VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

DISASSEMBLER A ANALYZÁTOR BINÁRNÍHO KÓDU

DISASSEMBLER AND BINARY CODE ANALYZER

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **DAVID BAYER**
AUTHOR

VEDOUCÍ PRÁCE — **Dr. Ing. PETR PERINGER**
SUPERVISOR

BRNO 2021

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav inteligentních systémů (UITS) Akademický rok 2020/2021

Zadání bakalářské práce

Student: **Bayer David**
Program: Informační technologie
Název: **Disassembler a analyzátor binárního kódu**
Disassembler and Analyser of Binary Code
Kategorie: Překladače

Zadání:

1. Seznamte se s existujícími nástroji pro zpětný překlad a také s použitelnými metodami pro analýzu kódu. Seznamte se s architekturami ARM a AVR.
2. Navrhněte aplikaci typu *disassembler*, která umožní automaticky analyzovat strukturu kódu a dovolí uživateli doplnit anotace (komentáře, jména funkcí a proměnných, atd.). Navrhněte uživatelské prostředí, které usnadní spouštění zpětného překladu, vizualizaci výsledku a editaci anotací.
3. Implementujte aplikaci v C++. Pro zpětný překlad instrukcí použijte existující knihovny. Zaměřte se především na platformy AVR a ARM.
4. Zhodnoťte dosažené výsledky a popište možné další směry vývoje aplikace.

Literatura:

- Dle pokynů vedoucího.

Pro udělení zápočtu za první semestr je požadováno:

- První dva body zadání.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Peringer Petr, Dr. Ing.**
Vedoucí ústavu: Hanáček Petr, doc. Dr. Ing.
Datum zadání: 1. listopadu 2020
Datum odevzdání: 12. května 2021
Datum schválení: 11. listopadu 2020

Abstrakt

Tato práce se zabývá problematikou zpětného překladu binárního kódu do jazyka symbolických instrukcí. Věnuje se popisu architektur ARM a AVR a průzkumu existujících řešení. Na základě získaných informací zpracovává návrh a implementaci aplikace typu disassembler. Poskytuje grafické uživatelské rozhraní usnadňující spuštění zpětného překladu a zobrazení jeho výsledku. Navržené prostředí je snadno rozšířitelné o disassemblery dalších architektur, algoritmy řízení zpětného překladu a analýzy kódu.

Abstract

This thesis is focused on binary code disassembly. It describes the design of ARM and AVR architectures and does a research into existing solutions. Based on this knowledge, we designed and implemented a disassembler-like application. The application provides a graphical user interface to facilitate the starting of disassembly and displays the result. The designed environment is extensible to more disassemblers, disassembly algorithms, and code analysis.

Klíčová slova

disassembler, instrukční sada, ARM, AVR, instrukce, spustitelný soubor

Keywords

disassembler, instruction set architecture, ARM, AVR, instruction, executable

Citace

BAYER, David. *Disassembler a analyzátor binárního kódu.* Brno, 2021. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Dr. Ing. Petr Peringer

Disassembler a analyzátor binárního kódu

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Dr. Ing. Petra Peringera. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
David Bayer
11. května 2021

Poděkování

Rád bych tímto poděkoval panu Dr. Ing. Petru Peringerovi za jeho rady a ochotu při tvorbě této práce.

Obsah

1 Úvod **3**

2 Zpětný překlad do jazyka symbolických instrukcí **4**
2.1 Základní pojmy 4
2.2 Architektury počítačů 5
2.3 Formáty spustitelných souborů 7
2.4 Metody disassemblování 8
2.5 Problematika statického disassemblování 9
2.6 Algoritmy pro statické disassemblování 9
2.7 Způsoby analýzy kódu 12

3 Instrukční sady **14**
3.1 ARM 14
3.2 AVR 18

4 Analýza a návrh **20**
4.1 Knihovny pro práci se spustitelnými soubory 20
4.2 Knihovny pro disassemblování 21
4.3 Existující disassemblery 23
4.4 Analýza požadavků 26
4.5 Východiska pro návrh 26
4.6 Výběr technologií 26
4.7 Architektura aplikace 27
4.8 Návrh modulu disassembleru 27
4.9 Návrh modulu analyzátoru 29
4.10 Návrh uživatelského rozhraní 30

5 Implementace a testování **31**
5.1 Prerekvizity 31
5.2 Třída Program 31
5.3 Třída Instruction 32
5.4 Třída ControlBase 33
5.5 Třída InsDisasmBase 34
5.6 Třída CFG 35
5.7 Uživatelské rozhraní 36
5.8 Testování 37

6 Závěr **39**

Literatura **40**

A Obsah přiloženého paměťového média **42**

Kapitola 1

Úvod

Assembler je program, který překládá kód napsaný v jazyce symbolických instrukcí do strojového kódu. Disassembler je program provádějící opak — překládá strojový kód do člověku srozumitelného kódu v jazyce symbolických instrukcí. Disassemblování neboli zpětný překlad do jazyka symbolických instrukcí je jednou z hlavních technik reverzního inženýrství v oblasti software. Je používáno k analýze škodlivého software, emulaci platforem, binární analýze a dalším. Je jedinou cestou k analýze spustitelných souborů, pokud není možné získat jejich zdrojový kód.

Cílem této práce je bližší seznámení s problematikou disassemblování. Na základě analýzy existujících řešení a dostupných technologií navrhnout a implementovat aplikaci typu disassembler, která bude modulární a snadno rozšiřitelná. Bude určen pro práci s menšími binárními soubory architektur ARM a AVR. Aplikace umožní uživateli skrze grafické uživatelské prostředí jednoduché spuštění zpětného překladu, vizualizaci výsledku, editaci komentářů a možnost přidávat anotace.

Práce se skládá z několika kapitol. Kapitola 2 pojednává o procesu disassemblování, jeho metodách a možnostech dalšího analyzování výsledku. Kapitola 3 je věnována popisu architektur ARM a AVR. Zabývá se jejich koncepcí, operačními kódy a principy práce s nimi. Kapitola 4 zkoumá existujících řešení z pohledu implementace disassembleru a návrhu uživatelského rozhraní. Následně popisuje návrh aplikace pro zpětný překlad. Kapitola 5 dokumentuje jeho implementaci a shrnuje výsledky testování.

Kapitola 2

Zpětný překlad do jazyka symbolických instrukcí

Tato kapitola se věnuje potřebným znalostem pro tvorbu disassembleru. Soustředí na proces disassemblování a možnosti dalších analýz jeho výsledku.

2.1 Základní pojmy

Slovo je základním prvkem designu procesoru. Jedná se o data pevné délky, která jsou základní jednotkou pro instrukční sadu nebo hardware procesoru. Délka slova ovlivňuje do velké míry strukturu a operace počítače. Většina registrů procesoru pro obecné použití má velikost jednoho slova. Často se také jedná o maximální délku dat, která mohou být během jedné operace přenášeny při práci s počítačovou pamětí. Od délky slova se většinou odvíjí i velikost celočíselných datových typů a maximální množství adresovatelné paměti na dané architektuře. Dnešní procesory mívají délku slova 8 bajtů.

Zarovnání dat se provádí kvůli rychlejšímu přístupu k nim. Data jsou v paměti organizována po slovech. Je proto výhodné, když je adresa dat dělitelná délkou slova beze zbytku. Pokud tomu tak není, může být při přístupu k datům větším než jeden bajt potřeba více přístupů do paměti, než by bylo teoreticky potřeba. Bajty zarovnání jsou tedy bajty nedefinovaného obsahu, které tvoří výplň prostoru za posledním bajtem dat po další adresu, která je násobkem velikosti slova. Překladače vyšších programovacích jazyků často provádějí zarovnání dat, datových struktur i kódu automaticky. Paměť, která je užita pro zarovnání, je cenou za vyšší výkon.

Endianita popisuje pořadí uložení jednotlivých bajtů slova dat v paměti. Existují dva základní typy endianity — little-endian a big-endian. Little-endian (obrázek 2.1a) ukládá nejméně významový bajt na nejnižší adresu, zatímco big-endian (obrázek 2.1b) jej ukládá na nejvyšší adresu. Každý z přístupů je výhodný pro odlišné použití. Většina procesorů je uzpůsobena k práci pouze s jedním typem endianity, existují však architektury, které podporují práci s oběma. Mimo práci s pamětí se endianita používá také pro specifikaci pořadí bajtů při komunikaci po síti, kde je využíván big-endian.

Strojová instrukce je označení pro příkaz ve strojovém kódu, který procesor dokáže rozpoznat a následně na jeho základě provést příslušnou operaci. V textové podobě se

(a) little-endian

(b) big-endian

Obrázek 2.1: Příklad uložení 4-bajtové hodnoty 0x0187D51A v paměti na adrese 0x100 u systémů různých endianit.

skládá z mnemoniky a operandů. Mnemonika je kód instrukce skládající se ze symbolů. Operand specifikuje, se kterými daty se má daná operace provést, nebo data sám definuje. Instrukce se dělí do skupin podle typů operací, které provádějí, např. přesunové, aritmetické, řídící, řetězcové, koprocesorové a další.

Vstupní bod je označení pro adresu instrukce, která je vykonána po spuštění programu jako první. Je buď dána přímo *application binary interface* (ABI) nebo *embedded-application binary interface* (EABI), pro které je spustitelný soubor přeložen, nebo je určen překladačem či programátorem při tvorbě programu.

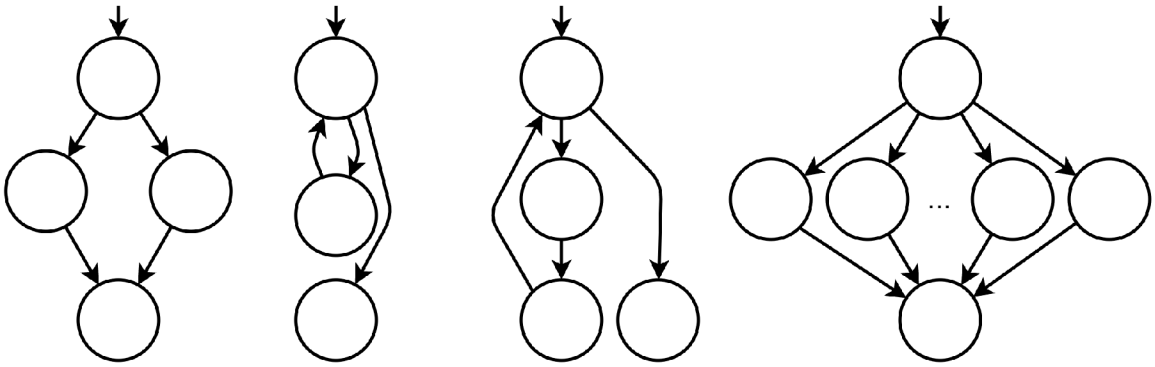
Basic block je přímočará sekvence instrukcí, která neobsahuje žádnou instrukci měnící tok programu, s výjimkou té poslední [1]. Má jeden vstupní bod. To znamená, že v programu neexistuje skok, jehož cílem by byl jiný bod z těla basic blocku než vstupní, a jeden výstupní bod, za kterým může následovat vykonávání jiného basic blocku. Tyto vlastnosti určují, že pokud je vykonána první instrukce basic blocku, nutně musí být vykonány všechny ostatní instrukce basic blocku v přesném pořadí právě jednou. Basic block, po kterém se daný basic block může přímo vykonat, se nazývá *předchůdce*. Daný basic block je pak svému předchůdci *následník*. Basic block může mít více předchůdců i následníků. Pokud nemá žádné následníky, nazývá se ukončovací basic block.

Control Flow Graph (CFG) je orientovaný graf, ve kterém uzly představují basic blocky a hrany reprezentují možné přechody mezi nimi (vizte obrázek 2.2) [1]. Během vykonávání programu mohou být použity všechny hrany grafu. Je často základem pro analýzu kódu a kompilátory jej využívají k optimalizacím a eliminaci nedosažitelného kódu. Lze díky němu detekovat např. nekonečné smyčky nebo zkoumat chování programu.

2.2 Architektury počítačů

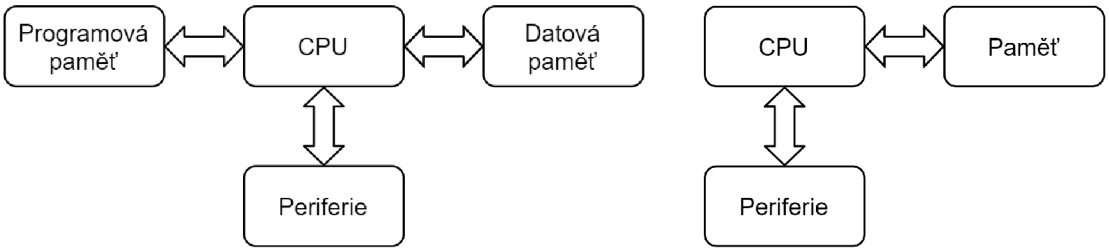
Pojem architektura představuje abstraktní model počítače. Její realizace se nazývá implementace. Definuje vlastnosti abstraktního modelu, jako jsou podporované datové typy, registry, způsob správy paměti, vstup-výstupní model a další. Specifikuje podporované instrukce a jejich chování. Existují dva hlavní typy architektur z pohledu paměťového modelu:

- **Harvardská architektura** od sebe fyzicky odděluje paměť pro program a paměť pro data. Obě paměti mohou mít zcela odlišné charakteristiky, jako je např. délka slova. Každá má také vlastní sběrnici, což znamená, že přístup k oběma pamětem se může vykonávat současně. Modifikovaná Harvardská architektura navíc přidává možnost adresace programové paměti a přistupovat tak k jejím datům.



Obrázek 2.2: Základní řídicí struktury v CFG

- **Von Neumannova architektura** uchovává data i instrukce ve stejné paměti. To sice vede k menším nákladům na výrobu, ale přináší s sebou problémy. Sdílená sběrnice znamená, že pokud např. nějaké externí zařízení pracuje s pamětí, procesor musí čekat, než je práce dokončena, aby mohl sám pokračovat ve vykonávání programu.



(a) Harvardská architektura (b) Von Neumannova architektura

Obrázek 2.3: Schéma počítačových architektur.

Architektury lze také dělit podle komplexity jejich instrukčních sad. Existuje celá řada typů, nejvýznamnější jsou však dvě skupiny:

- **CISC** — počítač podporuje rozsáhlou sadu instrukcí, které mívají různou velikost v závislosti na jejich komplexitě. Jedna instrukce může vykonat několik operací včetně přístupů do paměti v různých adresovacích módech. Jedná se o register-memory architekturu[1]. Výhodou je menší velikost programů a s tím spojených přístupů do paměti. Nevýhodou je složitost implementací těchto architektur.

