VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

DEKÓDOVÁNÍ BINÁRNÍHO KÓDŮ DO VYŠŠÍ FORMY REPREZENTACE

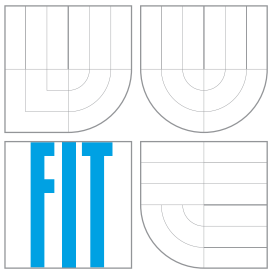
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. LUKÁŠ MACKO

BRNO 2015

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

DEKÓDOVÁNÍ BINÁRNÍHO KÓDŮ DO VYŠŠÍ FORMY REPREZENTACE

BINARY-CODE DECODING TO A HIGH-LEVEL REPRESENTATION

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. LUKÁŠ MACKO

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. PETER MATULA

BRNO 2015

Abstrakt

Diplomová práce se zabývá zpětným inženýrstvím v oblasti software. Představuje jeho uplatnění, používané nástroje a postupy. Podrobněji se věnuje problematice dekódování instrukcí. Uvádí dva základní postupy – lineární dekódování a rekurzivní sestup. Poukazuje na jejich silné stránky a nedostatky. Následně představuje zpětný překladač vyvíjený společností AVG Technologies. Tento překladač je rekonfigurovatelný, což umožňuje zpětný překlad z různých architektur do více cílových jazyků. Cílem práce bylo navrhnout a implementovat algoritmus pro dekódování binárních souborů do vyšší reprezentace. Navržený algoritmus vychází z algoritmu rekurzivního sestupu. Využívá informace o toku řízení programu. Pro zpřesnění výsledku dekódování jsou navíc použity záznamy z tabulky symbolů a další informace. Navržený algoritmus byl implementován pro rekonfigurovatelný zpětný překladač společnosti AVG Technologies. Testy implementovaného algoritmu ukázaly zlepšení detekce funkcí v dekódovaných programech. Implementované řešení navíc umožňuje dekódovat i soubory, jejichž analýza s aktuálním řešením nebyla možná.

Abstract

The thesis deals with reverse techniques in software engineering. It presents practical application of software reverse engineering, used tools and approaches. The topic of instruction decoding is discussed in detail. Two basic methods are presented – linear sweep and recursive descent. Their strengths and weaknesses are highlighted. Subsequently a decompiler developed by AVG Technologies is introduced. The decompiler is retargetable. This feature allows to decompile applications of multiple platforms into various target languages. The aim of the thesis is to design and implement algorithm for decoding binary files into high-level representation. The designed algorithm is based on modified recursive descent algorithm, which uses control flow information. In order to achieve more accurate decoding results, symbol table records and other additional information are used. The proposed algorithm was implemented for the AVG Technologies retargetable decompiler. The tests showed that the implemented algorithm improved the function detection in decoded programs. Furthermore, the implemented solution allows to decode files that could not be analysed using the previous version of the decompiler.

Klíčová slova

zpětný překladač, dekódovaní instrukcí, LLVM IR, hybridní analýza

Keywords

decompiler, instruction decoding, LLVM IR, hybrid analysis

Citace

Lukáš Macko: Dekódování binárního kódů do vyšší formy reprezentace, diplomová práce, Brno, FIT VUT v Brně, 2015

Dekódování binárního kódů do vyšší formy reprezentace

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Petra Matulu. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Lukáš Macko
25. května 2015

Poděkování

Chcel by som poďakovať vedúcemu práce Ing. Petrovi Matulovi, za cenné rady a pomoc pri vypracovaní diplomovej práce. Taktiež ďakujem rodičom a kamarátom, ktorí ma podporovali počas štúdia.

Obsah

Kapitola 1

Úvod

V dnešnej dobe sme svedkami vzniku nových technológií, ktoré sa snažia uľahčiť život v rôznych oblastiach. Vývoj nových technológii ovplyvňuje aj softwarové inžinierstvo a jeho aplikácie v rôznych odvetviach.

Spolu s nárastom aplikácii, ktoré vykonávajú činnosť prospešnú pre používateľa, sa rozširuje aj škodlivý software (angl. *malware*). Preto je potrebné zabezpečiť a chrániť vytvorené produkty pred zneužitím. Pri obrane voči škodlivému softwaru sa využívajú aj poznatky získané jeho manuálnou analýzou. Až na výnimočné prípady, zdrojový kód škodlivého softwaru nie je k dispozícii tým, čo ho analyzujú za účelom obrany. Zdrojový kód je ale pre svoju čitateľnosť vhodný na analýzu viac ako binárny súbor. Na zrekonštruovanie zdrojového kódu, respektíve vytvorenie vyššej reprezentácie binárneho súboru sa využívajú spätné prekladače – dekompilátory.

Tie vykonávajú netriviálnu úlohu, nakoľko sa pri preklade zo zdrojového jazyka vypúšťajú niektoré informácie, ktoré nie sú potrebné pre spustiteľný súbor. Existuje niekoľko spätných prekladačov. Jeden z nich je vyvíjaný aj spoločnosťou AVG Technologies. Špecifickou vlastnosťou tohto spätného prekladača je rekonfigurovateľnosť. Tá umožňuje podporu spätného prekladu spustiteľných súborov z rôznych architektúr do rôznych cieľových jazykov. Návrh spätného prekladača umožňuje použiť väčšinu jeho častí pri preklade z rôznych architektúr.

Cieľom práce je navrhnúť a implementovať metódu pre rekonfigurovateľný spätný preklad z binárnej formy do LLVM IR reprezentácie. Navrhovaný algoritmus využíva dostupné informácie pre spresnenie výsledku. Navrhnutý algoritmus je implementovaný pre použitie v spätnom prekladači spoločnosti AVG Technologies. Implementácia je doplnená sadou testov.

Kapitola 2 sa zaoberá teoretickými poznatkami, ktoré čitateľa uvádzajú do problematiky spätného inžinierstva. Po úvodnej definícii predstavuje niekoľko oblastí využitia v priemysle. Následne sú uvedené príklady uplatnenia v softwarovom inžinierstve. Ďalej sú predstavené skupiny nástrojov, ktoré je možné využiť pri reverznom softwarom inžinierstve. Záver kapitoly sa podrobnejšie zameriava na časť dekódovania inštrukcií. Predstavuje základné používané postupy, ich výhody a nedostatky. Kapitola 3 oboznamuje čitateľa s rekonfigurovateľným spätným prekladačom spoločnosti AVG Technologies. Na začiatku predstavuje jeho architektúru a členenie. Následne popisuje jednotlivé časti a vykonávané činnosti v nich. Podrobnejšie je popísaný dekodér inštrukcií, ktorého vylepšenie je cieľom tejto práce. Taktiež sú predstavené technológie použité v spätnom prekladači – platforma LLVM a jazyk ISAC pre popis architektúry. Kapitola 4 popisuje navrhovaný dekodér inštrukcií, ktorý využíva hybridnú analýzu a snaží sa eliminovať nedostatky aktuálne použí-

vaného riešenia. Implementácia je popísané v kapitole 6. Výsledná implementácia bola doplnená sadou testov. Cieľom testovania bolo overiť korektnosť dekódovacieho algoritmu – schopnosť správne dekódovať súbory a dekódovanie súborov, ktoré pri použití súčasného riešenia nebolo možné dekódovať. Pri testovaní bola taktiež skúmaná aj doba dekódovania implementovaným algoritmom. Popis jednotlivých testov a získané výsledky sú uvedené v kapitole 7. Táto kapitola taktiež uvádza výstupy profilácie použitej pre zrýchlenie dekódovacieho algoritmu. Zhrnutie dosiahnutých výstupov práce a možné vylepšenia sú popísané v záverečnej kapitole 8.

Kapitola 2

Spätné inžinierstvo

Táto kapitola sa venuje teoretickému úvodu do spätného inžinierstva. Uvádza všeobecnú definíciu spätného inžinierstva, následne sa zameriava na softwarové spätné inžinierstvo. Sú tu uvedené jeho rôzne aplikácie a taktiež využívané nástroje. Predstavuje dva základné postupy pri dekódovaní inštrukcií, ich princíp, výhody a nedostatky. Informácie sú čerpané z [11, 22, 21] a [10].

2.1 Definícia

Spätné inžinierstvo je proces získavania znalostí z ľudsky vytvorených objektov. Jedná sa o podobný proces ako vedecký výskum s tým rozdielom, že vedecký výskum sa zaoberá najmä prírodnými a prirodzenými javmi.

Cieľom spätného inžinierstva je získať nové poznatky o skúmanom objekte, ktoré nie sú k dispozícii. Dôvody ich nedostupnosti môžu byť rôzne: strata pôvodnej dokumentácie, ukončenie podpory výrobcom, prípadne ich vlastník nie je ochotný poskytnúť. Získané poznatky môžu byť následne použité pri údržbe, prípadne oprave, skúmaného objektu alebo pri výrobe duplikátu.

2.2 Spätné inžinierstvo v priemysle

Reverzné inžinierstvo nachádza uplatnenie v rôznych odvetviach priemyslu. S príchodom CAD (angl. *computer-aided design*) systémov je možné využiť jeho postupy pri tvorbe 3D modelov existujúcich fyzických objektov. Objekt je najprv analyzovaný laserovými skenermi a následne je z výsledkov merania rekonštruovaný 3D model skúmaného objektu.

Pri skúmaní integrovaných obvodov a čipových kariet sa reverzné inžinierstvo zameriava na odkrývanie ich funkčnosti. Jednotlivé vrstvy čipu sú postupne zbrusované a zaznamenávané elektrónovým mikroskopom. Zo získaných snímkov čipu je následne analyzovaná jeho funkčnosť. Úloha spätnej rekonštrukcie obvodu je úmyselne komplikovaná výrobcami skrývaním interne vykonávaných operácii.

Významné uplatnenie nachádza spätné inžinierstvo aj v zbrojnom priemysle a vojenských aplikáciach. Jednotlivé strany sa snažia analyzovať technológiu protivníka a využiť jeho poznatky. Nasledujúci text uvádza niekoľko príkladov z obdobia druhej svetovej vojny a následnej studenej vojny.

- Nemci používali prakticky navrhnuté, odolné a znovupoužitelné palivové kanistre. Britské a americké jednotky používali nádrže, ktoré zvyčajne neumožňovali opätovné

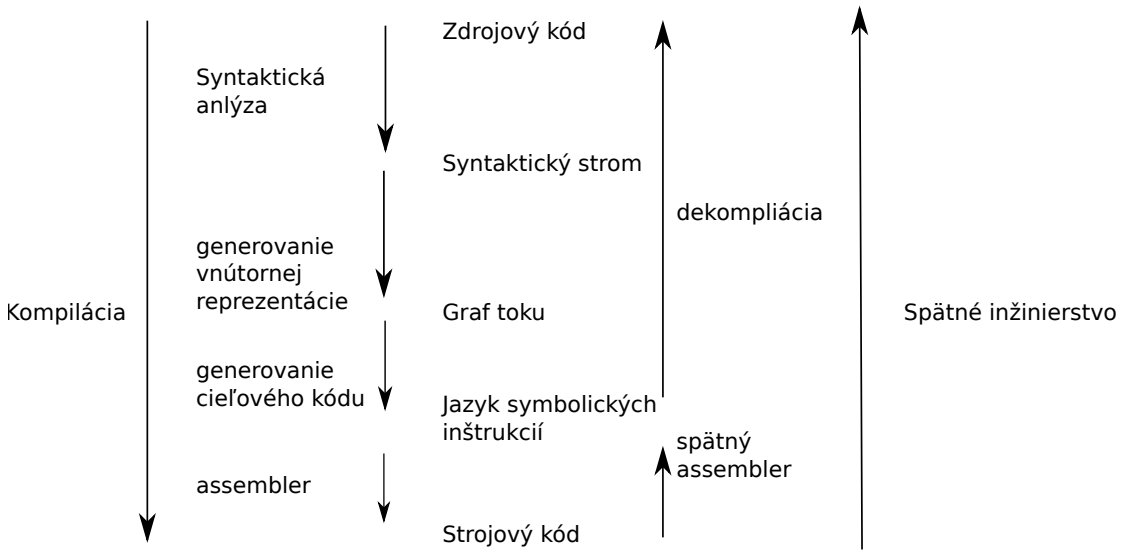
použitie alebo ich výrobné náklady boli vysoké. Potom ako spozorovali kanistre používané Nemcami, podarilo sa im získať niekoľko kusov a na ich základe začali vyrábať vlastné kópie. Kanistre boli známe pod anglickým názvom Jerry can.

