VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

ANALÝZA KÓDU V JAZYCE C PRO ÚČELY TESTOVÁNÍ ZPĚTNÉHO PŘEKLADU

ANALYSIS OF C CODE FOR TESTING OF DECOMPILATION

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **VIKTOR DÍTĚ**
AUTHOR

VEDOUCÍ PRÁCE **Ing. PETER MATULA**
SUPERVISOR

BRNO 2017

Zadání bakalářské práce/20127/2016/xditev00

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav informačních systémů Akademický rok 2016/2017

Zadání bakalářské práce

Řešitel: **Dítě Viktor**
Obor: Informační technologie
Téma: **Analýza kódu v jazyce C pro účely testování zpětného překladu**
Analysis of C Code for Testing of Decompilation
Kategorie: Překladače

Pokyny:

1. Studujte problematiku zpětného inženýrství. Zaměřte se na zpětný překlad binárního kódu do vyšší formy reprezentace.
2. Seznamte se s platformou LLVM, překladačem Clang a jeho podporou pro vytváření nástrojů analyzujících kód v jazyce C.
3. Seznamte se se zpětným překladačem společnosti AVG a současnou implementací analýzy kódu v jazyce C pro účely testování zpětného překladu.
4. Navrhněte rozšíření této analýzy, která umožní zpracování konstrukcí v jazyce C, pro které v současném řešení chybí podpora.
5. Po konzultaci s vedoucím implementujte rozšíření navržená v předchozím bodě.
6. Vytvořené řešení důkladně otestujte sadou minimálně padesáti testů. Zhodnoťte svou práci a diskutujte budoucí vývoj.

Literatura:

- E. Eilam: Reversing: Secrets of Reverse Engineering, Wiley 2005, ISBN 978-076457481.
- Popis platformy LLVM [online]. 2015 [cit. 2015-09-01]. Dostupné na URL: <http://www.llvm.org>
- Popis překladače Clang [online]. 2015 [cit. 2015-09-01]. Dostupné na URL: <http://clang.llvm.org/>
- Interní dokumentace společnosti AVG.

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

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

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

Vedoucí: **Matula Peter, Ing.**, UIFS FIT VUT
Konzultant: Zemek Petr, Ing., AVG
Datum zadání: 1. listopadu 2016
Datum odevzdání: 17. května 2017

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

L.S.

doc. Dr. Ing. Dušan Kolář
vedoucí ústavu

Abstrakt

Cílem této práce je rozšíření *aplikačního rámce* pro tvorbu regresních testů o novou funkcionalitu pro analýzu kódu v jazyce *C*. Tento aplikační rámec je vytvořen v jazyce *Python* a pro analýzu zdrojového kódu využívá překladač *clang*. Práce obsahuje popis oboru zpětného inženýrství a *zpětného překladače* společnosti *AVG*. Dále je stručně představena oblast testování software a jazyk *C*. Následuje popis navržených a implementovaných rozšíření. Tato rozšíření jsou předvedena na ukázkových testech. V závěru nalezneme shrnutí výsledků práce.

Abstract

The goal of this thesis is to extend *framework* for creation of regression tests with new functionality for analysis of *C* code. This framework is created in *Python* language and uses *clang* compiler for analysis of source code. The thesis contains description of area of reverse engineering and *decompiler* developed in *AVG* company. Then the area of software testing and *C* language are briefly introduced. Following chapters describe proposed and implemented extensions. These extensions are presented in sample tests. Summary of the results can be found in conclusion.

Klíčová slova

zpětné inženýrství, zpětný překlad, rekonfigurovatelný zpětný překladač, testování software, jazyk C, clang

Keywords

reverse engineering, decompilation, retargetable decompiler, software testing, C language, clang

Citace

DÍTĚ, Viktor. *Analýza kódu v jazyce C pro účely testování zpětného překladu.* Brno, 2017. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Matula Peter.

Analýza kódu v jazyce C pro účely testování zpětného překladu

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Petra Matuly. Další informace mi poskytl Ing. Petr Zemek, Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Viktor Dítě
22. května 2017

Poděkování

Rád bych poděkoval vedoucímu práce Ing. Petru Matulovi za odborné vedení a Ing. Petru Zemkovi, Ph.D. za poskytnuté rady, konzultace a věnovaný čas.

Obsah

Kapitola 1

Úvod

Obor zpětného inženýrství obecně, a v oblasti informačních technologií obzvlášť, stále poskytuje velký prostor k objevování nových poznatků, stejně jako k vývoji nových technologií a nástrojů. Při této průkopnické činnosti přikládají svoji ruku k dílu i vývojáři ze společnosti *AVG*, kteří pracují na zpětném překladači. Zpětný překlad je činností opačnou k překladu dopřednému, se kterým jsou všichni programátoři dobře obeznámeni. Cílem v tomto případě není ze zdrojových kódů získat spustitelnou aplikaci, ale naopak z přeložené aplikace získat zdrojový kód co možná nejvíce podobný tomu, ze kterého aplikace skutečně vznikla. Je zřejmé, že tato činnost je mnohem komplikovanější. Pro ilustraci bychom si mohli představit snahu o slepení rohlíku ze strouhanky a srovnat ji s úsilím, které předtím bylo potřebné k nastrouhání onoho rohlíku. V případě zpětného překladu nám situaci stěžuje hlavně nedostatek informací. Komentáře, jména proměnných a funkcí, původní dekompozice algoritmů ve zdrojových kódech, toto všechno a mnohá jiná data ztrácí po přeložení programu význam a z výsledné aplikace buď nemůžeme vůbec získat, nebo je snaha o jejich rekonstrukci značně náročná.

Vložené úsilí ale má smysl. Například si jistě dovedeme představit situaci, kdy jsme ztratili zdrojové kódy běžící aplikace a vznikne potřeba tuto aplikaci dále vyvíjet. V tomto případě by nám zpětný překlad mohl pomoci, ale bylo by to určitě velmi nerentabilní. Raději pečlivěji zálohujme. Hlavní uplatnění nalezneme ve studiu. Již samotný vývoj nástrojů pro zpětný překlad přináší nový vhled do oblasti vývoje a analýzy software. Z pohledu společností *AVG*, která se zabývá tvorbou antivirových programů je ovšem jistě nejdůležitější poznání vycházející z výsledků zkoumání nebezpečného kódu. Umožňuje jim nahlédnout do mysli nepřítele – tvůrce škodlivého software. Poznat, jaké cesty útoku volí a jakých zranitelností využívá.

Při vývoji *zpětného překladače*, stejně jako kteréhokoliv jiného programu, je samozřejmě potřeba ověřovat jeho funkčnost. Testování typicky probíhá na různých úrovních. Známe například jednotkové či integrační testování. V současné době, kdy se používání agilních metodik stalo v oblasti vývoje software naprostou samozřejmostí, málokterý programátor pochybuje o užitečnosti regresního testování. Ověření toho, že nově přidaná funkcionalita nenarušila funkcionalitu již existující je bezpochyby užitečné a zabraňuje odbočení vývoji do slepé uličky a ztrátám času a prostředků při hledání toho, kde v minulosti nastala chyba. Nijak výjimečný v tomto ohledu není ani vývoj *zpětného překladače* společnosti *AVG*. U tohoto produktu je k usnadnění tvorby regresních testů používán *aplikační rámec* vytvořený v jazyce *Python*. Cílem této práce bylo rozšíření tohoto *aplikačního rámce* o novou funkcionalitu.

Práce je rozčleněna následujícím způsobem. Po tomto úvodu následuje kapitola 2 představující obor zpětného inženýrství. Bližší zaměření je věnováno zpětnému inženýrství v oblasti informačních technologií, nalezneme zde několik příkladů využití reverzního inženýrství v tomto oboru. Zvláštní pozornost je věnována zpětnému překladu. Posléze se v kapitole 3 seznámíme se *zpětným překladačem* vyvíjeným ve společnosti *AVG* a stručně si rozebereme jeho strukturu. Představíme si projekt *LLVM* a také si stručně rozebereme jednotlivé fáze zpětného překladu. V kapitole 4 se podíváme na oblast testování software. Definujeme si některé základní pojmy z této oblasti a rozebereme si motivaci pro zájem o tento obor. Zvláštní pozornost věnujeme oblasti regresního testování. Kapitola 5 nás seznámí s jazykem *C*. Právě výstup *zpětného překladače* v tomto jazyku nás totiž v rámci této práce bude zajímat. Nalezneme zde ovšem skutečně pouze stručný přehled. K podrobnějšímu rozboru konstruktů, pro něž byla přidána podpora do *aplikačního rámce* se ještě vrátíme v pozdějších kapitolách. Kapitole 6 popisuje *aplikační rámec* regresních testů, jeho podobu před započetím této práce a navržená rozšíření. V kapitole 7 rozebereme jednotlivá rozšíření implementovaná v rámci této práce. Zde také nalezneme dříve avizovaný bližší popis prvků jazyka *C*, jichž se dané rozšíření týká. Na testy související s vyvíjeným *aplikačním rámcem* se podíváme v kapitole 8. Rozebereme si jak testování funkčnosti samotného rámce, tak testy, které je s jeho pomocí možné vytvářet. Podrobněji se zaměříme na několik ukázkových testovacích případů. Zjistíme, co je jejich cílem, a jakým způsobem je ho dosaženo. Nakonec si v kapitole 9 celou práci a její výsledky stručně shrneme.

Kapitola 2

Zpětné inženýrství a zpětný překlad

Tato kapitola se na mnoha místech opírá o informace získané z [7].

Zpětné (nebo též *reverzní*) inženýrství je proces opačný klasickému inženýrství. Jeho cílem je získat znalosti o již existujícím produktu. Může se například jednat o hledání procesu, který vedl ke vzniku daného produktu, zkoumání jeho struktury, chování. V některých případech může být cílem této snahy pouhá obnova dokumentace k produktu o němž nám chybí informace, v jiných případech může být záměrem jeho replikace, či zakomponování zajímavých charakteristik jeho výrobního procesu do výrobního procesu jiného produktu. V případě informačních technologií může být cílem například nalezení zdrojového kódu ve vyšším programovacím jazyce, který odpovídá analyzovanému spustitelnému souboru nebo hledání způsobu, jakým obejít ochrany zabudované do programu.

Motivací k této činnosti může být konkurenční boj společností zabývajících se podobnou oblastí trhu. Velké uplatnění nalezneme také ve vojenství, kde by neschopnost držet krok s technologickým vývojem protivníka mohla být pro danou armádu fatální. I některé oblasti standardního vědeckého výzkumu mohou mít v podstatě charakteristiku reverzního inženýrství. Jako příklad uveďme zkoumání struktury DNA živých organismů.

Právní otázky související s oblastí zpětného inženýrství jsou často velmi komplikované. Za určitých okolností může být považováno za nelegální činnost, jako například v případě průmyslové špionáže či pokud dochází k porušení práv duševního vlastnictví. K druhému případu může docházet například v oblasti informačních technologií při analýze softwaru. Tyto situace upravuje právní řád České republiky v autorském zákoně (121/2000 Sb.).

Zpětné inženýrství bylo v primitivních podobách používané zřejmě po celou historii lidstva. K velkému rozvoji poté dochází v době průmyslové revoluce. Další velké pokroky a nové rozměry této činnosti přichází s rozvojem informačních technologií. Právě poslední zmíněnou oblastí se budeme blíže zabývat v následujících částech této práce. Zvláštní zájem bude věnován hlavně oboru *zpětného překladu* počítačových programů.

2.1 Zpětné inženýrství v informatice

V kontextu počítačové vědy nachází zpětné inženýrství široké uplatnění. Předmětem zkoumání může být jak technické, tak programové vybavení počítačů. Tato práce se blíže zabývá druhou jmenovanou oblastí zpětného inženýrství. Zpětné inženýrství softwaru lze rozdělit do stejných fází jako klasické softwarové inženýrství, pouze s tím rozdílem, že probíhají

v obráceném sledu. Spektrum oblastí, ve kterých dochází k využití zpětného inženýrství softwaru je široké, pro přehled si několik významných oblastí krátce představíme a poté se blíže zaměříme na zpětný překlad.

