprosté většiny moderních překladačů. Metoda má šanci se osvědčit pouze u velmi krátkých a jednoduchých programů, kde může těžit ze své rychlosti. Naopak při dekódování delších strukturovaných programů je algoritmus téměř nepoužitelný a jeho úspěšnost správně rozpoznaných instrukcí klesá pod 50% hranici.

Algoritmus může být modifikován n-násobným průchodem binárního kódu. Při každém dalším průchodu se bude výsledný kód zpřesňovat. Avšak ani tak nebudou odstraněny všechny chyby způsobené nedokonalostí algoritmu. Navíc algoritmus přijde o svoji největší výhodu, kterou je rychlost. Ta se bude snižovat přibližně dvakrát při každém dalším průchodu.

2.4.2 Rekurzivní zanořování

Algoritmus založený na rekurzivním zanořování vznikl se záměrem eliminovat nedostatky lineárního přístupu. Hlavním cílem je odstranit dekódování datových úseku uvnitř kódových segmentů. Při zpětném překladu se instrukce dělí do tří skupin:

- *Instrukce nepodmíněného skoku a návratu z volání* – Při těchto instrukcích dochází ke změně lokace právě vykonávaného kódu. Děje se tomu tak na základě změny obsahu registru instrukčního ukazatele (*Instruction Pointer Register*).
- *Instrukce podmíněného skoku a volání funkcí* – Při podmíněném skoku a volání dochází k větvení programu. Děje se tomu podle dynamických podmínek, které jsou známi až za běhu programu. Jsou stěžejní částí algoritmu.
- *Ostatní instrukce* – Jedná se o všechny ostatní instrukce, které nemění lokaci právě vykonávaného kódu.

Vstupní soubor se začíná procházet lineárně od počátečního bodu programu (*Program Entry Point*), do té doby, než se objeví instrukce typu skok, volání či návrat z funkce. Pokud instrukce způsobuje pouze změnu adresy vykonávání programu, pak se přestane transformovat aktuální oblast a disassembler začne zkoumat cílovou adresu. Jestliže instrukce vykonává funkci větvení, tak se disassembler bude pokoušet zpracovat vzdálenou oblast a až poté pokračovat v dekódování oblasti stávající. Nastane tak situace, kdy disassembler zkoumá pouze programově dostupné adresy a nemůže

se tak dostat například do datového segmentu. Všechny nezpracované adresy jsou pak speciálně označeny jako nedostupné. Pro demonstraci poslouží obrázek číslo 3.

Adresa	Strojový kód	Disassembler	Správný význam
0x100	0x11	mov r1,	mov r1,
0x101	0x03	0x03	0x03
0x102	0x22	inc r2	inc r2
0x103	0x33	jmp	jmp
0x104	0x01	[0x106]	[0x106]
0x105	0x00	**nedosažitelné**	**zarovnání**
0x106	0x55	add r5,	add r5,
0x107	0x33	0x33	0x33
0x108	0x77	dec r7	dec r7
...	...	...	...

operační kód	operand

3 Ukázka výhod rekurzivního algoritmu

Jedná se o program z předcházející kapitoly. Stěžejní roli zde opět hraje instrukce skoku na adrese *103h*. Rekurzivní algoritmus bude postupovat od adresy 100h klasických způsobem, dokud nenarazí na instrukci skoku na adrese *103h*. Program zjistí, že dochází ke změně aktuální adresy na hodnotu *106h* a začne dekódovat až na této adrese. Stejným způsobem bude pokračovat do konce souboru. Na závěr tohoto postup jsou neprojité adresy označeny jako *nedosažitelné*.

Jak je z příkladu patrno, algoritmus řeší problém lineárního algoritmu, kdy velmi často docházelo ke snaze o zpětný překlad datových složek binárních souborů, což mělo za následek znehodnocení výstupního kódu. Rekurzivní zanořování tento problém pomocí sledování větvení programu úplně odstranilo. Avšak vyvstal jiný problém. Při použití instrukcí nepřímého skoku dochází k situaci, kdy nejsou dekódovány celé funkční bloky. Jako ukázka této situace poslouží obrázek číslo 4.

Adresa	Strojový kód	Disassembler	Správný význam
0x200	0x44	call	call
0x201	0x01	[r1]	[r1]
0x202	0x88	return	return
0x203	0x55	**nedosažitelné**	add r5,
…	…	…	…
0x205	0x88	**nedosažitelné**	return
0x206	0x65	**nedosažitelné**	sub r5,
…	…	…	…
0x208	0x88	**nedosažitelné**	return
…	…	…	…

operační kód operand

4 Problémy spojené s rekurzivním algoritmem

Binární soubor je zkompilován pro stejnou instrukční sadu jako v předchozích případech. Proces zpětného překladu začíná na adrese *200h*, kde se nachází instrukce nepřímého volání. Cíl volání je určen obsahem fiktivního registru *R1*. Jeho hodnotu není možné statickou analýzou zjistit. Procesor tuto hodnotu zjišťuje za běhu programu a může se tak dozvědět, že tok programu bude pokračovat například na adresách *203h* nebo *206h*, kde se nacházejí nějaké funkce ukončené instrukcí *ret* (návrat, z anglického *return*). Na tyto funkce není nikde jinde v programu již odkaz, a tudíž je algoritmus vyhodnotí jako nedosažitelné. Dochází tak tedy k nedokonalé transformaci programu.