- Britská inteligencia počas druhej svetovej vojny analyzovala získané nemecké šifrovacie stroje. Na základe ich analýzy navrhli takzvané bomby, ktoré testovali rôzne nastavenia šifrovacieho stroja a využívali ich k prelomeniu Nemcami šifrovaných správ.
- Tri americké lietadlá B-29 museli núdzovo pristáť na území Sovietskeho zväzu. Po ich rozobraní a analýze sa podarilo Rusom vyrobiť bombardéry Tupolev TU-4, schopné niesť ťažké zbrane.

2.3 Softwarové spätné inžinierstvo

Rovnako ako v iných odvetviach spätné inžinierstvo zasahuje aj do softwarového inžinierstva. Podľa [21] môžeme hovoriť o dvoch základných úlohách. V prvom prípade máme dostupný zdrojový kód a snažíme sa získať informácie o implementovanom algoritme na vyššej úrovni abstrakcie. Druhý prípad predpokladá, že zdrojový kód nie je dostupný a k dispozícii je len výstup kompilácie. Úlohou je rekonštruovať pravdepodobný zdrojový kód.

Nasledujúca časť práce sa zameriava najmä na druhý uvedený prípad. Náročnosť tohto procesu nemusí byť na prvý pohľad zrejmá nakoľko obsahuje podobné kroky ako kompilácia ale v chronologicky opačnom poradí. Proces dekompilácie je znázornený na obrázku 2.1.



Obrázok 2.1: Proces dekompilácie [18]

Zložitosť je spôsobená tým, že proces kompilácie je stratový. Výsledný strojový kód neobsahuje komentáre, mená premenných, funkcií ani informácie o dátových typoch. Tieto informácie nie sú potrebné k fungovaniu skompilovanej aplikácie. Pokiaľ ale chceme spätne zrekonštruovať zdrojový kód je potrebné analyzovať aký dátový typ reprezentuje skúmaná časť pamäte. Naviac, rovnaký zdrojový kód môže byť preložený pre jednu cieľovú platformu

rôznymi spôsobmi. Na základe použitého prekladača a zvolených optimalizácii tak vzniká odlišný výsledok.

Aj napriek týmto komplikáciám, sa spätný preklad rozvíja a nachádza uplatnenie vo viacerých oblastiach. Tie je možné podľa [11] rozdeliť na dve základné skupiny:

- bezpečnostné - výskum spojený s kryptografiou, skúmanie sily šifrovacích algoritmov, prelomenie ochrany proti kopírovaniu, analýza malware
- vývoj software - analýza nedokumentovaného produktu, analýza kvality produktu tretej strany, získanie informácii z konkurenčných produktov

Nasledujúca časť uvádza niekoľko príkladov z oboch uvedených skupín.

- **Analýza škodlivého software** – s rýchlym rozvojom technológii súvisí aj zvýšenie rýchlosti šírenia malwaru. Pred niekoľkými rokmi, keď sa vírusy prenášali najmä cez diskety, bola doba šírenia výrazne dlhšia v porovnaní s dnešným stavom. Taktiež bolo menej kanálov, ktorými mohlo dôjsť k nákaze. V súčasnosti, keď je väčšina počítačov pripojená k internetu, je riziko šírenia väčšie.

  Spätné inžinierstvo je používané oboma pomyselnými stranami – útočníkmi aj vývojármi antivírových spoločností. Útočníci jeho využitím hľadajú zraniteľnosti v operačných systémoch a iných aplikáciach. Na druhej strane vývojári antivírových spoločností analyzujú získaný škodlivý software pre zistenie jeho činnosti, minimalizáciu škôd a zabránenie jeho ďalšiemu šíreniu.

- **Kryptografia** – na základe Kerckhoffovho princípu [19] môžeme rozlíšiť dva spôsoby používané k utajeniu správy. Prvý sa spolieha výhradne na utajenie šifrovacieho algoritmu. V druhom prípade je algoritmus verejne známy a utajenie je založené na znalosti kľúča. Pokiaľ sa kryptografický algoritmus spolieha len na utajenie algoritmu, je len otázkou času, kedy po získaní prístupu k aplikácii vykonávajúcej šifrovanie alebo dešifrovanie, dôjde k prelomeniu algoritmu. Reverzné inžinierstvo je možné použiť k analýze algoritmu a návrhu postupu k jeho prelomeniu.

  Pri šifrách, kde je algoritmus verejne známy môže byť spätné inžinierstvo využité napríklad k analýze jeho konkrétnej implementácie. Verejne známy algoritmus síce môže byť bezpečný ale pri konkrétnej implementácii mohlo dôjsť k chybám, ktoré môžu byť zneužité útočníkom.

- **Správa digitálnych práv** – Od doby kedy sa počítače stali bežnou súčasťou väčšiny domácností, aj subjekty chránené autorským zákonom získali formu digitálnych informácií. Hudba, filmy a knihy už nie sú dostupné len na fyzickom analógovom médiu, ale aj v digitálnej forme. Na jednej strane to uľahčuje možnosti ich šírenia, na druhej to komplikuje kontrolu ich pohybu a neoprávneného distribuovania.

  Tvorcovia médií sa snažia vyvíjať technológie, ktoré by obmedzovali možnosti neoprávneného distribuovania digitálnych médií (angl. *DRM – digital right management*). Crackeri využívajú spätné inžinierstvo k pochopeniu týchto algoritmov a ich prelomeniu.

- **Programy s uzavretým zdrojovým kódom** – Programy, ktorých zdrojových kód nie je dostupný, môžu byť analyzované spätným inžinierstvom z viacerých dôvodov. Okrem hľadania zraniteľností to môže byť skúmanie nedokumentovaného API. Pomocou reverzného inžinierstva môže programátor získať presnejšie znalosti o API.

Tie môže následne využiť pri tvorbe programu, využívajúceho analyzovaný produkt. Ďalšou motiváciou môže byť skúmanie kvality softwaru s uzavretým zdrojovým kódom. Na rozdiel od open-source projektov, kde je možné kvalitu kódu posúdiť bez väčšej námahy, pri proprietárnom softwary využitie spätného inžinierstva môže byť potrebné.

- **Prekladače** – Spätné inžinierstvo môže byť použité pri testovaní prekladačov – overovaní, že generujú očakávaný výstup zodpovedajúci špecifikácii.

2.4 Nástroje

Nasledujúca časť predstavuje nástroje, ktoré sa využívajú v spätnom inžinierstve pri analyzovaní softwaru. Podľa [11] ich môžeme rozdeliť na dve základné kategórie:

- statická analýza – predstavuje transformáciu spustiteľného súboru do čitateľného formátu pre človeka. Samotný proces spočíva v načítavaní a analýze jednotlivých častí súboru.
- dynamická analýza – jedná sa tiež o transformáciu kódu ako pri statickej analýze. Na rozdiel od predchádzajúceho prístupu nie je kód len staticky čítaný, ale je spustený a následne je analyzované jeho správanie. Je možné skúmať interné dáta, obsah premenných a ich vplyv na tok programu. Preskúmaná je však len konkrétna časť programu na základe zvolených vstupov.

Systémové monitorovacie nástroje

Predstavujú kategóriu nástrojov, ktorá umožňuje skúmať vstupné a výstupné kanály medzi aplikáciami a operačným systémom. Využívajú sa na identifikáciu súborov, z ktorých aplikácia načítava vstup, do ktorých zapisuje, na analýzu použitia sieťového spojenia atď.

Debugger

Debugger je primárne určený vývojárom na identifikáciu chýb v kóde a ladenie. Umožňuje krokovanie programu a zisťovanie hodnôt premenných. Tieto informácie je možné využiť aj pri spätnom inžinierstve. Je však potrebné aby debugger podporoval možnosť krokovania kódu aj v prípade, že nie je dostupný zdrojový kód a ladiace informácie.

Disassembler

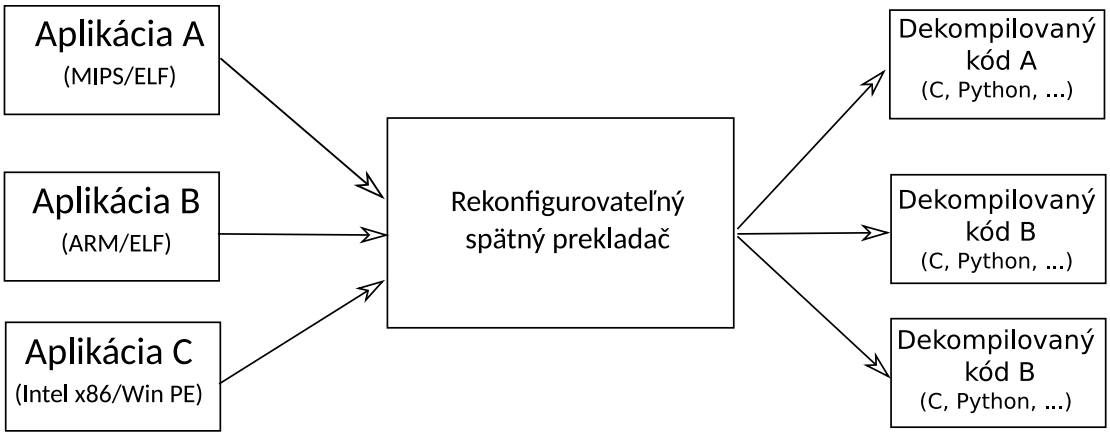
Disassembler prevádza binárny kód do čitateľnej formy – jazyk assembler. Úlohu spätného assembleru predstavuje analýza zvoleného binárneho súboru. Táto úloha zahŕňa potrebu rozlíšenia kódu od dát a následne samotný prevod binárnej reprezentácie do jazyka symbolických inštrukcií. Je dôležitá správna identifikácia oblasti kódu, aby niektoré inštrukcie neboli vynechané, alebo aby dáta neboli interpretované ako inštrukcie. Tento proces je platformovo závislý, nakoľko inštrukčná sada a používané registre sa vo všeobecnosti medzi rôznymi platformami líšia.

Spätný prekladač

Spätný prekladač (angl. *decompiler*) je nástroj, ktorý binárny kód transformuje do vyššej formy reprezentácie, podobnej zdrojovému kódu. Z dôvodov uvedených na začiatku 2.3 nie je možné získať úplne pôvodný zdrojový súbor. Na základe zložitosti problému spätného prekladu je táto úloha dekomponovaná do niekoľkých častí. Väčšina spätných prekladačov je určená pre jednu konkrétnu platformu. Existujú však aj riešenia podporujúce viacero architektúr. Túto problematiku približuje nasledujúca podkapitola.