- Kryptoanalýza

  Kryptoanalýza je věda zabývající se metodami pro rozkódování zašifrovaných zpráv [2]. K jejímu velkému rozvoji začalo docházet v období druhé světové války. I mezi širokou veřejností je známa například práce vědců v britském Bletchley Parku, kteří luštili německé tajné kódy produkované strojem Enigma. V posledních několika desetiletích nabývá kryptoanalýza a kryptografie s rozvojem internetu, potřebou šifrované komunikace, e-bankovnictví, apod. stále většího významu.

- Lámání ochran programů

  Snaha o prolomení ochran (tzv. *cracking*) je typická pro softwarové pirátství. Cílem pirátů je obejít ochrany zabraňující kopírování daného programu.

- Škodlivé programy

  V oblasti škodlivého softwaru (tzv. *malware*) je reverzní inženýrství používáno jak jeho tvůrci, tak těmi, kteří proti nim bojují. Tvůrci škodlivého softwaru zkoumají programy či informační systémy s úmyslem najít jejich slabiny a místa, skrze která lze na daný systém zaútočit. Vývojáři antivirových programů tyto útoky a škodlivý software analyzují s cílem najít nové a účinnější cesty, jak proti těmto útokům bojovat, či jim zcela zamezit. Dochází zde tedy k pomyslným závodům ve zbrojení, kdy jedna strana neustále hledá nové způsoby možných útoků a druhá proti nim vyvíjí nové způsoby obrany.

- Obnova ztracených zdrojových kódů

  V případě ztráty zdrojových kódů přeloženého programu je možné se pokusit o jejich získání pomocí zpětného překladu. Tato cesta bude ovšem zpravidla velice neefektivní a ekonomicky nevýhodná. Zpětný překlad je nástroj užitečný pro analýzu, nikoliv jako náhrada svědomitého zálohování.

2.2 Zpětný překlad

Cílem zpětného překladu je převod spustitelného binárního programu do takové formy reprezentace, která bude srozumitelná člověku, jenž chce tento program analyzovat. Míra sofistikovanosti získaného výsledku může být různá, od odpovídající posloupnosti instrukcí v jazyce symbolických adres získané pomocí tzv. *disassembleru*, po reprezentaci ve vyšším programovacím jazyce, například jazyce *C*. Dalšími výstupy, které napomáhají analýze programu mohou být například grafy toku řízení, volání funkcí, apod. Skutečností, která proces zpětného překladu a následné analýzy získaných výsledků značně stěžuje, je to, že mnoho informací nemůže být obnoveno. Získaný kód neobsahuje původní jména proměnných a podprogramů, všechna makra jsou rozbalena, chybí komentáře. Problém neúplnosti informací je v oblasti zpětného překladu zásadní a jedná se o fakt, který velmi zvyšuje jeho obtížnost ve srovnání s překladem dopředným.

2.3 Disassembling

Nejjednodušším příkladem převodu binárního programu do vyšší formy reprezentace je převod do jazyka symbolických adres. Binární zápis jednotlivých instrukcí pro procesor je takto převeden do podoby, která je pro člověka snáze čitelná. Přesto je výsledek stále hodně obtížně analyzovatelný. Jazyk symbolických adres operuje stále na velmi nízké úrovni abstrakce a zorientování v takovém kódu vyžaduje od člověka velké úsilí (stále se v podstatě jedná o instrukce pro procesor, pouze čitelněji zapsané), které s rozsahem zkoumaného projektu neúměrně roste. Nástroji používanými pro tuto činnost jsou takzvané *disassemblery*. Jako příklad uveďme *ndisasm*[1] nebo *IDA*[2].

2.4 Zpětný překladač

Pokročilejším nástrojem je tzv. *zpětný překladač*, jehož cílem je převod binárního programu do podoby některého vyššího programovacího jazyka. Proces zpětného překladu můžeme rozdělit do několika kroků. Strojový kód je nejprve převeden do *intermediální reprezentace*, na té jsou poté prováděny různé analýzy (analýza datových toků, typová analýza). Následně je *intermediální reprezentace* shlukována do konstruktů vyšších programovacích jazyků, jako jsou smyčky či podmíněný příkaz. V posledním kroku dochází k vygenerování kódu ve vyšším programovacím jazyce.

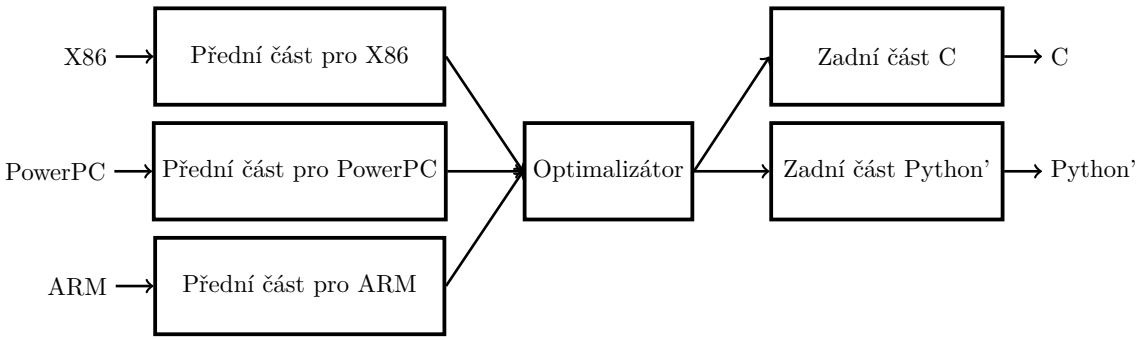
[1] http://www.nasm.us/doc/nasmdoca.html
[2] https://www.hex-rays.com/products/ida/

Kapitola 3

Zpětný překladač společnosti AVG

Tato kapitola čerpá informace z [7].

Společnost *AVG* vyvíjí *zpětný překladač*, který je *rekonfigurovatelný*. Jednoduchou představu o takové struktuře můžeme získat z obrázku 3.1. Obrázek byl převzat z [3] a upraven tak, aby odpovídal *zpětnému překladači* společnosti *AVG*. Lepší pochopení *rekonfigurovatelnosti* pak poskytne sekce 3.1.



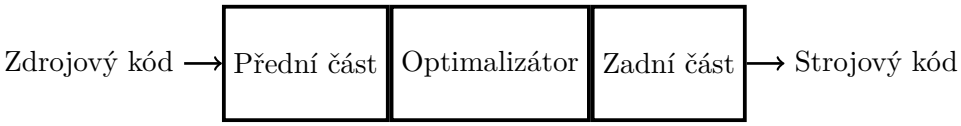
Obr. 3.1: Schéma rekonfigurovatelného zpětného překladače

V podstatě se jedná o nástroj, který je pokud možno co nejvíce nezávislý na konkrétní architektuře (*Intel x86*, *ARM*, *MIPS*, *PIC32*), formátu překládaného spustitelného souboru (*ELF*, *WinPE*), výstupním jazyku (v současnosti je podporován jazyk *C* a modifikovaná verze jazyka *Python*) či operačním systému. Nová funkcionalita může být přidávána bez narušení struktury *zpětného překladače* pomocí přidání nových zásuvných modulů a případnou úpravou konfiguračních souborů. Celý *zpětný překladač* je koncipován jako řetězec nástrojů, které spolu interagují pomocí *shellových skriptů*. Celou strukturu lze rozčlenit do čtyř základních částí – *předzpracování*, *přední část*, *střední část* a *zadní část*. Vzhledem k faktu, že proces zpětného překladu je obdobný překladu dopřednému, lze k vývoji *zpětného překladače* využít infrastrukturu existujících překladačů. V projektu společnosti *AVG* je využívána infrastruktura překladače *LLVM*.

3.1 LLVM

Projekt *LLVM* je tvořen kolekcí modulárních a znovupoužitelných technologií pro tvorbu překladačů. Vznikl jako výzkumný projekt na University of Illinois v prosinci roku 2000 [4].

Typický překladač je rozdělen do tří částí, jak vidíme na obrázku 3.2.



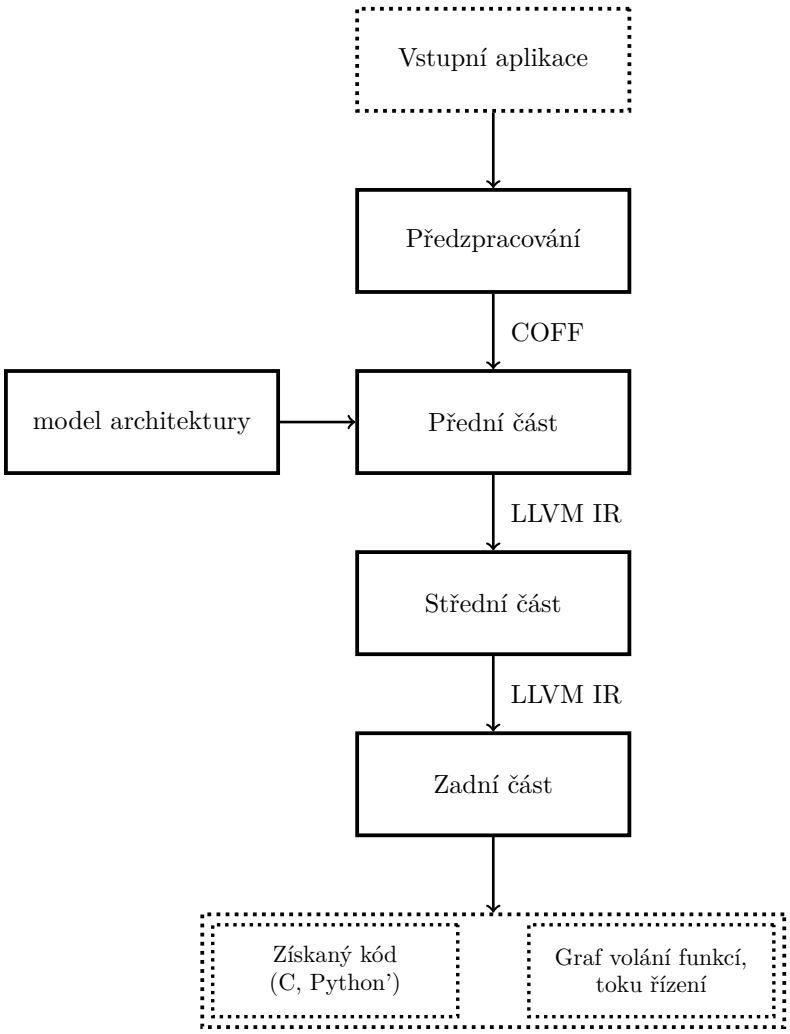
Obr. 3.2: Schéma běžného překladače, převzato z [3]

Přední část zpracovává zdrojový kód, hledá v něm chyby a vytváří abstraktní syntaktický strom reprezentující daný vstupní zdrojový kód. Optimalizátor poté provádí různé transformace, jejichž účelem je zrychlení běhu překládaného programu, popřípadě snížení jeho paměťové náročnosti. Může zde docházet například k odstranění přebytečných výpočtů a nedosažitelného kódu či zavedení určitých heuristik, vedoucích k efektivnějšímu provádění daného příkazu či podprogramu ve výsledném spustitelném programu. Zadní část poté vygeneruje posloupnost instrukcí pro cílovou architekturu odpovídající optimalizovanému zdrojovému kódu.

Z této struktury může plynout důležitá výhoda. Pokud je uvnitř optimalizátoru použita pro kód jednotná reprezentace nezávislá na vstupním jazyce, stačí pro přidání podpory pro nový jazyk vytvořit další přední část překladače. Obdobně pro přidání podpory pro novou cílovou architekturu postačí implementovat novou zadní část. Optimalizátor v takovém případě tvoří přemostění mezi přední a zadní částí. Takovýto překladač poté charakterizujeme označením *rekonfigurovatelný*.

Pro překlad z N vstupních jazyků na M cílových architektur by za normálních okolností bylo nutné vytvořit $N \times M$ překladačů. Při použití *rekonfigurovatelné* architektury stačí jeden společný optimalizátor, N předních a M zadních částí. Další výhodou pramenící z takovéto struktury je to, že jednotlivé části překladače mohou být vyvíjeny odděleně, což umožňuje snadnější rozdělení a paralelizaci práce vývojového týmu.

