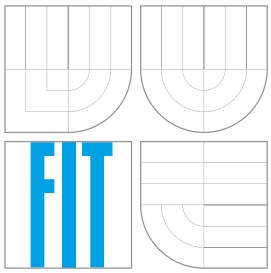
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

DETEKCE PODOBNOSTI ZDROJOVÝCH SOUBORŮ V JAZYCE C

C LANGUAGE SOURCE FILES SIMILARITY DETECTION

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

PETR REK

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. PETER MATULA

BRNO 2016

Zadání bakalářské práce/17996/2015/xrekpe00

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav informačních systémů Akademický rok 2015/2016

Zadání bakalářské práce

Řešitel: **Rek Petr**
Obor: Informační technologie
Téma: **Detekce podobnosti zdrojových souborů v jazyce C**
C Language Source Files Similarity Detection
Kategorie: Překladače

Pokyny:

1. Studujte problematiku zpětného inženýrství. Zaměřte se na zpětný překlad binárního kódu do vyšší formy reprezentace.
2. Seznamte se s překladačem Clang a jeho podporou pro vytváření nástrojů analyzujících kód v jazyce C.
3. Seznamte se se zpětným překladačem společnosti AVG a současnou implementací nástroje pro zjišťování podobnosti mezi dvěma soubory v jazyce C. Analyzujte jeho nedostatky.
4. Navrhněte novou verzi tohoto nástroje, která odstraní zjištěné nedostatky.
5. Po konzultaci s vedoucím implementujte nástroj navržený v předchozím bodě.
6. Vytvořené řešení důkladně otestujte sadou minimálně padesáti testů. Zhodnoťte svou práci a diskutujte budoucí vývoj.

Literatura:

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

Pro udělení zápočtu za první semestr je požadováno:

- První čtyři body zadání a rozpracování pátého bodu.

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
Datum zadání: 1. listopadu 2015
Datum odevzdání: 18. května 2016

L.S.

doc. Dr. Ing. Dušan Kolář
vedoucí ústavu

Abstrakt

Tato práce se zabývá návrhem, implementací a testováním nástroje *csim*, sloužícího pro porovnávání podobnosti dvou souborů v jazyce C. Primárním účelem vzniku tohoto nástroje je testování zpětného překladače vyvíjeného společností *AVG Technologies s.r.o.* Testování je prováděno na základě podobnosti abstraktního syntaktického stromu původního a dekompilovaného souboru. Čtenář je tedy seznámen se základy problematiky zpětného inženýrství, zejména zpětným překladem binárního kódu do vyšší úrovně reprezentace. Dále je popsán koloběh, kterým kód prochází od jeho vytvoření až po zpětný překlad, a jeho vliv na tento proces. Čtenáři je také poskytnut přehled o projektu LLVM a překladači Clang, který je základním stavebním kamenem nástroje *csim*.

Abstract

This thesis deals with design, implementation and testing of the *csim* tool, which compares two C source files by their similarity. The primary purpose of this tool is testing of a decompiler developed by *AVG Technologies s.r.o.* Testing is based on comparing abstract syntax trees of the original and decompiled source files. The reader is introduced to the basics of reverse engineering, especially reverse engineering of a binary file into a high-level programming language source file. The process of compiling followed by decompiling of a file is described along with its effect on reverse engineering. The LLVM project and the Clang compiler is introduced to the reader, since its libraries are the foundation upon which the *csim* tool is built.

Klíčová slova

Zpětné inženýrství, zpětný překladač, Clang, LLVM, podobnost zdrojových souborů, podobnost abstraktního syntaktického stromu, detekce malware, jazyk C

Keywords

Reverse engineering, decompiler, Clang, LLVM, source file similarity, abstract syntax tree similarity, malware detection, C language

Citace

REK, Petr. *Detekce podobnosti zdrojových souborů v jazyce C*. Brno, 2016. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Matula Peter.