- **RISC** — počítač pracuje s relativně malou sadou instrukcí, jež jsou vysoce optimalizované a mívají stejnou délku. Obecně obsahuje větší množství registrů než CISC architektury. Pro práci s pamětí slouží výhradně instrukce typu *load* a *store*. Nevýhodou je větší velikost programů a eventuálně menší komfort pro programátora, který musí nahrazovat komplexnější instrukce kombinací základních instrukcí.

[1]Register-memory architektura umožňuje provádění přesunových, aritmetických a dalších operací s daty nacházejících se přímo v paměti, ne pouze v registrech.

Instrukční sady mohou obsahovat také tzv. *ilegální* instrukce, které nejsou zmíněné v oficiální dokumentaci výrobce. Mají různý vliv na chod CPU v závislosti na konkrétní implementaci. Mohou být potenciální bezpečnostní hrozbou, způsobit pád systému, vykonat nedokumentovanou instrukci, nebo taky mohou být interpretovány jako instrukce `NOP`[2].

2.3 Formáty spustitelných souborů

Existuje celá řada formátů pro spustitelné binární soubory. Uchovávají data o jejich struktuře, obsahu a prostředí, pro které byly vytvořeny. Tyto informace mohou znatelně ulehčit i zpřesnit zpětný překlad a také zpřehlednit jeho výsledek. Jednotlivé formáty jsou užívány k různým účelům nebo jsou typicky používané na určitých operačních systémech. Nejznámějšími jsou *Executable and Linkable Format* (ELF) pro Unixové operační systémy, *Portable Executable* (PE) pro Microsoft Windows a *Mach-O* pro operační systémy macOS a iOS.

Různé formáty spustitelných souborů se mohou lišit obsahem. Disassembler pak může požadovat další informace, přestože disassemblovaný soubor je formátován. Minimálními požadavky pro spuštění zpětného překladu tak typicky jsou:

- specifikace architektury a endianita — jedná se o identifikaci instrukční sady architektury s možností doplnění o příznaky nebo specifikaci konkrétního procesoru. Disassembler tak může sám instrukční sadu omezit o na daném procesoru nedostupné instrukce nebo ji doplnit o instrukční sadu koprocesoru. Endianita může být dána implicitně typem architektury, definovaná ve struktuře formátu, nebo případně uživatelem. Existují však metody založené na strojovém učení, kterými lze zjistit specifikaci architektury i endianitu u neznámého souboru [9].
- vstupní bod — adresa, na které má vykonávání programu začít.

Sekce

Obsah spustitelných souborů může být rozdělen do sekcí podle jejich obsahu. Vlastnosti sekce jsou dány jejím typem a atributy. Je tak možné zjistit, které sekce obsahují kód, data, nebo jiné informace. Disassembler tak dokáže určit, které operace se sekcí provést nebo např. odhadnout obsah neznámé adresy. Mezi časté sekce vyskytující se ve spustitelných souborech patří:

- `.bss` — neinicializovaná data,
- `.rodata` — inicializovaná data pouze pro čtení,
- `.data` — ostatní data,
- `.text` — spustitelný kód,
- `.debug` — obsahuje dodatečné informace pro debugger.

[2] Instrukce pro provedení operace bez efektu. Její implementace se liší v závislosti na konkrétní architektuře.

```
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 03 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - GNU
  ABI Version:                       0
  Type:                              DYN (Shared object file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x60cc10
  Start of program headers:          64 (bytes into file)
  Start of section headers:          61060896 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         9
  Size of section headers:           64 (bytes)
  Number of section headers:         39
  Section header string table index: 38
```

Obrázek 2.4: Příklad výpisu ELF hlavičky pomocí programu *readelf*.

Tabulka symbolů

Jedná se o datovou strukturu, ve které jsou uloženy adresy, typy, jména a další informace o symbolech vyskytujících se v souboru. Pro disassembler mohou být tato data klíčová, protože poskytují adresy vstupních bodů do funkcí. Ve výstupu pak disassembler může nahrazovat adresy jmény symbolů a tím uživateli značně usnadnit porozumění kódu a orientaci v něm. Pokud spustitelný soubor obsahuje tabulku symbolů, bývá uložena jako samostatná sekce, nebo je přímo součástí formátu.

2.4 Metody disassemblování

Statická metoda čte jednotlivé úseky binárního souboru, které následně překládá do jazyka symbolických instrukcí [5]. Jejími výhodami jsou nižší náročnost na výkon (v závislosti na algoritmu) a vysoké pokrytí kódu programu. Umožňuje také disassemblovat nedosažitelný kód[3]. Díky těmto vlastnostem je tato metoda vhodná pro aplikace typu disassembler. Nevýhodou je však absence záruky správnosti disassemblované instrukce. To může vést k nesprávným výstupům disassembleru, a proto je třeba tomuto problému při tvorbě disassembleru věnovat zvýšenou pozornost.

Dynamická metoda vykonává kód binárního souboru v prostředí, které jeho chování za běhu analyzuje [5]. Hlavní výhodou je, že každá instrukce je disassemblována těsně před jejím vykonáním, což zaručuje její správnost. Na druhou stranu tím, že je disassemblován

[3]Kód, který se v souboru nachází, ale nemůže být nikdy vykonán, jelikož není součástí toku programu.

pouze vykonávaný kód, může tato metoda dosahovat velmi nízkého pokrytí kódu programu. Zásadní nevýhodou tohoto přístupu je vysoká náročnost na výkon. K samotnému vykonávání programu se přidávají nároky na předávání řízení a analýzy, což může prakticky způsobit nepoužitelnost této metody pro disassemblování náročnějších programů. Tato metoda je spíše vhodná pro aplikace určené k ladění a profilování programů.

2.5 Problematika statického disassemblování

Obecně by se dalo říct, že čím více informací o sobě a své struktuře soubor nese, tím kvalitnější výsledek statického disassemblování může být. Může s sebou nést mnoho problémů, jejichž řešení je často obtížné a ne vždy bezchybné [5][13]:

- Neznámá struktura spustitelného souboru. V případě, že o struktuře spustitelného souboru nejsou k dispozici žádné informace, jedinou spolehlivou jednotkou je bajt. Není však možné jednoduše zjistit, kde se nachází kód a kde data.

- Nepřímé řízení toku programu. Skoky a volání, jejichž cíle jsou vybírány dynamicky za běhu programu, představují nepřímé řízení toku. Adresa cíle je uložena do registru, který je pak použit jako operand pro instrukci skoku nebo volání. Při statickém disassemblování nelze bez dalších analýz zjistit ze jména registru jeho obsah a tudíž cíl nemůže být disassemblován. K obdobnému problému dochází při využívání změny řízení toku pomocí výjimek.

- Nekonstantní délka a překrývání instrukcí. K problému s proměnnou délkou instrukcí dochází z pravidla na CISC architekturách. Např. na architektuře x86 je naprostá většina možností z jednobajtových instrukcí zabrána, což znatelně zvyšuje pravděpodobnost, že případná mylná interpretace dat za kód bude úspěšná.

- Zamlžení (*obfuscation*). Tato technika je používána v případech, kdy se programátor snaží skrýt logiku, algoritmy nebo význam kódu a tím znesnadnit jeho analýzu. Některé části kódu mohou být dokonce šifrovány a dešifrovány programem až při jeho vykonávání. Statické disassemblování je tak velmi obtížné, v některých případech prakticky nemožné.

- Samoupravující se kód. Spustitelné soubory mohou za běhu generovat nebo modifikovat kód, který má být prováděn. Při disassemblování statickou metodou se však tyto změny neprojeví a není tak možné tento kód zpracovat. Používá se tak kombinace s dynamickou metodou, která modifikovaný nebo generovaný kód předává statickému disasassembleru.

- Bajty dat nebo zarovnání mezi kódem. Mezi bajty kódu se mohou nacházet také data. Data i kód bývají zarovnány na určitý počet bajtů v závislosti na architektuře kvůli důvodům popsaným v 2.1. Tyto bajty dat nebo zarovnání mohou pro statickou metodu znamenat problém, protože mohou být mylně považovány za bajty instrukcí.

2.6 Algoritmy pro statické disassemblování

Při disassemblování statickou metodou je třeba zvolit algoritmus pro řízení zpětného překladu instrukcí. V závislosti na typu spustitelného souboru a disassemblované architektury

mohou být vhodné nebo postačující různé algoritmy rozdílné složitosti, výpočetní náročnosti a správnosti výsledku. Pro hodnocení správnosti nálezu je užíváno spojení slov *true*, nebo *false* se slovy *positive*, nebo *negative*. První část vyjadřuje správnost interpretace a druhá část zda byl nález algoritmem shledán jako instrukce, nebo data. Cílem algoritmu je pokrýt co největší část kódu s co nejmenším množstvím nesprávně interpretovaných instrukcí a dat.

Linear sweep

Linear sweep je nejjednodušším algoritmem pro disassemblování. Je rychlý a velmi jednoduchý na implementaci [5][12]. Od dané počáteční adresy jsou přímočaře disassemblovány instrukce, dokud není překročena koncová adresa. Algoritmus pracuje s předpokladem, že se instrukce nacházejí v těsné blízkosti za sebou. To znamená, že po disassemblování instrukce je k aktuální adrese přičtena její velikost a tím je získána adresa další. Nejsou tak brány v potaz možné bajty dat nebo zarovnání mezi bajty instrukcí, což může potenciálně vést k velkému množství *false positive* nálezů.

I přesto se však jedná o hojně používaný algoritmus hlavně na RISC architekturách, kde je většinou dána pevná délka instrukce a při nesprávné interpretaci vznikne souvislý blok nesprávných instrukcí následovaný již správnou správnou sekvencí instrukcí. Na CISC architekturách však kvůli rozdílné délkou instrukcí může dojít k nesprávné interpretaci celého zbytku kódu.

Recursive traversal

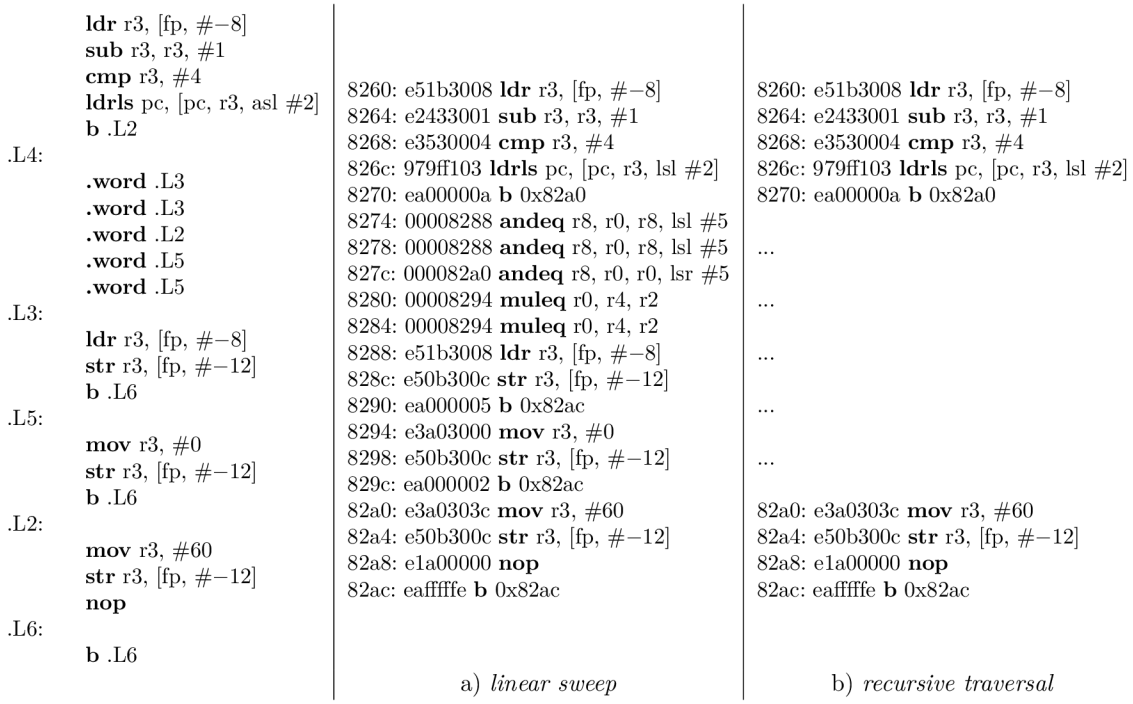
Recursive traversal na rozdíl od algoritmu *linear sweep* respektuje toku programu (vizte obrázek 2.5) [5][12]. Na základě typu disassemblované instrukce vykoná jednu z operací:

- v případě instrukce podmíněného skoku nebo volání funkce rekurzivně disassembluje jejich cíle a pokračuje další instrukcí,
- v případě instrukce nepodmíněného skoku rekurzivně disassembluje cíl a dále nepokračuje,
- v případě instrukce návratu z funkce ukončí disassemblování větve.

Je tak velmi malá pravděpodobnost, že dojde k *false positive* interpretacím. Problém však nastává v případě, kdy se v kódu vyskytují instrukce nepřímých skoků nebo volání. V základní formě nedokáže tento algoritmus zjistit adresy cílů a není tak možné jejich kód disassemblovat. To může vést k nižšímu pokrytí kódu. Existují však analýzy, kterými lze adresy cílů ve většině případů zjistit a tím pokrytí zvýšit. Tento algoritmus je použit v mnoha úspěšných disassemblerech.

Superset disassembly

Superset disassembly považuje každou adresu binárního souboru za potenciální začátek instrukce [7][12]. Algoritmy založené na tomto přístupu mívají vysoké pokrytí kódu, nicméně jejich implementace bývá výrazně složitější. Paměťová náročnost a nároky na výkon jsou podstatně větší než při použití *linear sweep* a *recursive traversal* algoritmů. Jsou vhodné k přepisování a instrumentaci binárních souborů.



Obrázek 2.5: Příklad ukazující rozdíly mezi algoritmy *linear sweep* a *recursive traversal* při zpětném překladu. První sloupec obsahuje původní kód programu v jazyce symbolických instrukcí, který byl přeložen a poté disassemblován. Lze vidět, že disassembler používající algoritmus *linear sweep* sice má větší pokrytí, nicméně v rozmezí adres *0x8274* až *0x8284* interpretoval data jako instrukce. Disassembler využívající algoritmus *recursive traversal* sice žádné *false positive* nálezy nemá, nedisassembloval však instrukce mezi adresami *0x8288* a *0x29c*.