3.2 Struktura zpětného překladače společnosti AVG



Obr. 3.3: Schéma zpětného překladače společnosti AVG

Na obrázku 3.3 vidíme velmi hrubý nástin struktury zpětného překladače *AVG*, který si nyní blíže představíme.

3.2.1 Fáze předzpracování

Před začátkem procesu zpětného překladu je nejprve nutné vstupní spustitelný soubor uvést do stavu, který tuto činnost umožní. V této fázi nejprve dochází k rozpoznání překladače, pomocí něhož daný binární soubor vznikl, jazyka v kterém byl zapsán jeho zdrojový kód, případně může být zjišťována přítomnost ladicích informací. Daný soubor mohl být zabalen pomocí tzv. *packeru*, v takovém případě je tedy nutné ještě detekovat použitý *packer*. Zjišťování těchto informací probíhá s pomocí databáze signatur. Detekce použitých nástrojů není zcela přesná a může dojít k označení několika možných použitých nástrojů. V případě, že bylo zjištěno použití *packeru*, je program nutné rozbalit použitím *unpackeru* určeného pro rozbalovaní souborů zabalených detekovaným *packerem*. Pokud by k rozbalení nedošlo,

nebylo by možné zpětný překlad provést nebo by byly získané výsledky velmi nepřesné. Použití *packeru* může být náznakem toho, že analyzovaný program patří mezi malware.

Dále dochází k detekci použitého objektového formátu. Tím například může být *ELF*, typický pro Unixové systémy, *WinPE* používaný systémy Windows, a další. Aby bylo možné se všemi programy pracovat jednotně, dochází k převodu na jednotný formát, který vychází z formátu *COFF*[1].

Pro rozbalování souborů i konverzi jejich formátů jsou používány generické nástroje. Přidání nového *unpackeru* či podpory pro nový objektový formát je řešeno implementací nového pluginu. Poslední důležitou informací je cílová architektura pro kterou byl program vytvořen. Jako příklad uveďme *x86*, *ARM* či *MIPS*. V rámci projektu zpětného překladače společnosti *AVG* jsou pro popis cílových architektur používány modely napsané v jazyce *ISAC*[2]. Jazyk *ISAC* patří mezi tzv. *smíšené ADL*[3]. Zaznamenává jak strukturu, tak chování dané architektury.

3.2.2 Přední část

V přední části dochází k převodu spustitelného souboru do posloupnosti instrukcí reprezentované v jazyce *LLVM IR*. K tomuto procesu je potřeba znát binární kódování instrukcí a jejich sémantiku specifické pro danou architekturu – tyto informace jsou obsažené v popisu dané architektury v jazyce *ISAC*. Pro zefektivnění procesu zpětného překladu mohou být v přední části volitelně použity i další vstupy, jako například informace o typech funkcí a parametrů či signatury staticky linkovaného kódu. V této sekci zpětného překladače je použit řetězec statických analýz, s jejichž pomocí dochází k odstranění staticky linkovaného kódu, obnově datových typů, detekci funkcí s jejich parametry, lokálních a globálních proměnných apod. Dochází též k hledaní jednoduchých idiomů a jejich nahrazení srozumitelnější reprezentací a odstranění kódu souvisejícího s voláním a návratem z funkcí. Po provedení těchto analýz je z transformovaného kódu vygenerovaná jeho nízkoúrovňová reprezentace v *LLVM IR*, která je dále zpracovávána ve střední části.

3.2.3 LLVM IR

LLVM IR je jazyk podobný jazyku symbolických adres, je ovšem platformně nezávislý a silně typovaný. Byl navržen s ohledem na efektivní provádění analýz a transformací v optimalizátoru překladače.

Převod vstupního kódu do *LLVM IR* je ve své podstatě překlad do virtuální instrukční sady *LLVM*. Instrukce odpovídají instrukcím typickým pro jazyk symbolických adres. Nalezneme zde tedy například instrukce pro porovnávání, provádění podmíněných a nepodmíněných skoků, aritmetické operace a jiné. Pro určení cíle instrukcí skoků jsou využívána návěští. Jazyk používá tzv. *tříadresnou formu*, to znamená, že instrukce vrací výsledek do registru různého od registrů, ve kterých byly předány vstupní parametry.

3.2.4 Střední část

Úkolem střední části je optimalizace kódu vyprodukovaného částí přední. Do této chvíle bylo na analyzovaný kód nahlíženo na úrovni jednotlivých instrukcí; optimalizátor analyzuje kód po větších blocích. Dochází k detekci složitějších idiomů, eliminaci nedosažitelného kódu.

[1] Common Object File Format, http://www.ti.com/lit/an/spraao8/spraao8.pdf
[2] Instruction Set Architecture C
[3] Architecture description language - jazyk popisující architekturu

Také jsou vyhledávány řídicí struktury, jako například smyčky (*for*, *while*), podmíněných příkazů (*if-then-else*) nebo konstrukce *switch*. Použity jsou jak optimalizace, které jsou součástí projektu *LLVM*, tak optimalizace vytvořené ve společnosti *AVG*. Také jsou zde využity různé podpůrné analýzy a pomůcky.

3.2.5 Zadní část

V zadní části zpětného překladače dochází k převodu optimalizovaného *LLVM IR* kódu do některého vyššího programovacího jazyka. V současnosti jsou podporovány dva jazyky – jazyk *C* a *Python'*. *Python'* je upravená podoba jazyka *Python*. Liší se od něho tím, že konstrukce, pro které *Python* nezná reprezentaci jsou nahrazeny jejich ekvivalentem v jazyce *C*. Kromě toho je také doplněný o ukazatele a příkaz *goto*, pole jsou v něm nahrazeny seznamy a struktury slovníky. Tato práce se bude dále zabývat případy, kdy je generován výstup v jazyce *C*.

Po vygenerování cílového kódu dochází v zadní části ještě k dalším optimalizacím, jejichž účelem je zvýšit čitelnost výsledku. V této části také dochází k vytvoření grafů toku řízení a volání funkcí. V ukázce 3.1 vidíme příklad výstupu zpětného překladače v jazyce *C*.

```
#include <stdint.h>
#include <stdio.h>

int32_t factorial(int32_t n) {
    int32_t x;
    if (n != 0) {
        x = factorial(n - 1) * n;
    } else {
        x = 1;
    }
    return x;
}

int main(int argc, char **argv) {
    printf("factorial(%d) = %d\n", 10, factorial(10));
    return 0;
}
```

Kód (3.1) Ukázka kódu získaného pomocí zpětného překladače

Kapitola 4

Verifikace softwaru

Cílem této práce není implementace nových vlastností zpětného překladače, ale ověřování jeho funkčnosti. Tato kapitola čerpá z [9].

Existuje několik přístupů k verifikaci správného fungování programů. Podle principů jejich fungování je můžeme rozdělit následujícím způsobem:

Statická analýza zkoumá vlastnosti programu bez jeho provádění. Pod tímto si můžeme představit například analýzu jeho zdrojového kódu.

Dynamická analýza ověřuje vlastnosti testované aplikace na základě jejího provádění. K uskutečňování takovýchto testů často není vůbec potřeba znalost vnitřní struktury testovaného systému. Mnohdy je taková znalost přímo nežádoucí – to v případě, kdy chceme testovat tzv. černou skříňku, tedy uzavřený systém, jehož implementační detaily neznáme a zaměřujeme se pouze na správnost výstupů pro zadané vstupy.

Formální verifikace s pomocí matematických metod ověřuje, že systém odpovídá specifikaci. Tuto metodu lze zařadit jak do dynamické, tak do statické analýzy.

Testování zkoumá programy jejich spouštěním za účelem zvýšení jejich kvality. Jedná se tedy o druh dynamické analýzy. Tato bakalářská práce se hlouběji zabývá právě touto oblastí verifikace programů.

Důležitým faktem, který je třeba zmínit v souvislosti s testováním je to, že pomocí testování nelze obecně dokázat, že testovaný systém žádné chyby neobsahuje, je pouze možné chyby nalézt.

Testováním se rozumí vyhodnocování testovaného systému jeho spouštěním a sledováním jeho běhu.

Selhání testu je spuštění testovaného systému, které vede k nežádoucímu stavu.

Projití testu je spuštění testovaného systému, které k selhání nevede.

Pro to, aby bylo selhání možné zjistit, musí být splněny tři základní podmínky:

Dosažitelnost (Reachability) Místo v kódu obsahující defekt musí být dosažena. To znamená, že při běhu testovaného programu musí procesor vykonat instrukce, na které byl daný úsek kódu přeložen.

Infikování (Infection) Po provedení daného místa musí být stav programu nekorektní.

Propagace (Propagation) Nekorektní stav programu se musí rozšířit či projevit tak, že způsobí nekorektní výstup programu.

Pro přehled nyní definujme několik základních termínů z oblasti testování.

Testovací sada je množina testovacích případů.

Testovací případ se skládá ze vstupních hodnot, očekávaných výsledků a z prefixových a postfixových hodnot.

Hodnoty testovacího případu jsou vstupní hodnoty nutné pro dokončení běhu.

Očekávaný výsledek jsou hodnoty produkované testovaným programem z hodnot testovacího případu pouze tehdy, splňuje-li program očekávané chování.

Prefixové hodnoty jsou vstupní hodnoty testovaného systému, které je nutné předat testovaného systému proto, aby bylo možné zjistit jeho stav.

Můžeme rozlišovat různé stupně testování.

- Během *akceptačního testování* dochází k zjišťování, zda testovaný systém splňuje požadavky zákazníka.
- Cílem *systémového testování* je ověření, že testovaný systém jako celek splňuje specifikaci požadavků.
- Účelem *integračního testování* testování je ověření rozhraní modulů v podsystému, jejich předpokládaných stavů a vzájemné komunikace.

Z pohledu této práce jsou zásadní následující dva stupně:

4.1 Jednotkové testování

Jednotka je fragment kódu, který popisuje chování systému. Například v jazyce *Python* může být za jednotku považována jedna metoda nějaké třídy. Jednotkové testování bylo zavedeno pro zmenšení rozsahu testování na vyšších úrovních. Odhalenými chybami jsou většinou chyby na nejnižší úrovni, tedy nesprávné návratové hodnoty, krajní hodnoty parametrů, nezamýšlené vedlejší účinky. V rámci implementační části této práce jsou jednotkové testy používány k ověřování správné funkčnosti vyvíjeného *aplikačního rámce*.

4.2 Regresní testování

Jak uvádí [5], regrese je situace, kdy zavedení nové funkcionality či oprava chyby způsobí vznik chybného chování v části vyvíjeného systému, která předtím už fungovala požadovaným způsobem. Smyslem regresních testů je takovýmto situacím zabraňovat. Systém je cyklicky po každé změně testován regresními testy. Pokud dojde k jejich selhání, znamená to, že poslední změna zanesená do projektu způsobila vznik regrese. Takovéto testování je v praxi velice časté a prakticky žádný větší projekt se bez něj neobejde. Regresní testy je velmi vhodné automatizovat a spouštět po každém přidání nového kódu do hlavní větve repozitáře projektu. Cílem této práce bylo rozšíření *aplikačního rámce*, který usnadňuje tvorbu regresních testů používaných při vývoji *zpětného překladače* ve společnosti *AVG*.

Kapitola 5

Jazyk C

Nyní si ve stručnosti představíme jazyk *C* a zaměříme se na ty jeho prvky, které jsou důležité z pohledu této práce. V rámci této kapitoly jsou použity informace získané z [6] a [8].

Programovací jazyk *C* může být výstižně kategorizován označením *jazyk nižší úrovně*. Z historického hlediska je validní označení *vysokoúrovňový jazyk*, protože používá vyšší úroveň abstrakce než *jazyk symbolických adres*. V současné době bývá ovšem často řazen mezi *jazyky nízkoúrovňové*, vzhledem k tomu, že jeho úroveň abstrakce je ve srovnání s modernějšími jazyky (např. *Python* nebo *Java*) mnohem nižší. Oba tyto pohledy mají svá opodstatnění a své zastánce, a mohly by být předmětem sáhodlouhých debat.