2.5 Rekonfigurovateľný spätný prekladač

Cieľom rekonfigurovateľného spätného prekladača je schopnosť dekompilovať aplikácie rôznych platforiem a formátov do jednotnej výslednej reprezentácie. Takýto spätný prekladač sa na najnižšej úrovni musí vysporiadať s analýzou špecifickou k danej platforme pri prevode do vnútornej reprezentácie. Ostatné časti ale pri korektnom návrhu môžu byť znovupoužité pre viaceré platformy. Používateľ rekonfigurovateľného prekladača môže následne profitovať z možnosti analyzovať aplikácie rôznych platforiem v jednotnej forme. Všeobecná schéma rekonfigurovateľného spätného prekladu je na obrázku 2.2. Príklad konkrétneho riešenia je popísaný v kapitole 3.



Obrázok 2.2: Schéma rekonfigurovateľného spätného prekladu, prevzaté z [15] a upravené

Odhliadnuc od počtu podporovaných architektúr, spätný prekladač, rovnako ako disassembler, musí na začiatku dekódovať binárny kód. Problematika dekódovania je podrobnejšie popísaná v nasledujúcej podkapitole.

2.6 Princíp disassembleru

Základný algoritmus spätného assembleru je možné podľa [10] definovať v štyroch krokoch.

1. Na začiatku je potrebné určiť oblasť kódu, ktorá bude disassemblovaná. Pri rozlišovaní kódu od dát sa využívajú informácie dostupné v objektovom formáte spustiteľných súborov ako napríklad Portable Executable (PE) alebo Executable and Linking Format (ELF).

2. V ďalšom kroku je načítaná hodnota na zadanej počiatočnej adrese. Na základe binárnej hodnoty je identifikovaná inštrukcia zo zoznamu dostupných inštrukcií danej platformy. V závislosti od inštrukčnej sady sa môže jednať o triviálny proces alebo aj o komplikovanejšiu úlohu, napríklad pri inštrukciách s rôznou veľkosťou.

3. Po načítaní inštrukcie a jej operandov je vytvorený zápis v jazyku assembler a inštrukcia je poslaná na výstup.

4. Následne je určená nová adresa, kde bude načítaná ďalšia inštrukcia a zopakovaný proces od kroku 2. Tento postup sa opakuje, kým nie sú disassemblované všetky inštrukcie.

Existujú viaceré algoritmy, ktoré sa snažia túto úlohu riešiť. Nasledujúca časť predstavuje princíp ich činnosti, výhody a slabé miesta.

2.6.1 Lineárne dekódovanie

Algoritmus lineárneho dekódovania (angl. *linear sweep*) využíva jednoduchý predpoklad pre identifikáciu nasledujúcej inštrukcie, ktorá bude dekódovaná: Tam kde jedna inštrukcia končí, ďalšia začína. Algoritmus začína prvým bajtom sekcie označenej ako kód. Následne algoritmus postupuje celou sekciou až po jej koniec. Pseudokód algoritmu je zachytený v ukážke 2.3.

```
global startAddr, endAddr
proc DisassembleLinear(addr, instrList)
{
  while ( startAddr <= addr <= endAddr ) do {
     instr = DecodeInstr(addr)
     add instr to instrList
     addr += length(instr)
  }
}
```

Obrázok 2.3: Algoritmus lineárneho dekódovania, prevzaté z [22] a upravené

Výhodou algoritmu je jeho jednoduchá implementácia a zaručené pokrytie celej kódovej sekcie, nakoľko algoritmus neberie do úvahy tok riadenia programu.

Hlavnou nevýhodou algoritmu je to, že nie je schopný odhaliť dáta v kódovej sekcii. Tie sú potom nesprávne dekódované ako inštrukcie. Naviac táto situácia nie je vždy rozpoznateľná. Je možné ju detekovať pokiaľ bol dekódovaný kód neexistujúcej inštrukcie. V iných prípadoch môžu byť dekódované inštrukcie, ktoré nezodpovedajú vstupnému súboru. Túto situáciu demonštruje ukážka 2.4. Kód v ukážke je určený pre architektúru Intel x86. Jedná sa o časť funkcie `strchr`, do ktorej boli vložené tri nulové bajty z dôvodu zarovnania. Nulové bajty začínajú na adrese `0x809ef47`. Ako je vidieť z inštrukcie pred nimi, pri vykonávaní budú preskočené inštrukciou `jmp`. Lineárne dekódovanie však nevyužíva informácie z dekódovaných inštrukcií a preto sú tieto bajty chybne dekódované ako inštrukcie. Chybne dekódované inštrukcie sú v ukážke vyznačené oranžovou farbou.

```
Adresa        Obsah pamäti      Výsledky disassembleru
              ...
0x809ef45:    eb 3c             jmp 0x809ef83
0x809ef47:    00 00             add %al, (%eax)
0x809ef49:    00                add %al,
0x809ef4a:    83 ee 04 83 ee    0xee8304ee(%ebx)
0x809ef4f:    04 83             add $0x83, %al
              ...
0x809efaa:    73 9e             jae 0x809ef4a
              ...
```

Obrázok 2.4: Príklad zlyhanie algoritmu lineárneho dekódovania na architektúre Intel x86, prevzaté z [22]

2.6.2 Rekurzívny zostup

Rekurzívny zostup (angl. *recursive descent*) sa snaží odstrániť nedostatky lineárneho dekódovania. Využíva informácie získané analýzou toku riadenia programu. Nasledujúci text predstavuje niekoľko konkrétnych príkladov ako sú využité informácie o toku riadenia programu pre dekódovanie inštrukcií, a následne uvádza pseudokód 2.6 algoritmu rekurzívneho zostupu.

Pri sekvenčnom toku riadenia algoritmus postupuje rovnako ako lineárne dekódovanie. Jedná sa napríklad o presuny medzi pamäťou a registrami, aritmetické operácie a operácie so zásobníkom. Po inštrukciách podmienených skokov môže program pokračovať dvoma spôsobmi. Pri splnenej podmienke sa vyhodnotí cieľ skoku a program pokračuje od získanej adresy. Pokiaľ podmienka nebola splnená, program pokračuje nasledujúcou inštrukciou. Keďže vo všeobecnosti nie je možné určiť ako bude podmienka vyhodnotená, disassembler berie do úvahy oba možné stavy. Pokračuje nasledujúcou inštrukciou a poznačí si cieľ skoku do zoznamu adries, od ktorých bude disassembler pokračovať neskôr.

Pri inštrukciách nepodmienených skokov, algoritmus nepokračuje nasledujúcou inštrukciou, na rozdiel od lineárneho dekódovania. Algoritmus sa snaží zistiť cieľ skoku a následne pokračuje v dekódovaní od získanej adresy. Na základe tohoto postupu je, na rozdiel od lineárneho dekódovania, schopný správne dekódovať kód, ktorý je zobrazený v ukážke 2.5. Bajt dát vložený pseudoinštrukciou `_emit` je preskočený. Komplikácie nastávajú pokiaľ cieľ

```
    ...
    jmp After
    emit 0x0f
After:
    mov eax, [SomeVariable]
    push eax
    call AFunction
```

Obrázok 2.5: Dáta v kódovej sekcii, prevzaté z [11]

skoku závisí od hodnoty vypočítanej počas behu programu. Príkladom môže byť skok na základe hodnoty registra. Nakoľko pri statickej analýze nie je známy obsah registra, nie je možné určiť, od ktorého miesta má pokračovať proces dekódovania.

Pri volaní funkcií dekódovanie postupuje podobne ako pri podmienených skokoch – pokračuje adresou volanej funkcie. Rovnako sa môže vyskytnúť problém s identifikáciou adresy volanej funkcie, pokiaľ je adresa odovzdávaná hodnotou registra. Algoritmus ďalej predpokladá, že po ukončení volanej funkcie pokračuje inštrukciou nasledujúcou za volaním funkcie. Problém môže nastať, pokiaľ volaná funkcia modifikuje návratovú adresu a program po návrate bude pokračovať na inom mieste.

Po inštrukcii návratu z funkcie, kedy by sa pri behu pokračovalo návratovou hodnotou uloženou na zásobníku, algoritmus pokračuje adresou, ktorá bola uložená pre ďalšie dekódovanie. Môže ísť napríklad o adresu nasledujúcu za volaním funkcie alebo o ciele podmienených skokov.

```
proc Disassemble(Addr, instrList)
{
 if (Addr.decoded == true )
   return;
 do {
    instr = DecodeInstr(Addr)
    Addr.decoded = true
    add instr to instrList
    if(instr is branch){
       T = set of possible control flow succesors of instr;
       for each (target in T) {
         Disassemble(target, instrList)
       }
    }
    else{
       Addr += instr.length
    }
 } while(Addr is valid)
}
```

Obrázok 2.6: Algoritmus rekurzívneho zostupu, prevzaté z [22] a upravené. Podmienka `is branch` predstavuje miesto, kde je analyzovaná dekódovaná inštrukcia. V prípade, že inštrukcia ovplyvňuje tok riadenia programu, sú zistené adresy možných nasledujúcich inštrukcií a dekódovanie pokračuje od týchto adries.

Hlavnou výhodou je vyššia pravdepodobnosť, že dáta budú korektne odlíšené od kódu, nakoľko pri postupe dekódovania využíva získané informácie o toku programu.

Kľúčovým predpokladom úspešného dekódovania je schopnosť rozpoznať všetky vetvy programu. Pokiaľ sa nepodarí identifikovať vetvy programu určené nepriamymi skokmi, môže nastať situácia, že nebudú dekódované všetky inštrukcie. Niektoré z problémov, s ktorými sa algoritmus rekurzívneho zostupu nevie vysporiadať, sú zachytené v ukážke 2.7. Bez analýzy predchádzajúcich inštrukcií nie je schopný určiť cieľ skoku inštrukcie `jmp eax`. Taktiež môže chybne analyzovať cieľ skoku inštrukcie `je Junk`. Síce sa jedná o podmienený skok, ale z kontextu je zrejmé, že podmienka nikdy nebude splnená. Uvedené postupy z ukážky 2.7 môžu použiť vývojári proprietárneho sofwaru alebo malwaru, aby zabránili analýze ich produktu.

```
    mov eax, 2
    cmp eax, 3
    je Junk
    mov eax, After
    jmp eax
Junk:
    _emit 0xf
After:
    mov eax, [SomeVariable]
    push eax
    call AFunction
```

Obrázok 2.7: Možné problémy rekurzívneho zostupu, prevzaté z [11]

2.6.3 Hybridná analýza

V predchádzajúcich dvoch častiach boli ukázané výhody a nedostatky dvoch základných algoritmov pre dekódovanie inštrukcií – lineárneho dekódovania a rekurzívneho zostupu. Ich kombináciou je možné vytvoriť hybridný algoritmus, ktorý bude spájať silné stránky z oboch čiastkových algoritmov.

Jedno z možných riešení je uvedené v článku [22]. Predstavený algoritmus začína lineárnym dekódovaním. Následne porovnáva výstup s výsledkom rekurzívneho zostupu a kontroluje, či získaná sekvencia inštrukcií je konzistentná – napríklad neobsahuje skoky do stredu inštrukcie. Vlastnosti výsledného hybridného algoritmu závisia od spôsobu skombinovania spomenutých dvoch základných algoritmov.

2.7 Príklady existujúcich nástrojov

Nasledujúca časť predstavuje niekoľko existujúcich spätných prekladačov a nástrojov umožňujúcich disassemblovanie.

- **objdump** [8] – nástroj, ktorý je súčasťou sady programov GNU binutils. Umožňuje získavať informácie z objektových súborov. Napríklad: informácie z hlavičky súboru, tabuľky symbolov, ladiace informácie a iné. Taktiež umožňuje aj disassemblovanie súborov. Využíva metódu lineárneho dekódovania.