U instrukce nepřímého skoku (některé architektury dovolují i nepřímá volání) je adresa cílového návěští získávána z dynamických dat, kterými jsou například obsahy registrů či hodnota uložená na vrcholu zásobníku. Tyto hodnoty není možné v průběhu zpětného překladu získat, jelikož se jedná o statickou analýzu. Proto je výsledný kód jazyka symbolických instrukcí často korektní, ale neúplný. Nejhůře končí pokusy o překlad programových knihoven a funkčních modulů.

Celkově je metoda vyspělejší než její lineární předchůdce, odstraňuje její hlavní chyby, ale přináší některé nové problémy. Z hlediska implementace je výrazně složitější. Experimentálně získaná data o procentuální úspěšnosti dekódování se pohybují lehce nad 80% hladinou. Úspěšnost není závislá na délce vstupního souboru.

Prostor pro modifikování tohoto algoritmu je poměrně velký. Nabízí se možnost spojení rekurzivního a lineárního algoritmu, ta bude zmíněna v další kapitole. Zástupcem dalších metod je postup nazvaný *krájení kódu* [15]. Jde o poměrně složitou analýzu, která zkoumá nedosažitelné adresy a pomocí předdefinovaných vzorů hledá funkční bloky. Ke své práci potřebuje kompletní graf toku, jehož konstrukce sama o sobě je dosti složitá. Jiná alternativa je spojená s vyhledáváním tzv. relokačních údajů, které jsou ukládány překladači do hlaviček binárních souborů. Relokační údaje

obsahují informace mimo jiné o počtu a umístění funkcí v binárním souboru. Bohužel tyto informace ukládají jen některé překladače a obecný disassembler se na tuto modifikaci tedy nemůže spoléhat.

2.4.3 Hybridní algoritmy

Programovou evolucí došlo ke spojení lineárního a rekurzivního přístupu. Tím se z velké části stírají problémy, které jednotlivé metody mají. Disassembler používající tuto metodu dokáže rozpoznat datové složky binárního souboru a najít funkce, které jsou rekurzivním algoritmem nedosažitelné. Existují dva přístupy k této metodě. První z nich používá rekurzivní algoritmus pro průchod programem a lineární pro následující dohledávání nenalezených funkcí. V této metodě jsou patrné známky *programového krájení*, které jsou zmíněny v minulé podkapitole. Druhým způsobem spolupráce je režim kontroly. Program je separátně zpracován lineárním i rekurzivním algoritmem. Dekódované instrukce na každé adrese jsou vzájemně porovnávány. Při shodě je instrukce vyhodnocena jako korektní, při neshodě je adresa znovu zkoumána nebo pouze označena za chybnou.

Hlavní nevýhodou algoritmu je jeho značná pomalost, ta je dána principem algoritmu, kdy se transformovaný soubor prochází dvakrát. Tento fakt ale zastiňuje vysoká úspěšnost při dekódování. Spojená síla obou přístupů dosahuje úspěšnosti přesahující 90%.

2.4.4 Párové gramatiky a párové automaty

Novinkou v oblasti transformace a analýzy kódů jsou překladové automaty, které používají atributované párové automaty (které jsou založeny na tzv. líných automatech) a párovou gramatiku. Program zapsaný ve strojovém kódu je zde chápán jako věta gramaticky popsaného jazyka. Tato technika má umožnit překlad z jazyka symbolických instrukcí do strojového kódu, ale i překlad opačný. Více o této myšlence lze nalézt v publikaci [16].

Metoda je použita v projektu Lissom [17], který je zaměřen na výzkum popisu různých mikroprocesorových architektur. Projekt má za cíl vytvořit pro zvolený vestavěný systém kompletní softwarovou sadu (překladač, assembler, linker, disassembler a simulátor).

V této bakalářské práci nebyla metoda prozatím použita, ale jeví se jako velmi nadějná náhrada předcházejících metod.

3 Implementace disassembleru

Cílem této práce je demonstrovat výše zmíněné metody a algoritmy v nástroji, který bude schopen převádět binární soubory obsahující strojový kód do jazyka symbolických instrukcí. Tento program ponese jméno *GDA*, což představuje zkratku odvozenou z počátečních písmen slov *Generický DisAssembler* (anglicky též jako *Generic DisAssembler*). Generičnost v názvu prozrazuje jeho hlavní vlastnost, kterou je zpětný překlad uživatelem definovaných sad. Program tedy není konečným produktem, nýbrž prostředkem, který každému uživateli dovolí rozšíření o další instrukční sady.

V následujících podkapitolách se nacházejí informace o funkci a konstrukci disassembleru. Pro detailnější pochopení programu je dostupná programová dokumentace, vytvořená pomocí nástroje *Doxygen*, která se nachází na přiloženém kompaktním disku společně se zdrojovými kódy programu (příloha č. 2).