Detekce podobnosti zdrojových souborů v jazyce C

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Petra Matuly.

.....
Petr Rek
15. května 2016

Poděkování

Tímto bych chtěl poděkovat vedoucímu práce Ing. Petru Matulovi za odbornou pomoc a cenné rady při jejím vypracování. Dále bych rád poděkoval vývojářům zpětného překladače *Retargetable Decompiler*, jmenovitě Ing. Jakubu Křoustkovi, za poskytnuté materiály a další rady. Speciální poděkování patří Ing. Petru Zemkovi za velmi užitečné revize zdrojových kódů a implementační tipy.

Obsah

Kapitola 1

Úvod

Zpětné inženýrství je proces extrakce znalostí z čehokoliv vytvořeného člověkem [24]. V této práci nás bude zajímat jeho využití z hlediska softwaru, kterých se nabízí velké množství. Je velmi užitečné v případě, kdy potřebujeme získat hlubší porozumění používaného nástroje nebo při snahách o zdokumentování tzv. *legacy systémů* (zastaralý software) či jejich převodu na nové platformy.

Se schopností analyzovat prakticky jakékoliv existující aplikace však vyvstává mnoho problémů z právního hlediska. Reverzního inženýrství můžeme využít k analýze produktů konkurence, a to jak v kontextu obchodním, tak vojenském. Časté je také využití hackery, kteří se snaží prolomit ochrany produktů, čímž následně dochází k porušování autorských práv, nebo detekovat slabiny systémů či aplikací za účelem jejich zneužití.

A právě naposledy zmíněný fakt je jedním z důvodů, proč jsou antivirové programy v dnešním světě naprostou nutností. Antivirové systémy také hojně využívají reverzního inženýrství, a to jak k analýze škodlivého kódů za účelem ochrany uživatelů, tak pro jeho detekci na základě vzorů. I proto vzniká zpětný překladač *Retargetable Decompiler* [17] vyvíjený společností *AVG Technologies s.r.o.*, ten provádí převod binárního kódu zpět do vyšší úrovně reprezentace (například do jazyka C). Pro jeho testování je nástroj *csim*, který je předmětem této práce, primárně určen.

Nástroj csim slouží k porovnání podobnosti dvou zdrojových souborů v jazyce C. Zaměřuje se zejména na analýzu původního a dekompilovaného kódu. Samotné porovnání je prováděno na úrovni *abstraktního syntaktického stromu* generovaného za pomoci knihoven překladače *Clang* [4]. Kromě testovacích účelů jej lze využít i pro obecné porovnávání dvou souborů a tudíž i detekci malwaru (škodlivého softwaru). Pro toto použití byl nástroj speciálně rozšířen.

Možných využití existuje více. To hlavní se však nachází v použití nástroje během nočních testů zpětného překladače, v těch je nástroj denně používán a odhaluje pokroky či regrese zavedené posledními změnami.

Práce je strukturována následovně. V kapitole 2 je čtenář seznámen se základy zpětného inženýrství, zejména nástroji, kterými zdrojový kód prochází od překladu po zpětný překlad, a jejich vlivem na tento proces. Dále je mu v kapitole 3 přiblížen projekt *LLVM* a pod něj spadající překladač Clang, jehož knihovny jsou základním stavebním kamenem řešení. V kapitole 4 jsou nastíněny problémy, které skýtá porovnání dekompilovaných souborů. V kapitole 5 je rozebráno aktuální řešení a identifikovány jeho klady a zápory. Na základě všech získaných znalostí je poté v kapitole 6 navržen nástroj csim a podpůrná knihovna *csiml*, jejichž implementace je popsána v kapitole 7. Vzhledem k náročnosti testování je tento proces popsán samostatně v kapitole 8. V kapitole 9 je zhodnocena kvalita výsledného řešení a diskutován jeho další vývoj. V závěrečné kapitole 10 je posouzena kvalita této práce.