- **IDA Pro** [10] – komerčný spätný prekladač, ktorý je vyvíjaný spoločnosťou Hex-Rays. Tento spätný prekladač využíva algoritmus rekurzívneho zostupu. Základný algoritmus naviac dopĺňajú heuristiky, ktoré sa snažia identifikovať kód, ktorý nebol dekódovaný základným algoritmom. Takisto je podporovaný aj interaktívny spätný preklad. Užívateľ môže označiť nedekódovanú oblasť a vynútiť jej spracovanie. Tento spätný prekladač taktiež umožňuje zobraziť aj graf toku riadenia pre jednotlivé časti kódu.

- **Boomerang** [2] – rekonfigurovateľný prekladač, ktorý je vyvíjaný ako projekt s otvoreným zdrojovým kódom. Umožňuje dekompilovať súbory určené pre Intel x86, Sparc a PowerPC. Jeho výstupom je kód v jazyku C. Tento spätný prekladač používa algoritmus rekurzívneho zostupu. Medzi jeho nedostatky patrí chýbajúca schopnosť

rozpoznať staticky linkované funkcie, ktoré spomaľujú spätný preklad. Posledná aktualizácia informácii na stránke projektu je z konca roku 2012, pravdepodobne už v súčasnosti nie je ďalej vyvíjaný.

- **SmartDec** [9] – spätný prekladač do jazyka C a C++. Je možné ho použiť samostatne alebo ako zásuvný modul pre IDA Pro. Stránka projektu uvádza, že cieľom je vytvoriť rekonfigurovateľný spätný prekladač. V súčastnosti podporuje architektúry Intel x86 a x86-64. Je možné analyzovať súbory vo formáte ELF a PE, prípadne aj ďalších, pokiaľ je použitý spolu s IDA Pro. Medzi silné stránky patrí rekonštrukcia dátových typov [12].

- **Microsoft WinDbg** [4] – nástroj, ktorý umožňuje ladiť užívateľské aplikácie, ale aj procesy z jadra operačného systému a ovládače. WinDbg umožňuje zobrazovať obsah pamäte, premenných, hierarchiu volaných funkcií a iné. Pri ladení kódu v assembleri sú oblasti pamäte disassemblované algoritmom lineárneho dekódovania. Medzi jeho hlavné prednosti patrí prepojenosť s operačným systémom a možnosť krokovať aj skoré fázy pri vytvorení procesu [11].

- **OllyDbg** [23] – ladiaci nástroj pre Microsoft Windows, ktorý vytvoril Oleh Yuschuk. Je dostupný zadarmo po registrácii a jeho funkčnosť je možné rozšíriť zásuvnými modulmi. Aj napriek tomu, že sa jedná primárne o ladiaci nástroj, obsahuje aj prepracovaný disassembler, ktorý využíva algoritmus rekurzívneho zostupu. Analyzátor kódu je schopný identifikovať cykly, príkaz switch a iné kódové konštrukcie. Taktiež umožňuje úpravu spustiteľného súboru vložením vlastného kódu.

Kapitola 3

Spätný prekladač AVG Technologies

Spätný prekladač vyvíjaný firmou AVG Technologies je nezávislý od konkrétnej architektúry, operačného systému a formátu súboru. Nasledujúca časť predstavuje architektúru spätného prekladača. Potom sú popísané činnosti vykonávané v jednotlivých častiach. Podrobnejšie je popísaný dekodér inštrukcií. Záver tejto kapitoly sa zaoberá technológiami využívanými v spätnom prekladači, konkrétne jazykom ISAC a kompilačnou platformou LLVM. Informácie popísané v tejto kapitole vychádzajú z [16, 14, 13, 24, 15] a [17].

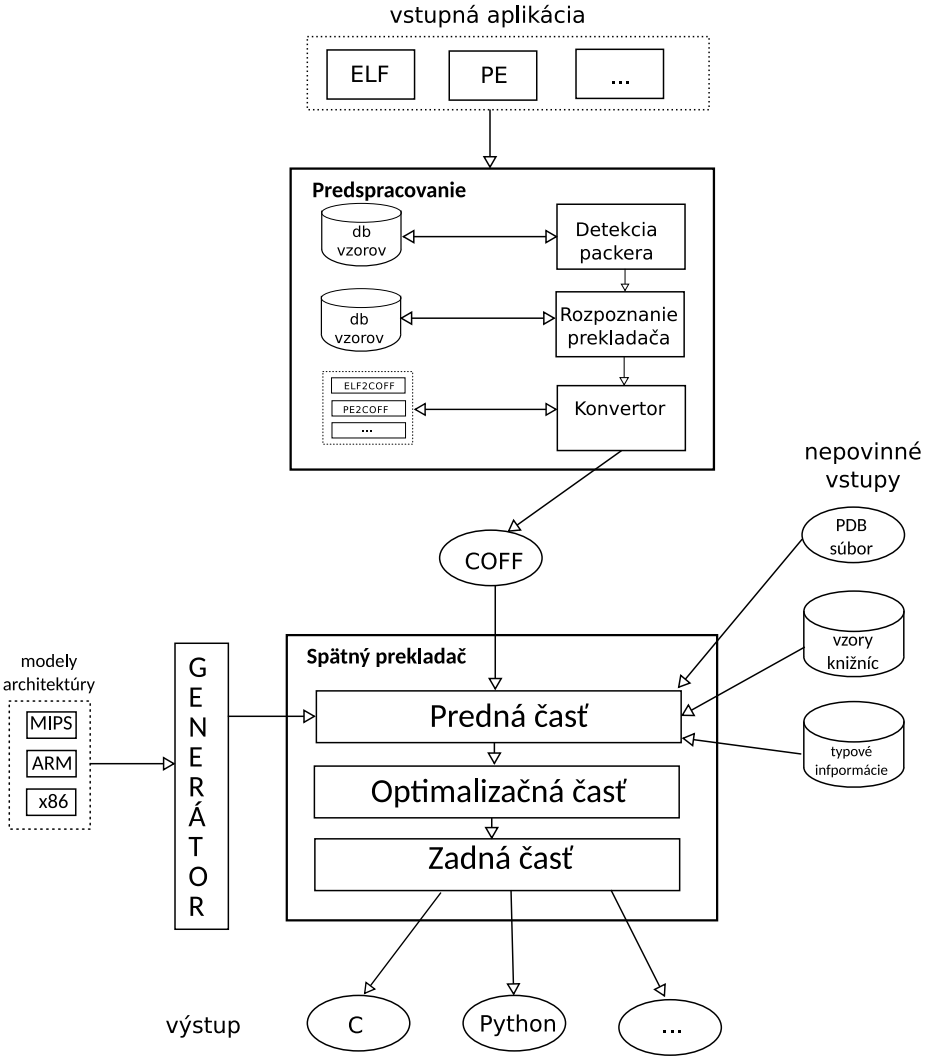
3.1 Architektúra spätného prekladača

Jednotlivé časti prekladača sú zobrazené na obrázku 3.1. Celý proces dekompilácie pozostáva zo spustenia viacerých nástrojov. Ich prepojenie zabezpečuje shell skript. Nakoľko je prekladač navrhnutý ako rekonfigurovateľný, popis konkrétnej platformy je načítaný po detekcii architektúry spustiteľného súboru. Popis platformy je v jazyku ISAC. Informácie o jazyku ISAC aj s ukážkou zápisu sú v časti 3.4. Samotné jadro spätného prekladača je členené na štyri základné časti – časť predspracovania, predná časť, optimalizačná časť a zadná časť. Pre vnútornú reprezentáciu analyzovaného programu je využívaný jazyk LLVM IR popísaný v časti 3.3.

3.1.1 Predspracovanie

Prvá časť spätného prekladu, predspracovanie, na začiatku zisťuje formát spustiteľného súboru a architektúru, pre ktorú je určený. Taktiež zisťuje použitý prekladač, jeho verziu a pôvodný implementačný jazyk analyzovanej aplikácie. Získané informácie sú uložené do súboru vo formáte JSON. Tieto informácie sú využité následne v ďalších fázach spätného prekladu, pre načítanie špecifikácie danej architektúry a inštrukčnej sady. Informácie o použitom prekladači a jeho verzii umožňujú transformovať rôzne idiomy – skupiny inštrukcií zaisťujúce efektívne vykonanie kódu ale ich vykonávaná funkčnosť nie je intuitívne zrejmá.

Analyzovaný spustiteľný súbor môže byť modifikovaný pomocou nástroja, ktorý môže brániť spätnému prekladu (angl. *packer*). Pokiaľ je zistené použitie takéhoto nástroja, časť predspracovania zaistí prevod do spracovateľného formátu. V závere je aplikácia prevedená do formátu Common-Object-File-Format (COFF), ktorý vstupuje do prednej časti prekladača.



Obrázok 3.1: Architektúra spätného prekladača, prebraté z [24] a upravené

3.1.2 Predná časť

Predná časť (angl. *front-end*) spracováva vstupný súbor vo formáte COFF. Jedná sa o jedinú platformovo závislú časť. Táto časť transformuje platformovo závislé inštrukcie do jednotnej vnútornej reprezentácie. Výstupom je textová reprezentácia LLVM IR inštrukcií.

Na začiatku sú zo vstupného súboru načítané informácie, ktoré môžu spresniť výsledky spätného prekladu. Jedná sa o ladiace informácie alebo informácie z tabuľky symbolov. Z týchto informácii je možné zistiť názvy premenných a funkcií a ich pozíciu v kóde. Tieto informácie ale nie sú povinnou súčasťou spustiteľného súboru. Najmä pre škodlivý software je typické, že tieto dodatočné informácie neobsahuje.

Následne na základe popisu platformy v jazyku ISAC je vytvorený dekodér inštrukcií. Ten je schopný rozpoznať inštrukcie v binárnom súbore a previesť ich na zodpovedajúcu reprezentáciu v jazyku LLVM IR. Podrobnejší popis dekodéru je uvedený v časti 3.2.

Spustiteľný súbor okrem vlastných inštrukcií programu môže obsahovať aj inštrukcie staticky linkovaných funkcií. Predná časť prekladača identifikuje oblasti, v ktorých sa tieto

funkcie nachádzajú. Tieto oblasti nie sú analyzované, aby spätný preklad prebehol rýchlejšie.

Po dekódovaní inštrukcií sú vykonávané statické analýzy nad vytvoreným LLVM IR kódom. Jednou z vykonaných analýz je aj analýza toku programu (angl. *control flow analysis*), ktorej hlavnou úlohou je rozpoznať základné bloky v programe. Okrem toho sú analyzované aj skokové tabuľky pre príkaz switch, globálne premenné, vyhľadávanie funkcii pôvodného programu a iné. Niektoré analýzy využívajú interpret statického kódu, ktorý je schopný vyhodnotiť operandy inštrukcií. Jedným z jeho využití je napríklad určenie cieľa skokovej inštrukcie.

Po vykonaní všetkých analýz nasleduje fáza generovania výstupu. Okrem výstupu vo formáte LLVM IR, je generovaný výstup aj vo formáte assembleru. Výstup v jazyku assembler je oproti výstupu bežného disassembleru obohatený o informácie z vyššej úrovne. Do kódu sú doplnené informácie o oblastiach kódu jednotlivých funkcií. K inštrukciám skokov a volaní podprogramu sú doplnené názvy volaných funkcií, a k použitým adresám z dátovej sekcie sú doplnené zodpovedajúce literály.