3.1 Návrh

Disassembler by měl mít jedinečné vlastnosti, které ho odliší od ostatních existujících konkurentů (viz kapitola 1.2.4). Důraz kladen především na tyto charakteristiky:

- *Open-source* – Program je šířen zkompilovaný i ve formě zdrojových kódů a je zdarma při dodržení licenčních ujednání.
- *Multi-platformní* – Existuje ve verzích pro všechny nejrozšířenější architektury a operační systémy.
- *Generičnost a modularita* – Disassembler se nezaměřuje pouze na jednu konkrétní architekturu a instrukční sadu. Poskytuje možnosti pro rozšíření o stávající či zatím pouze hypotetické instrukční sady.
- *Pohodlné uživatelské rozhraní* – Grafická nadstavba poskytuje komfortní a přehledný pohled na celou práci.
- *Použitelnost ve výuce* – Při zpětném překladu lze získat detailní informace o cílové architektuře, stejně tak o každé jednotlivé instrukci.
- *Vícejazyčné prostředí* – Pro použití v mezinárodním měřítku program podporuje anglický a český jazyk. Velice snadno je možné přidávat další lokalizační prostředky.

Některé z těchto vlastností poskytují i jiné disassemblery, ale téměř vždy jsou limitovány jinými nedostatky. *GDA* je koncipován tak, aby spojil nejlepší vlastnosti z ostatních řešení a umocnil je svými novými vlastnostmi.

3.2 Implementační prostředky

Pro implementaci byl zvolen programovací jazyk C/C++. Jazyk C je imperativní (procedurální) systémový programovací jazyk, který byl vytvořen v roce 1972. Jeho autorem je Dennis Ritchie, jenž v té době pracoval pro Bellovy Telefonní Laboratoře. C dovoluje programovat na nízkých vrstvách architektur (přístup do paměti, ovládání periferních zařízení, …), stejně tak jako na vrstvách vyšších. Výsledek překladu je velice blízký kódu napsanému v assembleru, z čehož vyplývá jeho čistá a rychlá činnost. C bylo určeno z počátku pouze pro operační systém UNIX. Velice brzy se však jazyk proslavil a jeho rozšíření na ostatní platformy bylo nasnadě. O tomto jazyku velice srozumitelně pojednává publikace [18].

Jazyk C++ vznikl v roce 1983 rovněž v Bellových Telefonních Laboratořích. Autorem je Bjarne Stroustrup, který C++ koncipoval jako rozšíření jazyka C. Přidány byly především třídy, virtuální funkce, přetěžování operátorů, šablony a zpracování výjimek. C++ povoluje několik programátorských přístupů jako je procedurální programování, objektově orientované programování či generické programování. Oba jazyky jsou rozšířené do té míry, že je podporuje i většina mikroprocesorů a vestavěných systémů.

Volba implementačního jazyka byla dána především požadavkem na přenositelnost a rychlost vykonávání programu. Alternativou k jazykům C/C++ byl programovací jazyk Java, ten je však kvůli své implementaci překladu programů často pomalejší a náročnější na systémové zdroje.

Jako nadstavbu pro uživatelské prostředí bylo nutné zvolit některý z programových balíčků různých firem. Tyto balíčky se častěji nazývají jako *toolkity*. Toolkit by měl být rovněž použitelný na co největším množství platforem, měl by poskytovat snadné a přehledné ovládací prvky, měl by být šiřitelný pod některou GNU licencí a měl by mít co nejlepší programovou dokumentaci. Z široké nabídky různých produktů byl zvolen toolkit Qt od firmy Trollech [19]. Qt je dostupné pro většinu operačních systémů. Tento toolkit má dlouhou historii, během níž byla kvalitně zpracována programová dokumentace, což velmi pomohlo při programování uživatelského rozhraní. Qt je pro nekomerční využití zdarma (licence GNU), což plně postačuje záměrům projektu *GDA*. Jako rozhodující klad při výběru se ukázala sbírka funkcí ve standardní programové knihovně tohoto produktu. Jsou zde obsaženy modifikace abstraktních datových typů obsažených v knihovnách jazyka C++, stejně tak obsahuje i nové datové typy. Velice kvalitně je zpracována i podpora vláken a optimalizací programu. Dalšími kandidáty byly toolkity FLTK a wxWidgets, které však výše zmíněné vlastnosti neměly, nebo byly zpracovány o stupeň horší úrovní.

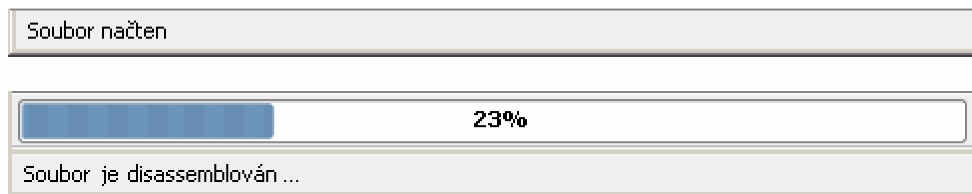
3.3 Grafické uživatelské rozhraní

Program je navržen jako aplikace s grafickým uživatelským rozhraním. Základem je hlavní okno s ovládacími prvky. Prvním z nich je tlačítková lišta s jednotlivými akcemi. Každá akce je

vyobrazena pomocí obrázkové ikony, která ji charakterizuje. Jelikož jsou tlačítka reprezentována graficky a ne textově, je pro usnadnění zobrazována textová nápověda. Lišta s tlačítky je vyobrazena na obrázku 5.