Speculative disassembly

Algoritmus *speculative disassembly* přidá disassemblovanou instrukci na konec každého basic blocku, na který by přímo navazovala, tzn. pokud součet adresy poslední instrukce v basic bloku a její velikosti se rovná adrese disassemblované instrukce [7]. Pokud žádný takový basic block není, je vytvořen nový a disassemblovaná instrukce se stává jeho vstupním bodem.

Basic blocky končící na stejné instrukci měnící tok programu jsou shlukovány do maximálních bloků. Maximální bloky mohou být spojeny, pokud alespoň jeden lineární blok maximálního bloku začíná těsně za koncem poslední instrukce maximálního bloku, ke kterému má být připojen.

Následně jsou maximální bloky zotaveny a výsledkem je množina všech validních basic blocků nacházejících se v binárním souboru. Při použití dalších analýz lze dosáhnout velice dobrých výsledků.

Probabilistic disassembly

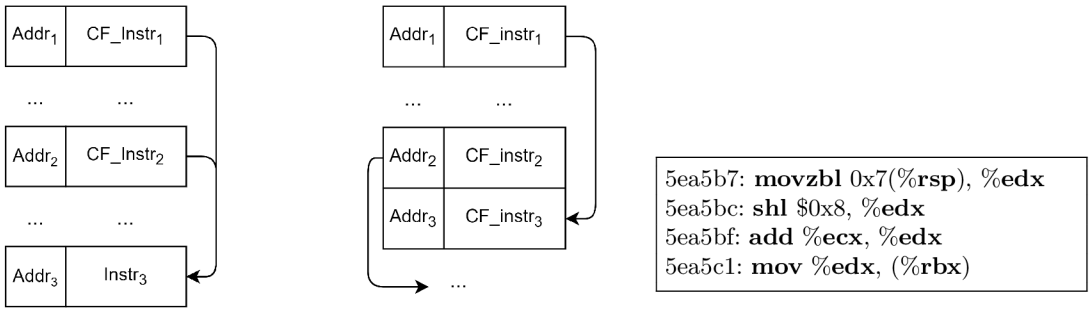
Probabilistic disassembly se zakládá na myšlence, že *true positive* instrukce vykazují společné rysy, které lze definovat a na jejich základě spočítat pravděpodobnost, že se skutečně jedná o *true positive* instrukce [12]. Tyto rysy mohou být například:

1. Konvergence toku programu (obrázek 2.6a). Pokud se mezi instrukcemi nachází skoky na stejné cílové adresy, s největší pravděpodobností to znamená, že se jedná o validní instrukce a na cílové adrese se rovněž nachází validní instrukce.

2. Přecházení toku programu (obrázek 2.6b). Pokud se v kódu nachází 3 validní skokové instrukce, přičemž první provádí skok na třetí, druhá a třetí jsou v těsné blízkosti za sebou a cílová adresa druhé je odlišná od adresy třetí instrukce, je velmi pravděpodobné, že se nejedná o byty dat, ale byty instrukcí. Podobné konstrukce jsou vytvářeny překladači při implementaci cyklů.

3. Vztah *definice-použití* registrů (obrázek 2.6c). Pokud jedna instrukce definuje nebo mění obsah registru a následující instrukce tento registr využívá, jedná se nejspíše o byty instrukcí. Čím větší počet registrů architektura obsahuje, tím je tento rys směrodatnější, protože je menší pravděpodobnost, že by byl náhodně užit.

Rysy lze přizpůsobovat konkrétní architektuře. Výsledkem algoritmu je pravděpodobnost *true positive* nálezu na každé adrese binárního souboru.



(a) Konvergence toku programu

(b) Přecházení toku programu

(c) Příklad vztahu definice a použití registru `edx` na architektuře x86-64

Obrázek 2.6: Příklady rysů pro počítaní pravděpodobnosti v *probabilistic disassembly*

2.7 Způsoby analýzy kódu

V této podkapitole jsou představuje vybrané analýzy, které lze s disassemblovanými instrukcemi provádět. Umožňují lepší pochopení kódu a orientaci v něm.

Control-flow analýza

Obecně užívanou metodou ke konstrukci CFG je sledování toku programu od zadaného vstupního bodu [13]. Instrukce jsou postupně procházeny a přidávány do basic blocku. Pokud je aktuální instrukce instrukcí skoku, je basic block ukončen a instrukce na adrese cílu skoku je začátkem nového basic blocku. V případě, že se jednalo o podmíněný skok, je navíc započat nový basic block na následující instrukci. Nové basic blocky jsou následníky ukončeného basic blocku. Tento přístup je velmi jednoduchý, avšak nedokáže vytvořit kompletní CFG, pokud se v kódu nacházejí nepřímé skoky.

Graf volání

Jedná se o orientovaný graf, ve kterém uzly představují funkce nebo metody a hrany jednotlivá volání. Znázorňuje vztahy mezi funkcemi. Lze tak díky němu lehce detekovat např. rekurzi a usnadnit pochopení a orientaci v programu. Vzniká při disassemblování funkce, kdy se vytváří množina volaných funkcí při analýze instrukcí volání. Pokud se vyskytují v kódu nepřímá volání, nemusí být graf kompletní.

Analýza tabulky skoků

Několikanásobné větvení programu, např. řídicí struktura `switch` v jazyce C, může být implementováno několika způsoby. Nejjednodušším způsobem je provedení několikanásobného porovnání jako lineární sekvence. Tento způsob je vhodný hlavně pro menší počet větví. Další možností je porovnání provést podle předem vytvořeného binárního stromu, čímž se sníží počet porovnání i časová náročnost.

Nejpoužívanější je implementace pomocí tabulky skoků, která je rovněž nejrychlejší metodou pro velké množství větví. Spočívá ve vytvoření tabulky s adresami jednotlivých větví. Na základě porovnání je spočítán index do této tabulky a následně je proveden nepřímý skok na adresu vybrané větve. V závislosti na architektuře mohou být tabulky skoků uloženy buď přímo v kódu, nebo v sekci `.rodata`.

Analýza pro obnovu tabulek skoků funguje tak, že pokud disassembler narazí na nepřímý skok, provádí se zpětná analýza přesunu registrů, dokud se nenarazí na přiřazení adresy tabulky do registru [8]. Pokud je adresa vrácena funkcí nebo je dosaženo začátku funkce bez přiřazení adresy, znamená to, že adresu tabulky nelze tímto způsobem nalézt. To se ale nestává příliš často.

Detekce funkcí

Nejpoužívanějším způsobem pro hledání funkcí je detekce cílů instrukcí volání a využití heuristik pro hledání prologu funkce. Prolog funkce je ustálená sekvence instrukcí pro vytvoření zásobníkového rámce (vizte obrázek 2.7). Pro běžné spustitelné soubory je tato metoda účinná, nicméně např. při použití na *zamlžené* binární soubory nedosahuje příliš dobrých výsledků. Další možností je využití strojového učení k rozpoznání hranic funkce [6], to ale vyžaduje kvalitní příklady k učení a ladění jemných detailů, analýza CFG, kdy se eliminují hrany volaných funkcí a uzly bez vstupujících hran jsou považovány za začátky funkcí, nebo přístup založený na kontrole toku programu a dat, který dokáže rovněž analyzovat argumenty a typ návratové hodnoty.

```
push ebp
mov ebp, esp
sub esp, N
..
mov esp, ebp
pop ebp
ret
```

(a) x86, N je velikost lokálních proměnných

```
enter N, 0
..
..
..
..
leave
ret
```

```
mov ip, sp
stmfd sp!, {fp, sp, lr}
..
..
..
ldmfd sp, {fp, sp, lr}
bx lr
```

(b) ARM

Obrázek 2.7: Příklady implementace prologů a epilogů funkcí na různých architekturách

Kapitola 3

Instrukční sady

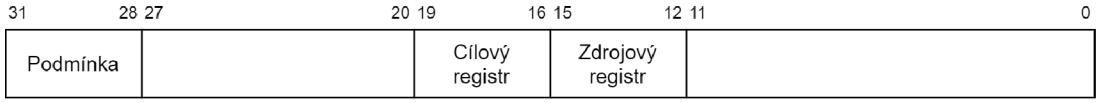
Tato kapitola se zabývá designem architektur ARM a AVR. Věnuje se vlastnostem a zvláštnostem jejich návrhu a popisem jejich instrukčí sady. Informace v této kapitole byly čerpány ze zdrojů [3][4][10][11].

3.1 ARM

Architektura ARM je jednou z nejrozšířenějších rodin architektur procesorů. Její počátky sahají do 80. let. Jedná se tedy o architekturu s poměrně dlouhou historií a několika verzemi. Čipy založené na této architektuře jsou často využívány v zařízeních, u kterých je kladen důraz na nízkou spotřebu elektrické energie.

Operační kód

Operační kód pro ARM je vždy little-endian. Instrukce mají velikost 32 bitů. Téměř všechny začínají čtyřbitovým prefixem (obrázek 3.1), který podmiňuje vykonání instrukce v závislosti na aktuálním stavu registru příznaků. Obsahuje také kombinaci pro nepodmíněné vykonání. Výhodou je, že ne každé podmíněné vykonávání musí být implementováno skoky, na druhou stranu je tímto omezen prostor pro zbytek dat v instrukci. Některé instrukce umožňují při přidání sufixu *S* za koncem mnemoniky. To znamená, že instrukce během svého vykonávání aktualizuje příznakový registr a není tak nutné používat další porovnávací instrukci.



Obrázek 3.1: Ukázka typického rozložení instrukcí instrukční sady ARM.

Thumb

Jelikož ARM je RISC architekturou, k vykonání ekvivalentní operace jedné instrukce na CISC procesorech může být potřeba provést více instrukcí na RISC procesorech. Z důvodu úspory paměti byla vyvinuta instrukční sada Thumb. Zkracuje velikost instrukcí na pouhých 16 bitů a tím velikost výsledného kódu na 65 %. Jejich funkcionalita je však oproti ARM instrukcím omezena - sada instrukcí není tak rozsáhlá, většina z nich může pracovat pouze

s omezeným počtem registrů a podmíněně mohou být vykonány pouze skokové instrukce. To může mít dopady na výkon systému.

Procesor se nachází v Thumb módu (pracuje s instrukční sadou Thumb), pokud je Thumb-bit v registru `CPSR` nastaven na hodnotu 1. Přechod mezi ARM a Thumb módem lze provést pouze pomocí instrukce skoku nebo volání se změnou instrukční sady.

Thumb-2

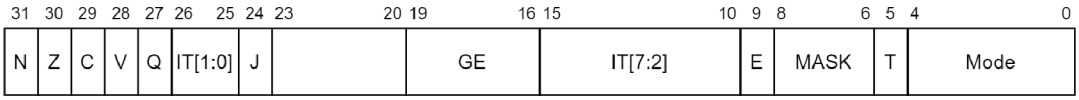
Thumb-2 slučuje výhody instrukčních sad ARM a Thumb. Obsahuje 32 i 16 bitové instrukce. Délkou kódu se tak blíží instrukční sadě Thumb, zároveň si však drží plnou funkcionalitu a téměř stejný výpočetní výkon jako instrukční sada ARM. Přináší také několik nových instrukcí včetně prefixové instrukce `IT`, která přináší nový způsob podmíněného vykonávání.

Registry

Všechny registry mají velikost 32 bitů. V user módu architektura nabízí pro práci celkem 16 registrů pro obecné použití označené `R0`–`R15`, z nichž některé mají zvláštní význam:

- `FP`/`R11` je ukazatelem zásobníkového rámce,
- `IP`/`R12` je dočasným registrem meziprocedurálních volání,
- `SP`/`R13` je používán jako ukazatel na vrchol zásobníku,
- `LR`/`R14` nese adresu pro návrat z funkce,
- `PC`/`R15` je čítač instrukcí.

Ke všem těmto registrům má programátor plný přístup a může měnit libovolně jejich obsah[1]. Registr `CPSR` (current program status register) uchovává status procesoru a další řídící informace. Obsahuje příznaky znaménka (N), vynulování (Z), přenosu (C), přetečení (V), bitové pole větší než nebo rovno (GE), aktuální endianitu (E), Thumb-bit (T), bity pro popis stavu vykonávání `IT` instrukce (IT) a aktuální mód procesoru (Mode).



Obrázek 3.2: Rozložení obsahu registru CPSR.

Většina Thumb instrukcí má přístup pouze k registrům `R0`–`R7`. Práce se zásobníkem probíhá výhradně pomocí instrukcí `PUSH` a `POP`, které implicitně pracují s registrem `SP`.

Banked registry jsou vlastní registry většiny privilegovaných módu. Stávají se aliasy pro jména běžných registrů. Všechny privilegované módy s výjimkou systémového mají vlastní registry `SP` a `LR`. Módy výjimek nepoužívají registr `CPSR`, ale *banked* registr `SPSR`, který nese uložený stav registru `CPSR` těsně před vyvoláním výjimky.

[1] Je však třeba v dokumentaci ověřit, zda daná instrukce neomezuje zápis do daného registru.

Barrel shifter

Jedná se o funkční jednotku umožňující vykonávat posuny nebo rotace posledního operandového registru o konstantní 5-bitovou hodnotu. Lze díky němu provést řadu operací bez použití dalších instrukcí. Na výběr je z pěti typů posunů nebo rotací:

- `LSL` — logický posun vlevo,
- `LSR` — logický posun vpravo,
- `ASR` — aritmetický posun vlevo,
- `ROR` — rotace vpravo a
- `RRX` — rotace vpravo zahrnující příznak přetečení jako 33. bit.

Adresování

ARM využívá load-store model pro práci s pamětí. To znamená, že pouze skrze instrukce `LDR` a `STR` (a jejich varianty) je možno přistupovat do paměti. Je tak podporováno pouze nepřímé adresování. Většina implementací umožňuje práci s daty různých endianit (little-endian a big-endian). K změně endianity slouží instrukce `SETEND`. V jazyce symbolických instrukcí je adresování značeno operandy v hranatých závorkách. Ke zjištění cílové adresy jsou k dispozici tři typy offsetu:

1. hodnotou — součet hodnot adresového registru a konstanty,

   ```
   LDR R0, [R1, #4] ; R0 = *(R1 + 4)
   ```

2. registrem — součet hodnot adresového registru a dalšího registru,

   ```
   LDR R0, [R1, R4] ; R0 = *(R1 + R4)
   ```

3. registrem v měřítku — součet hodnot adresového registru a dalšího registru, jehož hodnota posunutá doleva o hodnotu konstanty.