Kapitola 2

Zpětné inženýrství

Formální definici pojmu reverzní, neboli zpětné, inženýrství v souvislosti se softwarem, která byla poprvé zveřejněna E. J. Chikofskym v časopise *IEEE Software* v roce 1990 [21], lze volně přeložit do následující podoby:

> „Reverzní inženýrství je proces analýzy zkoumaného systému za účelem identifikace jeho komponent a jejich vzájemných vztahů, a vytvoření reprezentace systému v jiné podobě nebo na vyšší úrovni abstrakce."

Tento proces se typicky odehrává s žádnou nebo velmi omezenou znalostí zkoumaného systému.

Reverzní inženýrství zahrnuje širokou škálu technik. Jedním z hlavních způsobů klasifikace je podoba zkoumaného systému, ze které vycházíme. Může se jednat buď přímo o zdrojový kód nebo o spustitelné soubory. Právě zpětný překlad binárních souborů bude centrem zájmu v rámci této práce. Toto odvětví nazýváme *binární reverzní inženýrství* [24].

Pro snazší pochopení problémů, kterým zpětný překlad čelí, budou nyní popsány jednotlivé typy nástrojů, kterými zdrojový kód prochází od jeho vytvoření až po jeho opětovné získání a jejich vliv na tento proces.

2.1 Překladač

Překladač je program, jehož vstupem je kód napsaný ve zdrojovém jazyce a výstupem kód v cílovém jazyce [32]. Zdrojovým jazykem je typicky vysokoúrovňový programovací jazyk, jako například jazyk C. Výstupem pak strojový kód nebo tzv. *bajtkód*, který není závislý na platformě. Zatímco druhé možnosti využívá například jazyk Java [25], nás bude v této práci zajímat první možnost, používaná například jazyky C a C++.

Překlad se obecně sestává z několika fází, u kterých nás, z hlediska zpětného překladu, bude zajímat, které informace jsou v jejich důsledků ztraceny.

2.1.1 Fáze překladu

Preprocesor Před tím, než vůbec můžeme vstupní zdrojový kód přeložit, může být zapotřebí využít programu zvaného *preprocesor*, který například rozgeneruje použitá makra (a tím ztrácíme informaci o jejich použití), odstraní komentáře nebo spojí více zdrojových souborů do jednoho [20].

Lexikální analýza *Lexikální analýza*, první fáze překladu, postupně načítá tzv. *lexémy* zdrojového jazyka, které reprezentuje *tokeny* obsahujícími atributy, jako jsou názvy proměnných, hodnoty číselných konstant a další.

Syntaktická analýza Z tokenů je *syntaktickým analyzátorem* vytvářen derivační strom. Jeho vytváření je řízeno syntaktickými pravidly daného jazyka a zároveň tedy dochází k ověření syntaktické správnosti programu (zápis v jazyce je syntakticky správný, pokud je pro něj možno sestavit derivační strom). V této fázi se ztrácí informace neužitečné pro překlad.

Sémantická analýza Další fází je *sémantická analýza*, která z derivačního stromu vytváří stručnější *abstraktní syntaktický strom*. V průběhu tohoto procesu je ověřována sémantická správnost programu – použité proměnné byly deklarovány, proměnné jsou *ekvivalentní vůči přiřazení* (mají totožný typ nebo je za tímto účelem lze implicitně přetypovat) a další. Sémantická analýza může být spojena se syntaktickou, mluvíme pak o *syntaxí řízeném překladu* [20], v tomto případě je tvorba derivačního stromu simulována a vytvářen je pouze abstraktní syntaktický strom.

Generátor vnitřního kódu Strom z předchozí fáze je vstupem části překladače zvané *generátor vnitřního kódu*, kde dochází k jeho přetváření do interní reprezentace. Ta je překladači vlastní a dokáže s ní lépe pracovat, především ji lépe optimalizovat. Často je využíván *tříadresný kód* (skládající se z kódu instrukce a dvou operandů).

