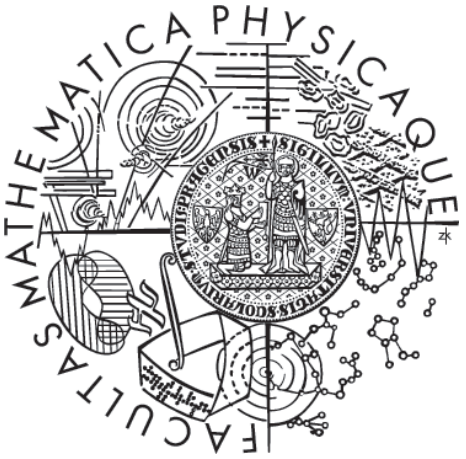
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE



Ladislav Láska

Anotující disassembler pro AMD64

Katedra Aplikované Matematiky

Vedoucí bakalářské práce:	Mgr. Martin Mareš, Ph.D.
Studijní program:	informatika
Studijní obor:	obecná informatika

Praha 2012

Rád bych poděkoval Mgr. Martinu Marešovi, Ph.D. za vedení práce, podnětné připomínky a jazykové i věcné korektury. Dále děkuji Mgr. Janu Hubičkovi, Ph.D. za cenné rady a Mgr. Petru Baudišovi za náměty a připomínky k programu. V neposlední řadě děkuji také Karlovi Lejskovi za poskytnutí reference instrukcí a pomoc s její reprezentací.

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

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

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

Název práce: Anotující disassembler pro AMD64

Autor: Ladislav Láska

Katedra: Katedra Aplikované Matematiky

Vedoucí bakalářské práce: Mgr. Martin Mareš, Ph.D.

Abstrakt: Cílem práce je vytvořit disassembler pro architekturu AMD64, který bude sloužit pro zjednodušení analýzy programů na úrovni strojového kódu. Práce popisuje formát instrukcí, binárních souborů a systémové konvence, které jsou následně využity ve vypracovaném disassembleru. Ten umí například analyzovat a vizualizovat skoky, pracovat s výchozím obsahem paměti, interpretovat volací konvence ABI, přejmenovávat a zjednodušovat výrazy, u kterých známe hodnoty a další. Nedílnou součástí je také skriptovací API pro Python, které umožňuje psát rozšiřující pluginy a přidávat funkce za hranice toho, na co myslel autor.

Klíčová slova: disassembler, AMD64, analýza kódu

Title: An annotating disassembler for AMD64

Author: Ladislav Láska

Department: Department of Applied Mathematics

Supervisor: Mgr. Martin Mareš, Ph.D.

Abstract: The goal of this work is to create a disassembler for the AMD64 architecture which would simplify program analysis in machine code. The work describes low-level instruction format, object file format and system conventions, on which we base our program. Its features are for example branch analysis and visualization, the ability to work with default memory contents, interpretation of ABI calling conventions, expression renaming and simplification based on known values. Scripting API for Python is integrated, which enables the user to write custom plug-ins, and extend its abilities beyond what the author has anticipated.

Keywords: disassembler, AMD64, code analysis

Obsah

1. Úvod

1.1 Motivace

Často se ocitneme v situaci, kdy potřebujeme přesně prozkoumat běh programu. Důvody mohou být různé.

Často jde o zkoumání chyb v programu, které lze těžko odhalit debuggerem – ať už proto, že jde o vícevláknový program a nebo díky optimalizacím neposkytuje vysokoúrovňový debugger užitečné informace. Někdy dokonce ani nemáme zdrojový kód od programu, který potřebujeme analyzovat.

Vždy máme možnost zkoumat kód pomocí *disassembleru*, tedy nástroje pro rozložení strojového kódu na jednotlivé instrukce v člověkem čitelné podobě. V takovém případě jsme ale na zkoumání programu zcela odkázáni sami na sebe.

Pokud je kód příliš komplikovaný a/nebo dlouhý, může proces zkoumání jednoho programu být časově náročný.

V některých případech si můžeme vypomoci *dekompilátorem*, tedy programem, který se pokusí strojové instrukce přeložit do vyššího jazyka (například C). To nám sice nepomůže při odhalování chyb v kódu, ale při zkoumání funkce neznámého programu může být cenným pomocníkem.

Bohužel dekompilace není zdaleka jednoduchý proces, protože při překladu programu do strojového kódu je ztraceno mnoho informací – od názvů funkcí a proměnných (pokud nejsou uloženy v symbolech, což ale mnohdy v produkčním prostředí nejsou), přes datové struktury až po mnohdy velké reorganizace kódu, které překladač provedl pro optimalizace. Zjistit původní záměr z dekompilovaného programu tedy může být stejně těžké, jako ze samotných strojových instrukcí.

I proto, že dekompilátor má relativně malý rozsah využití, většinou analyzujeme program pomocí disassembleru a textového souboru s poznámkami o zjištěných informacích – ať to jsou adresy funkcí a proměnných, rozložení zásobníku, volání funkcí a známé hodnoty.

Takový postup je sice funkční, ale časově náročný a náchlyný na chyby. Například přehlédnutím v adrese skoku může dojít k fatální misinterpretaci, které si můžeme, ale také nemusíme všimnout dřív, než investujeme mnoho zbytečného času.

Mnohem efektivnější by bylo použít to nejlepší z obou světů – absolutní přehled o kódu jako při užívání disassembleru, ale také částečné porozumění kódu programem na úrovni, aby uměl pomoci, ale nepřekážel ani nic nevnucoval. Například analýza skoků a jejich vizualizace, přejmenování elementárních výrazů (například pojmenování lokální proměnné na zásobníku) a nebo částečná aplikace volacích konvencí může velice pomoci.

1.2 Cíl práce

Cílem práce je napsat disassembler pro architekturu AMD64, běžící na Linuxu, který bude stát někde mezi „obyčejným“ disassemblerem a dekompilátorem. Primárně by měl ulehčit manuální práci a zmenšit prostor pro chyby. Autor takovéhoto softwaru ale nemůže nikdy myslet na všechny možnosti, měl by tedy umožňovat uživateli funkčnost programu dále rozšiřovat. Druhým hlavním cílem je tedy napsat skriptovací API pro Python, ve kterém půjde snadno napsat rozšiřující modul, mající přístup k dekódovaným instrukcím a grafickým prostředím pro interakci s uživatelem.

V první kapitole se věnujeme architektuře AMD64 ve světe Unixových systémů. Představujeme assembler z programátorského hlediska i z hlediska procesoru, formát souborů ELF a systémové konvence.

Pokračujeme představením vypracovaného programu v kapitole 3 instalací a rutinním používáním. V kapitole 4 se pak zabýváme pluginovým systémem a nakonec v kapitole 5 nahlédneme do zdrojových kódů.

Poslední část je Appendix A, kde uvádíme referenci API pro pluginy.

2. Architektura AMD64

V této kapitole se zběžně seznámíme s architekturou AMD64 na úrovni strojových instrukcí a některými konvencemi používanými na současných Unixových systémech. Většina textu v této kapitole vyžaduje základní znalosti assembleru a jazyka C.

2.1 Instrukce na AMD64

Architektura AMD64 vznikla jako zpětně kompatibilní rozšíření architektury IA-32, která je již mnoho let dominantní v osobních počítačích. Jde o logické rozšíření adres a přirozené velikosti operandů na 64 bitů.

Protože je AMD64 navržená jako zpětně kompatibilní, ale vyžadují se od ní nové vlastnosti, může být provozována v několika módech podle stupně kompatibility. V celé této kapitole budeme předpokládat tzv. *64-bit long mode*, který je nejméně kompatibilní a obsahuje všechny inovace. Vyznačuje se tím, že výchozí velikost adres je 64-bitová (data ale zůstávají 32-bitová), mnohé historické instrukce a módy jsou zrušeny (například segmentové adresování je značně omezeno) a některé instrukce mění svou sémantiku (například zásobníkové instrukce jako jedny z mála mají výchozí velikost dat 64-bitovou).

Tato část bude stručným úvodem do assembleru a formátu instrukcí, zejména jejich kódování na nejnižší úrovni. Pro jednoduchost je zde popsána pouze základní instrukční sada, tedy bez rozšíření jako SSE, AVX a dalších.

Čerpat můžeme ze dvou hlavních zdrojů. Pro potřeby této práce budeme používat manuály od firmy AMD, konkrétně AMD64 Architecture Programmer's Manual Volume 3: General-Purpose and System Instructions [3] (dále jen *manuál*). Alternativním zdrojem jsou manuály od firmy Intel, konkrétně Intel 64 and IA-32 Architectures Software Developer's Manual [4]. Protože tyto procesory jsou binárně identické, věcné odlišnosti mezi manuály příliš nenajdeme; názvosloví a některé pohledy se ale někdy výrazně liší.

2.1.1 Mnemonický formát instrukcí

Programátor v assembleru většinou neprogramuje v binárním kódu, ale píše instrukce v takzvané mnemonické formě, tedy formě srozumitelné pro člověka, která je samovysvětlující (nebo se alespoň snaží být).

Pro architekturu IA-32 existují historicky dvě základní varianty mnemoniky: *AT&T* a *Intel*. V Unixové světě je používanější varianta AT&T, nejprve se tedy podíváme na ni. Následně se podíváme, jak se liší od syntaxe Intelu (která je zavedenější ve světe Windows).

Varianta AT&T

Každá instrukce začíná tzv. *mnemonikou*, tedy jménem instrukce. Mnemoniku píšeme typicky malými písmeny a její poslední znak (*suffix*) může udávat velikost operace (pokud je zapotřebí). Tedy instrukci `mov` můžeme zapsat například jako `movl` nebo `movq` a tím provádět přesun 4 nebo 8 bytů.

Za mnemonikou následují čárkami oddělené operandy (pokud instrukce nějaké vyžaduje). Pokud je nějaký operand cílový (tj. je do něj zapsováno), uvádíme jako poslední. Každý operand může být buď literál, registr, nebo adresa paměti, kde se nachází hodnota operandu (nicméně každá instrukce nemusí podporovat všechny typy).

Literál píšeme se znakem `$` na začátku následovaným `0x` pro hexadecimální hodnotu, `0b` pro binární hodnotu a podobně.

Příklady: `$0x8` `$0x400bc0`, `$0b101`, `$0764`.

Registr píšeme se znakem `%` na začátku následovaným jménem registru psaným malými písmeny.

Příklady: `%rax`, `%eax`, `%al`, `%xmm0`.

Adresa v paměti se zapisuje obecně ve formátu:

```
displacement(base,index,scale)
```

Výpočet adresy je pak podle následujícího vzorce:

```
index * scale + base + displacement
```

Výrazy `index` a `base` jsou registry, `displacement` je literál a `index` je `1`, `2`, `4` nebo `8`. Každý ze sčítanců může být vynechán, vždy však musí zůstat alespoň jeden (tedy `index` a `scale` musí být přitomny oba nebo žádný).

Příklady: `$0xc(%rbx,%rcx,1)`, `(%rdi,%rsi,2)`, `(%rsp)`, `$0x18(,%rbp,8)`, `$0x401210()`.

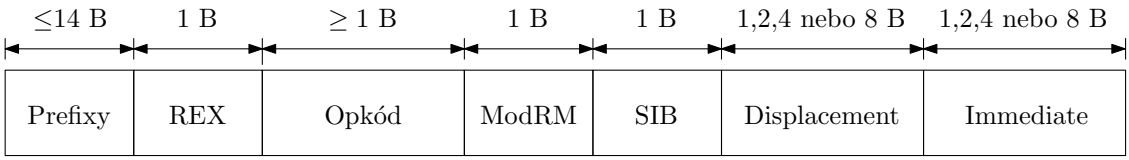
Varianta Intel

Syntaxe podle Intelu je v několika ohledech odlišná. Podívejme se, v čem tyto rozdíly spočívají. V tabulce 2.1 pak uvádíme několik příkladů zapsaných v obou variantách.

- Intel píše mnemoniku a jména registrů velkými písmeny, není to ale nutnost.
- Zápis jména instrukce: Intel nepřidává suffixy.
- Pořadí argumentů: Intel píše vždy cílový argument jako první. Pozor, toto pořadí je důležité i u instrukcí porovnávající argumenty, jako například `cmp`, které se chová, jako by odečítalo, ale ve skutečnosti svůj výsledek nezapíše a pouze nastaví příznaky.
- Velikost operace: Intel velikost operace hádá podle velikosti operandů. V případě, že není jasná (příklad s `dec` v tabulce 2.1), vyžaduje od programátora tuto velikost dospecifikovat modifikátorem velikosti, tedy například `DWORD PTR` nebo `BYTE PTR` atp.
- Zápis registrů: Intel píše registry bez znaku `%`.
- Zápis literálů: Intel píše literály bez znaku `$`, šestnáctkovké hodnoty se zapisují se suffixem `h`.
- Adresa v paměti: Intel píše infixový výraz adresy ve hranatých závorkách.

AT&T	Intel
`pushl %eax`	`PUSH EAX`
`movl %ebx, %eax`	`MOV EAX, EBX`
`addq $0x12, %rax`	`ADD RAX, 12h`
`decw (%ebx)`	`DEC WORD PTR [EBX]`
`cmpb $0x8, %al`	`CMP AL, 8`
`mov 0x4(%ebx), %eax`	`MOV EAX, [EBX+4]`
`mov 0x4(%ebx,%ecx,2), %eax`	`MOV EAX, [EBX + ECX*2 + 4]`

Tabulka 2.1: Rozdíly mezi syntaxí Intel a AT&T



Obrázek 2.1: Schéma instrukce

2.1.2 Přehled formátu

Každá instrukce má 1 – 15 bytů (delší instrukce způsobí výjimku procesoru). Instrukce nemusí být zarovnány a byty jsou uloženy v pořadí little-endian.

Jedinou nutnou součástí instrukce je tzv. *opkód* (operační kód), určující instrukci. Jak je znázorněno na obrázku 2.1, opkód může být doplněn dalšími informacemi:

- Před opkódem mohou být různé prefixy, které mění parametry instrukce (například velikost operandů a adres), viz tabulka 2.2.
- Prefix REX – musí být těsně před opkódem. Typicky rozšiřuje instrukce na 64-bitové verze.
- ModRM a SIB byty, pokud jsou vyžadovány instrukcí. Obsahují informace o operandech.
- Literály displacement (pro nepřímé adresování) a immediate (pro konstanty), v tomto pořadí (to je důležité, když se vyskytuje nepřímé adresování a konstanta v jedné instrukci, nebo mají různou velikost).