   ```
   STR R0, [R1, R3, LSL #2] ; *(R1 + R3 << 2) = R0
   ```

Architektura nabízí tři módy adresování, které popisují efekt na hodnotu v adresovém registru:

1. offset — čtení/zápis probíhá na adrese v adresovém registru s připočteným offsetem, hodnota v adresovém registru se nemění. Všechny operandy se nacházejí uvnitř hranatých závorek a žádné další se za jejím koncem nevyskytují (vizte předchozí příklady).
2. prefix — před provedením čtení/zápisu na adrese v adresovém registru je k němu přičtena hodnota offsetu. Je značeno stejně jako adresování offsetem, za hranatými závorkami se ale nachází vykřičník.

   ```
   LDR R0, [R1, #4]! ; R1 += 4, R0 = *(R1)
   ```

3. postfix — po provedení čtení/zápisu na adrese v adresovému registru je k němu přičtena hodnota offsetu. Offset je značen operandy za hranatými závorkami.

   ```
   STR R1, [R5], R2, LSL #4 ; *(R5) = R1, R5 += R2 << 4
   ```

Hromadný load a store

Pro větší efektivitu práce s pamětí umožňuje ARM hromadný zápis nebo hromadné čtení z/do několika registrů najednou. Využívá k tomu instrukce `LDM`*xy* a `STM`*xy*, kde písmeno *x* nahrazuje `I` (increase) nebo `D` (decrease) a písmeno *y* nahrazuje `B` (before) nebo `A` (after), což jsou způsoby přístupu k paměti.

Prvním operandem je registr s adresou do paměti[2] a druhým je seznam registrů určených k přesunu ve složených závorkách daných buď výčtem, nebo rozsahem. Použitím těchto instrukcí s adresovým registrem `SP` a správným způsobem přístupu do paměti lze docílit stejného efektu jako použitím instrukcí pro práci se zásobníkem (`PUSH` a `POP`).

Řízení toku

Většina architektur neumožňuje přímo pracovat s `PC` registrem a používají k tomu pouze vyhrazené instrukce. Architektura ARM však umožňuje s `PC` registrem pracovat stejně jako s ostatními. Mohou tak vzniknout obtížně detekovatelné změny v toku programu. Tato sekce bude však věnovat běžným způsobům pro provádění daných operací.

Skoky

Skok je možné provést instrukcí `B` (branch) na relativní adresu k `PC` nebo instrukcí `BX` (branch and exchange instruction set), která provede skok na adresu v daném registru a Thumb-bit v `CPSR` nastaví na hodnotu jeho nultého bitu. Může dojít ke změně instrukční sady na ARM nebo Thumb.

Volání a návrat z funkce

Volání funkce se provádí pomocí instrukce `BL` (branch with link), která uloží adresu následující instrukce do `LR` a provede skok na relativní adresu k `PC`. Návrat z funkce pak proběhne přepisem hodnoty v `PC` hodnotou, kterou obsahoval `LR` na začátku funkce.

Volání a návrat z funkce s výměnou instrukční sady

K volání funkce se změnou instrukční sady slouží instrukce `BLX` (branch with link and exchange instruction set). Uloží do `LR` adresu následující instrukce. Pokud je operandem hodnota, provede se skok na relativní adresu k `PC` a je převrácen bit Thumb-bit v `CPSR`. Pokud je operandem registr, provede se skok na adresu v něm a Thumb-bit v `CPSR` je přepsán nultým bitem z registru. Pro návrat z funkce je nutné provést skok s výměnou instrukční sady instrukcí `BX` a `LR` jako operandem.

Konvence volání

Argumenty funkce jsou uloženy v registrech `R0`–`R3`. Pokud je argumentů více, jsou předány přes zásobník. Registr `LR` obsahuje návratovou adresu. S registry `R4`–`R11` je možné ve funkci pracovat, je však nutné, aby na konci funkce byly jejich hodnoty obnoveny.

Návratová hodnota je uložena podle velikosti v registrech `R0`–`R3`. Pokud je větší než 128 bitů, je volající povinen alokovat paměťový prostor a jeho adresu předat funkci v registru `R0`.

[2] Instrukce umožňují zpětný zápis do adresového registru, tzn. že po provedení instrukce je jeho hodnota změněna na konečnou adresu sekvence.

3.2 AVR

Jedná se o architekturu 8-bitových mikrokontrolérů. Obsahuje několik rodin lišící se velikostí paměti, frekvenci procesoru a dalších vlastností. Využívají se hlavně ve vestavěných systémech.

Paměť

Architektura AVR je založená na modifikované Harvardské architektuře. Paměti pro program a pro data jsou tedy fyzicky odděleny a používají jinou sběrnici, což znamená, že k oběma pamětem může být přistupováno zároveň. Paměť pro program je organizována po 16 bitech a je rozdělena na dvě sekce — boot loader a application program sekci. Je určena pouze pro čtení. Mimo kód může obsahovat také konstantní tabulky. Paměť pro data je rozdělena na:

- registry — 32 8-bitových registrů R0-R31,
- I/O registry — 64 8-bitových registrů pro komunikaci s periferiemi nebo jinému využití,
- rozšířené I/O registry — některé AVR mikrokontroléry dokáží zmenšit adresový prostor interní statické paměti (*Static Random Access Memory*, dále jen SRAM) a o takto nabyté adresy rozšířit adresování dalších až 160 I/O registrů,
- interní SRAM — volatilní paměť s nízkou přístupovou dobou pro uložení dočasných hodnot,
- externí SRAM — některé AVR mikrokontroléry umožňují rozšířit interní SRAM paměť o externí a tím navýšit její celkovou kapacitu paměti.

EEPROM

Jedná se o nevolatilní paměť[3], ze které lze číst nebo do ní zapisovat jednotlivé bajty. Má svůj vlastní paměťový prostor. Práce s ní je umožněna pomocí EEPROM adresového, datového a řídícího I/O registru. Přístup k jejím datům je pomalý, proto je používána hlavně pro uložení konfiguračních souborů mikrokontroléru.

16-bitové registry

Jelikož paměťový prostor pro data je u většiny AVR mikrokontrolérů větší než 256 bajtů, 8-bitové registry pro obecné použití ho nedokáží adresovat celý. Proto vznikly větší 16-bitové registry konkatenací dvou 8-bitových registrů `X` (`R26` a `R27`), `Y` (`R28` a `R29`) a `Z` (`R30` a `R31`).

Adresování

AVR je load-store architekturou. Datová paměť může být adresována přímo i nepřímo. Pro AVR mikrokontroléry s SRAM pamětí je k přímému adresování využita 16-bitová konstanta, jinak lze využít adresování 7-bitovou konstantou. K nepřímému adresování dat jsou

[3] Po odpojení od zdroje napětí paměť nepřichází o uložené informace.

používány registry X, Y a Z. Jejich hodnota v závislosti na použitých operandech může zůstat nezměněna, být post inkrementována, pre dekrementována, nebo zvýšena o hodnotu konstanty.

Konstantní tabulky z programové paměti mohou být adresovány pomocí instrukce LPM (případně ELPM) a Z registru. Pokud je datový nebo programový paměťový prostor větší než 64 KiB, tak při nepřímé adresaci jsou ukazatelové registry dále konkatenovány s hodnotou příslušného RAMP I/O registru.

Řízení toku

Aktuální pozice v programu je uložena v registru PC, který má velikost 16 nebo 22 bitů v závislosti na velikosti programové paměti. Jeho hodnotu lze měnit pouze skokovými instrukcemi nebo instrukcemi volání. U obou skupin instrukcí existují 4 varianty (s různými prefixy mnemoniky):

1. přímý — do PC je zapsána 16-bitová konstanta,
2. relativní (R) — k PC je přičtena 12-bitová znaménková konstanta,
3. nepřímý (I) — do PC je načtena hodnota z registru Z,
4. rozšířený nepřímý (EI) — do PC je načtena hodnota z registru Z konkatenována s EIND I/O registrem.

Při volání funkce je návratová adresa uložena na vrcholu zásobníku za argumenty. Návrat z funkce je proveden instrukcí RET, která PC přepíše hodnotou na vrcholu zásobníku.

Konvence volání

Každý argument funkce je předán buď celý rozmístěný v registrech, nebo celý na zásobníku. Argumenty jsou vkládány do registrů R25–R8 tak, že jsou zarovnány na 16 bitů a jejich nejvýznamnější bity se nacházejí v registrech s vyšším číslem. Pokud je argument příliš velký a nevejde se do zbývajícího prostoru v registrech, je předán na zásobníku a každý další argument taktéž.

Pokud je návratová hodnota menší nebo rovna 8 bajtům, je předána v registrech na stejném principu jako první argument při volání funkce, jinak je volající povinen pro ni alokovat prostor a předat jeho adresu jako první argument funkce.

Funkce může měnit obsah všech registrů, na jejím konci však musí obnovit hodnoty registrů R1–R17 a Y.

Kapitola 4

Analýza a návrh

Tato kapitola se zabývá průzkumem již existujících řešení použitelných k implementaci disassembleru v jazyce C++ a dále se věnuje návrhu aplikace typu disassembler. Seznam popisovaných řešení představuje pouze vybrané zástupce, neobsahuje všechna existující řešení.

4.1 Knihovny pro práci se spustitelnými soubory

Tyto knihovny zapouzdřují práci s binárními soubory. Umožňují přístup k informacím uložených v souborových formátech a poskytují rozhraní k získávání nebo modifikaci souborových dat. Některé umožňují automaticky detekovat jim známé formáty, jiné tuto informaci potřebují znát.

Knihovna libbfd

Knihovna *libbfd* je součástí kolekce nástrojů GNU Binutils[1]. Jedná se o velmi rozsáhlou knihovnu podporující velké množství formátů i práci s neformátovanými soubory. Vedle spustitelných souborů umožňuje pracovat i s archivy. Její vývoj začal před více než 20 lety, i přesto je jejím největším problémem nedostatečná dokumentace.

Před zahájením práce s knihovnou by měla být zavolána funkce `bfd_init`. Ta sice prozatím neprovádí žádnou operaci, je ale možné, že v dalších verzích knihovny bude potřeba. Dále práce s knihovnou probíhá probíhá skrze strukturu `bfd`, která je dynamicky alokována po otevření nebo vytvoření souboru. Obsahuje informace o souboru a jeho formátu, jeho sekcích, architektuře, symbolech a další. Pomocí funkce `bfd_check_format` umožňuje automaticky detekovat formát neznámého souboru.

Knihovna a obsahuje řadu užitečných funkcí a datových typů např. pro práci s daty různých endianit, relokacemi nebo symboly.

Knihovna LIEF

LIEF[2] je multiplatformní knihovnou pro parsování, instrumentaci a upravování binárních souborů. Umožňuje konverze mezi jednotlivými formáty objektových souborů. Podporuje nejběžnější formáty ELF, PE a Mach-O spolu s dalšími formáty pro platformu Android.

[1] https://www.gnu.org/software/binutils/
[2] https://lief.quarkslab.com/

Podporuje také jejich automatické rozeznání. Je napsána v jazyce C++ a využívá třídního návrhu. Poskytuje rozhraní pro jazyky C, C++ a Python, pro jazyk C je však velmi omezené. Ke knihovně je dostupná přehledná dokumentace s řadou příkladů. Podstatnou nevýhodou je nemožnost pracovat s neformátovanými soubory.

Handler daného formátu je vytvořen statickou metodou `parse`. Umožňuje číst a měnit symboly, relokace, modifikovat a zjišťovat obsah sekcí nebo segmentů souboru a další. Metodou `write` lze všechny změny zapsat do souboru.

```
#include <iostream>
#include <bfd.h>

int main(void)
{
    bfd *abfd;
    bool match;

    bfd_init();
    abfd = bfd_openr("~/obj_elf_file", nullptr);

    if (abfd == nullptr) { /* handle error */ }

    match = bfd_check_format(abfd, bfd_object);
    if (!match) { /* handle error */ }

    if (bfd_get_flavour(abfd)
        != bfd_target_elf_flavour)
        { /* potentially handle error */ }

    std::cout << "Section count: "
              << bfd_count_sections(abfd)
              << std::endl;

    bfd_close(abfd);

    return 0;
}
```

(a) libbfd

```
#include <iostream>
#include <LIEF/LIEF.hpp>

int main(void)
{
    using namespace LIEF;

    std::unique_ptr<Binary> bin;
    std::unique_ptr<ELF::Binary> elf_bin;

    bin = Parser::parse("~/obj_elf_file");

    if (bin == nullptr) { /* handle error */ }

    if (bin->format()
        != EXE_FORMATS::FORMAT_ELF)
        { /* handle error */ }

    elf_bin = static_cast<ELF::Binary>(bin);

    const Header &elf_hdr = elf_bin->header();

    std::cout << "Section count: "
              << elf_hdr.numberof_sections()
              << std::endl;

    return 0;
}
```

(b) LIEF

Obrázek 4.1: Příklad programů napsaných v jazyce C++ pro zjištění počtu sekcí spustitelného souboru ELF formátu s využitím různých knihoven pro práci s binárními soubory. U obou knihoven je využita automatická identifikace formátu souboru souboru *~/obj_elf_file*.

4.2 Knihovny pro disassemblování

Knihovny pro disassemblování poskytují rozhraní k zpětnému překladu binárního kódu na instrukce.

Framework Capstone

Capstone[3] je multiplatformní framework pro disassemblování. Nabízí jednoduché a přehledné aplikační rozhraní a umožňuje disassemblovat více typů architektur. Pro práci s ním je třeba nejdříve vytvořit handler dané architektury. Disassemblovat je pak možné jednotlivé instrukce nebo celé buffery dat.

Disassemblování instrukcí probíhá pomocí *Machine Code* modulu projektu LLVM[4], který převádí strojový kód do interní reprezentace v podobě struktury `MCInst`. Tato struktura je dále vytisknuta do bufferu, jehož obsah je podle převodové tabulky dané architektury transformován na instrukci v podobě struktury `cs_insn`, která obsahuje její adresu, byty, řetěz s mnemonikou a řetězec s operandy.

Knihovna libopcodes

Knihovna *libopcodes* je (stejně jako knihovna *libbfd*) součástí kolekce nástrojů GNU Binutils. Je používána disassemblerem *objdump* a GNU Debuggerem. Jedná se o rozsáhlou statickou knihovnu s podporou mnoha architektur naprogramovanou v jazyce C. Je silně provázána s knihovnou *libbfd*. Své rozhraní definuje v hlavičkovém souboru *dis-asm.h*.