Právě v této fázi dochází ke ztrátě významného množství informací a to v závislosti na úrovni vnitřního kódu. Tyto úrovně existují tři hlavní – nízká, střední a vysoká [28]. Vysokoúrovňové reprezentace jsou bližší zdrojovému kódu a uchovávají více informací, zatímco nízkoúrovňové mají blíže cílovému kódu a spousta informací je ztracena (typy cyklů, přesná struktura podmíněných výrazů, přístupy do pole atd.).

Optimalizátor Vnitřní kód je dále předán *optimalizátoru*, který provádí platformně nezávislé optimalizace. Ty mohou sledovat více hledisek, například velikost výsledného kódu, typicky jsou však prováděny s ohledem na čas vykonání cílového programu. Optimalizací existuje velké množství a to, které budou spuštěny, může uživatel při překladu ovlivnit přepínači (viz sekce 2.1.5). V závislosti na zvolené úrovni optimalizace může docházet k podstatným ztrátám informací, které jsou pro proces zpětného překladu zásadní.

Generátor cílového kódu Generátor cílového kódu spadá pod tzv. *back-end* překladače (předchozím částem se říká *front-end*). V této části je optimalizovaný vnitřní kód přetvářen na cílový kód – tím je typicky jazyk symbolických adres nebo binární kód. Mimo to může také docházet k provádění dalších, platformně nebo jazykově specifických, optimalizací.

Jak je patrné, největší problém z pohledu zpětného inženýrství představují dvě části překladače – generátor vnitřního kódu a optimalizátor.

2.1.2 Symbolická jména

Symbolickými jmény máme na mysli názvy funkcí a globálních proměnných. Ty jsou většinou při překladu zachovány, a jak později uvidíme, porovnávání souborů se zachovanými symbolickými jmény je značně usnadněno.

V některých případech však může dojít k jejich ztrátě, především snahou tvůrce kódu a to právě za účelem znesnadnění procesu zpětného překladu. K tomuto účelu můžeme využít speciálních programů (které zvládají odstranit i ladicí informace, jako třeba *strip*), ale tuto činnost zvládá i překladač (přepínač `-s` u GCC či Clang).

2.1.3 Datové typy

Při převodu optimalizovaného vnitřního kódu do cílového kódu dochází ke ztrátě informací o datových typech, neboť jak na úrovni binárního kódu, tak v jazyce symbolických adres, existují pouze dva základní – s pevnou řádovou čárkou a plovoucí řádovou čárkou. Ty lze dále dělit podle jejich *šířky* (počet bitů), i tak však nutně nemusí odpovídat typu, který byl programátorem původně zvolen.

Zpětná inference datových typů je problematická a patří mezi jednu z největších výzev zpětného překladu. S jistotou můžeme rozeznat pouze typy těch proměnných, které byly použity v rozeznaných knihovních funkcích vyžadujících daný typ parametru [23]. Případně můžeme získat podrobnější informace o typu na základě použitých instrukcí (například porovnání znaménkových a bezznaménkových typů se na úrovni instrukcí liší).

Objektivní porovnání podobnosti datových typů ve zdrojových souborech je tedy problematické, neboť odlišnosti nemusí být nutně problémem zpětného překladače.

2.1.4 Ladicí informace

Při překladu můžeme (většinou přepínačem `-g`) aktivovat vkládání ladicích informací, určených pro programy zvané *debugger* (ladicí nástroj, viz sekce 2.4). Ty sice zvyšují velikost výsledného binárního souboru, ale slouží k jednoduššímu ladění kódu či hledání chyb. Ze své podstaty musí obsahovat přesné informace o struktuře kódu (aby například mohly uživateli poskytnout informaci o tom, na jakém řádku či v jaké funkci k chybě došlo). Uchovávanou informací je to, které konstrukce zdrojového jazyka odpovídají části kódu v jazyce cílovém.