Než se podíváme na jednotlivé části podrobněji, zaveďme si konvenci:

Značení: Mluvíme-li o nějaké obecném registru, píšeme prostě „registr“. Pokud mluvíme o konkrétním registru, v libovolné velikosti, píšeme „rCX“ pro registry něco-*CX.*

2.1.3 Prefixy

Prefixy jsou byty, které se mohou vyskytnout na začátku instrukce, jak je vidět na obrázku 2.1. Prefixy typicky nějak pozměňují chování instrukce, v rozšiřujících sadách instrukcí jsou ale někdy „zneužívány“ jako další bit opkódu a chování

Skupina	Mnem.	Byte	Popis
Operand-size override	–	66	Změní výchozí velikost operandu.
Address-size override	–	67	Změní výchozí velikost adresy.
Segment override	fs	64	Vynutí použití segmentu FS.
	gs	65	Vynutí použití segmentu GS.
Lock	lock	F0	Zaručí atomicitu některých instrukcí.
Repeat	rep repz repe	F3	Opakuje stringovou operaci a ověřuje (ne)nulovost rCX, případně ZF (Zero Flag). Konkrétní sémantika záleží na instrukci.
	repne repnz	F2	

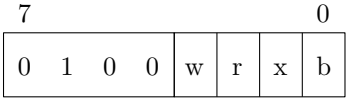
Tabulka 2.2: Přehled prefixů

instrukce mění úplně. Podle manuálu [3] by se prefixy neměly opakovat, většině procesorů to ale nevadí, a tak překladače běžně generují instrukce s vícenásobnými prefixy (zejména 66).

Přehled prefixů je v tabulce 2.2.

2.1.4 REX prefix

Instrukci také může předcházet jeden prefix REX. Jde o třídu prefixů, typicky rozšiřujících instrukci do 64-bitového módu.



Obrázek 2.2: REX byte

Každý REX prefix má 4 nejvýznamnější bity stejné, 4 nejméně významné bity pak určují jeho funkci (znázorněno na obrázku 2.2). Bit *w* přepíná velikost operandu na 64 bitů, zbylé se užívají jako rozšíření bytů ModRM a SIB, tedy umožňují přístup k přidaným registrům a adresním módům.

Pokud je použit prefix REX, musí to být poslední prefix před opkódem. Pozor také, že pouhá existence REX prefixu (tedy i „nulový“ 40) mění význam některých instrukcí (konkrétně čísla registrů, jak je vidět v tabulce 2.4).

Značení: Mluvíme-li o prefixu REX.w, myslíme tím prefix REX s nastaveným bitem w. Pro ostatní bity analogicky.

2.1.5 Opkód

Opkód určuje instrukci, která se má provést. Velmi často se jedná o jediný byte, protože ale existuje více než 256 instrukcí, může být rozšířen tzv. escapovacím znakem[1] (např. 0F), který změní význam dalšího bytu. Těchto escape znaků může být několik za sebou – instrukce s takto dlouhými opkódy patří často do nějaké rozšiřující sady, jako například SSE atp. Pokud budeme dále mluvit o opkódu, myslíme tím celou sekvenci i s escape znaky.

[1] Intel ve svých manuálech mluví o primárním a sekundárním opkódu (a případně dalších pro více než dvoubytové instrukce). AMD nazývá opkódem pouze poslední z této sekvence, všechny předchozí (podle Intelu) opkódy pak nazývá escape znaky. Mluví ale o tom samém.

Kromě vícebytových opkódů také můžeme potkat opkódy, které zasahují až do ModRM bytu (tj. obsah ModRM bytu určuje variantu instrukce zásadnějším způsobem, než definicí operandu). Takové instrukce nejsou výjimečné, většinou ale dále budeme mluvit o instrukcích – a v tom případě již myslíme konkrétní verzi po rozlišení pomocí ModRM bytu či jiných vlastností (prefixy).

2.1.6 ModRM a SIB byty

Instrukce typicky vyžadují operandy, se kterými se má pracovat. Pro každou instrukci se předem ví (podle opkódu), jaké má mít operandy. Některé instrukce však mohou mít operandy více druhů – registr, paměť, immediate hodnota, nebo nějaká kombinace.

(a) ModRM (b) SIB

Obrázek 2.3: ModRM a SIB byty

Některé opkódy jsou následovány byty ModRM a případně SIB. Každý z nich je sdružení tří hodnot do jednoho bytu, jak je znázorněno na obrázku 2.3.

Byte ModRM typicky specifikuje operandy instrukce, v některých případech obsahuje rozšíření opkódu v poli `reg` a také určuje, zda následuje SIB byte. Konkrétní interpretaci najdeme v sekci 2.1.8, která se operandy zabývá.

2.1.7 Displacement a Immediate

Obě pole slouží k zakódování konstanty přímo do instrukce, obě mohou být veliká 1, 2, 4 nebo 8 bytů. Displacement se využívá jako aditivní konstanta k adrese při relativním adresování. Immediate je konstantní operand, některé rozšiřující instrukce toto pole však využívají jako další rozšíření opkódu.

Každé z polí se může vyskytnout v rámci jedné instrukce nanejvýše jednou a pokud se vyskytují obě, immediate je vždy po displacementu.

2.1.8 Kódování operandů

Operandy mohou být v instrukci zkódovány dvěma způsoby: přímo v opkódu, nebo pomocí ModRM bytu. Jak jsou operandy (a kolik) kódovány víme přímo z opkódu.

Pokud je operand zakódován přímo v opkódu, pak 3 nejméně významné bity identifikují registr. Například tedy instrukci `pop` náleží opkódy `58` – `5F`.

Pokud je operand zakódován v ModRM bytu, je situace komplikovanější. Jak již víme, ModRM se skládá ze tří polí – `mod`, `reg` a `r/m`. Pole `reg` vyjadřuje vždy číslo registru (pokud je zpotřebí). Význam `mod` a `r/m` v 64-bitovém adresním módu udává tabulka 2.3 (`%rm` značí registr s číslem v poli `rm`, `disp8/32` displacement), pro 32-bitový mód je adresování stejné, až na to, že se místo `%rip` použije 0.

mod	r/m = 100	r/m = 101	Pro ostatní r/m
00	SIB	%rip + disp32	[%rm]
01	SIB	[%rm] + disp8	
10	SIB	[%rm] + disp32	
11	%rm		

Tabulka 2.3: Adresování pomocí ModRM v 64-bitovém adresním módu

Tam, kde je v tabulce vyznačeno SIB, následuje SIB byte a udává adresu operandu. Ta se vypočte podle vzorce:

```
(scale << index) + base + displacement
```

Kde << je bitový posun doleva, `index` a `scale` jsou registry. Displacement je přítomen pouze, pokud je `base=101`.

Poznámka: Tento vzorec již známe – používali jsme ho při nepřímém adresování, tam ale nebyl bitový posun. Pole `scale` *totiž nabývá hodnot 0–3, ale my jsme chtěli 1, 2, 4 a 8. Musíme tedy použít mocniny dvou, což právě splňuje bitový posun.*

1. `%rbp` není nikdy `base`. Namísto něj je použit 32-bitový displacement.

2. `%rsp` není nikdy `index`. Namísto něj je použita 0.

2.1.9 Čísla registrů

Každý registr má své číslo. V originální architektuře IA-32 bylo 8 adresovatelných registrů, s příchodem AMD64 bylo přidáno dalších 8. Tabulka 2.4 ukazuje čísla registrů. Ve sloupečku REX je vyznačeno „–“ pokud není REX prefix přítomen, „*“ pokud je přítomen, ale nezajímají nás jeho bity, a „b/r/x“ pokud je přítomen a příslušný bit je nastaven.

Který bit konkrétně rozšiřuje číslo registru záleží na tom, kde se to číslo vyskytuje: REX.r rozšiřuje pole `reg` v ModRM bytu, REX.x pole `index` v SIB bytu a REX.b pole `base` v SIB bytu pokud je přítomen, jinak `r/m` v ModRM.

Velikost	REX	0	1	2	3	4	5	6	7
8	–	%al	%cl	%dl	%bl	%ah	%ch	%dh	%bh
8	*	%al	%cl	%dl	%bl	%spl	%bpl	%sil	%dil
8	b/r/x	%r8l	%r9l	%r10l	%r11l	%r12l	%r13l	%r14l	%r15l
16	–	%ax	%cx	%dx	%bx	%sp	%bp	%si	%di
16	b/r/x	%r8w	%r9w	%r10w	%r11w	%r12w	%r13w	%r14w	%r15w
32	–	%eax	%ecx	%edx	%ebx	%esp	%ebp	%esi	%edi
32	b/r/x	%r8d	%r9d	%r10d	%r11d	%r12d	%r13d	%r14d	%r15d
64	–	%rax	%rcx	%rdx	%rbx	%rsp	%rbp	%rsi	%rdi
64	b/r/x	%r8	%r9	%r10	%r11	%r12	%r13	%r14	%r15

Tabulka 2.4: Čísla a jména registrů

2.2 Formát ELF

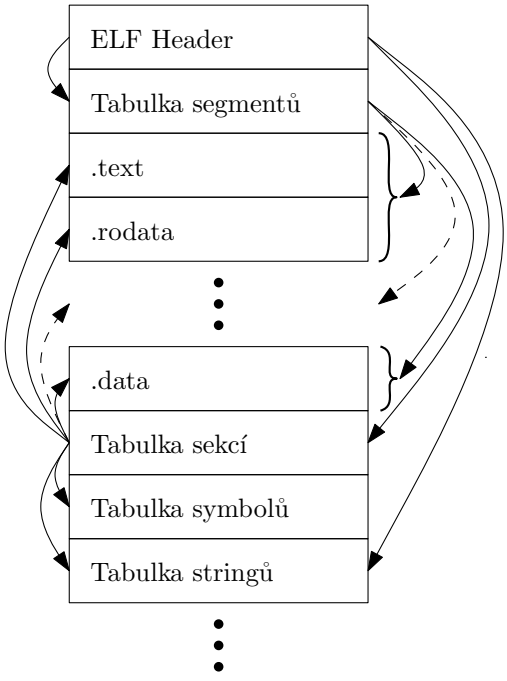
V současné době nejrozšířenější formát pro ukládání objektových, spustitelných a dalších druhů souborů, je ELF – *Executable and Linkable Format*. Je popsaný jako součást System V ABI [1], původně pro 32-bitové architektury, ale později rozšířen i pro 64-bitové. Pro naše účely budeme využívat právě tuto 64-bitovou variantu, jakožto nejpoužívanější na architektuře AMD64 v Unixovém světě.

2.2.1 Libelf

Pro zjednodušení přístupu k datům budeme využívat knihovnu `libelf` (dále jen *knihovna*), konkrétně implementaci z `elfutils` [7]. Knihovna nám exportuje struktury jazyka C, které obsahem odpovídají nízkoúrovňové struktuře souboru.

Dále budeme tedy nahlížet na soubor ELF z pohledu knihovny, tedy jako na serializované struktury. To je ale přesně to, čím ELF vlastně je (tj. jeho struktura je přímo mapovatelná na struktury jazyka C, které používáme).

Mluvíme-li dále o *makrech*, *strukturách* a *konstantách*, myslíme tím makra, struktury a konstanty definované v knihovně, pokud není řečeno jinak. Popisy struktur a hodnoty konstant nalezneme v hlavičkových souborech knihovny.



Obrázek 2.4: Struktura ELF souboru

2.2.2 Struktura

Formát ELF se skládá z *hlavičky*, která je
vždy na začátku soubor. Za ní následují další bloky dat, což můžou další informace o souboru, nebo přímo data programu. Tyto bloky nemají pevně dané pořadí a pro získání pozice daného bloku je potřeba mít strukturu, která na něj odkazuje. Jednotlivé vazby jsou nastíněny na obrázku 2.4.

Datová reprezentace

Protože je ELF stavěn jako univerzální formát, není omezen na jedinou architekturu. O jaký druh souboru je vždy specifikováno v hlavičce (`e_ident`). ELF32 a ELF64 jsou totiž zásadně rozdílné – jeden velikostí některých členů, ale v několika případech také přerovnáním členů struktur. Všechny toto drobnosti lze nalézst v ABI [1] a za nás je navíc vyřeší knihovna, nebudeme se jimi tedy dále zabývat.

Poznamenejme jenom, že formát dat programu je „přirozený“ pro danou architekturu (tedy korektně zarovnán na násobky vhodné mocniny dvou a v korektní endianitě), při manipulaci s nimi tedy není potřeba zvláštní pozornosti.

ELF Header

Hlavička ELF (dále jen *hlavička*) je vždy na začátku souboru a obsahuje všechny potřebné informace k získání všech potřebných dat ze souboru. Kromě informací o formátu (typ obsahu, cílový procesor, verze formátu atp.) obsahuje zejména odkazy na tabulky dalších struktur, ve kterých pro změnu nalezneme odkazy na data, která nás zajímají – instrukce, statická data, symboly, debugovací informace a další.

V knihovně je reprezentována strukturou `Elf64_Ehdr`, která má následující položky:

`e_ident`	Identifikace objektového souboru a informace o formátu dat. Zde je mimo jiné specifikováno, zda jde o 32 nebo 64-bitový formát (na indexu `EI_CLASS`) a endianita (na indexu `EI_DATA`).
`e_machine`	Identifikace cílové architektury. V našem případě musí být rovna `EM_X86_64`.
`e_version`	Verze formátu. V současné době je jediná platná verze a to 1.
`e_entry`	Adresa vstupního bodu programu.
`e_phoff`	Pozice v souboru (v bytech), kde začíná tabulka segmentů (*program header offset*)..
`e_shoff`	Pozice v souboru (v bytech), kde začíná tabulka sekcí.
`e_flags`	Příznaky specifické pro procesor.
`e_ehsize`	Velikost souborové hlavičky.
`e_phentsize`	Velikost jednoho záznamu v tabulce segmentů.
`e_phnum`	Počet záznamů v tabulce hlaviček segmentů.
`e_shentsize`	Velikost jednoho záznamu v tabulce sekcí.
`e_shnum`	Počet záznamů v tabulce sekcí.
`e_shstrndx`	Odkaz na tabulku řetězců (o tabulkách řetězců za chvíli).

2.2.3 Segmenty a sekce

V hlavičce nalezneme odkazy do dvou hlavních tabulek – tabulek segmentů a sekcí. Oba druhy záznamů slouží ke zpracování souboru pro různou činnost.

Segmenty existují několika druhů, všechny ale slouží pro zavedení a spuštění programu. Mohou obsahovat buď data, která se mají nahrát do paměti (spolu s informacemi, kam a jaké mají mít zarovnání), nebo různé další informace potebné ke korektnímu zavedení programu – například informace o potřebných knihovnách.

Sekce jsou abstrakcí používanou při linkování programu. Poskytují jednak alternativní pohled na některé informace obsažené v segmentech, ale také další informace pro linker, tabulky symbolů, debugovací informace a další. Oproti segmentům nejsou potřeba při spouštění programu.

Pro lepší pochopení se podívejme přesněji na záznamy v těchto tabulkách.