Pomocí funkce `disassembler` je na základě specifikace architektury, její verze a endianity disassemblovaného souboru získán ukazatel na disassembler. Disassembler reprezentuje funkci, která přeloží jednu instrukci do jazyka symbolických instrukcí tak, že její obsah postupně, nebo najednou tiskne na daný výstup předem určenou funkcí. Získaný ukazatel na funkci disassembleru je použit k inicializaci pomocné struktury `disassemble_info`, která obsahuje informace o formátu spustitelného souboru, zvolené architektuře, dostupných symbolech, funkce pro čtení paměti a emitace chyb, stylu tisku a další. Je používána disassemblerem pro získávání dat souboru, ukládání privátních dat architektury a případných dalších pomocných dat.

Disassemblování je provedeno voláním funkce disassembleru s adresou instrukce a informacemi v pomocné struktuře.

Knihovna libr_asm

Radare2[5] je framework pro reverzní inženýrství a analýzu binárních souborů. Jedná se o sadu utilit, ke kterým lze přistupovat najednou pomocí příkazu `r2` nebo jednotlivě pomocí jména utility. Jednotlivé utility využívají skupiny knihoven začínající prefixem *libr_*. Knihovna *libr_asm* umožňuje assemblování a disassemblování instrukcí. Obsahuje disassemblery pro řadu architektur, někdy i více než jeden na architekturu. Některé disassemblery pocházejí z knihoven jiných projektů, jiné jsou frameworku vlastní.

K assemblování a disassemblování je třeba vytvořit strukturu `RAsm`, která obsahuje informace o zvolené architektuře, endianitě a dalšího pomocných informací. Jednotlivé instrukční sady jsou implementovány jako pluginy, které je třeba poskytnout řídicí struktuře a dále vybrat, který má být použit. Disassemblování pak může probíhat pro celé buffery kódu nebo po jednotlivých instrukcích. Instrukce jsou uloženy do struktury `RAsmOp`, kde jsou uchovány v textové podobě.

[3] https://www.capstone-engine.org/
[4] https://llvm.org/
[5] https://rada.re/n/

	ARM	AVR	interní reprezentace instrukcí	Licence
Capstone	ano	ne	ano, textová	BSD
Libopcodes	ano	ano	ne	GPL3
Radare2	ano	ano	ano, textová	LGPL3

Tabulka 4.1: Srovnání knihoven pro disassemblování

4.3 Existující disassemblery

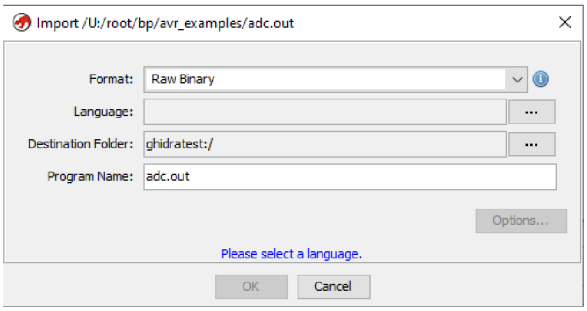
Tato podkapitola se věnuje průzkumu existujících disassemblerů s grafickým uživatelským rozhraním (GUI). Popisuje jejich obecné vlastnosti a dále se zaměřuje na zpracování jejich GUI pro spuštění zpětného překladu a zobrazení výsledného kódu.

Ghidra

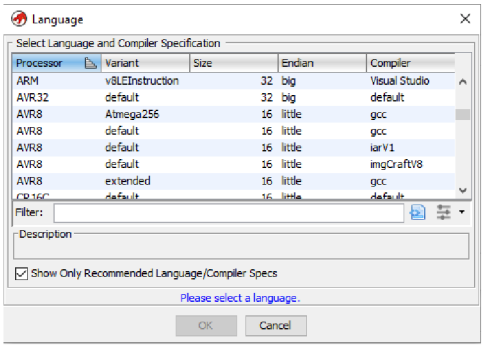
Ghidra[6] je svobodná aplikace pro reverzní inženýrství. Byla vyvinuta jako utajený projekt americké bezpečnostní služby NSA, která ji však v roce 2019 zveřejněna pod open-source licencí. Aplikace je napsána v jazyce Java. Může být použita jako celek s uživatelským rozhraním nebo také jako aplikační rozhraní pro disassemblování a analýzu. Umožňuje psát zásuvné moduly v jazyce Java nebo Python. Obsahuje také dekompilátor naprogramovaný v jazyce C++. Ten nejprve převede disassemblovaný kód do mezikódu *p-code*, který dále analyzuje, a vytváří ekvivalentní reprezentaci programu v jazyce C.

Spuštění zpětného překladu

Po spuštění programu je otevřeno okno pro správu projektů. S projekty lze provádět běžné operace jako např. vytvořit nový, otevřít existující apod. Při importu souboru k disassemblování do projektu je otevřeno dialogové okno pro výběr jeho vlastností. V případě, že program detekuje podporovaný formát, vyplní tabulku vlastností jeho parametry. V opačném případě je zvolen formát *Raw binary* a je nutné, aby uživatel zvolil z dialogového okna kombinaci architektury, její verze, délky slova, endianity a typu překladače. Po potvrzení importu je zobrazeno okno se souhrnnými informacemi. Zpětný překlad je zahájen použitím nástroje *CodeBrowser* na vybraný soubor.



(a) Výběr parametrů souboru



(b) Výběr konkrétního jazyka

Obrázek 4.2: Dialogová okna pro import souboru do projektu

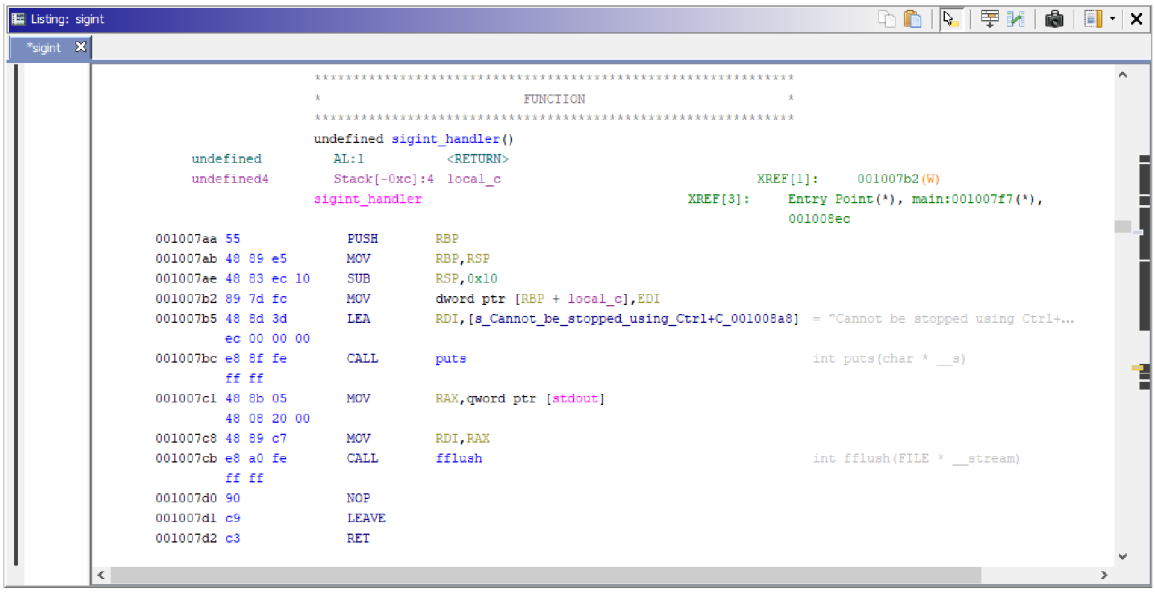
[6] https://ghidra-sre.org/

CodeBrowser

Po disassemblování souboru je uživatel dotázán, zda si přeje provést analýzy kódu. Množství a typy analýz se liší na základě disassemblované architektury. U výpočetně náročných analýz lze nastavit maximální dobu jejich trvání. Dále je zpřístupněno hlavní okno programu. Skládá se z menších oken konkrétních funkcí, které lze libovolně přesouvat a přizpůsobovat jejich velikost.

Disassemblované instrukce jsou zobrazeny spolu s jejich adresou v okně *Listing* (obrázek 4.3). Každý řádek obsahuje adresu, bajty a tělo instrukce. Pokud operandy obsahují odkazy do paměti a program zná jejich obsah, jsou automaticky nahrazeny vhodnými identifikátory, které při dvojkliku přesune pohled na jejich umístění v paměti. V komentářích se nacházejí další dodatečné informace jako jsou například deklarace funkcí, obsahy řetězců apod. Je možné zvýrazňovat části kódu a přidávat záložky.

Nástroj *Byte Viewer* umožňuje zobrazit bajty v paměti v různých formátech. Lze měnit počet zobrazených bajtů na řádek nebo například nastavit zarovnání adresy.



Obrázek 4.3: Ukázka okna listing a jeho formátování

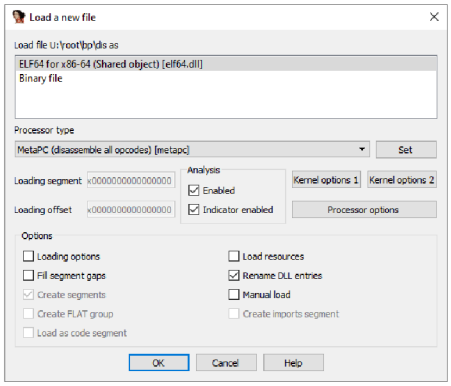
IDA Freeware

Jedná se o proprietární software pro disassemblování a debugování. Patří mezi nejúspěšnější disassemblery vůbec. Disassemblování je postaveno na základě *recursive traversal* algoritmu doplněného o heuristiky pro hledání odkazů na kód a analýzu informací uložených ve formátu souboru. Pro zpřesnění výsledku disassemblování poskytuje rozhraní pro skriptování v jazycích IDC, IdaRUB a IDAPython. Je tak možné například vytvářet automatické komentáře nebo přidávat externí anotace. Poskytuje také plugin pro dekompilaci programů naprogramovaných v jazycích C a C++. Program je dostupný v několika verzích, v této práci je použitá freeware verze programu[7], která je omezena na architekturu x86-64 a nepodporuje všechny formáty souborů.

[7] https://www.hex-rays.com/ida-free/

Spuštění zpětného překladu

Po vybrání souboru k disassemblování je zobrazeno okno k výběru jeho vlastností, vizte obrázek 4.4. Automaticky je detekován formát souboru, lze jej však vždy změnit na neformátovaný vstup. Dále je možno vybrat architekturu konkrétního typu procesoru a upravit další možnosti analýzy.



Obrázek 4.4: Okno pro načtení nového souboru

Prohlížení kódu

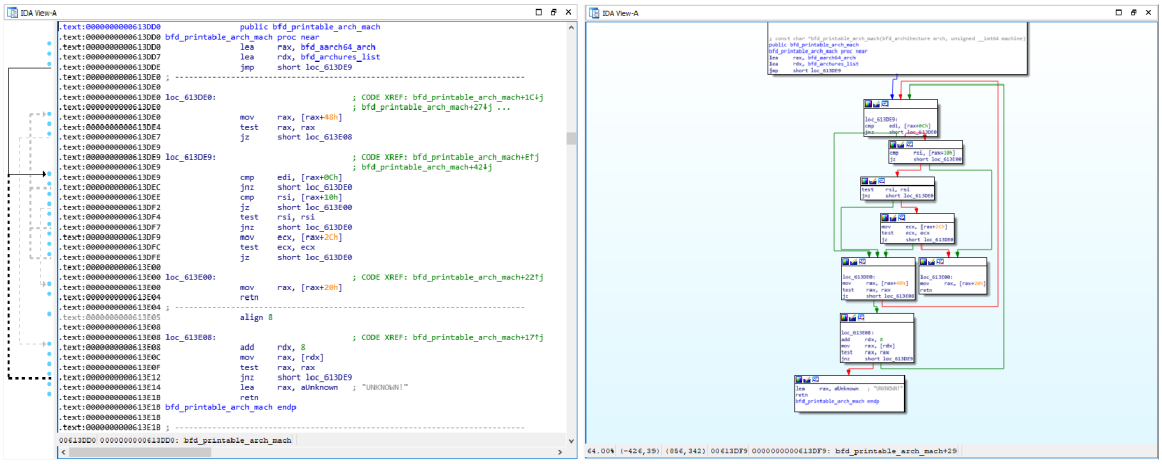
Po potvrzení vybraných možností je soubor disassemblován a zobrazen v hlavním okně. Program obsahuje několik možností prohlížení kódu.

- IDA Text View. Jedná se o typické zobrazení kódu, kdy se na jednom řádku nachází adresa instrukce, její mnemonika, operandy a dodatečný komentář. Program automaticky vytváří návěští pro cíle skokových instrukcí a v levé části okna skoky šipkami naznačuje. Funkce jsou vyznačeny pomocí samostatných komentářů na řádcích (obrázek 4.5a).
- IDA Graph View. Tento pohled umožňuje zobrazit CFG funkce nebo celého programu. Kód je rozdělen na basic blocky, které jsou vykresleny a propojeny barevnými šipkami (obrázek 4.5b). Lze je barevně vyznačit a měnit jejich jména. Změny jsou provedeny také v Text View.
- Hex View. Umožňuje prohlížet obsah souboru jako bajty v podobě páru hexadecimálních čísel a jejich zobrazení jako ASCII znaku.

Je možné mít otevřeno více oken pro prohlížení kódu najednou. Komentáře ke kódu lze přidávat pouze prostřednictvím dialogového okna.

Shrnutí

Oba popsané nástroje poskytují přehledné GUI. Dialog pro spuštění disassemblování nabízí u obou disassemblerů podobnou funkcionalitu, Ghidra však výběr možností rozčleňuje do více celků. Obdobně vypadá také zobrazení disassemblovaných instrukcí. Rozdílem tak zůstává pouze členění kódu, kdy IDA kód rozděluje na jednotlivé basic blocky, Ghidra pak zobrazuje kód funkcí souvisle.



(a) Text View (b) Graph View

Obrázek 4.5: Příklad zobrazení funkce v IDA View

4.4 Analýza požadavků

Požadovaná aplikace má být schopna diassemblovat binární soubory architektur ARM a AVR. Měla by usnadnit spuštění zpětného překladu do jazyka symbolických instrukcí a snadnou orientaci v něm. Uživatel by měl mít možnost doplnit anotace a komentáře kódu. Aplikace by měla být snadno rozšiřitelná o algoritmy řízení zpětného překladu, disassemblery dalších architektur, analýzy kódu a případně i emulátor.