3.1.3 Optimalizačná časť

Optimalizačná časť (angl. *middle-end*) upravuje kód získaný z prednej časti prekladača. Kód vstupujúci do tejto časti obsahuje úplný popis správania každej inštrukcie z analyzovanej aplikácie. Tento komplexný popis môže byť v praxi veľmi rozsiahly a svojou nadmernou podrobnosťou môže spomaľovať ďalšie analýzy. Optimalizačná časť stavia na LLVM nástroji `opt` a pridáva ďalšie optimalizácie:

- optimalizácia cyklov
- šírenie konštanty
- odstraňovanie mŕtveho kódu
- zjednodušovanie toku riadenia
- odstraňovanie inštrukčných idiómov

Výstupom tejto časti je redukovaný a optimalizovaný kód pre záverečné spracovanie a generovanie výstupného kódu.

3.1.4 Zadaná časť

Zadná časť spätného prekladača (angl. *back-end*) generuje zo získaného optimalizovaného kódu výstup vo vyššom jazyku. V súčasnosti je podporovaný výstup v dvoch jazykoch: C a jazyk podobný Python-u.

Druhá varianta je modifikácia jazyka Python, kde sú konštrukcie, ktoré nemajú v jazyku podporu, ako ukazovatele a `goto`, nahradené konštruktami jazyka C. Jedná sa o dynamicky typovaný jazyk. Z tohto dôvodu neobsahuje toľko explicitných pretypovaní ako výsledný kód v jazyku C. Python naviac využíva odsadenie namiesto zložených zátvoriek a výsledný kód je prehľadnejší.

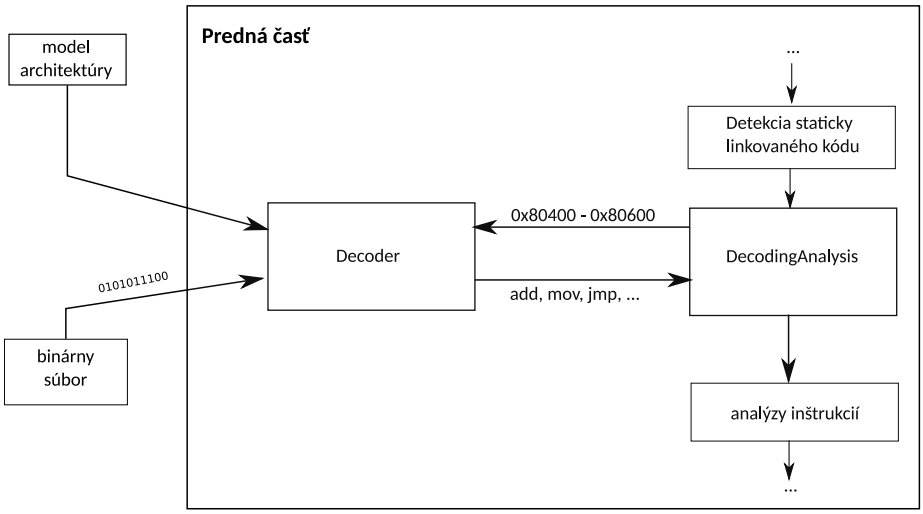
Pred samotným generovaním výstupu do jedného z dvoch cieľových jazykov je ešte vnútorný kód prevedený do ďalšej reprezentácie označovanej BIR (angl. *backend intermediate representation*). Tento prevod je vykonaný z toho dôvodu, že popis v jazyku LLVM IR je nízkoúrovňový. Pre rekonštrukciu prostriedkov riadenia programu vyššej úrovne, napríklad

cyklov a podmienených príkazov, je vhodnejšia reprezentácia BIR. Do kódu sú taktiež doplnené ladiace informácie, ak sú dostupné, aby bolo možné zrekonštruovať pôvodné názvy premenných a funkcií.

V závere je ešte poslednýkrát získaný BIR optimalizovaný a je generovaná reprezentácia v zvolenom jazyku. Ďalšími výstupmi tejto časti ešte sú: graf volaní, graf toku riadenia pre všetky funkcie a reprezentácia aplikácie v jazyku assembler.

3.2 Dekódovanie inštrukcií

Úlohu dekódovania inštrukcií v spätnom prekladači spoločnosti AVG zabezpečujú objekty triedy `Decoder` a `DecodingAnalysis`. `Decoder` zabezpečuje prístup k spracovávanému binárnemu súboru. Druhý menovaný predstavuje riadiaci algoritmus dekódovania. Určuje oblasti a poradie, v ktorom majú byť dekódované pomocou objektu triedy `Decoder`. Ich zaradenie v rámci spätného prekladača ilustruje obrázok 3.2.



Obrázok 3.2: Dekodér v spätnom prekladači

3.2.1 Decoder

Dekodér inštrukcií pri svojej inicializácií načíta súbor s textovou reprezentáciou požadovanej architektúry v jazyku ISAC. V tomto mieste je možné pridávať ďalšie podporované architektúry. Oddelenie popisu architektúry od samotného jadra spätného prekladača, umožňuje jeho rekonfigurovateľnosť – poskytovať spätný preklad z viacerých platforiem. Po pridaní modelu architektúry v jazyku ISAC je možné dekompilovať ďalšiu platformu. Z načítaného popisu je extrahovaná sémantika, ktorá umožní dekódovať binárny vstup na jednotlivé inštrukcie.

Samotné inštrukcie procesoru sú reprezentované vektorom operácii LLVM IR. Nakoľko existujú opakujúce sa podskupiny operácii, bol zavedený koncept subinštrukcií. Ten používa odkaz na opakujúce sa skupiny za účelom zníženie pamäťovej náročnosti ukladaných operácií.

Proces dekódovania oblastí spustiteľného súboru vykonáva metóda `decode`. Pseudokód tejto funkcie sa nachádza v ukážke 3.3. Zo vstupného súboru načítava dáta pre zadaný

rozsah adries. Veľkosť načítaných častí je určená možnými veľkosťami inštrukcií danej architektúry. Potom sa testuje, či načítaným dátam zodpovedá nejaká inštrukcia. Pokiaľ nie, pokračuje sa načítaním väčšieho kusu dát. V prípade, že ani pre jednu veľkosť inštrukcií nedôjde k nájdeniu zodpovedajúcej inštrukcie, je preskočený malý kúsok dát a dekódovanie inštrukcie pokračuje. Výstupom je vektor dekódovaných inštrukcií.

```
function decode(startAddress, endAddress){
    currentAddress = startAddress
    while(currentAddress < endAddress){
        for (length in allowed_lengths){
            data = read(currentAddress, length)
            inst = find(data)
            if inst{
                instructions.append(inst)
                current += inst.size
                break
            }
        }
        if not inst{
            currentAddress += skipSmallestSize
        }
    }
    return instructions
}
```

Obrázok 3.3: Dekódovanie inštrukcií

3.2.2 Decoding analysis

Objekt `DecodingAnalysis` riadi dekódovanie binárneho súboru. Na začiatku sú určené oblasti, ktoré budú dekódované. Oblasť môže byť zadaná užívateľom buď menom funkcie alebo rozsahom adries. Pokiaľ užívateľ nešpecifikuje oblasť, dekódujú sa celé kódové sekcie okrem staticky linkovaných funkcií. Pokiaľ sú dostupné ladiace informácie o lokalite funkcií, oblasť kódových sekcií je podľa nich rozdelená a dekóduje sa po menších častiach.

V súčasnej verzii spätného prekladača, je pre dekódovanie jednotlivých oblastí využitý algoritmus lineárneho dekódovania. Následne sú dekódované inštrukcie kontrolované. V prípade, že je zistený skok do stredu inštrukcie, je časť kódu znovu dekódovaná. Tento postup je opakovaný, pokiaľ nie je nájdená žiadna chyba alebo je prekročený maximálny počet opravných dekódovaní.

3.2.3 Vlastnosti aktuálneho riešenia

Táto časť sumarizuje vlastnosti aktuálne používanej implementácie dekódovania inštrukcií.

- Dochádza k viacnásobnému dekódovaniu niektorých oblastí, pri oprave chybne dekódovaných inštrukcií. Táto situácia môže nastať napríklad v prípade, že nie sú identifikované dáta v kódovej sekcii. To vedie k predĺženiu doby dekódovania.

- Nie sú využívané informácie o toku programu. To na jednej strane umožňuje rýchle dekódovanie, pokiaľ je kód dekódovaný na prvýkrát správne. Na druhej strane to môže viesť k chybám, ktoré dekódovací algoritmus nie je schopný identifikovať ani po následných analýzach.
- Nie je možné dekódovať súbory, ktoré neobsahujú informácie o sekciách.
- Okrem dekódovania inštrukcií spracovávaného súboru je časovo náročnou operáciou aj vytváranie objektu, ktorý sa používa na vyhľadávanie inštrukcií.

3.3 LLVM

Projekt LLVM začal ako výskumný projekt na Univerzite v Illinois. Jeho cieľom bolo poskytovať modernú kompilačnú platformu podporujúcu statický a dynamický preklad rôznych jazykov. V súčasnosti zastrešuje viacero modulárnych technológií využiteľných pre prekladače. V nasledujúcej časti je predstavených niekoľko z nich.

- LLVM knižnice tvoria optimalizačnú sadu, ktorá je zdrojovo a cieľovo nezávislá. Taktiež obsahujú podporu pre generovanie kódu pre rôzne platformy. Knižnice pracujú s vnútornou reprezentáciou LLVM IR (angl. *intermediate representation*).
- Clang – prekladač vystavaný nad LLVM pre jazyky C/C++ a Objective-C. Jeho cieľom je tvorba optimalizovaného a rýchleho kódu. Podporuje aj statickú analýzu kódu pre vyhľadávanie chybových vzorov v zdrojovom kóde.
- dragonegg spája optimalizačné nástroje LLVM so syntaktickou analýzou GCC. Toto spojenie umožňuje preklad jazykov ako napríklad Ada, Fortran a iných jazykov podporovaných prekladačom GCC.
- LLDB ladiaci nástroj, ktorý stavia na LLVM knižniciach a prekladači Clang. Dôraz je kladený na rýchlosť a efektívne využívanie pamäti. Podľa stránky projektu, je časovo aj pamäťovo efektívnejší ako ladiaci nástroj GDB.
- libc++ a libc++ ABI je optimalizovaná implementácia štandartnej knižnice jazyka C++. Obsahuje podporu aj novej normy jazyka C++11
- projekt polly implementuje optimalizácie na základe lokality dát a možnosť automatizovanej paralelizácie programu.
- projekt klee umožňuje skúmať možné priechody kódom funkcie za účelom hľadania chýb.
- lld je projekt, ktorého cieľom je vytvorenie linkovacieho nástroja pre prekladač clang.

3.3.1 LLVM IR

LLVM IR (angl. *intermediate representation*) je jazyk symbolických inštrukcií, používaný platformou LLVM na vnútornú reprezentáciu kódu. Je navrhnutý pre využitie v troch formách:

- v pamäti uložená pre spracovanie prekladačom

- uložiteľná na disku, vhodná pre Just-In-Time compiler
- čitateľná forma assembleru pre programátora

Predstavuje inštrukčnú sadu, ktorá je nezávislá na architektúre. Využíva SSA (angl. *static single assignment*) assembler, čo znamená, že hodnota je každej premennej priradená práve raz. Assembler síce predstavuje nízkoúrovňový jazyk, ale napriek tomu sa jedná o typovaný jazyk. Okrem základných dátových typov ako prázdny typ (*void*), celočíselný (*integer*) a typ pre desatiné čísla (*float*), taktiež podporuje aj funkcie (*function*), ukazovatele (*pointer*) a návestia (*labels*). Jazyk LLVM IR umožňuje použiť aj zložené dátové typy pre polia (*array*) a štruktúry (*structure*). Špeciálny typ pre ukladanie metadát (*metadata*) umožňuje pričlenenie dodatočných informácii k inštrukciám. Uvedené prostriedky umožňujú prehľadne reprezentovať aj jazyky vyššej úrovne. Medzi ďalšie kladné stránky patrí možnosť využiť optimalizačné nástroje navrhnuté pre túto vnútornú reprezentáciu.