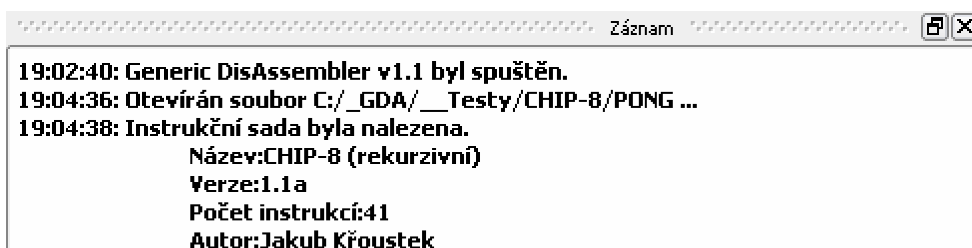
5 Lišta s tlačítky pro jednotlivé akce.

Při provádění příslušné akce je uživatel informován o jejím průběhu pomocí stavové lišty. Ta vyobrazuje informace, jako jsou načítání souboru, identifikování instrukce nebo uzavírání souboru. Při dlouhotrvajících akcích typu vykonávání transformace kódu se stavová lišta rozšíří o ukazatel pokroku (anglicky progressbar), který procentuelně vyjadřuje průběh operace. Více je patrno z obrázku 6.



6 Stavová lišta. Nahoře jednoduchá varianta, dole varianta s rozšířením o ukazatel pokroku.

Každá významnější událost je zapisována do okna se záznamem (logem) společně s časem jejího dokončení. To je výhodné například při testování a ladění instrukčních sad či nových funkcí disassembleru. Záznamové okno je zachyceno na obrázku číslo 7.



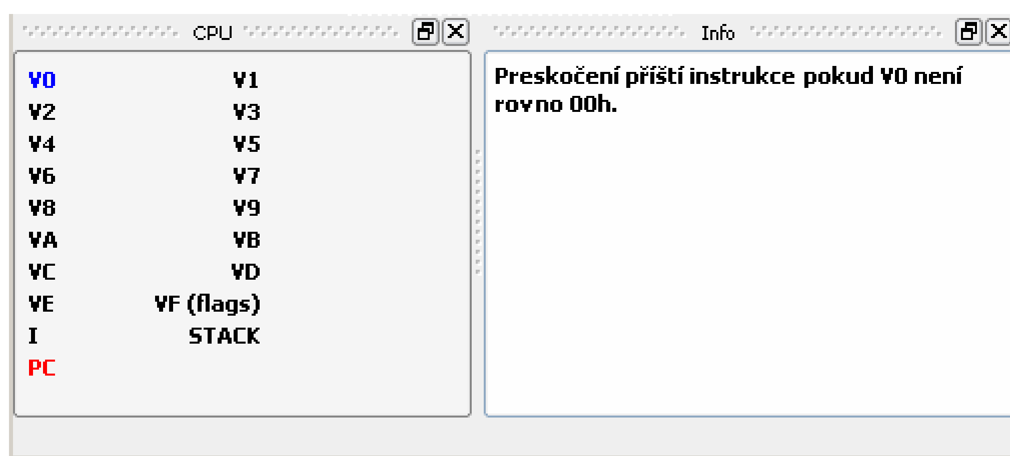
7 Okno s důležitými záznamy.

Jednotlivé dekódované instrukce jsou vykreslovány do přehledné tabulky, ve které je možné snadno se orientovat. Instrukce skoků a volání jsou rozšířeny o grafickou reprezentaci cíle skoku. (viz obrázek 8).

Adresa	Opcode	Instrukce	Komentář
0x00000210	30 00	skeq V0, 00h	podmíněný skok
0x00000212	DA B5	sprite VA, VB, 05h	
0x00000214	71 08	add V1, 08h	
0x00000216	7A 08	add VA, 08h	
0x00000218	31 30	skeq V1, 30h	nepodmíněný skok
0x0000021A	12 24	jmp [224h]	
0x0000021C	61 10	mov V1, 10h	

8 Tabulka s dekódovanými instrukcemi.

Z didaktického hlediska je vhodné každou instrukci vysvětlit a demonstrovat, jak ovlivňuje běh programu. Je důležité, aby tyto informace byly dostupné přímo v okně aplikace a uživatel je nemusel hledat například v manuálu dané instrukční sady. K tomuto účelu obsahuje *GDA* informační okno a okno s registry procesoru. V informačním okně je instrukce stručně popsána. V okně procesoru jsou zobrazeny registry, které procesor zvolené architektury obsahuje. Modře jsou zvýrazněny registry, ze kterých instrukce čte, červeně pak registry, které instrukce modifikuje. Obrázek číslo 9 tyto dvě okna prezentuje.



9 Okna s informacemi o instrukci.

Každé ze zmíněných oken jde libovolně přesouvat, zmenšovat nebo zavírat. To přispívá nemalou měrou ke zpříjemnění práce uživatele. Další důležitou funkcí je schopnost ukládat instrukce získané disassemblerem do různých formátů. Patří mezi ně jednoduchý textový formát (.txt), formát HyperText Markup Language (.html) či stále více populární formát PDF. Uživatel tak může výsledky dekódování okamžitě publikovat bez nutnosti cokoliv přepisovat či měnit.