Hlavička segmentu

`p_type` Typ segmentu. Zajímavé možné hodnoty:

- `PT_NULL` Nepoužitý segment. Jeho hodnoty jsou nedefinované a měl by být ignorován.
- `PT_LOAD` Segment pro přímé nahrání do paměti. Jeho umístění a velikost jsou definovány v příslušných dalších členech struktury. Pokud je dat v souboru méně než má být velikost paměti, doplní se nulami.
- `PT_DYNAMIC` Informace o dynamickém linkování.

`p_flags` Bitové pole oprávnění ke čtení, zápisu a spouštění daného segmentu.

`p_offset` Pozice v souboru, kde segment začíná.

`p_vaddr` Adresa ve virtuální paměti, kde má segment začínat.

`p_paddr` Adresa ve fyzické paměti, kde má segment začínat. Toto pole je většinou ignorováno, protože typicky uživatelské aplikace nemohou specifikovat fyzické adresování.

`p_filesz` Velikost dat v souboru. Toto číslo může být menší než `p_memsz` a dokonce i nula. V takovém případě se zbylá data považují za nuly.

`p_memsz` Velikost dat v paměti. Toto číslo může být větší než `p_filesz`. V takovém případě se chybějící data považují za nuly.

`p_align` Vyžadované zarovnání segmentu (jak v souboru, tak v paměti). Dříve uvedené adresy by jej měly respektovat. Typicky se jedná o kladnou mocninou dvou.

Hlavička sekce

`sh_name` Odkaz na jméno sekce. Toto je index do tabulky stringů, která je odkazována v souborové hlavičce.

`sh_type` Typ sekce. Některé zajímavé:

- `SHT_NULL` Prázdná sekce. Zbylé položky hlavičky nemají žádný význam a měly by být ignorovány.
- `SHT_PROGBITS` Data příslušná programu. Jejich význam není nijak interpretován.
- `SHT_NOTBITS` Data příslušná programu, až na to, že nezabírají žádné místo v souboru – je to jenom záznam o tom, že je pro ně potřeba alokovat paměť.
- `SHT_SYMTAB` Tabulka symbolů (viz níže).
- `SHT_STRTAB` Tabulka stringů (viz níže).

	`SHT_REL` `SHT_RELA`	Relokační informace.

`sh_flags` Bitové pole příznaků sekce. Zajímavé jsou následující:

`SHF_WRITE`	Data v této sekci lze měnit za běhu programu.
`SHF_ALLOC`	Data v této sekci jsou přítomna v hlavní paměti za běhu programu. Některá data (například debugovací symboly) se do paměti totiž nenahrávají.
`SHF_EXECINSTR`	Data jsou spustitelné instrukce.

2.2.4 Tabulka stringů

Již jsme potkali odkazy do tabulky stringů, ve které nalezneme například názvy sekcí. Je tedy na čase podívat se, jak tabulka stringů vypadá.

Jednotlivé záznamy jsou ukončené nulou (jako v C) zřetězené za sebe. Odkaz do této tabulky jednoduše ukazuje na první znak žádaného stringu.

První a poslední byte celé tabulky je nula, aby se zajistila existence prázdného řetězce (první položka) a každý string byl ukončený (poslední položka).

Pozor ale, že ačkoli je první a poslední znak definovaný jako nula, může existovat i prázdná tabulka stringů. Ta žádná data v souboru nemá, tudíž je nulu nutno „doplnit“. Všechny nenulové indexy do takovéto tabulky jsou neplatné.

2.2.5 Tabulka symbolů

Další důležitou tabulkou je tabulka symbolů. *Symbol* je záznam přiřazující adrese jméno a případně další informace.

Symboly obsahují informace potřebné pro relokacím symbolických referencí (tj. propojení na externí knihovny), vyhledávání napojení cizích knihoven na náš program, ladící informace a případně další.

Podívejme se tedy, co jednotlivé záznamy obsahují.

`st_name` Jméno symbolu. Odkazuje do globální tabulky stringů.

`st_info` Informace o viditelnosti symbolu, typu a dalších.

`st_other` Typicky 0, nyní nemá žádný význam.

`st_shndx` Odkaz to tabulky sekcí, ve které se symbol vyskytuje. Může být `SHN_UNDEF` v případě, že jde o externí symbol.

`st_value` Nese různé informace, podle typu symbolu. Ve spustitelných souborech většinou virtuální adresu symbolu.

`st_size` Velikost dat symbolu (například délka objektu).

2.2.6 Relokace a dynamické linkování

Protože se dnes často využívají sdílené knihovny, nejsou všechny reference na funkce dostupné při sestavení programu.

Každý soubor, který se účastní dynamického linkování, obsahuje sekci `.interp`, ve které je cesta k dynamickému linkeru (nulou ukončený string). Linker se nahraje do paměti místo našeho souboru a je zodpovědný za vyřešení závilostí.

Závislosti získáme v sekci `.dynamic`. Každý záznam v této sekci má pouze dvě položky: `d_tag`, která určuje význam pole `d_un`. Pole `d_un` je union obsahující buď virtuální adresu, nebo celé číslo. Dále se na ně budeme odkazovat jako na `d_ptr` (pro adresu) a `d_val` (pro celé číslo).

Pro nás jsou zajímavé následující hodnoty `d_tag`:

`DT_STRTAB` Záznam v `d_val` nese odkaz na tabulku stringů odkazovanou v ostatních záznamech této sekce.

`DT_NEEDED` Potřebná dynamická knihovna. `d_val` nese index do tabulky stringů udané záznamem `DT_STRTAB`.

Poznámka: Sekce `.dynamic` *má také svůj vlastní segment* `PT_DYNAMIC`*. V případě, že v souboru nejsou sekce, dá se vyhledat podle segmentu. Zároveň v ní nalezneme záznamy jako* `DT_RELA`*,* `DT_RELASZ`*,* `DT_SYMTAB` *a další, které odkazují na sekce potřebné dynamickému linkování. Opravdu se tedy bez sekcí ve spustitelném souboru obejdeme.*

Dynamický linker tedy rekurzivně prohledá tyto záznamy, dokud nenahraje všechny knihovny. Následně projde všechny relokační údaje všech nahraných knihoven (včetně našeho souboru) a provede na nich relokaci.

Relokační údaje se nacházejí v sekcích typu `SHT_REL` a/nebo `SHT_RELA` a obsahují následující informace:

`r_offset` Adresa, kde se má provést relokace. Konkrétní akce ja ale závislá na architektuře (pro AMD64 je několik desítek možností).

`r_info` Obsahuje informace o typu relokace, případně index do tabulky symbolů. Přesný význam je závislý na architektuře.

`r_addend` Obsahuje dodatečnou informaci pro relokaci. Pozor, tato informace je obsažena pouze ve struktuře `Elf64_Rela`. Struktura `Elf64_Rel` jej nemá a je o to kratší, nicméně by se podle ABI používat neměla.

V případě, že relokační záznam neobsahuje referenci na symbol (například protože jde o lokální relokace, například pomocí pozice sekce), vyřeší se lokálně.

V opačném případě přicházejí do hry zpět symboly. Ke každému externímu symbolu musí existovat globální symbol, linker je spáruje a použije jejich adresu (která u externího symbolu není známa, proto musí dojít k párování) a vykoná relokační akci na adrese udané v relokačním záznamu.

Ve skutečnosti je ale relokační proces o něco komplikovanější, podrobnosti zde však nebudeme rozebírat – lze je nalézt v článku Ulricha Dreppera How to write shared libraries [10].

2.3 ABI

Programy nejsou samostatné sekvence instrukcí, ale musí koexistovat v relativně komplexních prostředích – běžně se setkávají s operačním systémem a různými (dynamickými) knihovnami. Aby taková koexistence mohla fungovat, je potřeba mít ucelený soubor pravidel, jak spolu tyto součásti mají interagovat – takové sadě se říká *Application Binary Interface*, tedy „binární rozhraní pro aplikace".

Na UNIXových systémech dnes potkáme téměř výhradně tzv. System V ABI [1] (dále jen ABI), které bylo vyvinuto pro *Unix System V* firmou AT&T – s časem samozřejmě vylepšováno. Definuje celou paletu konvencí a rozhraní, mezi jinými například již zmíněný formát ELF, některé standardní knihovny a již zmíněné rozhraní zkompilovaných programů.

Některé části definované v ABI jsou společné pro všechny architektury – například bychom si přáli, aby standardní knihovny a formáty souborů byly pokud možno stejné. Různé nízkoúrovňové konvence ale musí být specifické pro procesor (nemůžeme mluvit o instrukci `call` a tvářit se, že na všech procesorech se chová stejně – skoro určitě se totiž bude chovat jinak napříč architekturami). V takových případech se ABI odkazuje na tzv. *processor-specific supplement*, který v našem případě bude pro AMD64 [2].

ABI se liší nejenom napříč architekturami, ale také napříč platformami. Například na platformě MS Windows plní podobný účel, jako System V ABI, *x64 Software Convention*[2]. Tato konvence je System V ABI místy podobná (zejména tam, kde konvenci do jisté části diktuje architektura procesoru), nicméně nekompatibilní. Pro účely této práce ji vůbec neuvažujme.

Vyčerpávající souhrn různých volacích konvencí můžeme nalézt v textu Software optimization resources od Agnera Foga [8].

2.3.1 Zásobník

Procesor umí zcela přirozeně pracovat s tzv. *zásobníkem* (tedy mají zabudované instrukce k tomu určené). Zásobník je obecně datová struktura, do které se přidávají a odebírají prvky od konce. Na vrchol zásobníku vždy ukazuje registr `%rsp`. Instrukce `push` a `pop` uloží, resp. vyzvednou, hodnotu z vrcholu zásobníku a příslušně upraví `%rsp`. Pozor, že pojem „vrchol" není úplně správný. Zásobník totiž roste směrem dolů, tedy od vysokých adres po nižší (jak tomu ostatní bývá na mnoha procesorech).

Instrukce `push` tedy sníží `%rsp` o vhodnou hodnotu a zapíše na adresu `%rsp` svůj argument. Instrukce `pop` naopak zapíše data z `%rsp` do svého argumentu a zvýší hodnotu `%rsp`. Vhodná hodnota přitom závisí na módu, typicky je to 8 pro 64-bitové architektury.

K čemu je nám takový zásobník dobrý? Jednak si na něj funkce může ukládat lokální proměnné a také nám pomůže při volání funkcí.

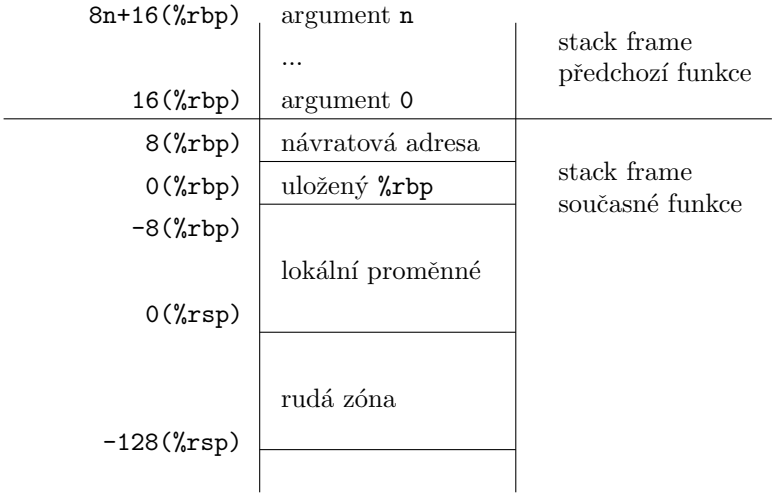
2.3.2 Volání funkcí

Voláme-li funkci, musíme vyřešit dva základní problémy: jak zajistit návrat na správné místo v kódu a jak jí předat argumenty. U funkcí, které nejsou přístupné

[2] `http://msdn.microsoft.com/en-us/library/9b372w95(v=vs.80)`

z vnitřku programu nebo knihovny, si může programátor (nebo překladač) zvolit libovolnou metodu – někdy je to dokonce výhodné a žádané z hlediska optimalizací. Nicméně pokud chceme komunikovat s okolním prostředím, je potřeba nějaká konvence, v našem případě definovaná v ABI.

Součástí volání každé funkce je tzv. *stack frame*. Je to část zásobníku vymezená pro každé volání funkce s předepsaným obsahem. Na obrázku 2.5 je schéma stack framu definované v ABI[2].



Obrázek 2.5: Schéma stack framu

Neboli volající funkce na zásobník zapíše argumenty funkce v opačném pořadí (za chvíli si ukážeme, že se na zásobník zapisují jenom některé) a adresu instrukce, na které se má pokračovat po návratu z funkce. Dále nepovinně následuje uložený registr `%rbp` (volitelně frame pointer, ukazatel na začátek stacku funkce) a další data, typicky lokální proměnné a nebo například argumenty pro další volání funkce. Na konec zásobníku vždy ukazuje `%rsp` (Stack Pointer, ukazatel na konec zásobníku).

Za koncem zásobníku začíná tzv. rudá zóna, veliká 128 bytů. To je místo, kam si funkce může téměř beztrestně ukládat lokální proměnné – data na tomto místě se nemusí zachovat přes volání vnořené funkce. Naopak obsluha přerušení si je tohoto místa vědoma, přeskočí ho a svůj stack frame (pokud potřebuje) založí až později.

Již jsme zmínili, že to s předáváním argumentů není tak jednoduché. Argumenty se totiž předávají preferovaně v registrech a až když dojdou registry, nebo se argument nevejde do registru, použije se zásobník. ABI definuje několik tříd argumentů a každá se chová trochu jinak.

Nejprve pro jednoduchost uvažujme jenom argumenty, které se vejdou do registrů. Argumenty se přednostně ukládají do registrů, po řadě konkrétně do `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8` a `%r9`. Pokud je argumentů více, použijí se již zmíněné sloty na zásobníku (které mají vždy velikost 8 bytů).

Pokud se míchají argumenty více typů (například floaty), mechanismus se zkomplikuje. ABI definuje několik tříd argumentů a ke každé třídě její sadu registrů, v jaké se mají předávat. Při předávání argumentu se tedy použije nejnižší registr z dané třídy a pokud není žádný volný, použije se zásobník. Výčet všech

Registr	Zachován	Význam
`%rax`	Ne	Návratová hodnota
`%rbx`	Ano	–
`%rcx`	Ne	Argument 4
`%rdx`	Ne	Argument 3
`%rsi`	Ne	Argument 2
`%rdi`	Ne	Argument 1
`%rbp`	Ano	Frame Pointer
`%rsp`	Ano	Stack Pointer
`%r8`	Ne	Argument 5
`%r9`	Ne	Argument 6
`%r10`, `%r11`	Ne	Scratch
`%r12` – `%15`	Ano	–

Tabulka 2.5: Význam některých registrů při volání funkcí

tříd a registrů nalezneme v ABI supplementu [2].