Jedná se o obecně využitelný programovací jazyk nezávislý na konkrétní platformě. Pro prakticky všechny platformy existuje nějaký jeho překladač. Při dodržení určitých zásad jsou aplikace v něm napsané velmi snadno přenositelné. Je standardizovaný (*ISO/ANSI*). Vzhledem k velké efektivitě dobře napsaného kódu v jazyce *C* bývá tradičně používán pro systémové programování. Stejně tak je však dobře použitelný pro psaní programů v oblasti numerických výpočtů, zpracování textu, databázových systémů a mnoha dalších. Jazyk *C* je otevřený jazyk s malým jádrem, které je dále rozšiřováno pomocí knihoven. Přestože první verze vznikla již před téměř polovinou století, je stále velmi často používaný. Podle TIOBE indexu se v současnosti jedná o druhý nejpopulárnější jazyk[1].

Často je jazyk *C* dáván do souvislosti s operačním systémem *UNIX*. Pod tímto systémem byl v 70. letech 20. století navržen a implementován Dennisem Ritchiem, a zároveň systém *UNIX* se svým programovým vybavením byl v tomto jazyku dále vyvíjen. Mnohé z myšlenek jazyka *C* vychází z jazyka *BCPL*, který byl vytvořen Martinem Richardsem. Dalším důležitým předchůdcem byl jazyk *B*, vyvinutý Kenem Thompsonem. Oba tyto jazyky byly ovšem beztypové.

Základní datové typy jazyka *C* jsou *znaky*, *celá čísla*, *čísla s pohyblivou řádovou čárkou* a *pole*, což jsou homogenní kolekce určitého typu a pevně dané délky. Dále je důležitá existence *struktur* a *ukazatelů*, které umožňují vytváření komplexnějších datových typů. S ukazateli je možné provádět ukazatelovou aritmetiku. Ukazatele můžeme sčítat, odečítat a porovnávat. Tyto operace mají význam při práci s poli nebo s dynamicky alokovanou pamětí. Ukazatelovou aritmetiku nemůžeme použít na jednoduché proměnné. Takové použití by ale ani nedávalo velký smysl. Jména polí jsou synonymem pro adresu jejich prvního prvku, jedná se tedy taktéž o ukazatele. Jazyk *C* je relativně slabě staticky typovaný jazyk. Je velmi benevolentní k typovým konverzím a mnoho z nich provádí implicitně, což

[1] http://www.tiobe.com/tiobe_index

může učinit napsaný kód rychlejším a kompaktnějším. Na druhou stranu toto ovšem zvyšuje obtížnost ladění programu.

Řízení toku programu je umožněno pomocí *složených příkazů*, jednoduchého větvení *if-then-else*, větvení s výběrem z množiny možných alternativ *switch* a smyček (*while*, *do while* a *for*). Průběh smyček můžeme ovlivňovat pomocí příkazů *continue* a *break* (tento nalézá využití i v konstruktu *switch*). Také máme možnost použít příkaz *goto*, což je ovšem v naprosté většině situací velice nevhodná praktika vedoucí ke znepřehlednění toku vykonávání programu. Poslední cestou vedoucí k ovlivnění toku programu je volání funkcí a návrat z jejich vykonávání pomocí příkazu *return*.

V jazyku *C* neexistují žádné operace pro manipulaci se složenými objekty, jako jsou znakové řetězce, množiny, seznamy nebo pole chápaná jako celek. Stejně tak zde nenalezneme automatickou správu paměti (*garbage collection*). Dynamická alokace paměti, nástroje pro vstup a výstup či práci se souborovým systémem nejsou přímo součástí jazyka, ale jsou zprostředkovány voláním knihovních funkcí.

Z faktu, že si programátor musí sám spravovat využívané zdroje, pramení zřejmě nejčastější chyby typické pro vývoj v tomto jazyce. Jedná se hlavně o tzv. *memory leaky*, tedy situace, kdy programátor zapomene uvolnit alokovanou paměť, kterou už dále nepoužívá. Komplementární k této situaci je tzv. *dangling pointer*, což je situace, kdy dochází k pokusu o dereferenci ukazatele, který dříve odkazoval na alokovanou paměť, ale tato už byla uvolněna. Dále může například dojít k opomenutí uzavření deskriptoru souboru, který byl použit pro vstup či výstup. Vzhledem k tomu, že jazyk *C* nekontroluje meze polí, nezkušení vývojáři se též často setkávají s tzv. *segmentation fault*, ke které dochází tehdy, když nastane pokus o přístup k úseku paměti, který procesor nemůže adresovat. S odstraňováním těchto a podobných problémů mohou programátorovi pomoci různé nástroje vyvinuté pro statickou (obecně známé pod označením *linter*) a dynamickou analýzu programů. Nejznámější z nich je zřejmě *valgrind*[2].

Typický program napsaný v jazyce *C* odpovídá *imperativnímu* paradigmatu. Vzhledem k tomu, že jazyk obsahuje *struktury* a *ukazatele na funkce*, je v principu možné v něm vyvíjet podle paradigmatu *objektového* či *funcionálního*. Jazyk ovšem toto nijak nepodporuje, nenalezneme tu syntaktické konstrukty usnadňující tvorbu objektů ani optimalizace typické pro funkcionální jazyky, jako například *tail call optimization*. Proto je tehdy, když nechceme programovat imperativně, vhodné zvolit jiný jazyk.

Mezi nevýhody jazyka *C* patří například to, že nedisciplinovanému programátorovi umožňuje psát naprosto nečitelné a nesrozumitelné programy. Také je třeba dát pozor na to, že některé věci nemůže překladač jazyka *C* kontrolovat – například konzistenci při sestavování spustitelného programu z více objektových souborů.

Argumenty funkcí se přenášejí kopírováním hodnoty, takže funkce nemůže změnit hodnotu argumentu ve volajícím programu. Volání odkazem lze dosáhnout použitím ukazatele – funkce má poté možnost změnit objekt, na který daný ukazatel odkazuje. Všechny funkce je možné volat rekurzivně a jejich lokální proměnné jsou automatické – to znamená, že při každém zavolání funkce dojde k jejich novému vytvoření na zásobníku. Definice funkcí nesmí být vnořené. Proměnné můžeme deklarovat blokově–strukturovaným způsobem, mohou být interní v rámci funkce, externí s oblastí platnosti v rámci jednoho zdrojového souboru, nebo zcela globální. Interní proměnné mohou být automatické nebo statické. Pro účely optimalizace je pomocí klíčového slova *register* sdělit překladači, aby automatickou proměnnou umístil do registru. Jedná se ale o pouhé doporučení, které může překladač zcela ignorovat.

[2] http://www.valgrind.org

Struktura programu v jazyku *C* sestávají na základní úrovni vždy z jedné nebo více funkcí. Navíc zde můžeme naleznout definice *globálních proměnných*, *struktur*, *unií* a *výčtových typů*. Každý program musí obsahovat *funkci* se jménem *main*, která je vstupním bodem jeho vykonávání. Definice jednotlivých funkcí můžeme rozdělit do více *modulů* (souborů). Pro případ, kdy chceme v jednom *modulu* použít funkci definovanou v jiném *modulu*, používá jazyk *C* tzv. *hlavičkové soubory*. Do těchto souborů umístíme *prototypy funkcí* definovaných v *modulu*, pro který hlavičkový soubor vytváříme. V hlavičkových souborech můžeme také často naleznout konstanty, výčty, definice komplexních datových typů apod. Ve zkratce můžeme říct, že to jsou části kódu, které svojí podstatou specifikují pevně dané neměnné hodnoty. Typicky byhom zde neměli nalézt logiku programu. *Prototyp funkce* zahrnuje její jméno, návratový typ a seznam parametrů s jejich typy. V podstatě se jedná o *API* dané *funkce*. Chceme-li využít *funkce* z jiného *modulu*, vložíme pomocí direktivy *#include* do našeho *modulu* hlavičkový soubor vkládaného *modulu*. Skutečné propojení těchto *modulů* se musíme poté postarat při vytváření spustitelného souboru během fáze *linkování*. Pojďme si tedy v závěru této kapitoly celý proces tvorby spustitelného souboru v jazyce *C* přiblížit.

Pro tvorbu spustitelných souborů ze zdrojového kódu používáme *překladač*. Příkladem může být *clang*, který byl již představen v předchozích částech této práce. Pomocí překladače můžeme z *modulu* jazyka *C* vytvořit tzv. *objektový soubor*. Tímto jsme ještě nevytvořili nic, co bychom mohli skutečně spustit. Pouze došlo k přeložení zdrojového kódu na odpovídající posloupnost instrukcí ve strojovém kódu. Pro získání funkčního programu musíme jednotlivé *objektové soubory* spojit v jeden *spustitelný soubor*. Tohoto dosáhneme pomocí tzv. *linkeru*. Během vytváření *objektových souborů* nekontroloval *překladač*, zda volané *funkce* skutečně existují. Pokud se nenacházely v překládaném *modulu*, jednoduše předpokládal, že se nachází v jiném. V této fázi ke kontrole již dochází. Typicky tedy můžeme narazit na chybu způsobenou neexistencí určité *funkce*. K tomuto zpravidla dochází z důvodu, že jsem funkci zapomněli definovat, nebo jsme zapomněli přeložit *modul*, který funkci obsahuje. Výsledkem úspěšného *slinkování* je spustitelný program. Nespornou výhodou tohoto rozdělení zodpovědnosti mezi *překladač* a *linker* poznáme ve chvíli, kdy provádíme změnu v již existujícím programu. V tuto chvíli stačí přeložit jen ty *moduly*, kterých se změna týkala, a celý program pouze znovu *slinkovat*.

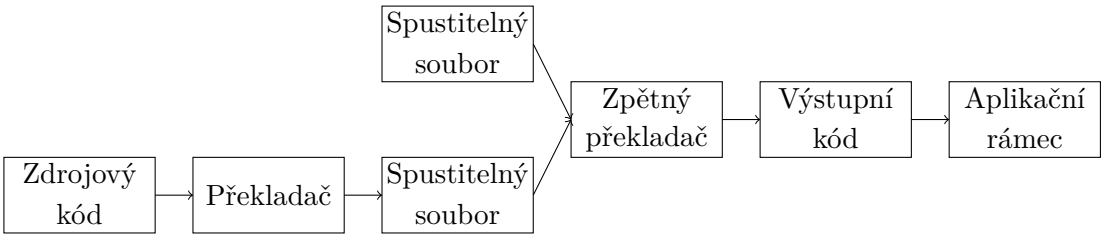
Toto stručné představení jazyka by mělo být pro účely této práce dostatečné. Jednotlivé konstrukty jazyka, pro něž byla v rámci práce vypracována podpora jsou blíže vysvětleny až v místech zabývajících se implementací jejich podpory ve vyvíjeném *aplikačním rámci*.

Kapitola 6

Aplikační rámec pro tvorbu regresních testů

Regresní testy zpětného překladače jsou tvořeny s pomocí *aplikačního rámce* napsaného v jazyce *Python*, který využívá provázání s překladačem *clang*, který je součástí projektu *LLVM*. Takto je umožněno s pomocí jazyka *Python* snadno psát různé analýzy ať už zpětně přeloženého zdrojového kódu, interní reprezentace *LLVM IR*, výstup z *disassembleru* či dalších souborů, které vzniknou během procesu zpětného překladu.

Motivací pro zavedení regresních testů byla kromě zabránění regresím při vývoji také možnost provádění kvalitativních testů. Dříve používané testovací metody byly prováděny tak, že zdrojový kód získaný zpětným překladem byl znovu přeložen a výstupy výsledného programu byly porovnávány s výstupy originálního programu. Takto bylo možné ověřovat například to, že zpětně přeložený zdrojový kód například obsahuje smyčku, už ale ne to, jakou konkrétně. *Aplikační rámec* regresních testů umožňuje provádět přímo analýzu získaných souborů, proto je možné ověřovat například to, že *for* cyklus nebyl v získaném kódu nahrazen třeba cyklem *while*.