Zpětný překlad binárního souboru obsahujícího tyto informace je zjednodušen a dosažené výsledky podobnosti se značně liší v závislosti na jejich dostupnosti.

2.1.5 Optimalizace

Prováděných optimalizací existuje větší množství a uživatel většinou může ovlivnit, které z nich budou spouštěny vybráním úrovně optimalizace. Výchozím úkolem překladače je redukovat časovou náročnost překladu a dohlédnout na to, že kód bude snadno laditelný (optimalizace ovlivňují strukturu kódu a ladění by pak nemuselo poskytovat korektní informace).

V překladači GCC (a také Clang) je tato defaultní úroveň označena přepínačem `-O0` [9]. Přepínačem `-O` či `-O1` instruujeme překladač, aby se pokusil redukovat velikost kódu a čas vykonání. I přesto, že časově náročné optimalizace nejsou využity, tak čas překladu mírně stoupá. Existuje také přepínač `-O2` a `-O3`. Větší číslo znamená efektivnější kód, z hlediska časové i paměťové náročnosti, ale delší čas překladu. Zároveň také dochází k čím dál rozsáhlejším úpravám zdrojového kódu v souvislosti s prováděnými optimalizacemi a podobnost vůči původnímu kódu klesá.

Dále je popsáno několik základních optimalizací, z nichž některé představují velký problém pro analýzu podobnosti i zpětný překladač samotný.

Eliminace mrtvého kódu Jedná se o jednu z nejčastějších optimalizací, která je z hlediska velikosti výsledného programu velmi podstatná. *Mrtvým kódem* se myslí takový kód, který není nikdy vykonán (funkce není volána, podmínka je vždy nepravdivá atd.) a jeho přítomnost v cílovém kódu je tedy zbytečná.

Z hlediska zpětného překladu, oproti jiným optimalizacím, představuje spíše výhodu, neboť snižuje množství kódu, který je třeba analyzovat, aniž by se to jakkoliv podepsalo na funkčnosti. V případě nástroje pro porovnání dvou zdrojových souborů však činí problém, neboť důsledkem je snížení podobnosti o chybějící kód, za což zpětný překladač nemůže.

Inlining *Inlining* je proces dosazení těla funkce na místo jejího volání. To se provádí, pokud je tělo funkce primitivní (například jednořádkové) a režie spojená s voláním funkce by byla neúměrně vysoká, i na vyžádání programátorem (klíčovým slovem `inline` v jazyce C – rozhodnutí o provedení je však stále na překladači). Názorně je efekt této optimalizace znázorněn v ukázkách níže, zde si můžeme všimnout, že tělo funkce `function_B()` bylo vloženo na místo jejího volání ve funkci `function_A()` (ukázka 2.1), výsledkem je funkce `function_A_B()` (ukázka 2.2).

Inlining představuje z hlediska analýzy podobnosti velký problém, neboť mění podobu grafu volání, který je pro porovnání, především správné promítnutí funkcí jednoho souboru na funkce v druhém, zásadní.

```
int function_A(int *arr, int n) {
        for (int i = 0; i < n; i++) {
                arr[i] = function_B(arr[i], 5);
        }
        printf("Values␣to␣the␣fifth␣power.");
}

static inline int function_B(int a, int b) {
        int retval = a;
        for (int i = 0; i < (b-1); i++) {
                retval *= a;
        }
        return retval;
}
```

Ukázka 2.1: Stav před provedením inliningu

```
int function_A_B(int *arr, int n) {
        for (int i = 0; i < n; i++) {
                int a = arr[i];
                for (int b = 0; b < 4; b++) {
                        arr[i] *= a;
                }
        }
        printf("Values␣to␣the␣fifth␣power.");
}
```

Ukázka 2.2: Stav po provedení inliningu

Zjednodušování matematických výrazů Matematické operace jsou v programech velmi časté a proto také existuje velké množství jejich optimalizací. Mezi ty základní patří dosazování do výrazu pro snížení počtu proměnných, výpočet statických matematických výrazů nebo jejich úpravy na ekvivalentní, ale efektivnější (typickým příkladem je převedení dělení či násobení na bitové posuvy).