Nastává vyřešit ještě jednu otázku: co se stane s registry, když voláme funkci? ABI definuje tzv. registry ukládané volajícím a volaným. Registry ukládané volajícím se po návratu z funkce mohly změnit, a tak pokud je chce zachovat, musí si je uložit (třeba na zásobník) a následně obnovit. Těmto se také říká *scratch* („črtací“) registry, protože si do nich črtáme dlouhodobě nepotřebná data (například mezivýpočty). Naopak registry ukládané volaným jsou registry, které musí při návratu z funkce zachovat svou hodnotu a volaný se tedy musí zasloužit o to, aby je zachoval. Již jsme si mohli všimnout, že ve stack framu je místo na `%rbp`, a to právě proto, že je potřeba ho obnovit při návratu z funkce (to ale můžeme obejít třeba tím, že ho nebudeme vůbec měnit, nebo jinak zařídíme, že v době návratu jeho původní hodnota bude korektní).

Tabulka 2.5 dává přehled vlastností některých registrů a jejich využití při volání funkcí.

2.3.3 Příklad

Pro lepší pochopení si ukažme na příkladě, jak takové volání funkce vypadá. Na obrázku 2.6 je ukázkový zdrojový kód a k němu odpovídající assembler[3]. Jednotlivé řádky jsou očíslovány pro rychlou referenci.

Podívejme se podrobně, co takový kód pro funkci `main` dělá (funkci `foo` si může zanalyzovat čtenář, nepřináší nic nového). Nejprve na řádku `1` alokujeme místo na zásobníku, konkrétně 18 bytů (ty lze využít pro lokální proměnné).

Pohledem na řádek `5` vidíme, že budeme volat funkci. Na řádcích `2-5` připravíme argumenty: Zapíšeme do `%rsi` adresu nějaké lokální proměnné (že je to lokální proměnná, zjistíme pohledem na adresu, která je pozitivní vůči `%rsp` a menší než místo na zásobníku, které jsme si před chvíli alokovali), do `%edi` hodnotu `$0x4006bc` a do `%eax` nulu.

[3]Vytvořen pomocí `gcc -O1`, protože bez optimalizací překladače generuje značně nepřehledný kód plný zbytečných přesunů dat a výpočtů.

```
int foo(int i) {
    printf("%i", i);
}

int main() {
    int i;
    scanf("%i", &i);
    foo(i);
}
```

```
000000000040059e <main>:
1    sub    $0x18, %rsp
2    lea    0xc(%rsp), %rsi
3    mov    $0x4006bc, %edi
4    mov    $0x0, %eax
5    callq  0x400490 <scanf@plt>
6    mov    0xc(%rsp), %edi
7    callq  0x400584 <foo>
8    add    $0x18, %rsp
9    retq
```

```
0000000000400584 <foo>:
10   sub    $0x8, %rsp
11   mov    %edi, %esi
12   mov    $0x4006bc, %edi
13   mov    $0x0, %eax
14   callq  0x400470 <printf@plt>
15   add    $0x8, %rsp
16   retq
```

Obrázek 2.6: Jednoduché volání funkce v jazyce C a jeho assembler

Hodnota v `%edi` je na první pohled trochu záludná, protože nevím, co se tam skrývá. Pohledem do souboru (například `objdump -x`) ale snadno zjistíme, že jde o data v sekci `.rodata`, a tak se jedná o nějaká statická data – v našem případě to je formátovací řetězec pro `scanf`, (proč je to zrovna `scanf` zjistíme za chvíli, zatím si vystačme s tím, že to víme, protože vidíme zdrojový kód).

Volání `callq` pak na zásobník přidá adresu následující instrukce a provede skok na zadanou adresu.

Na řádcích `6` a `8` pokračujeme dalším voláním funkce. Hodnotu již zmíněné lokální proměnné zapíšeme do `%edi` a zavoláme funkci `foo`.

Posledním úkonem je úklid registrů a návrta z funkce (řádky `8` a `9`). Přičteme 8 k `%rsp` (na začátku jsme odečetli) a protože jsme jinak měnili pouze scratch registry, můžeme se navrátit z funkce. Volání `retq` je jakýmsi opakem ke `call` – vyzvedne ze zásobníku návratovou adresu a skočí na ni.

Všimněme si, že v některých případech procesor používal 32-bitové registry. Konkrétně v místech, kde nebylo potřeba používat 64-bitové. Toto je z důvodu úspory místa potřebných pro instrukci, protože instrukce pracující se 64-bitových registrech jsou typicky delší (skoro všechny vyžadují buď REX prefix, nebo mají dlouhé immediate hodnoty). 32-bitové registry jsou prostou polovinou registrů 64-bitových a dají se různě zaměňovat, pozor ale na to, že při práci s 32-bitovým registrem se vynuluje horní polovina 64-bitového registru[4].

[4]Tedy nevinné `xchg %eax,%eax` vlastně znamená „prohoď `%eax` s `%eax` ... a smaž horní polovinu `%rax` “.

2.3.4 Externí funkce

V předchozím příkladě jsme volali funkce, u kterých jsme prohlásili, že to je `scanf`, resp. `printf`. Tyto funkce se ale nikde v našem kódu nevyskytují, jsou totiž součástí standardní knihovny `libc`. Jak tedy víme, kam skočit?

Když jsme se dívali na formát ELF, setkali jsme se s relokačními údaji, konkrétně relokačními symboly. To jsou symboly, které říkají, které externí funkce externí funkce si přejeme využívat. Dynamický linker si tyto symboly přečte a na předem určené místo v paměti (uložené v relokačních údajích) zapíše adresu, kde se funkce vyskytuje v našem paměťovém prostoru.

Překladač pak pro každou funkci vygeneruje tzv. *PLT stub*, což je funkce, která tuto hodnotu přečte a případně zavolá dynamický linker, aby načetl příslušnou knihovnu a až poté ji zavolá (a tedy dovoluje „líné načítání“ knihoven). Takový PLT stub může vypadat následovně následovně:

```
0000000000400460 <plt_lazyload>:
  400460: pushq  0x200b8a(%rip)
  400466: jmpq   *0x200b8c(%rip)
  40046c: nopl   0x0(%rax)

0000000000400470 <printf@plt>:
  400470: jmpq   *0x200b8a(%rip)
  400476: pushq  $0x0
  40047b: jmpq   400460 <plt_lazyload>
```

Vidíme, že funkce `printf@plt` je opravdu krátká. Okamžitě skočí na adresu uloženou v `0x200b8a(%rip)`, což je v sekci `.plt.got`. Tato sekce je obsahuje adresy funkcí z jiných dynamických knihoven a provádí se v ní relokace. Pokud se podíváme na hodnotu na této adrese před prvním voláním, je to `0x400476`, tedy další instrukce za naším skokem. Ta pak na zásobník přidá nulu a skočí na `plt_lazyload`, který již zařídí zavolání dynamického linkeru, napíše do `.plt.got` správnou hodnotu a rovnou danou funkci za nás zavolá.

Příští volání externí funkce půjde již skoro přímo, protože hned první `jmpq` skočí na žádanou funkci.

3. Uživatelská dokumentace

V této kapitole si předvedeme disassembler vypracovaný v rámci práci z pohledu uživatele.

3.1 Instalace

3.1.1 Závislosti

Následující knihovny jsou vyžadovány pro běh programu:

- python >= 2.7
- gtk+ >= 3.0

Pro překlad ze zdrojového kódu je navíc potřeba:

- GNU Make nebo kompatibilní
- GCC >= 4.0 (nebo jiný překladač C podporující normu GNU99)
- perl5
- SWIG >= 1.3

3.1.2 Distribuční balíčky

Balíčky pro distribuce Debian (Squeeze) a Gentoo lze je stáhnout ze stránky `http://idis.krakonos.org/downloads`. Při instalaci postupujte obvyklým způsobem pro danou distribuci.

3.1.3 Instalace ze zdrojového kódu

Zdrojový kód je na přiloženém CD. Budoucí verze na adrese projektu, konkrétně `http://idis.krakonos.org/downloads`. K dispozici je tarball a Gitový repozitář.

Po stažení a rozbalení zdrojového kódu zkompilujeme:

```
cd src
make
```

A nainstalujeme do adresáře `/usr/local`:

```
make PREFIX=/usr/local install
```

A to je vše!

3.2 Spuštění

K dispozici jsou dva soubory pro spuštění disassembleru. Jednodušší je řádkový disassembler `disasm`, který je určen spíše k automatickému testování. budeme dále spouštět program `idis`, kcož je grafické rozhraní disassembleru.

Programu `idis` můžeme nepovinně předat soubor, který má otevřít a jeho chování ovlivnit několika optiony. Ty se zpracovávají sekvenčně.

`-n` Nenačítat žádné pluginy.

`-m jmeno` Načíst plugin s daným jménem. Dodávané pluginy jsou v balíčku `plugins`, tedy například plugin `whatis` se zapíše jako `plugins.whatis`.

`-p cesta` Přidá cestu k modulům Pythonu. Cesta je přidána vždy jako první do seznamu cest, pluginy v tomto umístění tedy budou mít přednost.

O pluginech povíme více později.

3.3 Přehled možností

Nyní si ukážeme, jak disassembler ovládat a přistupovat k jeho funkcím. Nejdříve se ale podíváme, co všechno náš disassembler umí. Dále prozkoumáme nabízené možnosti blíže a popíšeme, jak je lze ovládat. Na konci této části pak zrekapitulujeme klávesové zkratky a textové příkazy.

- Vizualizace skoků pomocí názorných šipek, jednotlivé typy skoků rozlišené barvou.
- Automatické anotace volání funkcí a podmínek.
- Informace o instrukci či operandu. V případě vyčíslitelných operandů prozkoumá výchozí adresní prostor a vypreparuje číslo nebo řetězec, pokud nějaký nalezne.
- Přejmenování proměnných.
- Skriptování v Pythonu.
- Rychlá orientace programem pomocí vytváření vlastních značek.
- Textové komentáře k instrukcím.
- Vizuální značky na instrukcích.
- Vyhledávání podle adresy, symbolu, nebo fulltextově.
- Schování vícenásobných instrukcí `nop`.

3.3.1 Koncept ovládání

Než se pustíme dále, ujasněme si koncept ovládání:

Na obrazovce je *kurzor*, který můžeme posunovat po řádcích a tím označovat jednotlivé instrukce pro další příkazy. Kurzor posouváme klávesami (přehled později), případně klikem myši na část řádku instrukce, která nemá žádný jiný význam.

Kliknutím pravým tlačítkem myši na mnemoniku instrukce, nebo její operand vyvoláme *kontextové menu*. V něm můžeme zvolit akci, kterou chceme provést na daném objektu.

Dvouklikem na instrukci skoku se přesuneme na její cíl.

Z klávesnice pak můžeme program ovládat dvěmi způsoby: *klávesovými zkratkami* a *textovými příkazy*. Klávesové zkratky jsou jednotlivé klávesy nebo jejich kombinace, které okamžitě provedou nějakou akci – například posun kurzoru, položení značky atp.

Textové příkazy používáme, když potřebujeme dodat nějaký parametr (a nebo nechceme dovolit příkaz spustit nahodile – například uzavření souboru). Pokud budeme mluvit o textovém příkazu `X`, pro jeho zadání je potřeba zadat `:X` a případně pokračovat mezerou a zapsat parametry. Seznam dostupných příkazů nalezneme v části 3.3.7.

Speciálním případem textových příkazů je vyhledávání, které inicializujeme stiskem `/`. O vyhledávání se ještě zmíníme.

3.3.2 Orientace

Orientace programem je při práci velmi důležitá, protože assembler má tendenci být velmi zdlouhavý. Disassembler poskytuje několik možností, jak v kódu orientovat a pohybovat:

- Posunem o řádek, stránku: klávesami `j, k, J, K, PgUp, PgDown`.
- Dvouklikem na instrukci skoku: tím se přesuneme na cíl skoku a pohled se nastaví tak, aby cílová instrukce byla na místě původního skoku.
- Textovým příkazem `s`, který vyhledá symbol a přesune pohled na něj.
- Značkami: pokud se chceme na nějaké místo později vrátit, stačí tam položit značku klávesou `mX`, kde za `X` můžeme dosadit libovolný tisknutelný znak. Zpět na danou značku se dostaneme kombinací kláves `'X`.
- Vyhledáváním regulárního výrazu: po stisku `/` můžeme zadat regulární výraz, který bude použit k prohledání zobrazovaných dat (tak, jak jsou zobrazena). Mezi výskyty se pak můžeme pohybovat pomocí `n` a `p`.
- Přechodem na adresu: pokud vstoupíme do příkazového módu klávesou `:` a zadáme suffix adresy, disassembler nalezne všechny instrukce začínající na takovémto suffixu, přesune kurzor na první a dále se chová jako vyhledávání nad touto množinou. Samozřejmě takto nelze vyhledávat suffixy, které jsou zároveň příkazy, takových je ale minimum.

3.3.3 Vizualizace skoků

Každá instrukce skoku, u které známe její cíl, má pozici svého cíle znázorněnu šipkou nahoru nebo dolů. Pokud je nad instrukcí navíc kurzor myši, je šipka prodloužena až ke svému cíli (nebo mimo pohled, pokud není cíl vidět).

Barvy šipek odpovídají typu skoku: červená je pro nepodmíněné skoky a modrá pro podmíněné.

Zelené šipky vždy vedou do cíle skoku. Protože jedna instrukce (typicky například začátek funkce) může být cílem mnoha skoků, mají všechny cíle stejnou barvu bez ohledu zda jsou podmíněné nebo ne. Pokud je nad instrukcí, která je cílem skoku kurzor myši, znázorní se všechny viditelné instrukce, které tuto mají jako svůj cíl.

Jak vypadají šipky skoků je vidět na obrázcích 3.1 a 3.2 na konci této kapitoly.

3.3.4 Interpretace podmínek

Disassembler umí do jisté míry interpretovat podmíněné skoky. Pokud disassembler nalezne podmíněný skok, prozkoumá, zda se před ním vyskytuje operace nastavující registr `FLAGS`, kterou by uměl interpretovat (například `cmp`, `test`). Pokud tomu tak je, anotuje instrukci jako podmínku a porovnání přeloží jako porovnávací znaménko (`>`, `>=`, `<`, `<=`, `==` nebo `!=`).

Pokud je navíc operandem registr (alespoň jeden typicky registr je), prohledá předcházející instrukce, aby zjistil obsah daného registru. Pokud nalezne zápis do žádaného registru a je to opět registr, pokračuje s novým registrem. Zastaví se tehdy, když narazí na instrukci skoku, cíl skoku a nebo zápis něčeho, co není registr (typicky konstanta, nepřímé adresování). Získanou hodnotu poté použije při anotaci. Pokud nenalzene nic, použije registr nalezený v porovnávající instrukci.

3.3.5 Volací konvence

Další funkcí disassembleru je rozpoznávání volacích konvencí. Z kapitoly o ABI víme, že se argumenty instrukce předávají v registrech. Disassembler se pro každou instrukci `call` pokusí nalézt její parametry.