Obr. 6.1: Zpětný překladač a aplikační rámec v kontextu celého procesu

Diagram 6.1 zobrazuje celý proces testování *zpětného překladače*. Ve zkratce vše probíhá následujícím způsobem. *Spustitelný soubor* je zpracován *zpětným překladačem*, takto získáme *výstupní kód*. Tento kód je poté zkontrolován za pomoci *aplikačního rámce*. *Spustitelný soubor* může být buď dodaný přímo, nebo je nejprve vytvořen specifikovaným *překladačem* z dodaného *zdrojového kódu*.

6.1 Koncept regresních testů

Adresář `decompiler/testsuite/regression-tests` obsahuje jednotlivé testovací sady. Zde je každá testovací sada tvořena vhodně pojmenovaným adresářem, který lze uvnitř dále libovolně strukturovat. Specifikace testů je napsána v jazyce *Python* a nachází se v souborech pojmenovaných *test.py*. Tyto soubory se mohou nacházet v libovolném umístění v rámci adresáře dané testovací sady. Dále se zde mohou nacházet další soubory, jako například zdrojové kódy napsané v jazyce *C* či binární soubory. *Aplikační rámec* regresních testů prochází všechny podadresáře adresáře s testovací sadou, nalezne všechny soubory *test.py*, načte z nich informace o testovacích případech a ty následně provede a vyhodnotí.

U každého testovacího případu je třeba specifikovat, které třídy z *aplikačního rámce* mají být importovány a parametry, se kterými se má spustit zpětný překlad. Samotný testovací případ je pak vytvořen jako třída, která je potomkem bázové třídy Test.

```
settings = TestSettings(
    input = 'ack.c',
    arch = ['x86', 'arm', 'thumb', 'mips', 'powerpc', 'pic32'],
    format = 'elf',
    compiler = 'gcc',
    compiler_opts = '-O0',
    debug_info = True
)
```

Kód (6.1) Příklad nastavení testovacího případu, převzato z [1]

V ukázce 6.1 vidíme příklad nastavení testovacího případu.

- Parametr *input* označuje vstupní soubor se zdrojovým kódem v jazyce *C*, který bude přeložen překladači specifikovanými v proměnné *compiler* s nastaveními uloženými v *compiler_opts* na architekturách z proměnné arch a vzniklé spustitelné soubory budou mít formát z množiny specifikované v proměnné *format*.
- *Debug_info* určuje, zda budou ve výstupu přítomny ladicí informace.
- Proměnné *arch*, *format* a *compiler* mohou též nabývat hodnoty *ALL*, která říká, že má být test provedených na všech podporovaných architekturách, formátech či překladačích.
- Případně lze též použít zápis *ALL – [výčet prvků]*, který značí „vše, kromě vyjmenovaného“.

Provedeny jsou vždy všechny možné kombinace plynoucí z nastavení. V uvedeném příkladu tedy dojde ke spuštění šesti testů. Kdybychom například specifikovali tři různé architektury, dva formáty spustitelného souboru a dva překladače, provede se testů dvanáct. Tento příklad je ale pouze ilustrační, ve skutečnosti může být počet spuštěných testů nižší například kvůli tomu, že některá ze specifikovaných architektur nemusí podporovat některý z formátů spustitelného souboru. Regresní testy jsou spouštěny buď automaticky, například po přidání kódu do repozitáře projektu zpětného překladače, nebo mohou být spuštěny manuálně.

6.2 Libclang

Pro analýzu výstupů zpětného překladu v jazyce *C* je v *aplikačním rámci* regresních testů využívána knihovna *libclang*, která poskytuje API[1] pro přístup k abstraktnímu syntaktickému stromu, který z daného kódu vytvoří překladač *clang*. Pro přístup k této knihovně z prostředí jazyka *Python*, ve kterém je *aplikační rámec* napsán, slouží modul *cindex*. Pro účely analýzy kódu definuje *cindex* dva důležité objekty. Prvním je *TranslationUnit*, který zapouzdřuje abstraktní syntaktický strom dané překladové jednotky jazyka *C*. Druhým je poté *Cursor*, který reprezentuje jednotlivé uzly abstraktního syntaktického stromu. Objekt *Cursor* definuje mnoho atributů popisujících daný uzel. Za zmínku stojí například *spelling*, obsahující textovou reprezentaci uzlu, *kind*, označující druh uzlu či *type*, který obsahuje datový typ objektu reprezentovaného daným uzlem. Také zde nalezneme metodu *get_children*, skrze kterou získáme následovníky daného uzlu.

V ukázce 6.2 vidíme zdrojový kód funkce obsahující příkaz *switch* a v ukázce 6.3 abstraktní syntaktický strom, který tento kód reprezentuje. V ukázce abstraktního syntaktického stromu jsou uzly reprezentovány označením jejich druhu. Kód v jazyce *Python* z ukázky 6.4 je poté zkrácenou podobou třídy pro analýzu příkazu *switch* z implementované v testovacím *aplikačním rámci*.

```
void foo(char c) {
    float f = 1.0;

    switch (c) {
        case 'a':
            f = 2.0;
            break;
        default:
            bar();
    }
}
```

Kód (6.2) Funkce obsahující switch

[1] Application programming interface - rozhraní pro programování aplikací

```
TRANSLATION_UNIT
    FUNCTION_DECL
        PARM_DECL
        COMPOUND_STMT
            DECL_STMT
                VAR_DECL
                    UNEXPOSED_EXPR
                        FLOATING_LITERAL
            SWITCH_STMT
                UNEXPOSED_EXPR
                    UNEXPOSED_EXPR
                        DECL_REF_EXPR
                COMPOUND_STMT
                    CASE_STMT
                        CHARACTER_LITERAL
                        BINARY_OPERATOR
                            DECL_REF_EXPR
                            UNEXPOSED_EXPR
                                FLOATING_LITERAL
                    BREAK_STMT
                    DEFAULT_STMT
                        CALL_EXPR
                            UNEXPOSED_EXPR
                                DECL_REF_EXPR
```

Kód (6.3) Abstraktní syntaktický strom pro kód 6.2

Jak je patrné z ukázky abstraktního syntaktického stromu, vydáme-li se prvním podstromem uzlu příkazu *switch*, dosáhneme po přeskočení uzlů typu *unexposed_expr* k uzlu reprezentujícímu řídicí výraz příkazu *switch*. Tento postup vidíme použitý ve vlastnosti *switch_expr* třídy *SwitchStmt* testovacího *aplikačního rámce*. Tělo příkazu *switch* se nachází v druhém podstromu uzlu, který tento příkaz reprezentuje. V kódu testovacího *aplikačního rámce* k jeho analýze slouží metoda *_parse_switch_body*, která podstrom těla postupně projde a získá z něho uzly reprezentující jednotlivé případy *case* a také případ *default*, pokud byl uveden. Kód této metody nalezneme v ukázce 6.4.

```
def _parse_switch_body(self):
    body_node = list(self._node.get_children())[1]
    for child in body_node.get_children():
        if child.kind == cindex.CursorKind.CASE_STMT:
            self._cases.append(Case(child))
        elif child.kind == cindex.CursorKind.DEFAULT_STMT:
            self._default = Default(child)
```

Kód (6.4) Metoda _parse_switch_body ze třídy pro příkaz switch v aplikačním rámci

6.3 Počáteční stav a návrh rozšíření

Cílem této práce bylo vytvořit rozšíření pro analýz kódu v jazyce *C*, proto se zaměříme pouze na část *aplikačního rámce*, která s tímto souvisí. Před začátkem práce byl veškerý kód zodpovídající za tuto oblast umístěn v modulu *c_parser*. Zde byla implementována následující funkcionalita:

- *Module*

  Bylo možné analyzovat zvolený *modul* jazyka *C*. Získat *globální proměnné* v *modulu* obsažené, jejich počet a jména, zjistit, zda *modul* nějaké *globální proměnné* obsahuje, či jestli obsahuje (případně obsahuje pouze) *globální proměnné* specifikovaných jmen. Obdobné informace bylo možné zjišťovat také pro *funkce*. Dále bylo možné získat všechny komentáře a zjistit, zda se v *modulu* nachází *komentář* odpovídající zadanému regulárnímu výrazu. Bylo možné zjistit, které *moduly* byly vloženy pomocí direktivy *include*. V neposlední řadě šlo také zjistit hodnoty všech *řetězcových literálů* a dotázat se, zda *modul* obsahuje literál zadané hodnoty nebo odpovídající zadanému regulárnímu výrazu.

- *Function*

  Aplikační rámec umožňoval získat *jméno funkce*, její *návratový typ* a *seznam jejích parametrů* včetně jejich typů. Bylo možné se dotázat na počet parametrů, na to, zda vůbec *funkce* nějaké má, na to, jestli má (případně má pouze) parametry zadaných jmen. Také bylo možné zjistit, které *funkce* jsou v těle dané *funkce volány* a *for cykly*, které tělo obsahuje.

- *Variable*

  Tato třída umožňovala získat *jméno*, *typ* a *inicializátor* proměnné.

- *Comment*

  U komentářů bylo možné ověřit, že odpovídají zadanému *regulárnímu výrazu*.

- *Literal*

  Aplikační rámec obsahoval podporu pro čtyři druhy literálů – *celé číslo*, *číslo s plovoucí řádovou čárkou*, *znak* a *řetězec*. Z těchto objektů bylo možné získat hodnotu daného literálu.

- *Type*

  Implementovány byly následující datové typy – *char* (znak), *int* (celé číslo), *float* a *double* (číslo v plovoucí řádové čárce, po řadě s jednoduchou a dvojitou přesností), *void* (prázdný typ označující absenci hodnoty), *pointer* (ukazatel), *struct* (komplexní datový typ odpovídající záznamu sestávajícímu z více položek), *array* (pole hodnot stejného typu). Také existovala podpora pro vytváření objektů typu *Type* z charakteristiky *type* z uzlu stromu, který byl vytvořen zpracováním vstupního kódu pomocí *clang*.

- *Statement*

  Jediným podporovaným příkazem byl *cyklus for*.

- *Expression*

  Bylo možné vytvářet objekty typu *Expression* z uzlu stromu vytvořeného zpracováním vstupního kódu pomocí *clang*. Podporované byly dříve zmíněné *literály* a *proměnné*.

Aplikační rámec také obsahoval různé bázové třídy, jejichž detailní popis (kromě již zmíněných *Type*, *Expression* a *Statement*) nemá pro čtenářovu představu o funkcionalitě rámce před započetím práce hlubší přínos. Tyto třídy sdružují kód, který je společný více třídám. Jako příklad lze uvést bázovou třídu datové typy s plovoucí řádovou čárkou. Rámec též obsahoval funkci umožňující zpracování zadaného vstupního kódu v jazyce *C* pomocí *clang*. Výsledkem této operace je abstraktní syntaktický strom odpovídající danému kódu.

Veškerá funkcionalita týkající se zpracování kódu v jazyce *C* byla umístěna v jediném modulu. V takovémto stavu není na první pohled vůbec patrné, do jakých částí je *aplikační rámec* rozčleněn ani to, jaká funkcionalita už byla už naprogramována. Toto velice znepřehledňuje aplikaci a znesnadňuje další vývoj. Proto bylo prvním navrženým krokem v dalším vývoji rozčlenění *aplikačního rámce* do jemnější struktury balíčků a modulů obsahujících vždy jen kód týkající se jednoho prvku jazyka *C* (datového typu, výrazu, příkazu, apod.).

Dalším navrženým rozšířením bylo přidání prvků jazyka *C*, jejichž podpora nebyla dosud implementována.

- V oblasti datových typů chybí například *union* a *enum*.
- Datový typ pole podporuje pouze pole jednorozměrná, proto by bylo vhodné ho rozšířit o podporu libovolně velké dimenze.
- Dále neexistuje podpora pro většinu příkazů toku řízení.
- Stejně tak není implementována funkcionalita týkající se mnoha výrazů, hlavně operátorů.
- Během práce se také objevily požadavky na zavedení nové funkcionality od vývojářů zpětného překladače.

Kapitola 7

Provedená rozšíření

V této sekci si představíme změny v *aplikačním rámci* provedené autorem v průběhu práce.