Z dalších funkcí stojí za zmínku možnost komentovat dekódované instrukce, pohybovat se po jednotlivých adresách programu zapsaném v jazyce symbolických instrukcí či zvýrazňování syntaxe dekódovaných instrukcí. Zvýrazňování mění barvy a styl písma podle kategorie, do které ta

která instrukce spadá. Jinak jsou vypisovány instrukce přesunů a jinak například aritmetické instrukce. Styly písma u každé kategorie může uživatel libovolně měnit.

Vše je barevně sladěno ve snaze o co nejmenší rozptylování koncentrace uživatele. Ten má k dispozici dostatek prostředků pro vlastní úpravu vzhledu aplikace. Sic ne tolik výrazná, ale o to užitečnější je funkce ukládání všech nastavení. Bylo by totiž dosti nepraktické muset při každém startu aplikace vše nastavovat od začátku.

Podstatné je taktéž vícejazyčné rozhraní. Program dokáže sám zjistit jazykové nastavení operačního systému a podle něj přizpůsobit jazyk aplikace. V současné době *GDA* podporuje český a anglický jazyk. Není však problém tento výčet pomocí nástroje *Linguist* z toolkitu Qt dále rozšiřovat.

3.4 Programová implementace

Kostru programu tvoří hlavní okno. Je definováno jako třída `MainWindow` , která je potomkem třídy `QMainWindow` z definic toolkitu Qt. V konstruktoru této třídy jsou vytvářeny ovládací prvky okna. Jmenovitě se jedná o podtřídy `TextDockWindow`, `QTableWidget`, `QToolBar` a `QMenu`. Z pohledu funkčnosti je nejdůležitější třída `DecEngine`, jež vykonává celý proces analýzy a transformace kódů.

10 Diagram vytváření tříd.

11 Graf dědičnosti tříd.

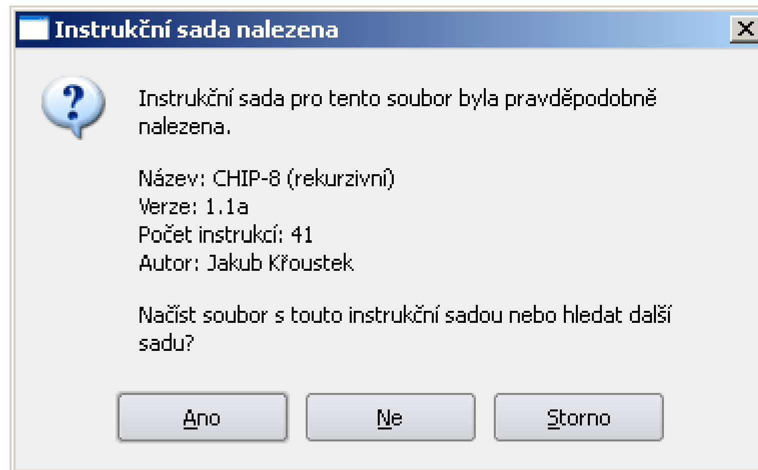
Třída `TextDockWindow` definuje rozhraní pro okno se záznamem událostí. Pomocí její veřejné metody poskytuje ostatním komponentám zapisovat do logu různé důležité akce. Pro lepší orientaci je ke každému záznamu evidován také čas vykonání. Celý seznam událostí lze ukládat či kopírovat.

Třída `QTableWidget` je použita pro práci s informačním oknem a oknem procesoru. Obsah těchto oken tvoří text formátovaný ve značkovacím jazyku HTML. Jazyk HTML umožňuje různorodé efekty, např. tvorba tabulek, barva a styl písma či vkládání obrázků, díky nimž informace v těchto oknech působí na uživatele vlídněji než prostý text. Okno procesoru prezentuje vliv instrukce na registry procesoru a okno informační nese jednoduchý popis o aktuálně vybrané instrukci.

`QToolBar` je název třídy, která obsluhuje panel nástrojů umístěný standardně v horní části hlavního okna. Je vytvořen pro urychlení uživatelské práce tak, že obsahuje nejpoužívanější akce, které jsou jinak obsaženy ve třídě `QMenu`. Těmi jsou: otevření a uložení souboru, editace komentáře instrukce, přechod na požadovanou adresu instrukce, spuštění zpětný překlad a změna nastavení. Jednotlivé akce jsou reprezentovány pomocí tlačítek s příslušnými ikonami. Pro větší komfort může být celá komponenta skryta či přesunuta na jiné místo v hlavním okně.

Prvek `QTableWidget` představuje tabulku, do které jsou vypisovány instrukce v jazyku symbolických instrukcí. Každý řádek tak obsahuje právě jednu instrukci. Sloupce této tabulky určují adresu, operační kód a operand instrukce, symbolické jméno instrukce a uživatelský komentář. Pro zvýšení čitelnosti kódu obsahuje tabulka ještě jeden, nepojmenovaný, sloupec, který slouží k určení cíle skoku a volání. Při vybrání takové instrukce se objeví malá barevná šipka, která uživateli ukáže polohu cílové adresy. Při výběru instrukce se také zobrazí ve zmíněných oknech její popis a funkce.