Pozorvání, ze kterého v tomto postupu užíváme, je, že překladač typicky před voláním funkce přesouvá její argumenty na příslušná místa. Ačkoli tedy neznáme prototyp volané funkce, můžeme odhadnout, jaké má argumenty.

Disassembler pro registry určené k předávání argumentů nalezne zápisy do nich (pokud jsou) a dohledá jejich hodnoty stejným způsobem, jako v případě interpretace podmínek. Poslední registr s argumentem, do kterého nalezl zápis, dále považuje za poslední argument a zaznamená příslušnou anotaci.

Varování: toto je pouze heuristika a zdaleka nefunguje vždy (příklady na konci kapitoly demonstrují i příklad, kdy selhala). Výsledky této anotace je tedy nutné brát s rezervou.

3.3.6 Kontextové menu

Kontextové menu je způsob, jakým dovolit uživateli snadno podnikat akce na menším celku, než je celá instrukce.

Je vyvoláno kliknutím pravého tlačítka myši na operand nebo mnemoniku instrukce a bez dalších pluginů obsahuje dvě položky – `What is it?` a `Rename`. Obě si později představíme na příkladu, nyní si ale řekneme co konkrétně dělají.

What is it?

Tato volba vypíše obecné informace o operandu v závislosti na tom, co je o něm možné zjistit. V obecném případě vypíše pouze jeho zápis v infixovém tvaru.

Užitečnějším se ukáže zejména ve chvílích, kdy máme jako operand hodnotu, kterou lze vyčíslit. V takovém případě hodnotu vypíše a ověří, zda se nenachází v staticky inicializované paměti – tak tomu je například pokud jde o adresu funkce, statický řetězec atp.

Pokud pro danou adresu nějaká data nalezne, vypíše je. Přičemž umí zdetekovat, zda se jedná o řetězec (prohledá konstantně mnoho bytů a odpoví, zda je to řetězec, pokud se skládá z tisknutelných a bílých znaků, zakončených nulou). V opačném případě hodnotu považuje za 64-bitové znaménkové číslo.

V případě, že byla označena instrukce, vypíše její základní mnemoniku a její krátký slovní popis.

Rename

Tato volba je dostupné pouze u operandů instrukcí. Zeptá se uživatele na řetězec, kterým se má hodnota nahradit. Ten bude nadále používán v operandech a některých pluginech namísto výrazu, který byl přejmenován.

Toto přejmenování je pouze vizuální, instrukci se operand samozřejmě nemění – můžeme tedy přejmenovávat na libovolná jména (i duplicitní).

Protože občas je dobré vidět původní výraz, je k dispozici klávesová zkratka `A`, která zapíná a vypíná zobrazování přejmenovaných operandů.

3.3.7 Textové příkazy

Po stisku klávesy `:` se dostaneme do příkazového módu. Příkazy píšeme hned za znak dvojtečky a parametry následují za příkazem oddělené mezerou. Příkazový mód můžeme ukončit stiskem klávesy `Enter` nebo `Esc` pro vykonání nebo zrušení příkazu.

Pokud neexistuje příkaz, je vložený text prozkoumán, zda neodpovídá suffixu nějaké virtuální adresy. V takovém případě program skočí na danou virtuální adresu.

Následuje seznam vestavěných příkazů:

`w soubor` Uloží stav práce do daného souboru.

`o soubor` Otevře daný soubor. Může to být buď serializovaný soubor, nebo soubor ELF. V druhém případě bude disassemblován. Pokud je otevřen již jiný soubor, je potřeba ho ručně zavřít.

`c` Zavře soubor.

`s symbol` Skočí na symbol s daným jménem.

`d soubor` Vypíše současný pohled na program do daného souboru, tak jak je vidět na obrazovce (nicméně vypíše program celý, ne jenom úsek).

Další příkazy mohou být přidávány pomocí pluginů.

3.3.8 Klávesové zkratky

Velká část ovládání probíhá pomocí klávesových zkratech. Výše jsme zmínili, jaké to jsou, zde je tedy pouze shrneme na jedno místo:

`j, k`	Posun kurzoru nahoru/dolů.
`J, K`	Posun pohledu nahoru/dolů.
`PgUp, PgDown`	Posun pohledu o stránku nahoru/dolů.
`H, M, L`	Skok kurzoru na první, prostřední a poslední řádku pohledu.
`n, p`	Další/předchozí položka vyhledávání.
`A`	Přepnutí viditelnost aliasů.
`B`	Přepnutí viditelnosti bytů instrukcí.
`v`	Vizuální značka na instrukci.
`C`	Smazání vizuálních značek ze všech instrukcí.
`c`	Přidání/změna uživatelského komentáře k instrukci.
`mX`	Nastaví značku `X` na daný stav zobraných instrukcí. `X` může být libovolná (tisknutelná) klávesa.
`'X`	Vrátí se na značku `X`. `X` může být libovolná značka nastavená pomocí příkazu `m`.
`/`	Začátek vyhledávání.
`:`	Začátek příkazového módu.

Další klávesové zkratky mohou být definovány v programu pomocí pluginů.

3.4 Pluginy

Již jsme se několikrát zmínili o tzv. *pluginech*. Je na čase si o nich něco povědět.

Plugin je modul v Pythonu, který je načten disassemblerem po spuštění a při různých událostech je zavolán, aby provedl nějakou akci. Kromě předdefinovaných akcí si mohou také zaregistrovat události, které si přejí přijímat – například volbu v kontextovém menu, klávesovou zkratku atp.

Mnoho zde popisovaných funkcí je dostupných pomocí pluginů – záměrně jsme to ale nerozlišovali, protože pro uživatele jsou pluginy dostupné transparentně.

O tom, jak pluginy fungují a jak napsat vlastní, povíme v kapitole 4.

3.5 Grafické rozhraní v příkladech

Již jsme si představili možnosti disassembleru, nyní se ještě podíváme na dva jednoduché příklady, jak vypadá a jak ho využít v praxi.

Na obrázku 3.1 vidíme typické rozhraní programu, které vás uvítá po otevření souboru. Co na něm vidíme?

- Kurzor: tmavý první řádek. S ním se můžeme po programu pohybovat. Některé příkazy se spouští právě na instrukci s kurzorem, svou roli také má při skocích mezi instrukcemi (o tom si ještě povíme).
- Adresy, instrukce, symboly: Všechny informace, na jaké jsme zvyklí z odstatních disassemblerů. Podoba je inspirována známým `objdump`.
- Komentáře: Napravo od instrukcí je vidět několik anotací od pluginů. Co se v nich přesně objevuje zjistíme později.
- Stavový a příkazový řádek: Na spodním okraji okna je stavový řádek. Na obrázku zobrazuje informace o adrese v paměti, ale také v něm editujeme psaný příkaz.
- Šipky skoků: Vlevo od instrukcí je několik šipek znázorňující skoky, v našem případě volání funkcí. Na instrukci `call`, kde je prodloužená šipka ke svému cíli, je umístěn ukazatel myši.

Podívejme tedy, co můžeme podniknout dále.

3.5.1 Volací konvence

Příklad, který je na obrázku 3.1 nám může přijít povědomý. Není to náhoda, jde o stejný program jako ukázkový kód z obrázku 2.6 na stránce 19, na kterém jsme si ukazovali volací konvence.

Můžeme se podívat, že za nás disassembler udělal kus práce. Na první pohled vidíme, že první `call` volá `scanf` s dvěmi argumenty. První je konstanta. Otevřením kontextového menu a zvolením `What is it?` se okamžitě dozvíme, že jde o adresu v sekci `.rodata` a dokonce i její hodnotu.

Hned druhé volání je také snadné a na první pohled korektní. Dvouklikem na druhý `call` nás disassembler přenese na začátek funkce `foo` (což bychom ocenili více, kdyby byla funkce `foo` daleko).

Ve funkci `foo` pak dojde k podobnému dosazení argumentů. Všimneme si ale, že poslední argument funkci `printf` je označen jako `%edx` – což je registr, který je podle ABI pro třetí argument určen. Důvod je ten, že byl předán instrukcí `mov %edi, %edx`. Psát ale `%edi` ve volání funkce by působilo podezřele, disassembler proto napíše prostě registr, ve kterém se daný argument běžně předává a nechá to tak být.

```
00000000004005c4 <foo>:
  4005c4:       sub    $0x8, %rsp
  4005c8:       mov    %edi, %edx
  4005ca:       mov    $0x4006fc, %esi
  4005cf:       mov    $0x1, %edi
  4005d4:       mov    $0x0, %eax
  4005d9:       callq  4004c0    <__printf_chk@plt>    # __printf_chk@plt($0x1, $0x4006fc, %edx)
  4005de:       add    $0x8, %rsp
  4005e2:       retq

00000000004005e3 <main>:
  4005e3:       sub    $0x18, %rsp
  4005e7:       lea    0xc(%rsp), %rsi
  4005ec:       mov    $0x400700, %edi
  4005f1:       mov    $0x0, %eax
  4005f6:       callq  4004d0    <__isoc99_scanf@plt>  # __isoc99_scanf@plt($0x400700, %rsp + $0xc)
  4005fb:       mov    0xc(%rsp), %edi
  4005ff:       callq  4005c4    <foo>                 # foo([%rsp + $0xc])
  400604:       add    $0x18, %rsp
  400608:       retq
  400609:       nop
  40060a:       nop
  40060b:       nop
  40060c:       nop
  40060d:       nop
  40060e:       nop
  40060f:       nop

0000000000400610 <__libc_csu_init>:
  400610:       mov    %rbp, -0x28(%rsp)
  400615:       mov    %r12, -0x20(%rsp)
  40061a:       mov    %r13, -0x18(%rsp)
  40061f:       mov    %r14, -0x10(%rsp)
  400624:       mov    %r15, -0x8(%rsp)
  400629:       mov    %rbx, -0x30(%rsp)
```

Obrázek 3.1: Ukázka rozhraní programu

3.5.2 Přejmenování proměnných, podmínky, cykly

Dalším příkladem je možnost přejmenování proměnných. Na obrázku 3.2 jsou dva obrázky, jeden před a jeden po přejmenování (přejmenování provedeme zvolením `Rename` v kontextovém menu instrukce. Zapínat a vypínat přejmenování lze klávesou `A` (přejmenování se tím ale vypne pouze dočasně, záznam se samozřejmě neztratí).

Co dělá tato funkce? Na zásobníku si naalokuje dvě lokální proměnné a jejich adresy zapíše do `%rbx` a `%rbp`. Poté volá `scanf` s oběmi adresami jako argumenty. Pohledem na formátovací řetězec bychom zjistili, že jde o `%i %i`. Načte tedy dvě celá čísla ze standardního vstupu. Poté jedno z nich přesune do `%edx` a porovná s druhým. Tutu skutečnost také jednoduše vidíme v komentáři podmíněného skoku, který se provede pouze, pokud `A < B`.

Zvýrazněním šipky našeho podmíněného skoku pak snadno zjistíme, že se vrací do úvodní části a skok opakuje. Funkce tedy skončí, pokud je `A >= B` a něco vypíše. Pohledem na řetězec zjistíme, že má v sobě formátovací řetězec – kam se ztratil argument? Při porovnávájí jsme `A` přesunuli do `%edx`. Žádná instrukce ho nepřepsala a tak tam zůstal. Disassembler to ale neodhalil, protože instrukcí `jl` končí basic blok a disassembler se za hranice basic bloků v současné verzi nevydává.

```
0000000004005c4 <main>:
  4005c4:       push   %rbp
  4005c5:       push   %rbx
  4005c6:       sub    $0x18, %rsp
  4005ca:       lea    0x8(%rsp), %rbx
  4005cf:       lea    0xc(%rsp), %rbp
  4005d4:       mov    %rbx, %rdx
  4005d7:       mov    %rbp, %rsi
  4005da:       mov    $0x4006fc, %edi
  4005df:       mov    $0x0, %eax
  4005e4:       callq  4004d0    <__isoc99_scanf@plt>   # __isoc99_scanf@plt($0x4006fc, %rsi, %rdx)
  4005e9:       mov    0xc(%rsp), %edx
  4005ed:       cmp    0x8(%rsp), %edx
  4005f1:       jl     4005d4    <main+0x10>            # if ([%rsp + $0xc] < [%rsp + $0x8])
  4005f3:       mov    $0x4006ff, %esi
  4005f8:       mov    $0x1, %edi
  4005fd:       mov    $0x0, %eax
  400602:       callq  4004c0    <__printf_chk@plt>     # __printf_chk@plt($0x1, $0x4006ff)
  400607:       add    $0x18, %rsp
  40060b:       pop    %rbx
  40060c:       pop    %rbp
  40060d:       retq
  40060e:       nop
  40060f:       nop
```

```
0000000004005c4 <main>:
  4005c4:       push   %rbp
  4005c5:       push   %rbx
  4005c6:       sub    $0x18, %rsp
  4005ca:       lea    B, %rbx
  4005cf:       lea    A, %rbp
  4005d4:       mov    %rbx, %rdx
  4005d7:       mov    %rbp, %rsi
  4005da:       mov    $0x4006fc, %edi
  4005df:       mov    $0x0, %eax
  4005e4:       callq  4004d0    <__isoc99_scanf@plt>   # __isoc99_scanf@plt($0x4006fc, %rsi, %rdx)
  4005e9:       mov    A, %edx
  4005ed:       cmp    B, %edx
  4005f1:       jl     4005d4    <main+0x10>            # if (A < B)
  4005f3:       mov    $0x4006ff, %esi
  4005f8:       mov    $0x1, %edi
  4005fd:       mov    $0x0, %eax
  400602:       callq  4004c0    <__printf_chk@plt>     # __printf_chk@plt($0x1, $0x4006ff)
  400607:       add    $0x18, %rsp
  40060b:       pop    %rbx
  40060c:       pop    %rbp
  40060d:       retq
  40060e:       nop
  40060f:       nop
```

Obrázek 3.2: Ukázka přejmenování a podmínky

4. Skriptování v Pythonu

Jednou z nejdůležitějších částí disassembleru je možnost psát pluginy v Pythonu. V této kapitole se podíváme nad čím plugin pracuje, jaké náležitosti musí splňovat a jak může interagovat s disassemblerem a uživatelem.

4.1 Program z pohledu pluginu

Plugin manipuluje nad sadou objektů reprezentujících jednotlivé objekty programu. Jednotlivé objekty jsou podrobně popsané v dodatku A, zde si ukážeme jenom přehled.

Disassemblovaný program je reprezentován objektem `program`. Obsahuje jméno souboru a seznamy sekcí, symbolů a basic bloků (co to je za chvíli). Jednotlivé sekce pak dále obsahují instrukce a informace o jejich uložení v paměti.

Kromě instrukcí a symbolů, které již důvěrně známe, jsme zmínili *basic bloky*. To jsou skupiny instrukcí, které jsou vždy vykonány všechny, nebo žádná (tedy jediná poslední instrukce může být instrukce skoku a pouze první instrukce může být cíl skoku). Mezi jednotlivými bloky je postaven graf podle známých skoků.