4.5 Východiska pro návrh

Cílem této práce tedy bylo implementovat disassembler s uživatelským rozhraním, který k disassemblování instrukcí využije již existující knihovnu a půjde snadno rozšířit o další architektury a jejich analýzy.

Z tohoto důvodu bylo nezbytné navrhnout univerzální struktury pro uchování jednotlivých částí disassemblovaného programu, které nejsou vázány na konkrétní architekturu, tak, aby bylo možné s nimi pracovat do určité úrovně hromadně. Stejně tak bylo třeba oddělit řízení disassemblování od zpětného překladu jednotlivých instrukcí. Je tak umožněno implementovat různé algoritmy pro statické disassemblování bez nutnosti zasahovat do disassemblerů instrukcí jednotlivých architektur.

Pro snazší a rychlejší provádění analýz byly pro data instrukcí vytvořeny interní reprezentace, aby nedocházelo k opakovanému parsování textu s instrukcemi. To by bylo jednak náročnější na výkon a také náchylnější k chybám.

4.6 Výběr technologií

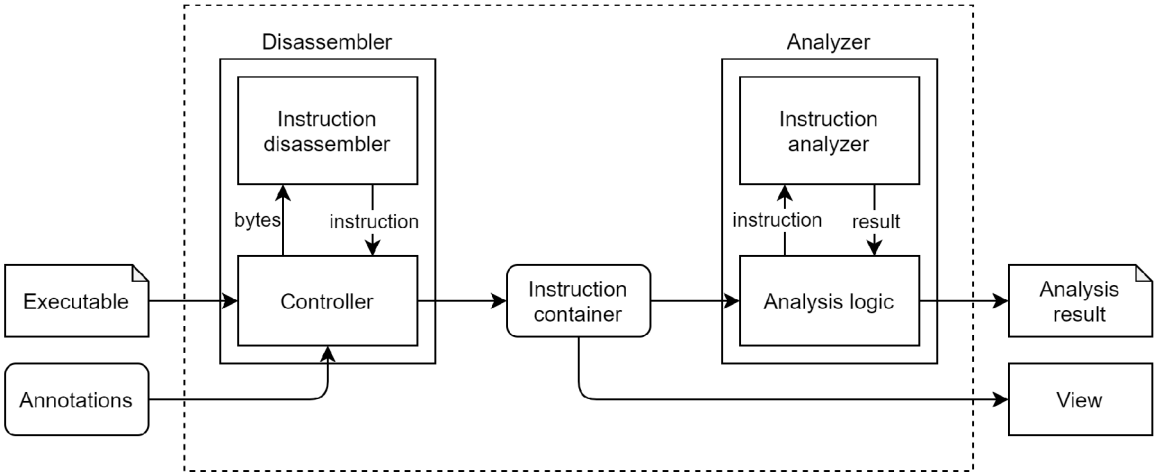
Po zvážení předností všech studovaných možností byla k práci se spustitelnými soubory vybrána knihovna *libbfd* a k disassemblování knihovna *libopcodes*. Jsou spolu velmi provázané a umožňují pracovat s největším množstvím formátů a architektur, tudíž nabízí širší možnosti pro další vývoj aplikace. Navíc obě pocházejí ze stejného projektu a je tak postačující instalace pouze jedné sady nástrojů.

Z knihovny *libopcodes* jsou použity zdrojové soubory jako základ pro disassemblery jednotlivých architektur tak, aby byly generovány rovnou instrukce v interní reprezentaci, nikoliv pouze jejich text, který by bylo třeba dále parsovat. Tento způsob má však za následek, že není možné jednoduše aktualizovat kód disassembleru. Nicméně disassemblery implementovaných architektur se již zásadně nemění, případné drobné změny by neměly být příliš náročné.

4.7 Architektura aplikace

Aplikace je rozdělena do několika částí znázorněných na obrázku 4.6. První je modul disassembleru, který na základě poskytnutých anotací a informací ve formátu souboru disassembluje daný spustitelný soubor. Disassembler se skládá z řízení zpětného překladu a disassembleru instrukcí. Modul řízení zpětného překladu umožňuje implementaci algoritmů pro řízení statického disassemblování, disassembler instrukcí vytváří na základě obdržených bajtů instrukce dané architektury. Výstupem modulu jsou disassemblované instrukce spolu s dalšími doplňujícími informacemi.

Druhou částí je modul pro provádění analýz, který výstup disassembleru analyzuje na základě logiky analýzy. Jednotlivé instrukce jsou analyzovány pomocí analyzátoru instrukcí, jehož výstupy jsou využity k tvorbě celkového výstupu analýzy.



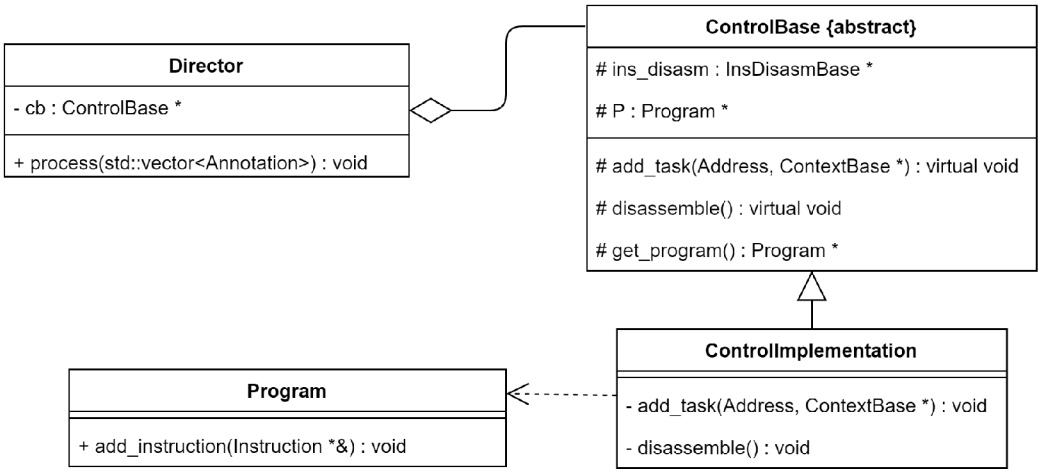
Obrázek 4.6: Diagram architektury aplikace

4.8 Návrh modulu disassembleru

Pro návrh řízení disassemblování byl vybrán návrhový vzor *builder*. Konkrétní implementace algoritmu pro řízení disassemblování bude plnit instrukcemi třídu `Program`. Bázová třída pro řízení zpětného překladu `ControlBase` implementuje společné metody pro inicializaci disassembleru a deklaruje metody pro přidávání adres ke zpracování a spuštění algoritmu. Je třeba zajistit, aby operace probíhaly ve správném pořadí, proto je celý proces řízen třídou `Director`.

Obrázek 4.7 znázorňuje návrh diagramu tříd s nezbytnými proměnnými a metodami pro tento modul. Jednotlivé instrukce třídy `Instruction` jsou získávány skrze objekt di-

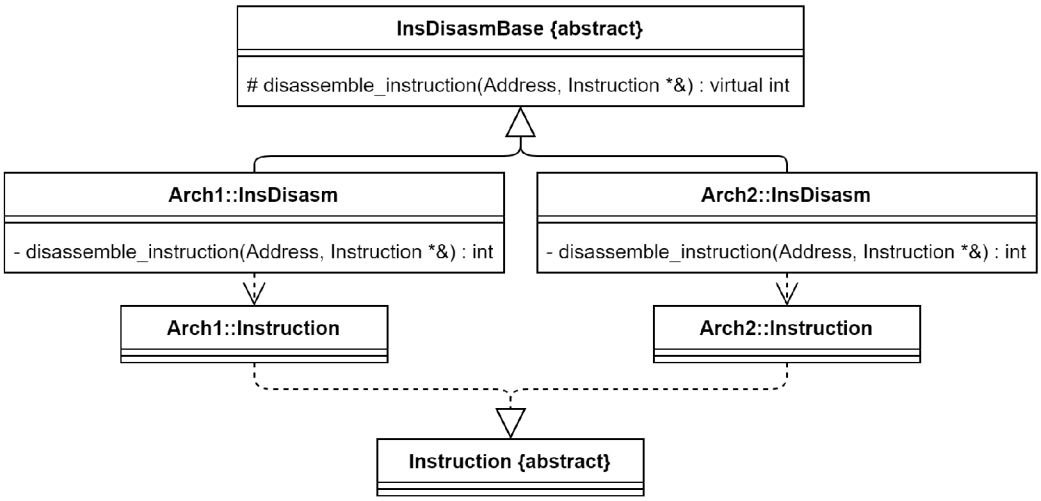
sassembleru instrukcí `ins_disasm`. Třída `ControlImplementation` symbolizuje konkrétní implementaci algoritmu pro řízení disassemblování.



Obrázek 4.7: Diagram tříd pro řízení zpětného překladu

Disassembler instrukcí bude generuje objekty instrukcí na základě daných dat. Instrukce se svou strukturou liší v závislosti na disassemblované architektuře. Proto byl pro návrh disassembleru instrukcí vybrán návrhový vzor *abstract factory*. Zpětný překlad probíhá skrze ryze virtuální metodu `disassemble_instruction` abstraktní třídy `InsDisasmBase`. Ta tvoří bázovou třídu pro jednotlivé disassemblery, které vytváří objekty instrukcí konkrétní architektury založené na abstraktní třídě `Instruction`.

Na obrázku 4.8 je diagram tříd odpovídající popsanému chování disassembleru instrukcí. Jmenné prostory `Arch1` a `Arch2` nahrazují jména konkrétních implementací architektur.



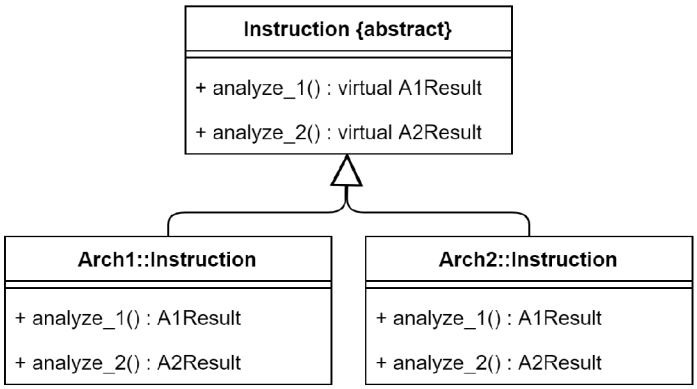
Obrázek 4.8: Diagram tříd pro disassembler instrukcí

Návrh třídy Program

Třída `Program` uchovává disassemblované instrukce, tabulku symbolů a další informace o souboru. Poskytuje také data pro modul analyzátoru a grafické uživatelské rozhraní. In-

strukce abstraktní třídy `Instruction` jsou uloženy v kontejneru podle svých adres, protože na jedné adrese souboru se nemůže nacházet více odlišných instrukcí. Je však třeba uchovávat pouze ukazatele na instrukce, aby bylo možné díky polymorfismu pracovat stejnými metodami s implementacemi instrukcí různých architektur.

Tabulka symbolů třídy `SymbolTable` obsahuje symboly na jednotlivých adresách obdržených v anotacích, nebo vytvořených automaticky modulem disassembleru. Symboly obsahují své jméno a typ.

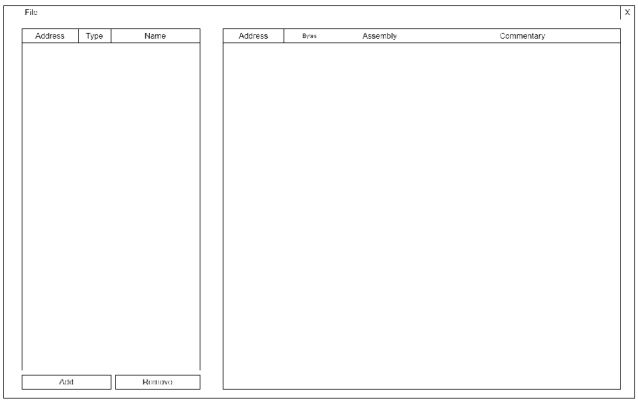


Obrázek 4.9: Diagram tříd pro analyzátor instrukcí

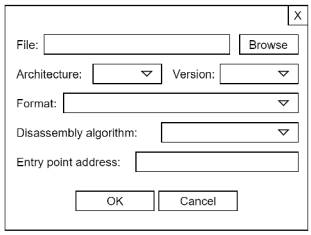
4.9 Návrh modulu analyzátoru

Modul pro logiku jednotlivých analyzátorů provádí na základě obsahu třídy `Program` danou analýzu za použití analyzátoru instrukcí. Analýzy mohou být velmi odlišné, proto není použit žádný pevný formát analyzátoru. Analýzy instrukcí jsou implementovány jako virtuální metody třídy `Instruction` vracející výsledek definovaný danou analýzou. Každá implementace architektury pak poskytuje vlastní způsob pro získání výsledku.

Obrázek 4.9 obsahuje diagram tříd `Instruction` a jejích konkrétní implementací architekturami `Arch1` a `Arch2` umožňující analýzy `analyze_1` a `analyze_2` s vracející dané výsledky.



(a) Hlavní okno



(b) Dialog pro spuštění disassemblování

Obrázek 4.10: Makety pro GUI

4.10 Návrh uživatelského rozhraní

Aplikace poskytuje grafické uživatelské rozhraní (GUI) a možnost spustit zpětný překlad pomocí parametrů z příkazové řádky. Je tak ušetřen čas uživatele při opakovaném otevírání souborů se stejnými parametry.

Hlavním prvkem GUI hlavní okno na obrázku 4.10a, které slouží k zobrazení výsledku disassemblování a je rozdělena na dvě části. V levé se nachází tabulka se symboly, v pravé pak jsou zobrazeny řádky s disassemblovanými instrukcemi a jejich komentáři. Skrze GUI je možné upravovat komentáře, přidávat a odebírat symboly. Dále umožňuje spuštění disassemblování pomocí dialogového okna na obrázku 4.10b.

Kapitola 5

Implementace a testování

Tato kapitola se věnuje implementaci disassembleru podle vytvořeného návrhu s využitím popsaných vědomostí.

5.1 Prerekvizity

Aplikace je naprogramována v jazyce C++20. K překladu je třeba nainstalovat překladač podporující tento standard. V našem případě byl použit GNU C++ compiler verze 10.3.0.