Programově nejzajímavější je třída `DecEngine`, jenž zastřešuje analýzu a transformaci kódu. Její metoda `detectInstrSet()` se stará o analýzu typu instrukční sady zvoleného binárního souboru. Ve spojení se zásuvnými moduly (viz dále) se snaží rozpoznat instrukční sadu tohoto souboru. Výsledky analýzy uživateli sdělí a poskytne mu možnost vybrat některou z metod dekódování (jak ukazuje obrázek č. 12). Druhou důležitou metodou je `disassemble().` Tato funkce převede vstupní soubor na vektor bajtů, pomocí zásuvného modulu ho transformuje na vektor symbolických instrukcí a ten vypíše. Formát jednotlivých položek tohoto vektoru je, stejně jako samotná implementace dekódování, vysvětlena v následující kapitole. Poslední významnější metodou je `getInstructionInfo().` Ta na požádání zjistí a vypíše vlastnosti a informace o jedné konkrétní instrukci.



12 Výběr zásuvného modulu.

3.5 Zásuvné moduly

Patrně nejobtížnější částí implementace byla etapa vývoje spojená se zásuvnými moduly. Podle vytyčených cílů musí být disassembler modulární a umožňovat rozšiřování o nové instrukční sady. Po prozkoumání těchto vlastností se jevily tři možná řešení problému instrukčních sad.

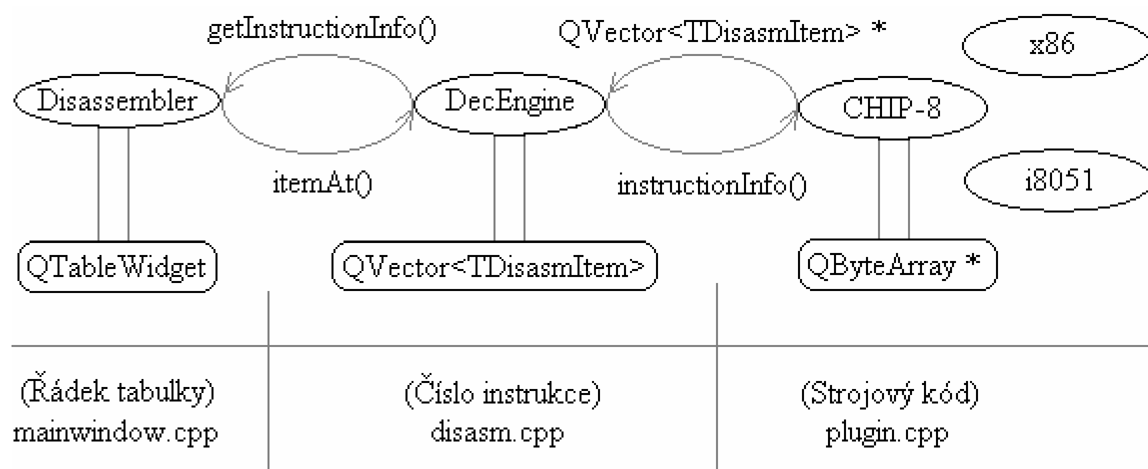
- *Přímé naprogramování sad do zdrojových souborů disassembleru.* Toto řešení se vyznačuje znamenitou rychlostí vykonávání, jelikož dekódování probíhá přímo v programu disassembleru. I přes jednoduchost implementace však dosti znehodnocuje zdrojový kód disassembleru obrovským nárůstem kódu. Největší nevýhoda je však prakticky nulová rozšiřitelnost programu o nové sady. Kvůli každé nové sadě by se program musel celí znovu kompilovat, což je velice nepraktické.
- *Instrukční sady definované pomocí XML*. Nové instrukční sady se v tomto přístupu načítají z externích souborů formátu XML. Uživatel by do XML šablony pouze dopisoval jednotlivé instrukce v předepsaném formátu a disassembler by je načítal. Pro koncového uživatele je tento způsob jednoduchý, rychlý a čitelný. Ne tak pro implementaci samotného disassembleru. Agenda spojená s načítáním je velice rozsáhlá. Disassembler musí obsahovat vlastní *parser*, který čte instrukce z XML souborů a až poté se může zpracovávat dekódovaný binární soubor. To velice zpomaluje průběh dekódování a zároveň neúměrně rozšiřuje obsah implementace disassembleru.
- *Instrukční sady formou zásuvných modulů (pluginů)*. Jedná se o kompromis mezi předchozími způsoby. Z XML návrhu čerpá jednoduchost přidávání nových sad a z kompilovaného řešení rychlost vykonávání. Uživatel nové sady vytváří pomocí šablon napsaných v jazyku C/C++. Je nutné, aby měl alespoň základní znalosti tohoto jazyka, ale na rozdíl od prvního řešení není nutné kompilovat vždy znovu celý program, ale postačí

kompilace pouze nové instrukční sady. Řešení je dostatečně rychlé i přes nutnost použití sdílené paměti mezi disassemblerem a zásuvným modulem.

Jak je patrné z názvu kapitoly, jako nejvýhodnější se ukázal způsob využívající *pluginy*. Ke zdrojovým souborům obsaženým v distribuci programu *GDA* je přidána i šablona pro zájemce o rozšíření programu o nové sady.

Pro demonstraci funkčnosti disassembleru byla zvolena instrukční sada CHIP-8 (viz příloha č. 1). Důvodem je lehce pochopitelný soubor instrukcí, jejich malý počet a srozumitelný formát symbolických instrukcí. Byly vytvořeny dva zásuvné moduly. První z nich ukazuje dekódování využívající lineární algoritmus (kapitola 2.4.1), druhý je založen na rekurzivním zanořování (kapitola 2.4.2). Hybridní algoritmus u této sady nemá využití, jelikož instrukční sada, kromě jedné výjimky, neobsahuje podporu nepřímých skoků. Z tohoto důvodu plně postačuje rekurzivní přístup.