Tato abstrakce se běžně využívá v překladačích, je tedy přirozené pokusit se ji zrekonstruovat. Bohužel to není jednoduché – při překladu se totiž ztratí některé informace o struktuře programu. Pokud by se pak do programu vloudil skok na adresu, kterou je potřeba vypočítat nebo načíst z paměti (dále jen *vypočítaný skok*), neumíme staticky určit jeho cíl.

Naštěstí většina programů vypočítané skoky téměř neobsahuje a tak nám to v praxi nevadí (často je potkáme například v kódu před spuštěním funkce `main`, o tento kód však máme zájem málokdy).

Disassembler poskytuje vypočítané basic bloky s tím, že cíle vypočítaných skoků ignoruje.

4.2 Formát pluginu

Nyní se již podíváme, jak takový plugin vytvořit a jak z něj přistupovat k datům o programu.

Plugin může být jakýkoliv Pythonový modul, tedy i prázný soubor. Aby mohl interagovat s disassemblerem, musí mít alespoň jeden *hook*. Dobrým zvykem je také importovat modul `idis`, který zpřístupní další funkce a konstanty.

Plugin musí být v adresáři, kde disassembler pluginy hledá. Pokud není při spuštění řečeno jinak, bývá to typicky adresář `$HOME/.idis/plugins`.

Hook je obyčejná funkce, která je zavolána při určité pevně dané události (jaké to jsou si řekneme za chvíli). Jakmile nastane událost, program projde všechny moduly (v pořadí v jakém byly nahrány) a pro každý plugin spustí příslušný hook.

Plugin má k dispozici tyto hooky:

`init_hook()`
: Volán při nahrávání pluginu.

`disassemble_hook( program )`
: Volán těsně po disassemblování programu.

`instr_hook( program, section, instr )`
: Volán pro každou instrukci. Proměnné `program` a `section` odpovídají programu a sekci, v jakých se instrukce nachází.

`bblock_hook( program, bblock )`
: Volán pro každý basic blok v programu.

`finalize_hook( program )`
: Volán jako poslední hook při disassemblování.

`serialize_hook( program )`
: Volán při uložení stavu. Zde může plugin uložit svá data pro příští spuštění.

`deserialize_hook( program )`
: Volán při načítení stavu. Zde program může načíst data, která si dříve uložil.

Například tedy plugin vypíše při inicializaci text „Ahoj, tady plugin!“ a pro každý načtený soubor jeho jméno:

```
import idis

def init_hook():
    print "Ahoj, tady plugin!";

def disassemble_hook( program ):
    print "Jméno programu: %s" % program.name;
```

Jak vidíme, tak objekt `program` má položku `name` se jménem načteného souboru. Jaké konkrétní objekty máme a co v nich najít zjistíme v Appendixu A: Python API.

4.3 Komunikace s uživatelem

Část komunikace s uživatelem probíhá pomocí výše zmíněných hooků, ty jsou ale pro efektivní komunikaci většinou příliš zbytečné. Zde popíšeme možnosti jak uživateli zobrazit výstup a jak od něj získat vstup.

Některé prvky používají tzv. *callbacky*, tedy funkce, které si plugin zaregistruje a budou zavolány, když dojde k žádané události.

4.3.1 Komentáře

Běžný úkon pluginu je sdělovat informace uživateli. Nejlepší způsob jak to udělat, je přidat komentář k instrukci, které se to týká. Slouží k tomu funkce:

```
instr.add_comment( str )
```

Ta zkopíruje daný řetězec a poznamená ho k dané instrukci jako komentář.

4.3.2 Dynamické anotace

Někdy obyčejný komentář nestačí, protože chceme anotaci měnit v závislosti na akcích uživatele (například přidáním aliasu chceme, aby se příslušná hodnota změnila i v komentáři).

Pro takový případ lze k instrukci zaregistrovat anotační callback. Ten pak bude zavolán kdykoli je potřeba data aktualizovat (což není často, data se cachují).

Pro registraci anotačního callbacku slouží funkce:

```
instr.add_annotate_hook( callback, data )
```

Kde `instr` je objekt instrukce a `data` jsou data, která budou funkci předána při zavolání. Callback musí být kompatibilní s následujícím prototypem:

```
annotate_callback( program, instr, data )
```

4.3.3 Zobrazování zpráv uživateli

Někdy je žádoucí uživateli pouze jednorázově sdělit informaci, o kterou požádal. V takovém případě není vhodné přidávat komentáře k jednotlivým instrukcím. Modul `gidis` má pro tento účel následující dvě funkce:

```
gidis.show_message( str )

gidis.show_error( str )
```

Obě zobrazí zprávu uživateli (na místě, kde bývá příkazový řádek), druhá zmíněná ji navíc zformátuje jako chybu (typicky tučně a červeně).

Zpráva je obyčejný řetězec a může obsahovat znaky `\n` a tím zabrat více řádků.

4.3.4 Zvýraznění instrukcí

Instrukce mouhou být zvýrazněny pomocí barvy pozadí. Plugin může zapsat do položky `instr.style` celé číslo, které určuje použitý styl. Disassembler podle něj vybere barvu z předpřipravené palety. Toto číslo může být kdykoliv smazáno grafickým prostředím.

4.3.5 Příkazový řádek

Plugin si může registrovat vlastní příkazy ve vestavěné příkazovém řádku. Lze to učinit pomocí funkce

```
gidis.add_command_hook( prikaz, callback )
```

Přičemž argument `prikaz` je řetězec se jménem příkazu. Callback musí být kompatibilní s následujícím prototypem a v argumentu `args` získá řetězec obsahující celý text po úvodním příkazu:

```
command_hook( program, instr, args )
```

Příkazový řádek však může být také vyvolán zevnitř programu. Následující funkce vyvolá editor příkazového řádku a přednastaví řetězec `text`. Po zavolání této funkce je potřeba vrátit kontrolu zpět disassembleru a počkat, než bude zavolán příslušný callback pro vložený příkaz (předpokládá se, že přednastavený text bude předvyplněný příkaz). Plugin by měl počítat s tím, že se uživatel může rozhodnout vstup nezadat.

```
gidis.editor_start_text( text )
```

4.3.6 Klávesové zkratky

Taktéž je možné registrovat obyčejné klávesové zkratky funkcí

```
gidis.add_hotkey_hook( key, callback )
```

Přičemž argument `key` definuje klávesu. Je to vždy řetězec obsahující jeden nebo více znaků. V případě jednoho znaku je tento znak považován za klávesu. V případě kombinace kláves se skládá řetězec z více znaků: poslední je považován za alfanumerickou klávesu, všechny předchozí mají speciální význam:

C	Ctrl
A	Alt
S	Shift

Znak mínus (`-`) je ignorován na všech místech, kromě posledního, a může být použi jako oddělovač. Tedy `C-A-A` znamená `Ctrl + Alt + 'A'` a `C--` znamená `Ctrl + '-'`.

Prototyp callbacku je:

```
hotkey_hook( program, instr )
```

4.3.7 Kontextové menu

Další jsou události z kontextového menu. Existují dva druhy těchto událostí: akce na mnemonice instrukce a akce na operandu. Obě události se registrují funkcí

```
gidis.add_context_hook( jmeno, typ, funkce )
```

Rozdíl je v tom, jaký se předá typ – možné hodnoty jsou `CMT_MNEMONIC` nebo `CMT_OPERAND`. Callbacky poté musí být kompatibilní s příslušným prototypem:

```
context_mnem_hook( program, instr )
context_operand_hook( program, instr, op )
```

4.3.8 Aliasy

Plugin má možnost ovlivňovat a používat tabulku aliasů (pro uživatele viděné jako přejmenování). Ty se vždy aplikují na výrazy, tj. objekty `expr`. Přístupné jsou tyto funkce:

```
gidis.alias_new( jmeno, expr )
gidis.alias_lookup( expr )
gidis.strexpr( expr )
```

První dvě zmíněné funkce přímo manipulují tabulku aliasů vložením nového aliasu a vyhledáním aliasu. Poslední zmíněný slouží pro pohodlné vyhledání aliasu a převedení do textové podoby.

Po volání těchto funkcí je zapotřebí obnovit vnitřní buffery.

4.3.9 Obnovení bufferů

Protože se všechny anotace cachují, je potřeba při jejich změně obnovit buffery. Někdy se toto děje automaticky – například, když je zavolána dynamická anotace, pokud k této změně dojde někdy jindy, je potřeba manuálně zavolat funkci:

```
gidis.rebuild_linebuf()
```

Ta zařídí vše potřebné.

4.4 Uložení stavu

V situaci, kdy je zapotřebí uložení současného stavu do souboru, je zavolán `serialize_hook`. V tomto hooku by měl plugin uložit všechna data, která není schopen získat bez vstupu od uživatele.

Slouží k tomu sada funkcí `seri_*`, ty jsou ale náchylné na chyby a není doporučené je používat. Modul `gidis` poskytuje funkci `serialize( data )`, jejíž argument je hash nebo list k serializaci, rekurzivně ho projde a zapíše do souboru.

Opačná funkce funkce `deserialize()` naopak data načte zpět do hashe nebo listu a vrátí ho pluginu, samozřejmě musí být volána v hooku `deserialize_hook`.

Do souboru lze uložit více objektů, každému volání `serialize` pak musí odpovídat jedno volání `deserialize`.

5. Programátorská dokumentace

Tato kapitola je seznámí s některými funkcemi vypracování programu, neslouží však jako úplná reference, ani jako výčet funkcí a struktur (potřebnou dokumentaci k funkcím a strukturám lze získat v komentářích zdrojového kódu).

Nejprve nastíníme některé konvence, kterými se kód programu řídí, stručně představíme kompilační mechanismus a rozdělení zdrojového kódu. Poté se podrobněji podíváme na formát databáze instrukcí, se kterou pracujeme, a popíšeme formát serializovaného souboru. Poslední část této kapitolu věnujeme představení programu SWIG, který byl použit pro generování rozhraní pro Python.

5.1 Konvence

Kód celého programu následuje několik konvencí, které je dobré znát:

- Názvy funkcí, proměnných a komentáře v kódu jsou psány anglicky.
- Makra jsou velkými písmeny, kromě případů kdy obalují nějakou funkci. Výjimkou jsou debugovací makra, která se mimikují funkce a také se píší malými písmeny.
- Většina funkcí začíná prefixem `idis_` v jádře, případně `gidis_` v gui. U pomocných funkcí, jako například `strimmf`, tato konvence není dodržována, většinou z pragmatických důvodů (jejich volání se vyskytují prakticky pouze jako argumenty jiných funkcí a zbytečně by to prodloužilo kód).
- Pokud se předává string, u kterého se nepředpokládá, že by se měl dlouhodobě uchovávat, je předán ve statickém bufferu. Názvy těchto bufferů většinou začínají podtržítkem (například `char _strimmbuf[128];`) a pokud se používají někde jinde než v příslušné funkci, většinou to značí chybu.
- Funkce začínající podtržítkem jsou lokální pro daný soubor a nikdy by se neměly volat z jiných souborů. Na jejich efekt se často nedá spoléhat, neboť to jsou logicky oddělené celky z větší funkce a předpokládá se nějak přednastavené prostředí. Výjimkou jsou některé funkce, které jsou pro pohodlí používání obalené makrem. V tom případě samotná funkce začíná podtržítkem a makro má stejné jméno bez podtržítka. Příkladem takové funkce je `_idis_python_call()`, což je variadická funkce a makro `idis_python_call` zjednodušuje její volání tím, že spočítá počet argumentů.

5.2 Kompilace a závislosti

Abychom se mohli ponořit do vnitřností programu, je potřeba porozumět tomu, jak se sestavuje.

V adresáři se zdrojovým kódem (`src`, dále jen „kořenový adresář“) a všech jeho podadresářích, ve kterých je co kompilovat, se nachází jeden makefile. Ten je zodpovědný za sestavení zdrojového kódu v daném adresáři. V kořenovém adresáři

navíc nalezneme soubor `Makefile.inc`, který by měl být includován každým dalším makefilem, protože definuje proměnné potřebné pro správný překlad.

Protože jednotlivé soubory na sobě netriviálním způsobem závisí, existuje systém závislostí. V každém makefilu je tedy cíl `deps`, který vytvoří rekurzivně pro každý adresář soubor `Makefile.deps`, který nese informace o závislostním stromě. Tento soubor je vhodné přegenerovat po každé změně, která by mohla ovlivnit závislosti (tj. změny týkající se hlavičkových souborů).

5.3 Průlet zdrojovým kódem

Začněme velmi stručným úvodem do zdrojového kódu. Následujících několik odstavců slouží jako rychlé seznámení čtenáře se strukturou a neslouží jako reference.

Zdrojový kód se skoro celý, až na generátor syntaxe, který se nachází v `util/syntax-generator`, nachází v adresáři `src` a jeho podadresářích. Ze zajímavého kódu na hlavní úrovni není skoro nic. Pouze konfigurační API a vstupní body do programu.

Adresář `python` obsahuje všechen kód v Pythonu, který není generovaný – zejména tedy různé pluginy a pomocné moduly.

Adresář `core` obsahuje samotný disassembler, `gui` obsahuje grafickou část. Obě části generují vlastní Pythoní modul (`idis` a `gidis`).

Názvy souborů vysvětlují svůj obsah. V místech, kde není umístění některých funkcí nebo struktur jasné, je jejich umístění v hlavičkovém souboru poznamenáno jako komentář (například struktura `bblock` je v souboru `program.h`, abychom se vyhli cyklickým závislostem mezi hlavičkovými soubory).

Všechny funkce jsou okomentovány (stručně v hlavičkovém souboru, podrobně ve zdrojovém kódu), komplikovanější funkce mají okomentovány i jednotlivé kroky.

5.4 Databáze instrukcí

Jak již víme, instrukční sada AMD64 je komplikovaná a obsahuje mnoho navzájem různých instrukcí. Abychom mohli efektivně dekódovat instrukce, musíme mít při ruce databázi instrukcí. Ta nám vždy řekne, jaké argumenty očekávat a jak je dekódovat.

Pro naše účely je dobré také vědět další informace o instrukci, zejména modifikace příznaků, a možnost rozšiřitelnosti – například přidání aritmetického výrazu, který instrukce reprezentuje. Mezi volně dostupnými se nejvíce nabízí reference od Karla Lejsky [6], která podrobně dokumentuje většinu opkódů (neobsahuje jenom AVX a pár velmi exotických instrukcí, zejména z důvodu odlišného kódování). Pro účely této práce bylo uděleno povolení referenci využívat jako součást softwaru.