K získání knihoven *libbfd*, *libopcodes* a *libiberty* je třeba přeložit GNU Binutils verze 2.36.1, které jsou dostupné ke stažení na oficiálních stránkách projektu. Není třeba překládat všechny součásti kolekce, postačí přeložit pouze zmíněné knihovny. Pro usnadnění překladu byl vytvořen skript *install_binutils.sh*, který do dočasné složky stáhne a extrahuje daný archiv, přeloží potřebné knihovny a nainstaluje jejich obsah do adresáře binutils ve složce aplikace. Dočasnou složku následně odstraní ze systému. K překladu je dále nutné mít v systému přítomné knihovny *zlib* a *libdl*. Moduly uživatelského rozhraní požadují aplikační rámec Qt 5.9.5 a program *qmake*.

5.2 Třída Program

Třída `Program` je vytvořena pro uchování informací o binárním souboru a výsledku disassemblování. Obsahuje jméno disassemblovaného souboru, proměnnou `target` se jménem formátu souboru, dále pak jméno architektury a adresu vstupního bodu do programu. V proměnné `algh` se nachází informace o zvoleném algoritmu pro disassemblování. Symboly vytvořené na základě anotací jsou uchovávány v tabulce symbolů `symtab`. Třída obsahuje také kontejner s disassemblovanými instrukcemi třídy `Instruction` datového typu `std::map`. Poskytuje rozhraní pro práci se symboly a instrukcemi včetně jejich zobrazení.

Třída SymbolTable

Tabulka symbolů `SymbolTable` umožňuje přiřadit adresám symboly, určit jejich typ, pojmenovat je a dále s nimi manipulovat. Podporovanými typy symbolu jsou funkce, proměnná nebo návěští. Pokud se pak nacházejí v instrukcích adresy do paměti, je možno je vyměnit za jména symbolů. Pro získání dat k vykreslení slouží metoda `to_lines`.

5.3 Třída Instruction

Třída `Instruction` byla vytvořena jako bázová třída pro implementace tříd instrukcí jednotlivých architektur a zapouzdření jejich analýz na instrukční úrovni. Obsahuje společná data pro všechny architektury, jako jsou adresa, vektor bajtů a komentář. Poskytuje rozhraní k převedení instrukce do podoby struktury `InstructionLine`, která nese data k zobrazení instrukce, pomocí metody `to_line`. Ta využívá mimo jiné ryze virtuální metodu `ins_to_str`, která slouží k získání řetězce disassemblovaného kódu konkrétní instrukční sady. Při volání této metody je předáván odkaz na tabulku symbolů `SymbolTable`, aby mohly být případné odkazy do paměti přeloženy na jejich jména.

Ryze virtuální metoda `analyze_cf` slouží k zjištění vlivu instrukce na tok programu. Ten symbolizuje výčtový datový typ `FlowControl`. Tato analýza je vhodná pro tvorbu důležitá pro implementaci *recursive traversal* algoritmu nebo při tvorbě CFG.

Podtřídy instrukční sady ARM

Architektura ARM obsahuje více instrukčních sad a umožňuje práci s koprocesory. Tato implementace podporuje disassemblování pouze instrukčních sad ARM, Thumb a Thumb2. Reprezentace jejich instrukcí je založena na třídě `Arm::InsBase`, která sdružuje jejich společné vlastnosti. Obsahuje informaci o podmíněnosti vykonávání kódu, vektor operandů, příznaky predikovatelnosti a validity operandů. Konkrétní implementace podtříd instrukcí pak obsahují vlastní výčtové třídy `Code`, které obsahují seznam všech dostupných instrukcí, a jimiž jsou jednoznačně identifikovány. Instrukce instrukčních sad Thumb a Thumb2 obsahují navíc specifikaci jejich šířky.

Pro operandy je vytvořena třída `Arm::Op`, která jejich typ uchovává v proměnné výčtového datového typu `Arm::Op::Type`. Jeho výčet neodpovídá přímo specifikaci podle instrukční sady architektury, nicméně v této podobě napomáhá snazší interpretaci. Data operandu mohou mít mnoho podob, ale vždy bude v závislosti na typu vybrána pouze jedna. Proto se jednotlivé datové proměnné nacházejí v anonymní unii. Pro některé z nich bylo třeba vytvořit komplexní struktury, např. pro popis adresování nebo posunu registru.

Metoda `analyze_cf` je implementována pro každou instrukční sadu zvlášť. U architektury ARM může být analyzování toku programu problematické, a tak jsou implementovány pouze překladači užívané konstrukce.

Podtřída Avr::Ins

Podtřída `Avr::Ins` je implementací interní reprezentace instrukcí architektury AVR. Obsahuje kód instrukce výčtové třídy `Code`, který obsahuje kódy všech dostupných instrukcí. Dále obsahuje dva operandy, které v závislosti na dané instrukci nemusí být využity, pro snížení paměťové náročnosti jsou však napevno do vloženy.

Operand reprezentován třídou `Avr::Op` obsahuje svůj typ a hodnotu. Typ je dán výčtem všech druhů operandů popsaných instrukční sadou architektury. Hodnotami jsou vhodně zvolené datové typy pro reprezentaci daného typu v anonymní unii.

Analýza instrukcí z pohledu řízení toku programu metodou `analyze_cf` je prováděna porovnáváním kódu instrukce se skupinami kódů instrukcí sledovaných vlastností pomocí dalších privátních metod.

```
class Avr::Op {
    enum class Type {
        EMPTY, IMMED, PORT, REG, ADDR, PTR_REG,
        PTR_REG_DIS, PC_REL_OFFSET,
        BAD
    };

    Type type;
    union {
        uint i;
        bfd_vma addr;
        PtrReg ptr_reg;
        struct {
            PtrReg ptr_reg;
            uint dis;
        } ptr_reg_dis;
        int rel_addr;
    };
};
```

```
class Avr::Ins {
    enum class Code {
        ADC, ADD, ... ,
        XCH, UNDEF_INS
    };

    Code ins_code;
    Op op1;
    Op op2;
};
```

Obrázek 5.1: Ukázka implementace struktur pro operand a instrukci architektury AVR

5.4 Třída ControlBase

Třída `ControlBase` je bázovou třídou pro implementaci řízení disassemblování. Její objekt lze získat statickou metodou `create` se zvoleným algoritmem jako parametrem. Tento objekt je dále třeba předat třídě `Director`, která poskytuje rozhraní pro práci s disassemblerem. Pomocí metody `open` proběhne inicializace, metodou `process` pak zpracování anotací a disassemblování.

K práci s binárními soubory využívá knihovnu *bfd*. Ta umožňuje jejich otevření, rozpoznání formátu a manipulaci s obsahem. Otevřen souboru daného formátu je implementováno metodou `open_bfd`. Pokud formát nebyl specifikován, je automaticky detekován funkcí `bfd_check_format`. Poté proběhne výběr disassembleru instrukcí, buď na základě zadané architektury, nebo automaticky podle informací uložené v hlavičce formátu. Dále je inicializován disassember instrukcí a následuje extrahování spustitelných sekcí ze souboru do struktur `SectionData`.

Pro získání jednotlivých sloupců do tabulky anotací a jejich povolených formátů ve formě regulárního výrazu třída obsahuje metodu `get_annotation_columns`. Ta využívá statickou metodu třídy `SymbolTable`, která vrací sloupce nutné pro přidání symbolu, a metodu `get_context_columns` disassembleru instrukcí pro získání sloupců kontextu pro disassemblování adresy. K vložení anotace slouží metoda `add_annotation`, která ji parsuje a přidá do objektu programu `P`. Pokud se jedná o anotaci funkce nebo návěští, je její adresa a kontext zařazena do fronty ke zpracování ryze virtuální funkcí `add_task`. Případné vstupní body nevhodného formátu jsou přeskočeny.

Ke spuštění disassemblování slouží ryze virtuální metoda `disassemble`. Výsledná data v podobě třídy `Program` umožňuje získat metodou `get_program`.

Podtřída LinearSweep

Tato podtřída implementuje algoritmus *linear sweep* popsaný v kapitole 2.6. Od vstupního bodu disassembluje přímočaře všechny následující spustitelné sekce. Pokud je prostřednictvím anotací určen počátek kódu na adrese menší, než aktuální vstupní bod, je vstupní bod nahrazen adresou v anotaci. Získané instrukce jsou ukládány do kontejneru instrukcí třídy `Program` pomocí funkce `push_instruction`. Pokud se mezi instrukcemi nachází instrukce přímého volání, jsou do tabulky symbolů přidány symboly funkcí volaných cílů.

Podtřída RecursiveTraversal

Jedná se o implementaci algoritmu *recursive traversal* (viz kapitola 2.6). Obsahuje frontu adres s jejich kontexty `tasks`, ze které jsou postupně odebírány a zpracovávány jednotlivé úlohy. Do fronty jsou během disassemblování přidávány další detekované počátky basic blocků na základě výsledku analýzy vlivu instrukce na tok programu metodou `analyze_cf`. Skrze anotace lze přidat další úlohy ke zpracování.

5.5 Třída InsDisasmBase

Třída `InsDisasmBase` je bázovou třídu pro disassemblery instrukcí. Je přizpůsobena pro práci s extrahovanými a upravenými disassemblery z knihovny *opcodes*. Využívá její strukturu `disassemble_info` při čtení dat z alokovaných sekcí binárního souboru. Inicializace disassembleru proběhne pomocí virtuální funkce `init`, ve které mohou být např. na základě zvolené verze architektury vyřazeny nepodporované instrukce.

Některé architektury mohou ke správnému disassemblování požadovat dodatečné informace. Aby bylo možné tyto informace předávat nezávisle na algoritmu disassemblování, je vytvořena struktura `ContextBase`. Nemá žádný obsah, jedná se čistě o bázi pro struktury kontextu implementované konkrétními architekturami. Pokud daná architektura práci v kontextu nepotřebuje, je předáván `nullptr`. Pro získání sloupců pro požadovaný kontext slouží metoda `get_context_columns`, ke zpracování pak metoda `parse_context`.

K získávání disassemblovaných instrukcí jako objektů třídy `Instruction` slouží ryze virtuální funkce `disassemble_instruction`, kterou je nutné implementovat pro každý disassembler.

Podtřída Arm::InsDisasm

Podtřída Arm::InsDisasm implementuje disassemblování instrukcí architektury ARM. Využívá možnosti disassemblování ve zvoleném kontextu struktury `Context`. Ta obsahuje proměnnou `state` reprezentující aktuální nastavení Thumb-bitu. Inicializace disassemblování metodou `init` přizpůsobí prohledávaný seznam instrukcí podle zvolené verze architektury. Disassemblování instrukce je zahájeno metodou `disassemble_instruction`, která v závislosti na aktuálním stavu Thumb-bitu v proměnné `context` zvolí instrukční sadu a její disassembler. V případě instrukční sady Thumb je zvolena její verze a k ní vybrána patřičná délka instrukce. Disassembler je založen na kódu disassembleru ARM knihovny *libopcodes*.

Podtřída Avr::InsDisasm

Disassembler instrukcí pro architekturu AVR je poměrně jednoduchý. Metoda `init` vytvoří masku dostupných instrukcí pro danou verzi architektury. Při disassemblování instrukce

je nejprve ve vektoru instrukcí `opcodes` nalezen kód instrukce, poté jsou pomocí metody `operands_decoder` získány hodnoty a typy operandů. Původní kód metod pochází z disassembleru pro AVR knihovny *libopcodes*.

5.6 Třída CFG

Třída `CFG` slouží k vytvoření CFG programu na základě daného vstupního bodu a kontejneru s instrukcemi. Výsledkem analýzy je kontejner objektů třídy `BasicBlock`, které reprezentují basic blocky. Tvorba probíhá postupným zpracovávání zásobníku `todo` prvků privátní struktury `Task`. Ta obsahuje adresu nového basic blocku a množinu basic blocků, které jsou jejími předchůdci.

Basic block je vytvářen tak, že od jeho počáteční adresy jsou instrukce postupně přidávány do jeho těla `body` a následně analyzovány, z pohledu jejich vlivu na řízení toku programu metodou `analyze_cf`. V případě, že se jedná o instrukci

1. bez vlivu na tok programu — je pokračováno další instrukcí,
2. volání — je cíl přidán do fronty ke zpracování a je pokračováno další instrukcí,
3. nepodmíněného skoku — je cíl přidán do fronty ke zpracování a ukončen aktuální basic block,
4. podmíněného skoku — je cíl a adresa následující instrukce přidána do fronty ke zpracování a ukončnen aktuální basic block,
5. návratu z procedury — je ukončen aktuální basic block.

U každého basic blocku je uchovávána množina jeho předchůdců i následníků. Pokud se začátek basic blocku nachází v již existujícím basic blocku, existující je rozdělen v místě začátku nového, následníci existujícího jsou předáni novému, nový se stává jediným následníkem existujícího a existující se stává předchůdcem nového.

```
class CFG {
    struct Task {
        bfd_vma addr;
        std::set<bfd_vma> pre;
    };

    std::vector<Task> todo;
    std::map<bfd_vma, BasicBlock> bbs;
};
```

```
class BasicBlock {
    bfd_vma addr;
    std::set<bfd_vma> pre;
    std::vector<Instruction *> body;
    std::set<bfd_vma> succ;
};
```

Obrázek 5.2: Třídy CFG a BasicBlock

5.7 Uživatelské rozhraní

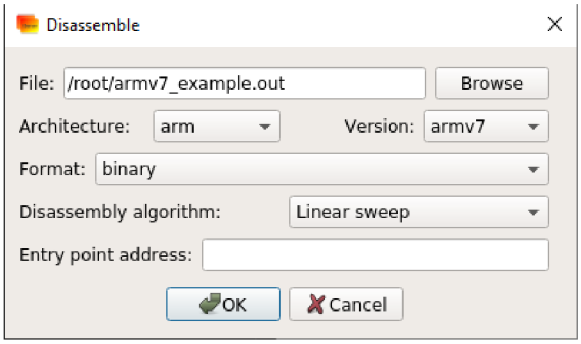
Spuštění zpětného překladu

Dialog pro spuštění disassemblování je implementován třídou `StartDisasmDialog` založenou na třídě `QDialog`. Umožňuje vybrat soubor k disassemblování a jeho vlastnosti. Cestu k souboru je možné vložit jako text nebo vybrat pomocí dialogového okna `QFileDialog`.

K výběru architektury je využit `QComboBox`. Pokud architektura není uživatelem vybrána, je použita architektura z formátové hlavičky. Konkrétní verze architektury pak může být specifikována v combo boxu pro výběr verzí, který je aktuální vždy pro zvolenou architekturu. Nabízené možnosti jsou získány metodou `get_versions` vybrané architektury třídy `Architecture`.