7.1 Rozčlenění modulu analyzátoru jazyka C

V první fázi provádění změn byla část *aplikačního rámce* sloužící k analýze kódu v jazyce *C* z jediného modulu obsahujícího veškerý kód rozčleněna na množství modulů, z nichž každý zodpovídá za jediný úkol. Původní modul *c_parser* byl nahrazen balíčkem stejného jména. Pro jemnější rozčlenění byly uvnitř balíčku zavedeny podbalíčky *exprs*, obsahující moduly týkající se výrazů, *stmts*, obsahující moduly příkazů, a *types*, který obsahuje moduly pro datové typy. Balíček *exprs* je rozdělen ještě dále, a to na podbalíčky *binary_ops* pro binární operátory, *literals* pro literály a *unary_ops* pro unární operátory. Výsledná struktura je naznačena v ukázce 7.1.

```
c_parser
    exprs
        binary_ops
        literals
        unary_ops
    stmts
    types
```

Kód (7.1) Struktura balíčku c_parser

Toto rozčlenění umožňuje mnohem snadnější orientaci v celém kódu. Okamžitě je patrné z jakých částí se analyzátor kódu skládá, a pro které prvky jazyka *C* existuje podpora. Totožným způsobem byly rozčleněny odpovídající jednotkové testy. Toto opět vede k větší přehlednosti. Podstatnou výhodou je mnohem snazší dohledatelnosti toho, které případy jsou či nejsou pokryty testy.

7.2 Předávání objektu místo řetězce se jménem

Ve funkcích, které očekávají jako argument řetězec se jménem (např. *funkce*, *struktury*) byla přidána možnost předat místo řetězce objekt reprezentující daný element.

7.3 Dump modulu nebo funkce

Byla přidána možnost vypsat obsah zvoleného *modulu* nebo *funkce*. V případě *modulu* se jedná například o seznam *globálních proměnných*, *funkcí* nebo definovaných *enumů*. V případě *funkce* poté o její *návratový typ*, *parametry*, seznam *for cyklů* v jejím těle apod. Do zpětného překladače byl také přidán skript, který zpracuje zadaný kód v jazyce *C*, vypíše chyby při překladu a *dump* daného *modulu*.

7.4 Podpora nových datových typů

Do *aplikačního rámce* byla přidána podpora pro datový typ *union*. Jedná se o typ, který sdílí mnoho vlastností s typem *struct*, proto byla zároveň pro oba zavedena bázová třída. Navenek vypadají oba typy prakticky totožně, ale na rozdíl od struct, *union* má vždy definovanou nejvýše jednu svoji složku. U obou typů *aplikační rámec* umožňuje zjištění, zda jsou pojmenované, výpis jejich jména (s názvem typu nebo bez). Dále je možný výpis jejich složek, zjištění jejich existence a počtu. Příklad *unionu* vidíme na ukázce 7.2.

```
union U {
    int i;
    double d;
};
```

Kód (7.2) Definice unionu se jménem U

Dalším přidaným typem byl *enum*, výčtový typ. Konstanty definované v těle *enum* mají celočíselné hodnoty. Pokud nejsou explicitně definovány, začínají jejich hodnoty postupně od 0 a každá další je o 1 vyšší než předchozí. *Aplikační rámec* umožňuje zjištění toho, zda má daný *enum* jméno a případně jaké. Také je možné vypsat jeho prvky a zjistit, či existují a kolik jich je. Příklad *enum*u vidíme na ukázce 7.3

```
enum Color {
    RED = 1, GREEN, BLUE
};
```

Kód (7.3) Enum reprezentující základní barvy

Přidán byl také typ *funkce*. Díky tomu bylo umožněna analýza *ukazatele na funkci*. Lze se dotázat na *návratový typ* dané *funkce*, *počet* a *datové typy* jejích *parametrů* a na to, zda je *funkce variadická*, tedy jestli je schopna přijímat proměnlivý, předem nespecifikovaný, počet parametrů. Ukazatel na funkci vidíme v ukázce 7.4.

```
int addOne(int x) { return x + 1; }

int (*addOnePtr)(int);
```

Kód (7.4) Ukazatel na funkci addOne

Posledním přidaným datovým typem byl *bool*. Ten byl zaveden do jazyka *C* ve verzi *C99*. Proměnná tohoto typů může nabývat booleovské hodnoty *true* a *false*.

Datový typ *pole* původně podporoval pouze jednodimenzionální variantu. V rámci této práce byl rozšířen o podporu dimenze libovolné velikosti. *Aplikační rámec* nově umožňuje dotaz na počet dimenzí daného pole. Výpis počtu prvků byl oproti původnímu stavu pozměněn a místo jedné skalární hodnoty je nyní vrácena n-tice obsahující počet prvků pro jednotlivé dimenze pole.

7.5 Rozšíření třídy Module

Přidání nových datových typů bylo reflektováno také ve třídě analyzující *moduly* jazyka *C*. Nově je možné se dotazovat na *struct*, *union* a *enum* definované v *modulu*. Pro každý z vyjmenovaných je možné vypsat všechny zástupce daného typu v daném *modulu*, případně zvlášť jejich pojmenované a nepojmenované varianty. Stejně tak je možné zjistit jejich existenci, případně počty a jména. Také lze zjistit, zda jsou (případně jsou pouze) v *modulu* prvky specifikovaných jmen. V případě *enumů* je také možné vypsat jména všech definovaných položek a jejich počet.

V ukázce 7.5 vidíme příklad ověření toho, že *modul* obsahuje právě *pojmenované struktury S* a *S2*, počet unií definovaných v *modulu* je roven *2* a nalezneme zde právě jeden *enum* (se jménem *e*) obsahující právě jeden prvek, který se jmenuje *a*.

```
self.assertTrue(module.has_just_named_structs('S', 'S2'))

self.assertEqual(len(module.unions), 2)

self.assertEqual(len(module.enums), 1)
enum = module.named_enums[0]
self.assertEqual(enum.name, 'e')
self.assertEqual(enum.item_count, 1)
self.assertEqual(enum.item_names[0], 'a')
```

Kód (7.5) Ukázka testu obsahu modulu

7.6 Přidané příkazy jazyka C

V rámci této práce byla přidána podpora pro analýzu většiny (podpora pro *cyklus for* existovala před zahájením práce) základních příkazů jazyka *C*. V této sekci si je postupně představíme.

Nejjednodušším příkazem je *prázdný příkaz*. Tento příkaz nemá žádný efekt a neexistuje informace, kterou by mělo smysl o něm zjišťovat. *Aplikační rámec* pouze umožňuje vytvořit jeho instanci z uzlu stromu získaného zpracováním vstupního kódu pomocí *clang*. Prázdný příkaz je reprezentován samostatným středníkem, jak je patrné z ukázky 7.6.

```
;
```

Kód (7.6) Prázdný příkaz

Dalším příkazem implementovaným v rámci této práce byl *return*, sloužící pro návrat z *funkce*. Je možné se dotázat, zda tento příkaz vrací nějakou hodnotu a případně získat výraz, jehož hodnota je vracena. Příklady příkazu *return* nalezneme v ukázkách 7.7 a 7.8.

```
void func(void) {
    return;
}
```

Kód (7.7) Příkaz return nevracející hodnotu

```
int addOne(int x) {
    return x + 1;
}
```

Kód (7.8) Příkaz return vracející hodnotu výrazu x + 1

Větvení toku programu je umožněno příkazem *if-then-else*, jehož podpora byla v rámci této práce zavedena také. *Aplikační rámec* umožňuje zjištění výrazu v podmínce *if*, který rozhoduje o tom, která větev programu se vykoná. Dále je možné zjistit existenci *else* větve. Na tomto místě je vhodné zmínit jedno specifikum tohoto příkazu. Pokud programátor potřebuje rozdělit tok programu do více než dvou větví, typicky přidá další příkaz *if* do těla *else if*u, ve kterém se právě nachází. Je ustálenou konvencí tento nový *if* začít na stejném řádku, na kterém je zapsáno klíčové slovo *else* předchozího *if*u a neobalovat nový *if* složenými závorkami. Příklad nalezneme v ukázce 7.9. Takto může vznikat dojem, že jazyk *C* umožňuje v rámci jediného příkazu *if* rozdělit tok programu do více než dvou větví, podobně jako například jazyk *Python* se svojí strukturou *if-elif-else*. Reálně se ale vždy jedná o několik příkazů, jak je vidět ukázce 7.10.

```
if (x == 1) {
    foo();
} else if (x == 2) {
    bar();
} else {
    baz();
}
```

Kód (7.9) Tradiční zápis rozvětvení programu do více než dvou větví

```
if (x == 1) {
    foo();
} else {
    if (x == 2) {
        bar();
    } else {
        baz();
    }
}
```

Kód (7.10) Reálná struktura skrývající se za else if

Přidána byla také podpora pro příkaz *goto*, od jehož používání jsou programátoři v rámci zásad praktik správného vývoje krom výjimečných případů odrazováni. Je možné zjistit, jaké *návěští* je cílem skoku. Příklad tohoto příkazu nalezneme v ukázce 7.11.

```
goto zacatek_vnejsi_smycky;
```

Kód (7.11) Příkaz goto

Dalším přidaným příkazem je příkaz *definice proměnné*. *Aplikační rámec* umožňuje získat objekt dané proměnné, ve kterém je obsažena informace o jejím jménu a typu. Také je možno zjistit, zda byla proměnná zároveň inicializována a případně získat její hodnotu. Příklad vidíme na ukázce 7.12.

```
int x = 42;
```

Kód (7.12) Definice proměnné a inicializace její hodnoty

Také smyčky *while* a *do while* byly přidány. Těla obou smyček se vykonávají v závislosti na pravdivostní hodnotě zadaného výrazu. Jedinou odlišností je to, že v případě smyčky *while* je daný výraz vyhodnocován na začátku každé iterace, zatímco v případě smyčky *do while* je vyhodnocen až po provedení jejího těla. Tělo smyčky *do while* se tedy vždy vykoná alespoň jednou. Pro obě smyčky je umožněn dotaz na jejich řídicí výraz. Příklad smyčky *while* vidíme v ukázce 7.13, *do while* poté v 7.14.

```
while(0) {
    doNotCallMe();
}
```

Kód (7.13) Smyčka while

```
do {
    callMeOnce();
} while(0);
```

Kód (7.14) Smyčka do while

Posledním skutečně zajímavým příkazem je *switch*. Pomocí tohoto příkazu můžeme tvořit vícecestné rozhodovací konstrukce. Po vyhodnocení řídicího výrazu pokračuje vykonávání programu první alternativou *case*, která má odpovídající hodnotu. Poté se pokračuje dalšími příkazy v těle *switch* až do jeho konce nebo do té doby, dokud nedojde k vykonání příkazu *break*, po kterém se vykonávání programu přenese za konec daného příkazu *switch*. Pokud hodnotě řídicího výrazu neodpovídá žádná alternativa *case* a je definována alternativa *default*, pokračuje vykonávání programu zde. *Aplikační rámec* umožňuje získat řídicí výraz, zjistit, zda existují nějaké *case* a případně získat jejich hodnoty a zjistit, zda je definována alternativa *default*. Příklad příkazu *switch* nalezneme v ukázce 7.15.

```
switch (x) {
    case 1:
        foo();
    case 2:
        bar();
        break;
    default:
        baz();
}
```

Kód (7.15) Příkaz switch

Zbývá už pouze zmínka o příkazech *break* a *continue*. Oba můžeme naleznout v tělech smyček, *break* navíc také v těle příkazu *switch*. Příkaz *continue* slouží k ukončení vykonávání současné iterace smyčky a přesunu na začátek iterace další. Pomocí příkazu *break* přerušíme vykonávání těla nadřazeného příkazu a přesuneme vykonávání programu na instrukci následující po konci nadřazeného příkazu. U těchto příkazu neexistují žádné smysluplné vlastnosti, které by bylo možné získávat. Pro všechny příkazy smyček byla přidána podpora pro zjišťování existence a výpis *break* a *continue* v jejich tělech. Příklady příkazu *continue* nalezneme v ukázce 7.16, *break* v ukázce 7.15 (case 2).

```
while(1) {
    continue;
}
```

Kód (7.16) Příkaz continue

Pro všechny příkazy byla přidána možnost dotázat se, který příkaz po nich následuje. Byla také implementována funkcionalita umožňující instancovat objekty příkazů z uzlů abstraktního syntaktického stromu získaného pomocí *clang*.