Pro předání dekódovaných instrukcí hlavnímu programem používá každý modul vektor prvků typu `TDisasmItem`. Tato datová struktura obsahuje informace typu adresa, strojový kód, symbolický název, komentář a další. Pro vnitřní reprezentaci dekódovaných instrukcí moduly používají abstraktní datové typy mapa a tabulka s rozptýlenými prvky (*hash table*). Celá vnitřní implementace může však být uživatelem změněna. Nutné je pouze dodržet základní parametry rozhraní mezi disassemblerem a modulem. Princip komunikace je znázorněn na obrázku 13.



13 Princip komunikace mezi disassemblerem a zásuvnými moduly.

3.6 Ovládání

Standardní průběh použití disassembleru probíhá v těchto fázích:

- Spuštění disassembleru a výběr binárního souboru.
- Volba zásuvného modulu a výběr instrukční sady.

- Dekódování.
- Komentování transformovaného kódu, jeho analýza a studie.
- Uložení do požadovaného formátu.
- Ukončení programu nebo pokračování práce s jiným binárním souborem.

4 Závěr

V této bakalářské práci byla představena většina metod používaných ve vědním oboru zvaném reverzní inženýrství. Detailněji byly rozebrány metody analýzy a transformace kódů používané v informačních technologiích, a to především ty, které se uplatňují při zpětném překladu do jazyka symbolických instrukcí.

Nejpodstatnější znalosti jsou způsoby analýzy binárních souborů a transformační algoritmy. Byly prezentovány vlastnosti a využití většiny dnes používaných metod a diskutovány jejich rozšíření a modifikace. Jako další byly popsány problémy, které tuto činnost znesnadňují či mnohdy zcela vylučují.

Získané teoretické znalosti byly uvedeny do praxe při vývoji nástroje, jenž vznikl společně s touto prací. Tento nástroj, disassembler, slouží k převodu binárních, zkompilovaných souborů zpět do jazyka symbolických instrukcí. Při tomto procesu názorně ukazuje zmíněné metody a algoritmy. Mimo to dále poskytuje mnoho funkcí, které současné konkurenční nástroje stejného zaměření neumožňují. Mezi ty nejdůležitější patří podpora šablon, zásuvné moduly a modulárnost disassembleru.

Projekt by se dále mohl rozvíjet dvěma směry. Prvním z nich by byla evoluce stávajícího programu. Zvláště na optimalizaci rychlosti transformace by bylo dobré zapracovat. Rovněž by bylo vhodné přidat některé další funkce jako editace instrukcí, vyhledávání či vytvoření grafu toku programu. Program toto, díky své modulárnosti, umožňuje implementovat a problém by zde neměl nastat. Druhý směr vývoje je více zaměřen na spolupráci s jinými programy. Jak bylo již v práci uvedeno, dobrý disassembler se hodí například jako základ pro dekompilátor. V úvahu by tedy připadala spolupráce na dalším vývoji projektu [20]. Mezi další obory využití patří antivirové programy a jiné bezpečnostní programy. Zde by disassembler vyhledával řetězce instrukcí a na základě definovaných posloupností by prováděl heuristickou detekci škodlivého kódu. Neméně zajímavá je myšlenka využít stávající program v některém z vývojových prostředí pro popis mikroprocesorových architektur. Například v projektu Lissom [17], který využívá odlišný postup transformace založený na párových automatech, by mohla být kombinace obou technik velkou výzvou.

Literatura

[1] Ondřej, M., Drobník, J.: Trangenoze rostlin, 1. vydání. Academia, Praha, 2002.

[2] WWW stránka projektu Folding@Home, URL http://folding.stanford.edu (březen 2007).

[3] WWW stránka projektu Rosetta@Home, URL http://boinc.bakerlab.org/rosetta (březen 2007).

[4] WWW stránka portálu Genetika, URL http://genetika.wz.cz (březen 2007).

[5] Eilam, E.: Reversing: Secrets of Reverse Engineering. Wiley, 2005.

[6] Menezes, A., Van Oorschot, P., Vanstone, S.: Handbook of Applied Cryptography. Crc Press, 1996.

[7] Klíma, V.: Several observations regarding Chinese collisions of MD5, 3rd International Scientific Conference Security and Protection of Information. Brno, 2005.

[8] Halfhill, T.: AMD and Intel Harmonize on 64. In: MPR 3/29, 2004, s. 1-2.

[9] WWW stránka portálu Program-Transformation, URL http://www.program-transformation.org (březen 2007).

[10] Abd-El-Barr, M., El-Rewini, H.: Fundamentals of Computer Organization and Architecture. Wiley, 2005.

[11] Schwarz, J., Růžička, R., Strnadel, J.: Studijní opora pro předmět IMP. VUT Brno, FIT, 2006.

[12] Murdoccas, M. J., Heuring, V. P.: Principles of computer architecture. Prentice Hall, 1999.

[13] Goppit: Portable Executable File Format – A Reverse Engineer View, In: Code Breakers Journal, Vol. 2, No 3, 2005, s. 1-100.