5.4.1 Struktura

Reference je XML soubor strukturovaný podle opkódů. Na obrázku 5.1 je zjednodušené DTD (mnoho technikalit je pro názornost vynecháno, kompletní DTD lze získat na webu[6]).

```
<!ELEMENT x86reference (one-byte, two-byte, gen_notes, ring_notes)>
<!ELEMENT one-byte (pri_opcd+)>
<!ELEMENT two-byte (pri_opcd+)>
<!ELEMENT pri_opcd (proc_start?, proc_end?, entry+)>
<!ELEMENT entry (
            opcd_ext?,
            pref?, sec_opcd?,
            proc_start?, proc_end?,
            syntax+,
            instr_ext?, grp1?, grp2*, grp3*,
            test_f?, modif_f?, def_f?, undef_f?, f_vals?,
            test_f_fpu?, modif_f_fpu?, def_f_fpu?, undef_f_fpu?,
            f_vals_fpu?, note?)>
```

Obrázek 5.1: Struktura databáze instrukcí

Na hlavní úrovni dokumentu je tag `x86reference`. Ten obsahuje tagy `one-byte` a `two-byte`, které vymezují jednobytové, resp. dvoubytové instrukce. Pozor, že vícebytové instrukce jsou kódovány jako dvoubytové pomocí speciálních polí (za chvíli).

Obě sekce dále postupují stejně – obsahují tagy `pri_opcd`, kde každý reprezentuje jeden opkód (ale teoreticky více instrukcí), jak ho chápe Intel. Pro každou instrukci pak máme jeden tag `entry` (těch může být pro jeden opkód více). Jednotlivá pole mají následující význam:

`opcd_ext` Rozšíření opkód v ModRM bytu (konkrétně reg části).

`pref` Prefix, který tato instrukce vyžaduje. Většinou se originální význam prefixu ignoruje.

`sec_opcd` Sekundární opkód. Toto je opkód jak ho chápe AMD. Pokud najdeme toto pole, element `pri_opcd`, který zrovna zpracováváme, je vlastně escape znak.

`proc_start` `proc_end` První a poslední procesory, ve kterých měl tento opkód daný význam.

`syntax` Popis mnemonické syntaxe (Intel).

`instr_ext` Rozšiřující sada instrukcí, ve které se poprvé tato instrukce objevila.

`grp1,grp2, grp3` Skupina (interní) instrukcí. Mimo jiné jsou to skupiny `arith`, `compar`, `branch`, `conditional` a další, které dávají rychlou představu o účelu instrukce (zejména pro program).

Dále pak pole `*_f` nesou informace o manipulacích s flagy v registru rFlags.

5.4.2 Zpracování databáze

Protože pro každou dekódovanou informaci číst XML soubor je nepraktické, je potřeba databázi přednačíst do paměti. To je možné udělat za běhu programu, ale není to příliš praktické, protože se pokaždé dělá stejná práce. My tedy budeme generovat paměťové struktury při překladu.

O generování struktur se stará skript napsaný v Perlu[1]. Jeho úkolem je předzpracovat instrukce, některé rovnou vyřadit (například ty, které nejsou platné v 64-bitovém módu) a ze zbytku vygenerovat rozumný struct, který bude snadno čitelný z C. Pro každou instrukci, kterou skript nalezne, udělá zhruba následující úkony:

1. Zjistí, zda je platná v 64bitovém módu a dostupná na těchto procesorech (některé instrukce byly nahrazeny a tak má daný opkód více významů, podle toho na jakém procesoru je vykonáván). Také přeskočí instrukce, které nechceme generovat (prefixy, x87fpu atp.).
2. Detekuje instrukce skoku.
3. Vybere mnemoniku (v případě více možných se spolehne na atribut `redundant`).
4. Zkoriguje operandy podle skupin (některé MMX/SSE instrukce nemají explicitně udanou velikost implicitních operandů).
5. Označí vstupní a výstupní argumenty.

Následně se ze získaných informací vytvoří pole instrukcí a uloží do souboru `instr_data.c`, který je includován z `core/instr.c`. Soubor do jisté míry kopíruje strukturu XML dokumentu – tedy obsahuje dvě pole `_opcds` a `_opcds0f`, ve kterých jsou pointery na pole instrukcí s daným opkódem. V těchto polích jsou pak již samotné definice instrukcí `struct instr_def`, které jsou znázorněny na obrázku 5.2. Jejich pole do značné míry kopírují elementy v XML dokumentu a význam by tedy měl být zřejmý. Jenom dodejme, že v souboru `instr_data.h` jsou jak definice těchto struktur, tak zadefinované některé konstanty, které se vyskytují v příslušných polích (zejména velikosti a typy operandů).

Na obrázku 5.2 je také struktura `op_def`, která nese informace o operandech instrukce. Její pole mají následující význam:

`addr` Adresování operandu. Možné hodnoty jsou definované v `instr_data.h` a popisují, kde hledat informace o operandu (ModRM, immediate a další). Význam kopíruje význam v XML referenci.

`type` Typ operandu. Možné hodnoty jsou definované v `instr_data.h` a popisují velikost v bytech, případně hodnotu operandu, včetně rozlišení o znaménkovém rozšíření.

`def` Konkrétní hodnota operandu v případě, že je pevně definovaný. Potom je `addr` roven buď `A_REG` (v tom případě se v `def` nachází číslo registru), nebo `A_def` a v tom případě je `type` roven `T_CONST` a v `def` je konstanta.

[1] V souboru `utils/syntax-generator/gen.pl`

```
typedef struct op_def {
    unsigned char addr;
    unsigned char type;
    unsigned char def;
    unsigned char flags;
} _op_def_t;

typedef struct instr_def {
    char *mnem;
    unsigned char prefix;
    unsigned char x0f;
    unsigned char opcd;
    unsigned char opcd2;
    unsigned char mask;
    unsigned char extension;
    unsigned int flags;
    struct op_def **ops;
} _instr_def_t;
```

Obrázek 5.2: Struktura definice instrukce a operandu

`flags` Příznaky argumentu. Ty mohou mít následující hodnoty:

- `OF_DEST` Tento operand je cílový a ponese výsledek operace.
- `OF_VDEST` Tento operand by byl cílový, ale nic do něj ve skutečnosti zapsáno nebude (například `cmp` se chová jako odečítání, ale nezapisuje výsledek).
- `OF_HIDDEN` Tento operand je implicitní a není zobrazen v mnemonice.

5.5 Serializační soubor

Protože někdy nelze práci na programu dokončit ihned, je praktické mít možnost uložit dosažené výsledky do souboru. Vytyčme si, jaké požadavky na takový soubor s metadaty budeme klást požadavky:

1. Textový formát čitelný člověku.
2. Kompatibilní mezi verzemi (v rámci možností).
3. Možnost ukládat data pluginům (skoro libovolná).
4. Musí obsahovat všechny informace důležité k obnovení stavu práce.

Je tedy vhodné zvolit nějaký obecný a dobře definovaný formát, který navíc bude umět snadno reprezentovat strukturovaná data. K tomu nám postačí několik přirozených struktur, jako skalární hodnoty (řetězce, čísla) a jednoduché kolekce – seznamy a hashe. To je zejména výhodné pro Python, ve kterém je ukládání dat v hashi velmi jednoduché a přirozené.

Jako implementaci jsem zvolil YAML (akronym pro *YAML Ain't Markup Language*). Jde o jednoduchý serializační jazyk se stabilní C knihovnou. Na obrázku 5.3 je jednoduchý soubor v YAML.

```
file: cond
module:
  handler: module
  name: plugins.comments
  data:
    4195825: ! '} while (...)'
    4195796: do {
module:
  handler: module
  name: ...
    ...
```

Obrázek 5.3: Ukázkový YAML

Přesněji se takovýto formát chová podle gramatiky 5.4 dokumentované v [5]. Čtení a zapisování souboru se podle ni řídí také. Při čtení se ve smyčce volá parser, který se vrátí pokud narazí na nějakou událost (v gramatice velkými písmeny). Při zápisu se naopak tyto události generují a posílají tzv. emitteru, který je validuje a generuje soubor.

```
stream ::= STREAM-START document* STREAM-END
document ::= DOCUMENT-START node DOCUMENT-END
node ::= ALIAS | SCALAR | sequence | mapping
sequence ::= SEQUENCE-START node* SEQUENCE-END
mapping ::= MAPPING-START (node node)* MAPPING-END
```

Obrázek 5.4: Gramatika YAML

5.6 SWIG

SWIG, *Simple Wrapper and Interface Generator* [9], je program, který umí generovat rozhraní mezi jazykem C a mnoha skriptovacími jazyky, mimo jiné pro Python. V projektu je použit ke generování rozhraní pro pluginy z existujících funkcí a struktur v jazyce C. V této části stručně povíme, jak SWIG funguje a jak se používá.

Dále budeme uvažovat, že pracujeme s jazykem C a Python.

5.6.1 Idea fungování

Tak jako program v jazyce C má zdrojový kód, SWIG má tzv. *interface file*. Ten definuje rozhraní, které chceme exportovat do Pythonu. Formát je velmi podobný hlavičkovým souborům, jak je známe z C, dokonce se do něj přímo dají

vložit direktivou preprocesoru, může ale obsahovat mnoho direktiv upravující jeho chování.

Při generování rozhraní SWIG načte soubor s definicí rozhraní, expanduje makra preprocesoru a pro všechny struktury a prototypy vygeneruje tzv. *wrappery*. SWIG vygeneruje dva soubory – soubor s wrappery v originálním jazyce (v našem případě C) a modul pro Python (dále jen „modul“), který tyto wrappery využívá. V praxi poté budeme používat právě tento modul.

Jednotlivé wrappery funkcí se postarají o konverzi argumentů z typů Pythonu do typů C, vloží volání originální funkce, a zkonvertuje návratové hodnoty zpět do typů Pythonu. V modulu je pak pouze zavolá.

Pro struktury se vytvářejí *proxy třídy*. Jsou to třídy v modulu, které nesou ukazatel na svou strukturu. Pro každou položku struktury pak SWIG vygeneruje wrappery pro její čtení a zápis. Proxy třída potom pomocí těchto funkcí emuluje instanci struktury.

V případě polí je situace komplikovanější. Statická pole SWIG umí přeložit snadno, protože o nich ví všechny potřebné informace při kompilaci. U dynamických polí (jako například naše pole instrukcí) je to těžší – protože pole může být v jazyce C reprezentováno různými způsoby. Je nutné SWIGu pomoci.

Existuje několik způsobů, jak to udělat. Většina těch od autorů ale myslí na předávání polí z Pythonu do C, ale ne naopak. Součástí projektu je sada maker v souboru `wrap_array.i`, která implementují náš model polí (tedy pole ukazatelů jako jeden prvek a délka jako prvek druhý).

5.6.2 Příklad

Následuje jednoduchý wrapper, který generuje modul `gidis.py` v disassembleru.

```
%module gidis

%rename("%(regex:/gidis_(.*)/\\1/)s") "";
%rename("%(regex:/gidis_view_(.*)/\\1/)s") "";

%include "context_menu.h"
%include "view.h"

%{
#define SWIG_FILE_WITH_INIT
#include "context_menu.h"
#include "view.h"
%}
```

Vidíme zde několik direktiv:

`%module` Jméno modulu, tato direktiva musí být vždy přítomna.

`%rename` Přejmenování funkcí, v našem případě je využíváme k odstranění prefixu `gidis_`, abychom nemuseli funkce volat příliš zdlouhavě `gidis.gidis_view_...`.

`%include` Vloží na místo direktivy obsah souboru a zpracuje ho. Narozdíl od direktivy `#include` preprocesoru C každý soubor načte nanejvýš jednou.

`%{ ... %}` Takto uvozený kód bude vložen do souboru s wrappery nezměněn, nebudou pro ně generovány ale žádné wrappery. Tady by měly být vloženy všechny hlavičkové soubory nebo definic, které jsou potřeba pro volání funkcí, pro které generujeme wrappery. Taktéž zde můžeme vložit další funkce, které chceme do modulu zakompilovat (to děláme například v `core/python.i`).

5.7 Debugging

Některé části programu nelze snadno ladit. Například při ladění dekodéru instrukcí se může zdát, že program funguje bezchybně, ale ve skutečnosti některé instrukce překládá jinak, než by měl – často si ale na první pohled nemusíme všimnout zaměněného `%rax` za `%eax` a podobných chyb.

V hlavním makefilu jsou dva cíle určené k ladění: `test` a `valgrind`.

`test` Slouží k otestování správnosti instrukcí. Spustí program v módu čistého disassembleru a porovná jeho výstup s `objdump -d`. Navíc dělá některé bezpečné substituce, jako drobné odchylky v mnemonice a ignoruje bílé znaky. Některé rozdíly v něm ale nejsou zaneseny a tedy rozdílný výstup nutně neznamená problém – stále je potřeba ověřit rozdíly ručně, ale jejich počet je drasticky zmenšen.

`valgrind` Slouží ke spuštění programu ve valgrindu. Jeho jedinou funkcí je zjednodušení typické spuštění disassembleru s potlačením vybraných varování, která jsou generována z externích knihoven (Python, GTK).

Kromě externích nástrojů jsou v kódu zabudované jednoduché mechanismy pro ladění. V souboru `core/debug.h` jsou nadefinována makra `debug` a `warn`, která vypíší předaný řetězec na chybový výstup, přičemž makro `debug` lze vypnout oddefinováním makra `DEBUG`, makro `warn` nikoliv. V témže souboru jsou nadefinovány i varianty `debugf` a `warnf`, kterým lze předat formátovací řetězec a doplňující argumenty. Tyto varianty se chovají jako `printf`, konkrétně `printf` volají bez jakýchkoliv ověřování argumentů.

Závěr

Zhodnocení

V práci bylo dosaženo většiny stanovených cílů. Výsledný disassembler je velmi užitečným pomocníkem při analýze programů.

Grafické prostředí je jednoduché a přehledné, pohodlně se ovládá pomocí kombinace klávesnice a myši. Vizualizace skoků a jejich automatické následování velmi urychluje práci a usnadňuje orientaci. Také možnost nechat si přeložit adresu relativní k instrukci (které se vyskytují velmi často) a zjistit, na jaké misto v paměti ukazuje, velmi urychluje orientaci v kódu.

Pluginy v Pythonu mají přístup ke kompletní sadě funkcí, které se v disassembleru objevují (kromě grafického rozhraní, které vystavuje jenom některé) a je tedy možné s instrukcemi pracovat téměř libovolně.

Jádro disassembleru je navíc samostatně použitelné jako dynamická knihovna a modul pro Python, je tedy možné pomocí něj skriptovat rozdílné úkoly automaticky.

Nápady na vylepšení a plány do budoucna

Jako v každém programu, i v našem disassembleru je mnoho prostoru pro vylepšování.

Jednou z největších problémů je, že jádro disassembleru nepodporuje práci s pokročilými instrukcemi řad SSE apod. Podpora těchto instrukcí je asi nejdůležitější úkol, který čeká v blízké budoucnosti.