Príklad zápisu reprezentácie programu v jazyku LLVM IR je v ukážke 3.4. Identifikátory majú dva základné typy – globálne a lokálne. Globálne používané pre funkcie a globálne premenné začínajú znakom `@`. Lokálne začínajú znakom `%`. Komentáre začínajú znakom `;` a metadata znakom `!`. Zodpovedajúci kód v jazyku C je zobrazený v ukážke 3.5

```
; Declare the string constant as a global constant.
@.str = private unnamed_addr constant [13 x i8] c"hello ↩
    world\0A\00"

; External declaration of the puts function
declare i32 @puts(i8* nocapture) nounwind

; Definition of main function
define i32 @main() {   ; i32()*
  ; Convert [13 x i8]* to i8  *...
  %cast210 = getelementptr [13 x i8]* @.str, i64 0, i64 0

  ; Call puts function to write out the string to stdout.
  call i32 @puts(i8* %cast210)
  ret i32 0
}

; Named metadata
!0 = !{i32 42, null, !"string"}
!foo = !{!0}
```

Obrázok 3.4: Program v jazyku LLVM

3.4 Jazyk ISAC

Jazyk ISAC bol vyvinutý v rámci projektu Lissom [7]. Patrí medzi takzvané zmiešané jazyky pre popis architektúry (angl. *mixed architecture description language*). To znamená, že jeden model je možné využiť na popis štruktúry a zároveň aj správania danej architektúry. Model sa skladá z dvoch základných častí.

```c
#include <stdio.h>
#include <stdlib.h>

int main(){
   puts("Hello world!");
   return 0;
}
```

Obrázok 3.5: Program v jazyku C

V časti `RESOURCES` sú popísané jednotlivé prostriedky, ktorými disponuje procesor – registre a pamäťová hierarchia. Sekcia `OPERATION` popisuje inštrukčnú sadu a činnosť jednotlivých inštrukcií. Textová forma inštrukcie je popísaná v časti `ASSEMBLER`, binárna v časti `CODING` a činnosť inštrukcie je popísaná pomocou jazyka ANSI C v časti `BEHAVIOR`. Príklad popisu v jazyku ISAC je v ukážke 3.6.

Popis architektúry pomocou jazyka ISAC sa používa v rekonfigurovateľnom spätnom prekladači spoločnosti AVG Technologies. Na základe popisu je vytvorený dekodér pre danú platformu. Dekodér je využívaný na prevod binárnych súborov danej architektúry do LLVM IR, ako bolo uvedené v 3.2. Prínosom tohto jazyka je, že v jednom modeli umožňuje popísať štruktúru aj správanie architektúry. Zjednodušuje tak náročnosť pridávania ďalších podporovaných architektúr do rekonfigurovateľného spätného prekladača.

```
RESOURCES {
  // HW zdroje
  PC REGISTER bit[32] pc;    // programovy citac
  REGISTER bit[32] gprs[32];// subor registrov
  RAM bit[32] memory {
    ENDIANESS = little,
    SIZE = 0x20000,
    FLAGS = {R, W, X}
  };
}

/* popis registrov */
OPERATION reg REPRESENTS gprs
{ ... }

OPERATION op_add {// popis instrukcie
  INSTANCE gprs ALIAS {rd, rs, rt};
  ASSEMBLER { "ADD" rd "," rs "," rt };
  CODING { 0b000000 rs rt rd 0b000001...};
  // cinnost instrukcie
  BEHAVIOR {gprs[rd] = gprs[rs]+gprs[rt];};
}
```

Obrázok 3.6: Ukážka popisu architetkúry v jazyku ISAC, prevzaté z [25] a upravené

Kapitola 4

Návrh rekonfigurovateľného dekodéru

Obsah tejto kapitoly je klasifikovaný ako utajený, viď licenčné podmienky.

Kapitola 5

Serializácia

Obsah tejto kapitoly je klasifikovaný ako utajený, viď licenčné podmienky.

Kapitola 6

Implementácia

Obsah tejto kapitoly je klasifikovaný ako utajený, viď licenčné podmienky.

Kapitola 7

Experimenty a testovanie

Táto kapitola sa venuje testovaniu funkčnosti riešenia. Cieľom testovania bolo overiť správnosť dekódovania inštrukcií pri spätnom preklade. Testovanie sa taktiež zameriava na čas potrebný na dekódovanie implementovaným algoritmom. Na testovanie funkčnosti bola počas vývoja používaná sada regresných testov vytvorených počas vývoja spätného prekladača. Určitá podmnožina z týchto testov je priložená aj na odovzdanom CD. Týmto testom sa venuje časť 7.1. Ďalšie časti tejto kapitoly sa zameriavajú na testovanie niekoľkých špecifických prípadov. Testovaná bola aj serializácia navrhnutá v kapitole 5. Časť 7.9 zachytáva analýzu vykonanú počas implementácie algoritmu za účelom identifikácie časovo náročnej časti pri dekódovaní.

7.1 Testy spätného prekladu

Korektnosť správneho dekódovania inštrukcií v rámci spätného prekladača bola testovaná sadou niekoľkých malých programov. Tie boli preložené pre všetky podporované architektúry a následne dekompilované. Obsahovali volania funkcií, cykly a podmienené príkazy. Túto sadu testov tvorí 10 programov. Na preklad boli využité prekladače Clang a gcc. Cieľom testov je overiť správnosť spätného prekladu. Získané súbory zo spätného prekladu sú porovnávané s referenčnými výstupmi. Test je považovaný za úspešný v prípade zhody s referenčným výstupom. Prehľad výsledkov pre jednotlivé architektúry uvádza tabuľka 7.1. Úspešný test je označený slovom OK v príslušnej bunke tabuľky.
Testy je možné spustiť príkazom: `./test.sh integration`

7.2 Detekcia funkcií

Jedným z očakávaných prínosov implementovaného algoritmu je presnejšia detekcia funkcií, nakoľko algoritmus využíva informácie o toku programu. Cieľom testov bolo detekovať viac funkcií v programoch ako umožňovalo súčasné riešenie. Testovacími vstupmi boli aplikácie pre architektúru Intel x86. Táto časť predstavuje niekoľko príkladov zlepšenia, kde nový algoritmus umožnil detekovať funkcie, ktoré predtým neboli objavené. Prehľad počtu detekovaných funkcií v jednotlivých programoch uvádza tabuľka 7.2.
Testy je možné spustiť príkazom: `./test.sh functions`

program	x86	ARM	Thumb	MIPS	PIC32	PowerPC
ackermann	OK	OK	OK	OK	OK	OK
array-global	OK	OK	OK	OK	OK	OK
bitcnt-1	OK	OK	OK	OK	OK	OK
dowhile	OK	OK	OK	OK	OK	OK
factorial	OK	OK	OK	OK	OK	OK
many-params	OK	OK	OK	OK	OK	OK
palindrome	OK	OK	OK	OK	OK	OK
stack	OK	OK	OK	OK	OK	OK
strlen	OK	OK	OK	OK	OK	OK
switch	OK	OK	OK	OK	OK	OK

Obrázok 7.1: Prehľad výsledkov testov pre jednotlivé architektúry

program	existujúce riešenie	implementovaný algoritmus
RLE	10	16
BinaryCards2	1	267
BigCombos2	1	186
SciGraph	21	330
87aa	1	8

Obrázok 7.2: Počet detekovaných funkcií

7.3 Duplicitné dekódovanie inštrukcií

Implementovaný algoritmus nedekóduje opakovane oblasti skúmaných súborov, nakoľko využíva informácie o toku programu. Dekodér v aktuálnej verzii v niektorých prípadoch opakovane dekódoval oblasti za účelom opravy chybného dekódovania. V niektorých prípadoch dochádzalo k duplicitnému dekódovaniu inštrukcií, aj keď si to priamo nevyžadovala oprava chýb. Táto situácia bola riešená použitím iterátoru `unique`, ktorý odstránil duplicitné inštrukcie. Správnosť dekódovania v tomto prípade testujú dva príklady na architektúru MIPS, pre ktoré bolo v súčasnej verzii spätného prekladača nutné použiť `unique`. Implementovaný algoritmus je schopný správne dekódovať uvedené súbory aj bez dodatočného odstraňovania viacnásobne dekódovaných inštrukcií.
Testy je možné spustiť príkazom: `./test.sh repeated`

7.4 Binárne súbory bez sekcií

Testy súborov, ktoré neobsahujú informácie o sekciách, sú predstavené v tejto časti. Implementovaný algoritmus je schopný sa vysporiadať s týmto problémom využitím informácií o segmentoch súboru. Táto funkčnosť je demonštrovaná na spustiteľných súboroch troch rôznych architektúr – Intel x86, MIPS, PowerPc. Výpis informácií o sekciách pre testované súbory zodpovedá výstupu 7.3. Cieľom testov bolo overiť schopnosť dekódovania súborov, ktoré neobsahujú informácie o sekciách. Počet detekovaných funkcií v jednotlivých súboroch sumarizuje tabuľka 7.3. Existujúce riešenie neumožňovalo dekódovať súbory, pokiaľ neobsahovali informácie o sekciách.

```
Sections:
Idx Name          Size      VMA       LMA       File off  Algn
```

Obrázok 7.3: Prázdne informácie o sekciách

program	implementovaný algoritmus	existujúce riešenie
intelx86-30782	3	-
mips-elf-efbb7	4	-
powepc-elf-c8e7	49	-
powerpc-elf-e9e8	38	-

Obrázok 7.4: Počet detekovaných funkcií v súboroch bez sekcií

Testy je možné spustiť príkazom: `./test.sh nosections`

7.5 Dekódovanie v dátových sekciách

Táto časť testuje dekódovanie v sekciách, ktoré sú označené ako dátové aj napriek tomu, že obsahujú kód. Tento problém vzniká pri spracovaní súborov, ktoré boli zabalené a nepodarilo sa zrekonštruovať charakteristiky jednotlivých sekcií. Testovaný súbor je určený pre architektúru Intel x86 a bol zabalený programom MPRESS2. Obsahuje sekcie ako uvádza ukážka 7.5. Označenie kódovej sekcie má len druhá sekcia v poradí, nakoľko obsahuje vstupný bod programu. Očakávaný výstup je zachytený v ukážke 7.6. Komentáre nad funkciami v ukážke udávajú na akých adresách sa nachádzajú jednotlivé funkcie. Prvá a tretia funkcia sa nachádzajú v sekcii `.data0` respektíve `.data4`, ktoré sú označené ako dátové v 7.5. Algoritmus je napriek tomu schopný ich dekódovať. Existujúce riešenie neumožňovalo dekódovanie takýchto súborov.
Test je možné spustiť príkazom: `./test.sh datasections`

```
Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .data0        00001000  00401000  00401000  00000400  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  1 .text         00001000  00402000  00402000  00001400  2**2
                  CONTENTS, ALLOC, LOAD, CODE, DATA
  2 .data4        00001000  00403000  00403000  00002400  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  3 .data6        00001000  00404000  00404000  00003400  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  4 .data7        00001000  00405000  00405000  00004400  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  5 .MPRESS2      00000320  00406000  00406000  00005400  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  6 .imports      00000200  00407000  00407000  00005800  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
```