[14] Schwarz, B., Debray, S. K., Andrews, G. R.: Disassembly of Executable Code Revisited. University of Arizona, Tucson, 2002.

[15] Cifuente, C., Van Emmerik M.: Recovery of Jump Table Case Statements from Binary Code. Proceedings of the International Workshop on Program Comprehension, květen 1999.

[16] Lukáš, R., Hruška, T., Kolář, D., Masařík, K.: Two-Way Deterministic Translation and Its Usage in Practice, In: Proceedings of 8th Spring International Conference - ISIM'05, Ostrava, CZ, MARQ, 2005, s. 101-107.

[17] WWW stránka projektu Lissom, URL http://merlin.fit.vutbr.cz/Lissom (březen 2007).

[18] Herout, P.: Učebnice jazyka C, 4. přepracováné vydání. Kopp, České Budějovice, 2006.

[19] WWW stránka firmy Trolltech, URL http://www.trolltech.com/products/qt (březen 2007).

[20] Šomlo, I.: Transformace kódů aplikovaná pro dekompilaci, bakalářská práce, Brno, FIT VUT v Brně, 2007.

Seznam příloh

Příloha 1. CHIP-8 a jeho instrukční sada.
Příloha 2. Obsah CD

Příloha 1 – CHIP-8 a jeho instrukční sada

CHIP-8 je souhrnný název pro koncept architektury, instrukční sadu a programovací jazyk. Byl vytvořen Josephem Weisbeckerem. Používal se především v 8mi bitových procesorech, které byly součástí herních zařízení během sedmdesátých let minulého století. Později bylo v různých zařízeních používáno rozšíření nazvané Super CHIP-8.

- Paměťový prostor se nacházel mezi adresami 200h a FFFh a byl 3584 bajtů velký. Program začínal vždy na adrese 200h.
- Procesor obsahoval 16 osmibitových registrů pro volné použití. Jejich jména jsou V0 až VF. VF je používán jako registr příznaků
- Zásobník pro ukládání návratových adres při volání je 48 bajtů velký, což umožňuje maximálně 12 zanoření.
- Dva čítače pracující na kmitočtu 60Hz. Jeden z nich pracuje jako časovač, druhý jako časovač zvukových efektů.
- Jediným vstupem je tlačítková klávesnice s 16 klávesami.
- Výstup tvoří monochromatický displej s rozlišením 64 x 32 bodů a jeden reproduktor.

Základní instrukční čítá 35 instrukcí, rozšířená instrukční sada Super CHIP-8 přináší dalších 6 instrukcí.

Operační kód	Význam
0NNN	Volání systémové funkce na adrese NNN.
00E0	Vymazání obrazovky.
00EE	Návrat z funkce.
1NNN	Skok na adresu NNN.
2NNN	Volání funkce na adrese NNN.
3XNN	Přeskoční následující instrukce pokud VX se rovná NN.
4XNN	Přeskoční následující instrukce pokud VX se nerovná NN.
5XY0	Přeskoční následující instrukce pokud VX se rovná VY.
6XNN	Nastavení VX na hodnotu NN.
7XNN	Přičtení NN k VX.
8XY0	Nastavení VX na hodnotu VY.
8XY1	Nastavení VX na hodnotu VX OR VY.
8XY2	Nastavení VX na hodnotu VX AND VY.
8XY3	Nastavení VX na hodnotu VX XOR VY.
8XY4	Přičtení VY do VX. Generuje se příznak přenosu.
8XY5	VX je sníženo o hodnotu VY. Generuje se příznak výpůjčky.
8XY6	Bitový posun VX o jedna vpravo. Generuje se příznak přenosu.
8XY7	Odečtení VY od VX. Generuje se příznak výpůjčky.

8XYE	Bitový posun VX o jedna vlevo. Generuje se příznak přenosu.
9XY0	Přeskoční následující instrukce pokud VX se nerovná VY.
ANNN	Nastavení indexového registru na NNN.
BNNN	Nepřímý skok na adresu NNN + V0.
CXNN	Nastavení VX na náhodné číslo < NN.
DXYN	Vykreslení bodu na souřadnici [VX, VY].
EX9E	Přeskoční následující instrukce pokud VX se rovná zmáčknuté klávese.
EXA1	Přeskoční následující instrukce pokud VX se nerovná zmáčknuté klávese.
FX07	Nastavení VX na hodnotu časovače.
FX0A	Čekání na stisk klávesy a její uložení do VX.
FX15	Nastavení časovače na VX.
FX18	Nastavení zvukového čítače na VX.
FX1E	Přičtení VX k indexovému registru.
FX29	Nastavení indexového registru na hodnotu VX.
FX33	Uložení BCD reprezentace registru VX na I, I+1 a I+2.
FX55	Uložení hodnot registrů V0 až VX na paměťové místo I.
FX65	Načtení registrů V0 až VX z paměti na adrese I.

Příloha 2 – Obsah CD

Adresář	Obsah
./readme.txt	Obsah CD.
./Bin/	Spustitelná verze programu (GDA.exe) včetně knihoven.
./Bin/_Testy/	Ukázkové příklady.
./Doc/	Elektronická verze této písemné zprávy.
./Install/	Programy nutné k překladu programu.
./Manual/	Programová dokumentace.
./Src/	Zdrojové soubory programu.