Práce s některými objekty v Pythonu není tak přirozená, jak by mohla být. Toto je dáno zejména tím, že z velké části kopírují sémantiku struktur v jazyce C, které jsou navíc v případě instrukcí a operandů rozděleny na část instance a definice. Plán do budoucna je navrhnout rozhraní, které bude struktury exportovat přirozeněji a poskytovat všechny pohodlné funkce, které nyní musí programátor spouštět ručně (například vyhodnocování v kontextu, zjednodušování a uvolňování paměti).

Zobrazovací rozhraní by si také zasloužilo vylepšení. Jeho jednoduchá implementace se zprvu zdála dostačující, nicméně se ukázalo, že tomu tak není. V současnou dobu například rozhraní neumí poskytovat události nad objekty, které vložil plugin. Tento nedostatek je v blízké budoucnosti potřeba opravit.

Jedním z nápadů, který přišel v pozdějších fázích vývoje a nezbyl čas ho implementovat, je možnost přímější manipulace s instrukcemi – například jejich lokální prohazování. Tato možnost by umožnila snazší orientaci například v případech, kde byly dvě smyčky vloženy do sebe z důvodu optimalizace.

A. Python API

V této části uvádíme stručný přehled důležitých objektů a funkcí, které lze při psaní pluginů využít.

A.1 Objekt `program`

Objekt `program` reprezentuje disassemblovaný program. Obsahuje několik položek:

`file`	Jméno souboru.
`sections`	Seznam sekcí.
`symbs`	Seznam symbolů.
`bblocks`	Seznam basic bloků.

Dále má následující funkce:

`add_symbol()`
: Přidá nový symbol do programu a vrátí na něj referenci.

`lookup_symbol( adresa )`

`lookup_symbol_by_addr( adresa )`
: Vrátí referenci na symbol na dané adrese, případně `None`, pokud na dané adrese žádný není.

`lookup_symbol_nearest( adresa )`
: Vrátí referenci na nejbližší symbol na menší nebo rovné adrese, případně `None`, pokud takový symbol neexistuje.

`lookup_symbol_by_name( jmeno )`
: Vrátí referenci na symbol s daným jménem, nebo `None`, pokud žádný neexistuje.

`finalize_symbols()`
: Ukončí přidávání symbolů. Tuto funkci je nutno volat pokaždé, když dojde ke změně symbolů, protože je setřídí podle adres a případně aktualizuje další informace.

`lookup_section_by_name( jmeno )`
: Vrátí referenci na sekci s daným jménem, nebo `None`, pokud taková neexistuje.

`lookup_section_by_addr( adresa )`
: Vrátí referenci na sekci obsahující danou adresu, nebo `None`, pokud adresa není v žádné sekci.

`lookup_instr_by_addr( adresa )`

Vrátí referenci na instrukci začínající na dané adrese, nebo `None`, pokud taková neexistuje.

`assumption_getfor( instr, key_type, key )`

Vrátí referenci na objekt předpokladu pro danou instrukci. Hodnota `key_type` udává typ hodnoty v `key`. V současné době jsou platné pouze klíče typu `AK_REG`.

A.2 Objekt `section`

Objekt `section` popisuje sekci programu, odpovídající sekcím, jak je známe z formátu ELF. Jejich vztah k sekcím ale nyní zanedbáme a budeme se na ně většinou dívat jako na kontejnery pro intstrukce, ale dostupné jsou všechny nalezené sekce v souboru.

Najdeme v něm následující položky:

`name`	Jméno sekce..
`addr`	Virtuální adresa začátku sekce.
`data`	Pole bytů v sekci.
`data_size`	Velikost data v sekci.
`instrs`	Seznam instrukcí v sekci.
`prg`	Reference na objekt `program` pod který patří.

Vlastní funkce žádné nemá.

A.3 Objekt `bblock`

Objekt `bblock` je má následující parametry:

`firsti`	První instrukce basic bloku.
`lasti`	Poslední instrukce basic bloku.
`instrs`	Seznam instrukcí.
`instrn`	Počet instrukcí.
`vmaddr`	Virtuální adresa začátku basic bloku.
`jumpsto`	Spojový seznam objektů `bblock`, ve kterých může pokračovat běh programu.
`jumpedfrom`	Spojový seznam objektů `bblock`, ze kterých mohlo být skočeno na tento basic blok.

Spojové seznamy mají položky `bblock` a `next` se zjevným významem.

Basic bloky tedy tvoří graf možných běhů programu. Pozor ale, že tento graf se může lišit od skutečnosti, protože některé skoky nelze staticky předvídat (speciálně skoky, které závisí na hodnotě registru, která ale není známa) – tyto situace jsou ale vzácné a překladač je generuje pouze zřídka.

A.4 Objekt `instr`

Asi nejdůležitější objekt ze všech, `instr` reprezentuje instrukci programu. Každý objekt reprezentuje jednu instanci instrukce.

`mnem`	Mnemonika instrukce, malými písmeny bez suffixu.
`desc`	Krátký popis instrukce.
`def`	Reference na definici instrukce. Ta nese obecné informace o instrukci.
`raw`	Pole bytů instrukce.
`raw_length`	Délka instrukce v bytech.
`modrm`	ModRM byte (pokud je použit).
`sib`	SIB byte (pokud je použit).
`flags`	Příznaky instrukce jako bitové pole. Mohou být buď IF_MODRM a/nebo IF_SIB v závislosti na tom, zda je přítomen ModRM nebo SIB byte.
`prefixes`	Bitové pole značící přítomné prefixy. Konkrétní hodnoty za chvíli.
`vmaddr`	Virtuální adresa, na které se instrukce nachází.
`ops`	Seznam operandů instrukce.
`style`	Styl instrukce v grafickém prostředí. Toto pole obsahuje možnost zvýraznění instrukce pro pluginy.
`src`	Zdrojový operand.
`dst`	Cílový operand.

Konstanty pro prefixy jsou pak následující:

```
PF_REX    PF_FS        PF_REPZ
PF_REX_B  PF_GS        PF_REPNZ
PF_REX_X  PF_OPSIZE    PF_0F
PF_REX_W  PF_ADDRSIZE
PF_REX_R  PF_LOCK
```

Definice instrukce odkazovaná v poli `def` má následující formát:

`name`	Mnemonika instrukce, tak jak je uvedeno v XML souboru (o něm později). Velkým písmenem a bez suffixu.
`desc`	Krátký popis instrukce, tak jak je uvedeno v XML souboru.
`prefix`	Vyžadovaný prefix instrukce. Pozor, není to samé jako pole prefix objektu `instr`.
`x0f`	Pole značící potřebu přítmnosti prefixu `0F`.
`opcd`	Primární opkód instrukce.
`opcd2`	Sekundární opkód instrukce (pokud je vyžadován).
`mask`	Maska opkódu. Určuje, jaká část bytu opkódu je opkód a jaká operand, pokud je zakódován v opkódu.
`extension`	Rozšíření opkódu v ModRM bytu.
`flags`	Příznaky instrukce. V současné existují pouze příznaky skoku: `IDF_BRANCH`, `IDF_CONDITIONAL`.

Objekt `instr` má navíc následující funkce:

`add_userdata( ident, data )`

Přidá uživatelská data daného identifikátoru.

`add_comment( str )`

Přidá komentář k instrukci.

A.5 Objekt `op`

Objekt `op` definuje instanci operandu dané instrukce. Jeho struktura je následující:

`type`	Typ operandu (registr, literál, ...).
`val`	Hodnota operandu (číslo regisru, konstanta).
`disp`	Displacement (pokud je použit).
`scale`,`index`,`base`	Přeparsované hodnoty SIB bytu, pokud je tento operand definovaný SIB bytem.
`expr`	Objekt `expr` odpovídající tomuto operandu.

Hodnoty pole `type` definují význam ostatních položek. Následující tabulka popisuje interpretaci:

`OT_REG`	Operand je registr, jeho číslo je v poli `val`.
`OT_IMMEDIATE`	Operand je immediate, jeho hodnota je v poli `val`.
`OT_DISPLACEMENT`	Operand je nepřímo adresován, pouze displacement. číslo je v `val`).
`OT_INDIRECT_DISP`	Operand je nepřímo adresován, je použit registr a displacement, hodnota registru je ve `val`
`OT_JUMP_REL`	Operand je cíl nepřímého skoku. Relativní offset je ve `val`.
`OT_STACK`	Operand je vrchol zásobníku.
`OT_SIB`	Operand je určen jako SIB, příslušné hodnoty jsou v `scale`, `index` a `base`.

A.6 Objekt `expr`

Objekt `expr` reprezentuje aritmetické výrazy. Každý z těchto objektů reprezentuje jeden n-ární operátor (jejich výčet později). Jeho struktura vypadá následovně:

`type`	Typ objektu nebo operátor.
`ops`	Operandy.
`val`	Hodnota (pouze v případě některých operátorů).

Typy operandů jsou následující:

`EX_REG`	Registr. Pole `val` obsahuje registr (jaké jsou hodnoty za chvíli).
`EX_CONST`	Konstanta. Pole `val` obsahuje její hodnotu.
`EX_DEREF`	Dereference operandu. Tento operátor smí být pouze unární.
`EX_PLUS`	Sčítání.
`EX_MINUS`	Odečítání. Druhý a všechny další operandy jsou odečteny od prvního.
`EX_MUL`	Násobení.
`EX_DIV`	Dělení. Druhý a všechny další operandy budou vynásobeny a první vydělen výsledkem.
`EX_ASSIGN`	Přiřazení. Tento operátor smí být pouze binární a přiřadí prvnímu operandu hodnotu druhého.

Modul `idis` pak má několik funkcí pro manipulaci s operandy:

`expr_new( opn )`
: Vrátí nový výraz s `opn` operandy.

`expr_new_value( val )`
: Vrátí nový výraz `EX_CONST` s hodnotou `val`

`expr_new_reg( reg )`
: Vrátí nový výraz `EX_REG` pro registr `reg`

`expr_new_unary( type, expr )`
: Vrátí nový unární výraz typu `type` a operandem `expr`.

`expr_new_binary( type, expr1, expr2 )`
: Vrátí nový binární výraz typu `type` a operandy `expr1` a `expr2`

`expr_free( expr )`
: Rekurzivně uvolní paměť používanou výrazem `expr` a všech jeho operandů. Tento výraz lze také volat na objektu metodou `expr.free()`.

`expr_enumerate( expr, program, instr )`
: Pokusí se vyčíslit výraz `expr` tak, jak by byl vyhodnocen v instrukci `instr` a programu `program`.

`expr_simplify( expr, program, instr )`
: Pokusí se zjednodušit výraz `expr` za předpokladu, že by se nacházel v instrukci `instr` a programu `program`. Vrací nový výraz, který musí být uvolněn metodou `free()`.

A.7 Konstanty

Některým funkcím je potřeba předávat konstanty, jako jsou čísla registrů. Uvádíme zde některé konstanty, vždy dostupné v modulu idis.

R_AL	R_CL	R_DL	R_BL	R_AH	R_CH	R_DH	R_BH
R_R8B	R_R9B	R_R10B	R_R11B	R_R12B	R_R13B	R_R14B	R_R15B
R_AX	R_CX	R_DX	R_BX	R_SP	R_BP	R_SI	R_DI
R_8D	R_9D	R_R10W	R_R11W	R_R12W	R_R13W	R_R14W	R_R15W
R_EAX	R_ECX	R_EDX	R_EBX	R_ESP	R_EBP	R_ESI	R_EDI
R_R8D	R_R9D	R_R10D	R_R11D	R_R12D	R_R13D	R_R14D	R_R15D
R_RAX	R_RCX	R_RDX	R_RBX	R_RSP	R_RBP	R_RSI	R_RDI
R_R8	R_R9	R_R10	R_R11	R_R12	R_R13	R_R14	R_R15
R_X_SPL	R_X_BPL	R_X_SIL	R_X_DIL	R_RIP	R_ZERO		

Registry `R_X_` jsou osmibitové registry dostupné pouze s REX prefixem a nulovým příslušným rozšiřujícím bitem. `R_RIP` představuje adresu instrukce. Registr `R_ZERO` je vždy nula (používá se v některých neúplných SIB adresacích).

B. Obsah přiloženého CD

`idis-1.0.tar.gz`	Kompletní zdrojové kódy programu.
`idis-1.0_amd64.deb`	Balíček pro Debian Squeeze
`idis-1.0.ebuild`	Balíček pro Gentoo Linux.
`prace.tar.gz`	Zdrojový text práce.
`prace.pdf`	Přeložená práce ve formátu PDF.

Seznam použité literatury

[1] SCO, *System V Application Binary Interface* [online]. Verze 4.1, aktualizace 18. 3. 1997. Dostupné z: `www.sco.com/developers/devspecs/gabi41.ps`. [cit. 23. 5. 2012]

[2] Matz, Michael, Hubička, Jan, Jaeger, Andreas, Mitchell, Mark. *System V Application Binary Interface, AMD64 Architecture Processor Supplement* [online]. Draft 0.99.6, aktualizace 15. 5. 2012. Dostupné z: `http://www.x86-64.org/documentation/abi-0.99.pdf`. [cit. 23. 5. 2012]

[3] AMD, *AMD64 Architecture Programmer's Manual Volume 3: General-Purpose and System Instructions* [online]. Aktualizace 13. 12. 2011. Dostupné z: `http://developer.amd.com/documentation/guides/`. [cit. 23. 5. 2012]

[4] Intel, *Intel 64 and IA-32 Architectures Software Developer's Manual Combined Volumes 2A, 2B, and 2C: Instruction Set Reference, A-Z* [online]. Aktualizace květen 2012. Dostupné z: `http://download.intel.com/products/processor/manual/325383.pdf` [cit. 23. 5. 2012]

[5] Ben-Kiki, Oren, Evans, Clark, Ingerson, Brian. *YAML Ain't Markup Language* [online]. Verze 1.1. Dostupné z: `http://yaml.org/spec/1.1/` [cit. 23. 5. 2012]

[6] Lejska, Karel. *X86 Opcode and Instruction Reference* [online]. Verze 1.11, aktualizace 20. 1. 2009. Dostupné z: `http://ref.x86asm.net/` [cit. 23. 5. 2012]

[7] Drepper, Ulrich, MCGRATH, Rolan, MACHATA, Petr. *Elfutils* [online]. Dostupné z: `https://fedorahosted.org/elfutils/`. [cit. 23. 5. 2012]

[8] Fog, Agner. *Software optimization resources* [online]. Aktualizace 29. 2. 2012. Dostupné z: `http://www.agner.org/optimize/`. [cit. 23. 5. 2012]

[9] Fulton William, aj. *SWIG* [online]. Aktualizace 30. 4. 2012. Dostupné z: `http://www.swig.org/`. [cit. 23. 5. 2012]

[10] Drepper, Ulrich. *How to write shared libraries* [online]. Aktualizace 10. 12. 2011. Dostupné z: `http://www.akkadia.org/drepper/dsohowto.pdf` [cit. 23. 5. 2012]