Obrázok 7.5: Sekcie súboru multitext.exe

```c
#include <stdbool.h>
#include <stdint.h>
#include <stdlib.h>
#include <windows.h>

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

int32_t function_401000(void);
int32_t entry_point(void);
int32_t function_403000(void);

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

int32_t g1 = 0; // 0x40400f
bool g2 = false; // gpr0
/* ------------- Functions -------------- */

// Address range: 0x401000 - 0x401022
int32_t function_401000(void) {
    int32_t hConsoleOutput = GetStdHandle(-11); // 0x401002
    WriteConsoleA(hConsoleOutput, "Hello World!\r\n", 14, &↩
       g1, NULL);
    function_403000();
    return 42;
}

// Address range: 0x402000 - 0x402fff
int32_t entry_point(void) {
    // 0x402000
    function_401000();
    ExitProcess(0);
    unsigned char v1 = *(char *)42; // 0x40200d
    *(char *)42 = (char)((0x1000000 * (int32_t)v1 - 0↩
       x7a000000) / 0x1000000 + 84);
    return g2 ? 42 : 0;
}

// Address range: 0x403000 - 0x403005
int32_t function_403000(void) {
    // 0x403000
    g2 = true;
    return 42;
}
```

Obrázok 7.6: Výstup spätného prekladu pre multitext.exe

7.6 Selektívna dekompilácia

Rovnako ako existujúce riešenie používané v prekladači AVG Technologies, aj implementované riešenie umožňuje zúžiť oblasť, v ktorej budú inštrukcie dekompilované. Je možné zadať rozsah adries, ktorý má byť dekompilovaný, alebo názov funkcie.
Testy je možné spustiť príkazom: `./test.sh selDecomp`

7.7 Dekódovanie bez symbolov

Algoritmus využíva ladiace informácie a informácie z tabuľky symbolov pre spresnenie výsledku dekódovania. Tieto informácie však nemusí obsahovať každý spustiteľný súbor. Sada testov obsahuje aj súbory, z ktorých boli tieto informácie odstránené pomocou nástroja `strip`. Cieľom testov bolo overiť, že funkcie v súbore bez symbolov sú detekované na rovnakej adrese ako v súbore so symbolmi. Prehľad výsledkov uvádza tabuľka 7.7. Úspešný test, v ktorom boli nájdené funkcie na rovnakej adrese po odstránení symbolov, je označený slovom OK v stĺpci výsledok.
Testy je možné spustiť príkazom: `./test.sh strip`

program	architektúra	adresy začiatkov funkcií	výsledok
ack	ARM	0x8574, 0x8628, 0x86b8, 0x87b8	OK
ack	x86	0x8048520, 0x80485e0, 0x8048660, 0x8048660	OK
ack	MIPS	0x8900368, 0x890043c, 0x89004e0, 0x8900628	OK
ack	PowerPC	0x100004bc, 0x10000588, 0x10000630, 0x1000072c	OK
factorial	Thumb	0x8194, 0x81c0	OK
factorial	x86	0x804851c, 0x8048543	OK
factorial	MIPS	0x8900368, 0x89003dc	OK
factorial	PowerPC	0x100004bc, 0x10000520	OK

Obrázok 7.7: Výsledky dekódovania bez symblov

7.8 Časová náročnosť

Vplyv nového dekódovacieho algoritmu na celkový čas prekladu bol testovaný na väčšom množstve vstupov pre rôzne architektúry. Porovnanie časov behu navrhnutého algoritmu a existujúceho riešenia je v ukážke 7.8. Časy sú približne rovnaké. Veľkosť dekompilovaných programov sa pohybuje rádovo do `500kB`. Pri testovaní na väčších programoch sa prejavila náročnosť analyzovania jednotlivých inštrukcií a doba spätného prekladu bola približne dvakrát pomalšia oproti existujúcemu riešeniu.

	existujúce riešenie	implementovaný algoritmus
real	29m26.528s	29m28.351s
user	80m9.210s	80m0.621s
sys	3m49.822s	3m43.414s

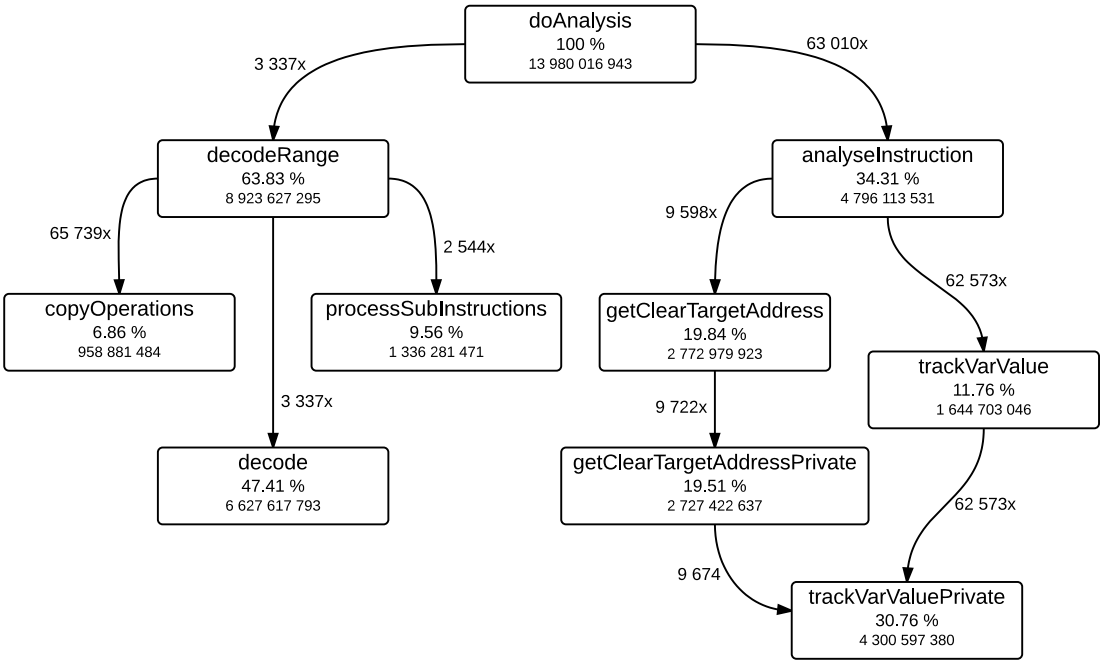
Obrázok 7.8: Čas behu

Parametre testovacieho prostredia

- `Fedora 21 (64-bit)`
- `Intel Core i5-3210M CPU @ 2.50GHz × 4`
- `8GB RAM`

7.9 Profilácia

Táto časť zachytáva profiláciu, ktorá bola vykonaná počas implementácie riešenia. Cieľom profilácie bolo zistiť, v ktorej časti v rámci dekódovania trávi algoritmus značné množstvo času. Následne boli navrhnuté úpravy, ktoré sa snažia vylepšiť implementáciu na základe získaných výsledkov profilácie. Pre profiláciu bol využitý program `callgrind` a pre následnú vizualizáciu výsledku program `KCachegrind`. Predmetom analýzy bola fáza dekódovania inštrukcií. Aby program strávil väčšinu času práve v tejto fáze, bola použitá upravená verzia spätného prekladača. Táto upravená verzia zastavila preklad po dekódovaní inštrukcií.

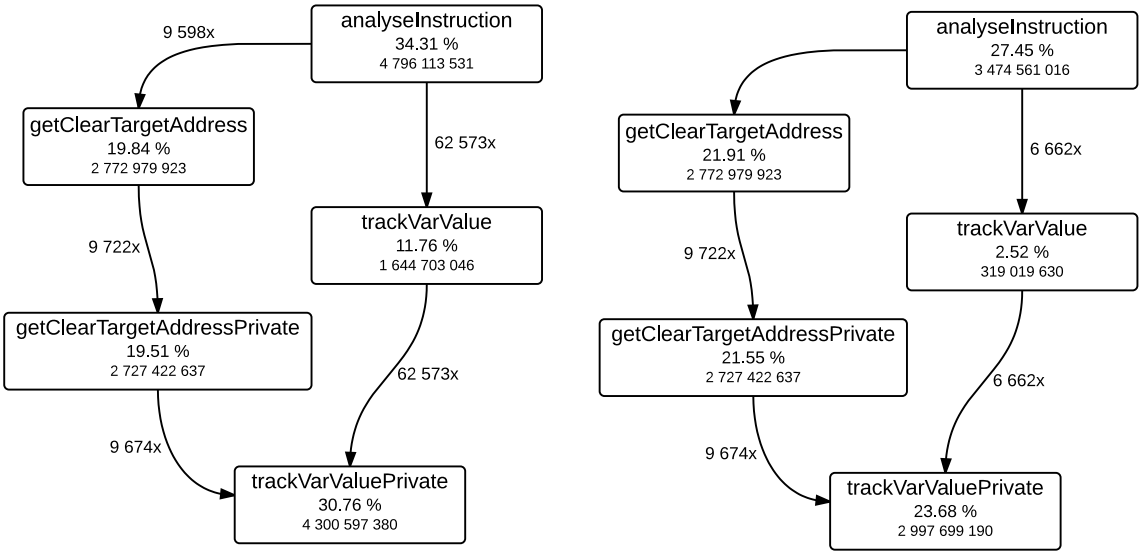


Obrázok 7.9: Vizualizácia výsledku profilácie

Výstup programu `KCachegrind` je v ukážke 7.9. Vo vrchnej časti grafu je zobrazená metóda `doAnalysis`, ktorá riadi proces dekódovania. Z tejto metódy sú volané `decodeRange` a `analyseInstruction`. V rámci metódy `decodeRange` najviac času zaberá `decode`, ktorá načítava dáta zo vstupného súboru. Táto metóda nebola cieľom optimalizácie, nakoľko jej implementácia nebola súčasťou práce. V druhej vetve `analyseInstruction` významnú

dobu behu programu zaberá volanie interpreta statického kódu – uzol grafu vpravo dole `TrackVarValuePrivate`. Následne boli analyzované jednotlivé miesta, v ktorých dochádza k volaniu interpreta. V niektorých prípadoch dochádzalo k opakovanému vyhodnoteniu niektorých výrazov interpretom. Kód bol upravený tak, aby sa po vypočítaní výsledok uložil a nevypočítaval sa zakaždým nanovo.

Vplyv tejto úpravy ilustruje ukážka 7.10. V ukážke je zachytený len detail volania metódy `TrackVarValuePrivate`. Vpravo je stav pred úpravou a vľavo po úprave. Na obrázkoch je uvedený počet inštrukcií procesoru. Počet volaní spomínanej funkcie sa navrhnutou úpravou podarilo znížiť pre testovanú aplikáciu približne na desatinu – pôvodne 62 573 a po úprave len 6 662.