7.7 Rozšíření třídy Function

Třída Function byla obohacena o možnost získávání informací o všech příkazech zmíněných v předchozí sekci v jejím těle. Je možné se dotazovat na přítomnost určitého druhu příkazu (jednotlivé příkazy daného druhu lze navíc blíže specifikovat jejich přesnou podobou) a příkazy vypsat.

Ukázka 7.17 obsahuje otestování toho, že *funkce* obsahuje nějaké příkazy *return*, konkrétně potom takový, který vrací hodnotu *0*. Také je ověřena přítomnost právě jednoho příkazu *if*, který má *else* větev a řídícím výrazem je *1*. *Funkce* by neměla obsahovat žádnou smyčku *while*. Také bychom ve *funkci* měli najít právě jeden příkaz *switch* s řídícím výrazem *'a'*, dvěma *case* a také *default*. V neposlední řadě nás zajímá existence právě jednoho *goto*, jehož cílem je *návěští abc*.

```
self.assertTrue(func.has_any_return_stmts())
self.assertTrue(func.has_return_stmts('return 0'))

self.assertEqual(len(func.if_stmts), 1)
self.assertTrue(func.if_stmts[0].has_else_clause())
self.assertEqual(func.if_stmts[0].condition, '1')

self.assertFalse(func.has_any_do_while_loops())

self.assertEqual(len(func.switch_stmts), 1)
self.assertEqual(func.switch_stmts[0].switch_expr, 'a')
self.assertTrue(func.switch_stmts[0].has_cases())
self.assertEqual(len(func.switch_stmts[0].cases), 2)
self.assertTrue(func.switch_stmts[0].has_default_case())

self.assertEqual(len(func.goto_stmts), 1)
self.assertEqual(func.goto_stmts[0].target, 'abc')
```

Kód (7.17) Ukázka testu obsahu funkce

7.8 Přidané výrazy jazyka C

V rámci této práce byla do *aplikačního rámce* přidána podpora pro mnoho výrazů jazyka *C*.

Jako první uveďme výraz *volání funkce*. Lze zjistit *jméno* volané *funkce*, a to zda byla volána s nějakými argumenty a případně získat jejich hodnoty. Výraz *volání funkce* vidíme na ukázce 7.18

```
foo(1, 'hello')
```

Kód (7.18) Volání funkce se dvěma parametry

Dalším z přidaných výrazů je *přetypování*. *Aplikační rámec* umožňuje získat přetypovávaný výraz a cílový typ. Příklad přetypování vidíme v ukázce 7.19.

```
(int) 5.0
```

Kód (7.19) Přetypování desetinného čísla na typ int

Přidán byl také *inicializátor pole*, který umožňuje získat seznam hodnot použitých k inicializaci. Příklad nalezneme v ukázce 7.20.

```
int pole[] = {1, 2, 3};
```

Kód (7.20) Přetypování desetinného čísla na typ int

Dále byla přidána podpora pro množství *operátorů*. Jako první se podívejme na speciální případ – *ternární operátor*. Tento výraz nabývá jednu ze dvou hodnot v závislostí na dané podmínce. Je možné získat výraz řídicí podmínky a výrazy obou možných hodnot. Ternární operátor vidíme na ukázce 7.21.

```
(i > 5) ? 1 : 0
```

Kód (7.21) Ternární operátor nabývající hodnotu 1 nebo 0

O jeden operand méně mají *binární operátory*. U všech je možné se dotázat na *hodnoty* na jejich levé a pravé straně a také na to, jestli se jedná o *operand* očekávaného *typu*. Podporovány jsou *operátory* uvedené v tabulce 7.1.

Tab. 7.1: Podporované binární operátory

+	součet
−	rozdíl
∗	násobení
/	dělení
%	modulo
&&	logické AND
||	logické OR
&	bitové AND
|	bitové OR
∧	bitové XOR
<<	bitový posun doleva
>>	bitový posun doprava
[]	přístup k položce pole
=	přiřazení
,	čárka
==	rovnost
! =	nerovnost
<	menší
>	větší
<=	menší nebo rovno
>=	větší nebo rovno
+ =, − =, ∗ =, / =, % =, & =, | =, ^=, <<=, >>=	složená přiřazení

Pouze jeden operand mají *unární operátory*. Můžeme zjišťovat *hodnotu* tohoto *operandu* a to, jestli se jedná o *operand* očekávaného *typu*. V tabulce 7.2 najdeme podporované *unární operátory*.

Tab. 7.2: Podporované unární operátory

∗	dereference ukazatele
&	reference ukazatele
−	negace
!	logická negace
++	inkrementace (pre i post)
−−	dekrementace (pre i post)

Kapitola 8

Testování a výsledky

Tato sekce se zabývá testováním. Postupně se podíváme na dvě podoby této činnosti. Nejprve si představíme *jednotkové testování*, kterým byla ověřováno to, že se samotný *aplikační rámec* chová očekávaným způsobem. Následně si ukážeme, jak lze tento *aplikační rámec* použít při tvorbě *regresních testů* pro *zpětný překladač*.

8.1 Jednotkové testy

Účelem těchto testů je ověření očekávaného chování vyvíjeného *aplikačního rámce* postupně po krátkých úsecích jeho kódu – například po jednotlivých metodách testované třídy. V ukázce 8.1 vidíme příklad metody z modulu *function.py*.

```
def has_any_return_stmts(self):
    """Does the function contain any return statements?"""
    return bool(self.return_stmts)
```

Kód (8.1) Metoda has_any_return_stmts modulu function.py

U takovéto metody budeme zřejmě chtít ověřit, že v případě, kdy objekt reprezentuje *funkci* obsahující v těle příkaz *return*, bude metoda vracet hodnotu *True*. Odpovídající jednotkový test si můžeme prohlédnout v ukázce 8.2.

```
def test_has_any_return_stmts_return_stmt(self):
    func = self.get_func("""
        void func() {
            return;
        }
    """, 'func')
    self.assertTrue(func.has_any_return_stmts())
```

Kód (8.2) Jednotkový test očekávající hodnotu True

V situaci, kdy *funkce* žádný příkaz *return* neobsahuje, očekáváme návratovou hodnotu *False*. Toto chování ověřuje test v ukázce 8.3

```
def test_has_any_return_stmts_no_return_stmt(self):
    func = self.get_func("""
        void func() {}
    """, 'func')
    self.assertFalse(func.has_any_return_stmts())
```

Kód (8.3) Jednotkový test očekávající hodnotu False

Takovýchto testů bylo v průběhu práce vytvořeno zhruba sedm set. Autorovou snahou bylo dosažení pokrytí řádků kódu co nejvíce se blížícího 100%. Kromě klasických případů ověřování očekávané hodnoty byl také v případech, kde takováto věc měla smysl, testován kontrakt mezi metodami *__hash__* a *__eq__*. Platí-li, že dva objekty jsou si rovny, musí také platit, že hodnoty jejich *__hash__* jsou totožné. Pokrytí kódu jednotkovými testy kromě ověřování jeho funkčnosti také například velice usnadňuje jeho refaktorování.

8.2 Použití aplikačního rámce

Autorovi práce se bohužel nepodařilo sestavit takový vstupní kód, ze kterého by získal výstupní kód obsahující použitelný příklad nějakého zajímavého konstruktu jazyka *C*, pro které byla do *aplikačního rámce* regresních testů přidána podpora. První testovací případ provedl autor na starší verzi zpětného překladače a na aktuální se mu ho nepodařilo zreprodukovat s uspokojivým výsledkem. Ve zbývajících ukázkách se pro účely představení další funkcionality budeme muset uchýlit k použití falešných výstupů. Představme si, že takovýto kód *zpětný překladač* skutečně produkuje a my chceme otestovat jeho vlastnosti.

8.2.1 Výsledek regresních testů

Při spouštění *regresních testů* nás v první řadě zajímá jejich výsledek. V ideálním případě chceme získat informaci o tom, že všechny testy proběhly v pořádku. V případě, že došlo k chybě, jistě budeme chtít zjistit, co přesně přineslo neočekávaný výsledek.

Na ukázkách 8.4 a 8.5 si můžeme prohlédnout výstupy pro obě situace. V obou případech je spouštěn testovací případ *testcase* umístěný v adresáři `testsuite/regression-tests`. Testovací případ používá kód umístěný v souboru *code.c*, překlad probíhá pomocí *clang* bez použití optimalizace, cílovou architekturou je *x86* a použitý formát spustitelného souboru je *elf*.

```
Running 1 test case in /testsuite/regression-tests/testcase

testcase.Test (code.c -a x86 -f elf -c clang -C -O0) [  OK  ] (1.64s)

SUCCESS (3/3)
```

Kód (8.4) Úspěšný průběh regresních testů

V prvním případě test proběhl úspěšně. Vidíme informaci o spuštění testů v zadaném adresáři, dále následuje seznam spuštěných testovacích tříd (*Test*), pro každou třídu poté parametry běhu testu, celkový výsledek a dobu trvání běhu. Na konci výstupu vidíme celkový výsledek.

```
Running 1 test case in /testsuite/regression-tests/testcase

testcase.Test (code.c -a x86 -f elf -c clang -C -O0) [ FAIL ] (1.24s)
F..
======================================================================
FAIL: test_for_main (testcase.Test)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/testsuite/regression-tests/testcase/test.py", line 19,
  in test_for_main
    self.assertEqual(len(main.return_stmts), 2)
AssertionError: 3 != 2

----------------------------------------------------------------------
Ran 3 tests in 0.073s

FAILED (failures=1)


FAIL (failed 1 out of 3 tests)
```

Kód (8.5) Neúspěšný průběh regresních testů

V druhém případě byla v testu pozměněna jedna očekávaná hodnota, a test proto skončil neúspěchem. Z výstupu je zřejmé, že k chybě došlo ve třídě *Test*. O řádek níže vidíme výsledky jednotlivých metod testovací třídy. Písmeno *F* na první pozici nám prozrazuje, že chyba se vyskytla v první spuštěné metodě. Následuje detailní popis problému. Na řádku *19* v souboru *test.py* v metodě *test_for_main* bylo očekáváno, že velikost listu příkazů *return* je *2*; ve skutečnosti to ovšem bylo *3*. Třída *Test* tedy končí selháním, a stejně tak i celý běh regresních testů.

8.2.2 Testovací případy

Ve zbytku této kapitoly si demonstrujeme použití *aplikačního rámce* a podrobněji si rozebereme několik ukázových testovacích případů.