Volba formátu je rovněž implementována pomocí combo boxu, jehož možnosti se mění v závislosti na zvolené architektuře. Pokud je ponechán prázdný, je formát souboru automaticky detekován. Pomocí třídy `QComboBox` je také implementován výběr algoritmu, jejichž seznam je získán z globální proměnné `Disasm::algorithms`. Pole pro vložení adresy vstupního bodu očekává hexadecimální číslo s hlídanou délkou osmi znaků. Pokud zůstane nevyplněno, je použita hodnota z formátu souboru, nebo pokud soubor není formátován, začíná disassemblování od adresy 0x0.

Po potvrzení je spuštěno disassemblování podle daných parametrů. Pokud dojde k chybě, je zobrazeno chybové okno s jejím popisem.



Obrázek 5.3: Dialogové okno pro spuštění zpětného překladu

Hlavní okno

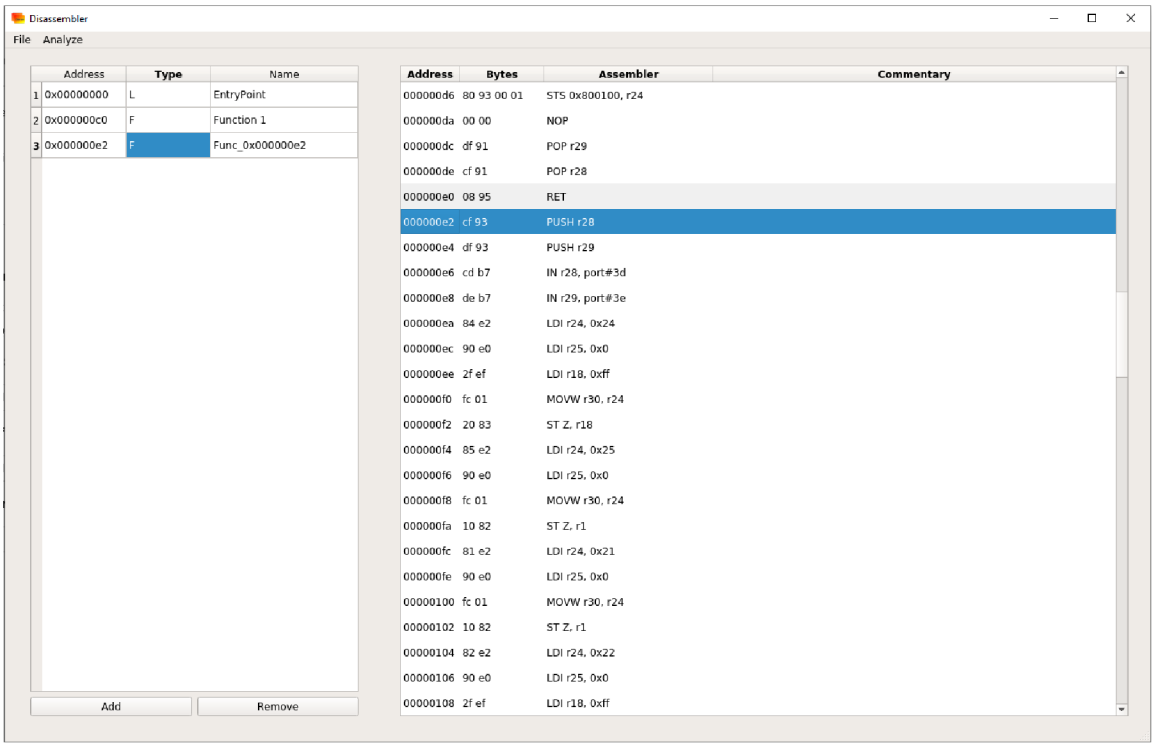
Data k zobrazení instrukcí jsou získány metodou `get_ins_lines` třídy `Program` jako vektor struktur `InstructionLine`. Tato struktura obsahuje textové řetězce jednotlivých položek instrukce a dále doplňující informaci o tom, zda se jedná o řídicí instrukci. Řádky s řídicími instrukcemi jsou podbarveny šedou barvou pro větší přehlednost. K zobrazení disassemblovaného kódu spolu s dalšími informacemi je využit `QTableWidget`.

Tabulka k zobrazení symbolů je naplněna daty z tabulky symbolů `symtab` uložené v aktuální třídě `Program` pomocí metody `get_sym_lines`. Jednotlivé symboly jsou vloženy na vlastní řádek tabulky. Lze editovat jejich adresy a jména, přičemž změny jsou následně zaneseny do zdrojové tabulky symbolů. Při kliknutí na jméno nebo typ symbolu je pohled v tabulce s instrukcemi přenesen tak, aby zobrazoval jeho adresu (pokud se v programu nachází). Přidat další symboly do tabulky je možné buď pomocí tlačítka `Add`, dvojklikem

na adresu instrukce nebo dvojklikem na buňku s disassemblovanou instrukcí, ze kterého je případně vybrána hodnota operandu s adresou.

Přidání je provedeno pomocí nově otevřeného dialogového okna třídy `AnnotationDialog` založeném na třídě `QDialog`. Umožňuje přidat anotace uložené v souboru, nebo jejich ruční zadání pomocí tabulky, která obsahuje sloupce pro adresu, typ a jméno symbolu. Pokud daná architektura požaduje vyžaduje pro disassemblování další informace (např. aktuální nastavní Thumb-bitu u ARM), jsou přidány další sloupce, které je nutné vyplnit pro disassemblování návěšní nebo funkcí. Po potvrzení výběru jsou anotace předány disassembleru, který je vloží do tabulky symbolů a případně disassembluje. Nevalidní anotace jsou přeskočeny.

V záložce *File* je možné spustit disassemblování nového souboru nebo exportovat stávající výsledek v textové podobě do souboru. V záložce *Analyze* se nachází možnost vytvořit CFG programu.



Obrázek 5.4: Výsledná podoba GUI hlavního okna.

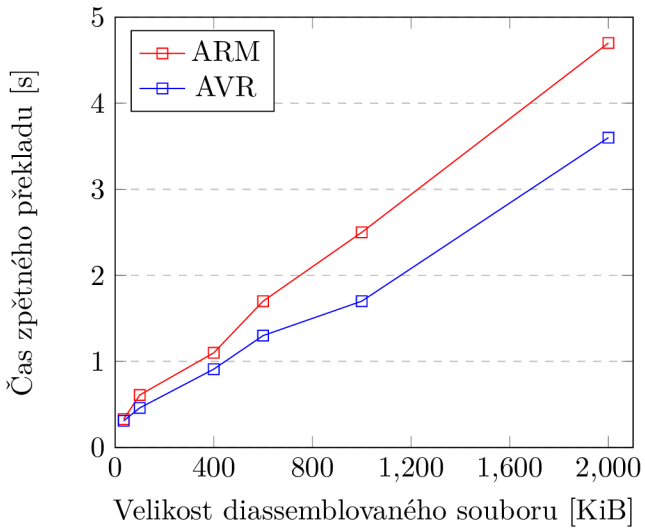
5.8 Testování

Testování aplikace probíhalo ve dvou fázích. Nejprve byla testována správnost zpětného překladu. K testování byla použita sada neformátovaných binárních programů pro mikrokontroléry implementovaných architektur. Aplikace byla spouštěna s parametry dané architektury, formátem *binary*, algoritmem *linear sweep* a vstupním bodem 0x0. Výstup aplikace pak byl porovnáván s výstupem lineárního disassembleru *objdump* z GNU Binutils, který byl spuštěn s obdobnými parametry.

Druhou částí bylo testování rychlosti zpětného překladu. Bylo prováděno ve Windows subsystému pro Linux verze 2 na procesoru Intel Core i7-3632QM s omezeným množstvím

4 GB paměti. K měření časových údajů byl použit program *time*. Měřen byl čas od spuštění programu prostřednictvím příkazové řádky po čas ukončení programu funkcí *exit* umístěné v těsné blízkosti za dokončením zpětného překladu.

Z výsledků v grafu na obrázku 5.5 je patrné, že program funguje rychle pro menší soubory. Se zvětšující se velikostí souborů roste čas jejich zpracování víceméně linerně. Dále je patrné, že disassembler architektury AVR pracuje rychleji než disassembler architektury ARM i přesto, že naprostá většina instrukcí architektury AVR má poloviční velikost oproti instrukcím architektury ARM a to znamená, že musí disassemblovat větší množství instrukcí. Program zvládá pracovat i s většími soubory, nicméně doba jejich zpracování je o poznání delší.



Obrázek 5.5: Graf závislosti času zpětného překladu na velikosti disassemblovaného souboru pro disassembery různých architektur.

Kapitola 6

Závěr

Tato práce se zabývala problematikou zpětného překladu do jazyka symbolických instrukcí a jejich analýzou. Věnovala se průzkumu již existujících řešení a na základě získaných poznatků byl vytvořen objektově orientovaný návrh aplikace s využitím návrhových vzorů. Návrh umožňuje snadné přidání dalších algoritmů pro řízení disassemblování a disassemblerů instrukcí architektur. Na rozdíl od různých existujících řešení uchovává instrukce v interní podobě, což může značně usnadnit jejich analýzu a tím usnadnit další rozvoj aplikace.

Implementovaná aplikace poskytuje dva algoritmy pro řízení disassemblování a disassemblery pro architektury ARM a AVR. Nabízí grafické uživatelské rozhraní pro usnadnění spuštění překladu, přehledný výpis výsledku a editaci anotací. Umožňuje přidávat komentáře k disassemblovaným instrukcím a snadno vytvářet symboly z jejich adres. Dokáže vytvořit CFG programu a uložit jej do souboru.

Prostor pro další vývoj aplikace je široký. Bylo by možné aplikaci rozšířit o podporu dalších architektur nebo implementovat další algoritmy pro řízení disassemblování, například některé z ostatních popsaných v této práci. Dále by bylo možné implementovat automatické zavádění symbolů z tabulek symbolů souboru do interní tabulky symbolů programu. Rovněž by do ní mohly být automaticky zaneseny známé proměnné vybraných architektur. Aplikaci by bylo možné obohatit o další analýzy a implementovat grafické prostředí, ve kterém by jejich výsledek mohl být zobrazen a editován.

Literatura

[1] Allen, F. E. Control Flow Analysis. In: *Proceedings of a Symposium on Compiler Optimization* [online]. New York, NY, USA: Association for Computing Machinery, 1970, s. 1–19 [cit. 2021-04-10]. DOI: 10.1145/800028.808479. ISBN 9781450373869. Dostupné z: https://doi.org/10.1145/800028.808479.

[2] Alves Foss, J. a Song, J. Function Boundary Detection in Stripped Binaries. In: *Proceedings of the 35th Annual Computer Security Applications Conference* [online]. New York, USA: Association for Computing Machinery, 2019, s. 84–96 [cit. 2021-04-23]. ACSAC '19. DOI: 10.1145/3359789.3359825. ISBN 9781450376280. Dostupné z: https://doi.org/10.1145/3359789.3359825.

[3] ARM Limited. *ARM® Compiler v5.06 for µVision®* [online]. 5. 2015 [cit. 2021-04-20]. Dostupné z: https://developer.arm.com/documentation/dui0379/latest.

[4] ARM Limited. *Procedure Call Standard for the Arm Architecture* [online]. 2020 [cit. 2021-04-20]. Dostupné z: https://developer.arm.com/documentation/ihi0042/latest/.

[5] Ayyangar, A. *Static Disassembly of Stripped Binaries* [online]. New York, USA, 2010. [cit. 2021-04-10]. Thesis. Stony Brook University. Dostupné z: https://ir.stonybrook.edu/xmlui/bitstream/handle/11401/70937/Ayyangar_grad.sunysb_0771M_10271.pdf.

[6] Bao, T., Burket, J., Woo, M., Turner, R. a Brumley, D. BYTEWEIGHT: Learning to Recognize Functions in Binary Code. In: *Proceedings of the 23rd USENIX Conference on Security Symposium*. USA: USENIX Association, 2014, s. 845–860. SEC'14. ISBN 9781931971157.

[7] Ben Khadra, M. A., Stoffel, D. a Kunz, W. Speculative disassembly of binary code. In: *2016 International Conference on Compliers, Architectures, and Sythesis of Embedded Systems (CASES)*. ACM, 2016, s. 1–10. ISBN 9781450344821.

[8] Cifuentes, C. a Van Emmerik, M. Recovery of jump table case statements from binary code. *Science of Computer Programming* [online]. 2001, sv. 40, č. 2, s. 171–188, [cit. 2021-04-25]. DOI: https://doi.org/10.1016/S0167-6423(01)00014-4. ISSN 0167-6423. Special Issue on Program Comprehension. Dostupné z: https://www.sciencedirect.com/science/article/pii/S0167642301000144.

[9] Kairajärvi, S. *Automatic identification of architecture and endianness using binary file contents* [online]. 2019. [cit. 2021-04-15]. Thesis. University of Jyväskylä. Dostupné z: https://jyx.jyu.fi/handle/123456789/63543.

[10] Microchip Technology Inc. *AVR® Instruction Set Manual* [online]. DS40002198A. Microchip Technology Inc, May 2020 [cit. 2021-04-15]. Dostupné z: https://ww1.microchip.com/downloads/en/DeviceDoc/AVR-Instruction-Set-Manual-DS40002198A.pdf.

[11] Microchip Technology Inc.. *8-bit AVR® Microcontrollers* [online]. 2021 [cit. 2021-04-15]. Dostupné z: https://microchipdeveloper.com/8avr:start.

[12] Miller, K., Kwon, Y., Sun, Y., Zhang, Z., Zhang, X. et al. Probabilistic Disassembly. In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, 2019-, s. 1187–1198. ISBN 9781728108698.

[13] Zeng, B. *Static Analysis on Binary Code* [online]. 2012 [cit. 2021-04-28]. Dostupné z: https://engineering.lehigh.edu/sites/engineering.lehigh.edu/files/_DEPARTMENTS/cse/research/tech-reports/2012/lu-cse-12-001.pdf.

Příloha A

Obsah přiloženého paměťového média

- `/`
  - `disassembler/` ..... složka aplikace
    - `src/` ..... složka se zdrojovými soubory
    - `ui/` ..... složka s konfiguračními soubory uživatelského rozhraní
    - `disassembler.pro` ..... projektový soubor programu *qmake*
    - `install_binutils.sh` ..... skript pro instalaci knihoven
    - `LICENSE` ..... licence programu
    - `README.md` ..... informace o programu
  - `xbayer09/` ..... složka se zdrojovými soubory této práce v LaTeX
  - `xbayer09.pdf` ..... tato práce ve formátu PDF