Obrázok 7.10: Detail zníženia počtu volaní interpretu kód

7.10 Test serializácie

Sada testov demonštruje správnosť dekompilácie aj s využitím serializácie. Obsahuje testy pre architektúry, na ktorých sa `TDecoderLevelOne` vytvára pri každom spustení a aj tie, na ktorých sa využíva serializovaný obsah. Na testovanie boli použité rovnaké programy ako v časti 7.1. Prehľad správnosti výsledkov je v tabuľke 7.11. Porovnanie časov s využitím serializácie a bez nej, sa nachádza v ukážke 7.12. Algoritmus dáva korektné výsledky aj pri využití serializácie. Na skúmanej sade testov prinieslo využitie serializácie zrýchlenie približne 35 sekúnd, čo predstavuje zrýchlenie približne o 15% z pôvodného času. V testovacej sade bola serializácia využitá v 44 prípadoch. Ostatných 68 testovaných aplikácii bolo určených pre architektúry, na ktorých sa serializácia nevyužíva.
Testy je možné spustiť príkazom: `./test.sh serialization`

program	x86	ARM	Thumb	MIPS	PIC32	PowerPC
ackermann	OK	OK	OK	OK	OK	OK
array-global	OK	OK	OK	OK	OK	OK
bitcnt-1	OK	OK	OK	OK	OK	OK
dowhile	OK	OK	OK	OK	OK	OK
factorial	OK	OK	OK	OK	OK	OK
many-params	OK	OK	OK	OK	OK	OK
palindrome	OK	OK	OK	OK	OK	OK
stack	OK	OK	OK	OK	OK	OK
strlen	OK	OK	OK	OK	OK	OK
switch	OK	OK	OK	OK	OK	OK

Obrázok 7.11: Prehľad výsledkov testov pre jednotlivé architektúry

	bez serializácie	so serializáciou
real	5m11.446s	4m55.372s
user	3m33.480s	2m56.042s
sys	0m11.493s	0m13.779s

Obrázok 7.12: Čas behu testov, ktoré obsahovali spustiteľné súbory rôznych architektúr. Z toho 44 prípadov tvorili testy pre architektúru ARM, kde bol využitý serializovaný objekt `TDecoderLevelOne`. V 68 prípadoch sa jednalo o aplikácie pre architektúry, v ktorých sa nevyužíva serializácia.

Pri testovaní sa objavili problémy pri dekompilácii niektorých aplikácii, ktoré vo verzii bez serializácie fungujú a s využitím serializácie nie. Tieto problémy sú s najväčšou pravdepodobnosťou spôsobené zmenou dátového typu používaného na ukladanie inštrukcií. V rámci diplomovej práce sa nepodarilo navrhnúť riešenie tohto problému.

Kapitola 8

Záver

V tomto dokumente bolo predstavené spätné inžinierstvo a jeho využitie. Boli uvedené jeho možné uplatnenia v priemysle, ale najmä v oblasti sofwarového inžinierstva. Taktiež boli predstavené skupiny nástrojov, ktoré môžu byť využité pri tomto procese. Podrobnejšie bol vysvetlený princíp dekódovania inštrukcií. Boli uvedené dve základné metódy – lineárne dekódovanie a rekurzívny zostup. Bolo poukázané na ich silné stránky a nedostatky. Následne bol čitateľ oboznámený s aktuálnym stavom rekonfigurovateľného spätného prekladača vyvíjaného spoločnosťou AVG Technologies. Spätný prekladač je členený na štyri časti – predspracovanie, predná časť, optimalizačná časť, zadná časť. Práca sa podrobnejšie venuje prednej časti spätného prekladača a v rámci nej sa zameriava na problematiku dekódovania inštrukcií. Uvádza nedostatky aktuálneho riešenia. Hlavným nedostatkom je chybné dekódovanie spôsobené nerozoznaním dát v kóde a s tým súvisiaca nutnosť opakovaného dekódovania.

Hlavným prínosom práce je návrh algoritmu pre rekonfigurovateľný dekodér inštrukcií. Navrhnutý algoritmus vychádza z rekurzívneho zostupu. Využíva informácie o toku programu. Postup dekódovania je ovplyvnený sémantikou rozpoznaných inštrukcií – skokové inštrukcie, volanie funkcií a inštrukcie pre návrat z funkcie. Keďže navrhovaný algoritmus je rekonfigurovateľný, súčasťou návrhu je aj rozpoznanie sémantiky inštrukcií na základe ich jednotného popisu v jazyku LLVM IR. Okrem uvedených informácií o toku programu navrhnutý algoritmus využíva aj doplňujúce informácie pre spresnenie dekódovania. Medzi využívané informácie patria napríklad záznamy z tabuľky symbolov. Návrh zohľadňuje aj možné rozšírenie v budúcnosti pridaním ďalších spresňujúcich informácií pre dekódovanie. Pre zníženie časovej náročnosti procesu dekódovania bolo navrhnuté využitie serializácie dátovej štruktúry používanej pre vyhľadávanie inštrukcií.

Navrhnutý algoritmus bol implementovaný pre spätný prekladač AVG Technologies. Výsledná implementácia bola testovaná za účelom skúmania vlastností navrhnutého riešenia. Sadou testov bola overená základná funkčnosť dekódovacieho algoritmu, kde nový algoritmus dával zhodné výsledky s aktuálne používaným riešením. Následne boli testované vlastnosti, ktorými aktuálne riešenie nedisponovalo. Implementovaný algoritmus umožňuje dekódovať inštrukcie aj v sekciách, ktoré boli chybne označené ako dátové. Taktiež je možné dekódovať aj súbory, ktoré neobsahujú informácie o sekciách, využitím informácií o segmentoch. Prínosom je aj v niektorých prípadoch presnejšia detekcia funkcií oproti existujúcemu riešeniu. Doba behu navrhnutého algoritmu je približne zhodná s dobou behu aktuálneho riešenia pre menšie programy. Pri väčších programoch je síce doba samotného dekódovania približne dvojnásobná oproti existujúcemu riešeniu, ale implementovaný algoritmus umožňuje dosiahnuť presnejšie výsledky. Testy taktiež potvrdili, že navrhnuté

využitie serializácie umožňuje zrýchlenie procesu dekódovania pre niektoré architektúry.

Možnosťou pre ďalšie rozšírenie práce je optimalizácia časovej náročnosti algoritmu. Jedná sa najmä o zefektívnenie interpretu statického kódu, ktorý sa využíva pre výpočty hodnôt operandov analyzovaných inštrukcií, prípadne zníženie počtu prípadov, kedy sa používa. Námetom pre ďalšie skúmanie je taktiež voľba dátového typu pre uloženie inštrukcií, ktorý by umožňoval efektívne využitie serializácie a zároveň zachovanie funkčnosti pôvodného riešenia. Ďalším možným pokračovaním práce je pridávanie nových informácií pre spresnenie dekódovania. Príkladom môžu byť adresy uložené v kódovej sekcii programov určených pre architektúru ARM.

Literatura

[1] Apache Avro. [online] [cit. 2015-05-13].
URL http://avro.apache.org/docs/current/

[2] Boomerang Decompiler. [online] [cit. 2015-01-10].
URL http://boomerang.sourceforge.net/

[3] Cereal. [online] [cit. 2015-05-13].
URL http://uscilab.github.io/cereal/index.html

[4] Debugging Enviroments (Windows debuggers). [online] [cit. 2015-04-18].
URL https://msdn.microsoft.com/en-us/library/windows/hardware/hh406268(v=vs.85).aspx

[5] Developer Guide - Protocol Buffers. [online] [cit. 2015-05-13].
URL https://developers.google.com/protocol-buffers/docs/overview

[6] The DWARF Debugging Standard. [online] [cit. 2015-05-10].
URL http://www.dwarfstd.org/

[7] Lissom. [online] [cit. 2015-04-28].
URL http://www.fit.vutbr.cz/research/groups/lissom/

[8] Objdump - GNU Binary Utilities. [online] [cit. 2015-01-10].
URL https://sourceware.org/binutils/docs/binutils/objdump.html

[9] SmartDec. [online] [cit. 2015-04-28].
URL http://decompilation.info/

[10] Eagle, C.: *The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler*. William Pollock, 2011, ISBN 9781593272890.

[11] Eilam, E.; Chikofsky, E. J.: *Reversing: secrets of reverse engineering*. Wiley Publishing, Inc., 2005, ISBN 0-7645-7481-7.

[12] Fokin, A.; Derevenetc, E.; Chernov, A.; aj.: SmartDec: Approaching C++ Decompilation. In *Proceedings of the 2011 18th Working Conference on Reverse Engineering*, WCRE '11, Washington, DC, USA: IEEE Computer Society, 2011, ISBN 978-0-7695-4582-0, s. 347–356, doi:10.1109/WCRE.2011.49.
URL http://dx.doi.org/10.1109/WCRE.2011.49

[13] Křoustek, J.: AutoIt Approach: Infecting Your Browser Via Code Recycling. [online], March 2014, available on http://blogs.avg.com/news-threats/autoit-approach-infecting-browser-code-recycling/.

[14] Křoustek, J.; Matula, P.: Malware Analysis: Cryptocurrency-mining Malware Running on DVRs. [online], April 2014, available on http://blogs.avg.com/news-threats/cryptocurrency-mining-dvr-malware/.

[15] Křoustek, J.: *Retargetable Analysis of Machine Code*. disertační práce, Brno, FIT VUT v Brně, 2015.

[16] Křoustek, J.; Matula, P.; Kolář, D.; aj.: Advanced Preprocessing of Binary Executable Files and its Usage in Retargetable Decompilation. *International Journal on Advances in Software*, ročník 7, č. 1, 2014: s. 112–122, ISSN 1942-2628.
URL http://www.fit.vutbr.cz/research/view_pub.php.en?id=10531

[17] Lattner, C.: The LLVM Compiler Infrastructure Project. [online] [cit. 2015-01-05].
URL http://www.llvm.org

[18] Linn, C.; Debray, S.: Obfuscation of Executable Code to Improve Resistance to Static Disassembly. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, CCS '03, New York, NY, USA: ACM, 2003, ISBN 1-58113-738-9, s. 290–299, doi:10.1145/948109.948149.
URL http://doi.acm.org/10.1145/948109.948149

[19] Menezes, A.; van Oorschot, P.; Vanstone, S.: *Handbook of Applied Cryptography*. Discrete Mathematics and Its Applications, Taylor & Francis, 1996, ISBN 9781439821916.

[20] Ramey, R.: Serialization. [online] [cit. 2015-05-13].
URL
http://www.boost.org/doc/libs/1_58_0/libs/serialization/doc/index.html

[21] Roebuck, K.: *IT Security Threats: High-impact Strategies - What You Need to Know: Definitions, Adoptions, Impact, Benefits, Maturity, Vendors*. Emereo Publishing, 2012, ISBN 9781743048672.

[22] Schwarz, B.; Debray, S.; Andrews, G.: Disassembly of Executable Code Revisited. In *Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE'02)*, WCRE '02, Washington, DC, USA: IEEE Computer Society, 2002, str. 45.
URL http://dl.acm.org/citation.cfm?id=882506.885138

[23] Yuschuk, O.: OllyDbg. [online] [cit. 2015-04-18].
URL http://www.ollydbg.de/

[24] Ďurfina, L.; Křoustek, J.; Matula, P.; aj.: A Novel Approach to Online Retargetable Machine-Code Decompilation. *Journal of Network and Innovative Computing (JNIC)*, ročník 2, č. 1, 2014: s. 224–232, ISSN 2160-2174.
URL http://www.fit.vutbr.cz/research/view_pub.php.en?id=10676

[25] Ďurfina, L.; Křoustek, J.; Zemek, P.; aj.: Design of a Retargetable Decompiler for a Static Platform-Independent Malware Analysis. In *The 5th International Conference on Information Security and Assurance*, Communications in Computer and Information Science, Volume 200, Springer Verlag, 2011, ISBN 978-3-642-23140-7, s. 72–86.
URL http://www.fit.vutbr.cz/research/view_pub.php?id=9582

Příloha A

Obsah CD

`decfront/`	zdrojové súbory vytvorené v rámci diplomovej práce a súbory potrebné pre preklad
`decompiler/`	adresár s nainštalovaným spätným prekladačom
`diff/`	referenčné výstupy testov
`outputs/`	adresár pre výstupy testov
`doc/`	technická správa v plnej a cenzurovanej verzii
`tests/`	vstupy testov
`test.sh`	BASH script na spúšťanie testov
`Makefile`	súbor zabezpečujúci preklad zdrojových súborov
`README`	súbor popisuje obsah CD a tiež obsahuje informácie potrebné pre preklad zdrojových súborov