8.2.3 Testovací případ 1

První testovací případ je pro svoji délku umístěn v příloze 1. Test je spuštěn s nastavením specifikovaným v ukázce 8.6.

```
settings = TestSettings(
    input='testcase.c',
    arch='x86',
    format=ALL,
    compiler='clang',
    compiler_opts=['-O0'],
    debug_info=False,
    strip=False
)
```

Kód (8.6) Nastavení testovacího případu 1

Zdrojový kód 1 byl přeložen na architektuře *x86* překladačem *clang*. Parametr *-O0* říká, že při překladu nebude použita optimalizace. Vytvořen je spustitelný soubor formátu *PE*. Tento soubor je předán zpětnému překladači, který pro něj vyprodukuje výstup umístěný v příloze 2. Nad vyprodukovaným kódem můžeme spouštět různé testy. Například by nás mohlo zajímat, zda výsledný kód obsahuje funkci se jménem *main*. Takovýto test vidíme v ukázce 8.7.

```
def test_for_main(self):
    assert self.out_c.has_funcs('main')
```

Kód (8.7) Ověření existence funkce main

Také můžeme chtít ověřit, že vyprodukovaný kód obsahuje funkci se jménem *bar* nebo *_bar*, která obsahuje právě jeden příkaz *return* vracející hodnotu *97*. Tento případ si můžeme prohlédnout v ukázce 8.8.

```
def test_function_bar(self):
    bar = self.out_c.func('bar', '_bar')
    self.assertTrue(bar.has_any_return_stmts())
    self.assertEqual(len(bar.return_stmts), 1)
    self.assertEqual(bar.return_stmts[0].return_expr, 97)
```

Kód (8.8) Ověření existence a vlastností funkce bar

Jako poslední si ukažme otestování toho, že se nám pomocí zpětného překladače podařilo získat kód, který obsahuje funkci se jménem *foo* nebo *_foo*, která ve svém těle obsahuje právě jeden výraz *přiřazeni hodnoty*, a to navíc ve tvaru $g1 = 5$. Testovací případ nalezneme v ukázce *testcase-foo*.

```
def test_function_foo(self):
    foo = self.out_c.func('foo', '_foo')
    self.assertEqual(len(foo.assignments), 1)
    self.assertTrue(foo.has_assignments('g1 = 5'))
```

Kód (8.9) Ověření existence a vlastností funkce foo

8.2.4 Testovací případ 2

Mějme výstupní kód z ukázky 8.10.

```
int main(int argc, char **argv) {
    int a;

    switch(a) {
        case 7: ;
        case 5: ;
        default:
            break;
    }

    return 0;
}
```

Kód (8.10) Testovací případ 2 - výstup

V takovémto případě zřejmě budeme chtít ověřit, že ve funkci *main* se nachází příkaz *switch*. Dále nás může zajímat, že *řídicím výrazem* je hodnota proměnné *a*. Můžeme také testovat, že se v těle příkazu nacházejí větve *case*, a to dvě. Zajímat nás nejspíš budou i podmínky pro vstup do daných větví. Ověřit můžeme i to, že se zde nachází také větev *default*. Všechny tyto případy vidíme v ukázce 8.11.

```
def test_switch(self):
    main = self.out_c.func('main')
    self.assertEqual(len(main.switch_stmts), 1)

    switch_stmt = main.switch_stmts[0]
    self.assertEqual(switch_stmt.switch_expr, 'a')
    self.assertTrue(switch_stmt.has_cases())
    self.assertTrue(switch_stmt.has_default_case())
    self.assertEqual(len(switch_stmt.cases), 2)
    self.assertEqual(len(switch_stmt.cases[0].condition), 7)
    self.assertEqual(len(switch_stmt.cases[1].condition), 5)
```

Kód (8.11) Ověření vlastností příkazu switch

8.2.5 Testovací případ 3

Jako další příklad použijme výstupní kód v ukázce 8.12, který obsahuje dvě smyčky.

```
int main(void) {

    do {
        continue;
    } while(1+1);

    int a = 7;
    while(a++) {
        break;
    }

    return 0;
}
```

Kód (8.12) Testovací případ 3 - výstup

V ukázce 8.13 vidíme test zjišťující, zda výstupní kód skutečně obsahuje jednu smyčku *do while*, jejíž řídicím výrazem je *součet*, který má na levé i pravé straně hodnotu *1*. V těle smyčky očekáváme příkaz *continue*. Ve výstupu též předpokládáme existenci smyčky *while* s řídicím výrazem *postinkrementace* proměnné *a*. V jejím těle by se měl nacházet příkaz *break*.

```
def test_while_loops(self):
    main = self.out_c.func('main')
    self.assertEqual(len(main.do_while_loops), 1)
    self.assertEqual(len(main.while_loops), 1)

    do_while_loop = main.do_while_loops[0]
    self.assertTrue(do_while_loop.condition.is_add_op())
    self.assertEqual(do_while_loop.condition.lhs, 1)
    self.assertEqual(do_while_loop.condition.rhs, 1)
    self.assertTrue(do_while_loop.has_any_continue_stmts())

    while_loop = main.while_loops[0]
    self.assertTrue(while_loop.condition.is_post_increment_op())
    self.assertEqual(while_loop.condition.op, 'a')
    self.assertTrue(while_loop.has_any_break_stmts())
```

Kód (8.13) Testování smyček while a do while

8.2.6 Testovací případ 4

Nyní si ukažme testování *ternárního operátoru* na němž závisí *návratová hodnota* funkce *main* z ukázky 8.14.

```
int main(int argc) {
    return (argc < 5) ? foo(4, 5) : bar();
}
```

Kód (8.14) Testovací případ 4 - výstup

Ukázka 8.15 obsahuje test kontrolující plastnost následujících předpokladů. Funkce *main* obsahuje jeden příkaz *return*, který vrací hodnotu *ternárního operátoru*. Podmínkou tohoto operátoru je výraz *menší než*, který má na levé straně proměnnou *argc* a na pravé hodnotu *5*. V případě, že je výraz pravdivý, nabývá *ternární operátor* hodnoty získané zavoláním funkce *foo*, které je provedeno s předáním dvou argumentů. Pro nepravdivý případ je bez argumentů zavolána funkce *bar*.

```
def test_while_loops(self):
    main = self.out_c.func('main')
    self.assertEqual(len(main.return_stmts), 1)

    return_stmt = main.return_stmts[0]
    self.assertTrue(return_stmt.return_expr.is_ternary_op())
    ternary_op = return_stmt.return_expr
    self.assertTrue(ternary_op.condition.is_lt_op())
    self.assertEqual(ternary_op.condition.lhs, 'argc')
    self.assertEqual(ternary_op.condition.rhs, 5)
    self.assertTrue(ternary_op.true_value.is_call())
    self.assertEqual(ternary_op.true_value.name, 'foo')
    self.assertEqual(len(ternary_op.true_value.args), 2)
    self.assertTrue(ternary_op.false_value.is_call())
    self.assertEqual(ternary_op.false_value.name, 'bar')
    self.assertFalse(ternary_op.false_value.has_args())
```

Kód (8.15) Analýza příkazu return a ternárního operátoru

8.2.7 Testovací případ 5

Jako poslední představme výstupní kód obsahující několik případů přiřazení v ukázce 8.16.

```
int main(void) {
    int a = 5;
    a = 6;
    int b = 2;
    b = b + a;
    return 0;
}
```

Kód (8.16) Testovací případ 5 - výstup

V ukázce 8.17 nalezneme otestování očekávaných hodnot na levých a pravých stranách *přiřazovacích výrazů*. Za povšimnutí stojí fakt, že kromě přístupu pomocí *indexu*, můžeme k jednotlivým položkám polí obsahujících prvky jazyka *C* nacházející se v dané funkci přistupovat také pomocí jejich *řetězcové reprezentace*. Tato možnost byla také přidána v rámci této práce.

```
def test_while_loops(self):
    main = self.out_c.func('main')
    self.assertEqual(len(main.assignments), 4)

    self.assertEqual(main.assignments['a = 5'].lhs, 'a')
    self.assertEqual(main.assignments['a = 5'].rhs, 5)
    self.assertEqual(main.assignments['a = 6'].lhs, 'a')
    self.assertEqual(main.assignments['a = 6'].rhs, 6)
    self.assertEqual(main.assignments['b = 2'].lhs, 'b')
    self.assertEqual(main.assignments['b = 2'].rhs, 2)
    self.assertEqual(main.assignments['b = b + a'].lhs, 'b')
    self.assertEqual(main.assignments['b = b + a'].rhs, 'b + a')
```

Kód (8.17) Testování přiřazení

Kapitola 9

Závěr

Na tomto místě si shrňme výsledky dosažené v rámci této práce. Cílem bylo seznámení s problematikou zpětného inženýrství se zvláštním zaměřením na oblast informačních technologií. Dále bylo nutné nastudovat *zpětný překladač* společnosti *AVG* a technologie, které jsou v něm použity – jmenovitě *LLVM* a *clang*. Poté bylo třeba se zaměřit na oblast verifikace programů. Hlavním výstupem této práce mělo být rozšíření *aplikačního rámce* pro tvorbu regresních testů. Bylo tedy nejprve nutné zjistit, jakým způsobem jsou testy tvořeny, jak se nastavují jejich parametry a jak jsou spouštěny. Pro vývoj *aplikačního rámce* bylo samozřejmě také nutné pochopit jeho strukturu před začátkem této práce a poté se rozhodnout, jaká rozšíření by bylo vhodné vytvořit. Provedená práce se týkala hlavně přidání podpory pro chybějící elementy jazyka *C*. Jmenovitě se jednalo o některé složitější datové typy, většinu příkazů toku řízení a mnoho výrazů, hlavně operátorů. Dále byla přidána funkcionalita rozšiřují analýzu *modulů* a funkcí jazyka *C* o nové elementy. Byly opraveny některé chyby a přidána možnost získávat *dump modulu* či *funkce*. S provedenými rozšířeními byly zároveň dodávány *jednotkové testy* pro tuto novou funkcionalitu. Nová funkcionalita byla použita v ukázkových regresních testech.

Z pohledu budoucího vývoje by bylo možné se zaměřit na analýzu těl jednotlivých konstruktů. Tedy například umožnit vypsání všech příkazů nalézajících se v těle *else* větve určitého příkazu *if*. Stále také existují prvky jazyka, pro které nebyla přidána podpora. Takovým je třeba *typedef*.

Literatura

[1] Interní dokumentace AVG.

[2] An introduction to cryptography and cryptanalysis. [cit. 2016-12-21].
URL http://math.scu.edu/~eschaefe/book.pdf

[3] LLVM. [cit. 2016-05-10].
URL http://www.aosabok.org/en/llvm.html

[4] LLVM. [cit. 2016-05-10].
URL http://llvm.org/

[5] Progresní a regresní testy. [cit. 2016-05-11].
URL http://testovanisoftwaru.cz/tag/regresni-testy/

[6] Kernighan, B. W.; Ritchie, D. M.: *Programovací jazyk C*. Computer Press, 2006, ISBN 80-251-0897-X.

[7] Křoustek, J.: *Rekonfigurovatelná analýza strojového kódu*. Dizertační práce, Fakulta informačních technologií VUT v Brně, 2014.

[8] Peringer, P.: Materiály k předmětu Jazyk C, FIT VUT v Brně. [cit. 2016-05-12].

[9] Smrčka, A.: Materiály k předmětu Testování a dynamická analýza, FIT VUT v Brně. [cit. 2016-05-11].

Přílohy

```
char bar(void) {
    return 'a';
}

int foo(void) {
    int x = 5;
    return x;
}

int main(int argc, char **argv) {
    if(argc < 4) {
        int i = foo();
        while(i > 0) {
            ;
            i--;
        }
    } else {
        do {
            continue;
        } while(0);
    }

    while(1) {
        bar();
        break;
    }

    return 0;
}
```

Kód (1) Testovací případ 1 - vstup

```c
#include <stdint.h>

// ------------------- Function Prototypes --------------------

int32_t _bar(int32_t a1);
void _foo(void);

// --------------------- Global Variables ---------------------

int32_t g1 = 0; // eax

// ------------------------ Functions -------------------------

// Address range: 0x401560 - 0x401565
int32_t _bar(int32_t a1) {
    // 0x401560
    return 97;
}

// Address range: 0x401570 - 0x40157c
void _foo(void) {
    // 0x401570
    g1 = 5;
}

// Address range: 0x401580 - 0x4015df
int main(int argc, char ** argv) {
    // 0x401580
    ___main();
    g1 = (int32_t)argv;
    int32_t v1;
    if (argc > 3) {
        // 0x4015c7
        _bar(v1);
        return 0;
    }
    // 0x4015a4
    _foo();
    char * v2 = *(char **)g1; // 0x4015ac2
    if (v2 < (char *)1) {
        // 0x4015c7
        _bar(v1);
        return 0;
    }
    int32_t v3 = (int32_t)v2 - 1; // 0x4015b5
    g1 = v3;
    char * v4 = *(char **)v3; // 0x4015ac
    while (v4 >= (char *)1) {
        // 0x4015b2
        v3 = (int32_t)v4 - 1;
        g1 = v3;
        v4 = *(char **)v3;
        // continue -> 0x4015b2
    }
    // 0x4015c7
    _bar(v1);
    return 0;
}
```

Kód (2) Testovací případ 1 - výstup (x86, PE, clang -O0)