Zatímco zjednodušování představuje výzvu pro tvůrce zpětného překladače (je vhodnější převést zjednodušený výraz do původní podoby snáze pochopitelné programátorem), nástroj pro analýzu podobnosti to neovlivní, neboť porovnávání dvou souborů je typicky prováděno na vyšší úrovni, než jsou jednotlivé výrazy.

Reorganizace kódu ve smyčkách V případě, že je smyčka vykonána pouze několikrát, je možné její tělo do výsledné podoby vložit několikrát za sebe. Tím sice zvýšíme velikost programu, avšak ušetříme čas spojený s podmíněnými skoky. V rámci smyček je také časté přeuspořádání instrukcí a vyřazení těch, které jsou časově nezávislé, před tělo smyčky (takzvaný *hoisting*).

Eliminace jednotlivých smyček problémem pro analýzu podobnosti je, neboť ty lze jednoznačně k porovnávání využít.

2.2 Assembler a disassembler

Assembler slouží k převodu kódu v platformně specifickém jazyce symbolických instrukcí (taktéž zvaném jazyk symbolických adres nebo nepřesně assembler) do binární podoby, kterou procesor zvládne vykonat. Tato činnost není příliš složitá, ale může se komplikovat prováděnými optimalizacemi.

Úkolem *disassembleru* je pak převod opačným směrem. Tento proces již tak jednoduchý není. Problematické je například rozeznání dat od instrukcí ve *Von Neumannově architektuře* (obsahuje společnou paměť pro data i instrukce), tento problém se poté násobí v případě, že jsou využívány instrukce s proměnnou délkou. Jen jeden jediný špatně rozeznaný bajt nebo špatně rozeznaná instrukce mohou způsobit velké problémy ve zbytku analyzovaného binárního kódu – v případě, že místo instrukce o délce dva bajty rozeznáme instrukci o délce tři bajty, všechny další instrukce jsou taktéž posunuty a tudíž chybně rozeznány.

Kód v jazyce symbolických instrukcí může posloužit k pochopení činnosti aplikace, avšak vyžaduje více úsilí, znalostí a je také časově náročnější než zkoumání kódu ve vysokoúrovňové reprezentaci. Také úprava takto získaného kódu může být problematická, například kvůli převodům mezi absolutními a relativními skoky, které assembler provádí. Proto vznikají složitější nástroje zvané *zpětný překladač*.

2.3 Zpětný překladač

Zpětný překladač se značně liší od disassembleru. Jeho výstup je kratší a lépe srozumitelný. Zatímco zpětný překlad bajtkódu je již do jisté míry zvládnutý [26], neboť je zachováno mnohem větší množství informací o typech či struktuře kódu, proces dekompilace strojového kódu je stále velmi složitým problémem vzhledem k velkému množství chybějících informací. Právě zpětný překlad binárního programu nás bude dále zajímat.

> „Dekompilátor, neboli zpětný překladač, je program snažící se provést opak překladu. Cílem je vytvoření funkcionálně ekvivalentního programu ve vysokoúrovňovém jazyce z vstupního binárního programu. Vstup je platformně specifický, zatímco výstup jazykově specifický.“ [22]

Důvodů, proč zpětný překladač využívat, existuje více. Může být použit například při snaze o získání ztracených zdrojových kódů nebo pro převod starších programů na nové platformy. Zpětný překladač společnosti AVG se zaměřuje zejména na jeho využití pro detekci škodlivého softwaru.

2.3.1 Obecná struktura zpětného překladače

Zpětný překladač je možno rozdělit na tři hlavní části – strojově specifický *front-end* [30], nezávislý *middle-end* a jazykově specifický *back-end* [22]. Jazykově i strojově nezávislý middle-end je podstatný zejména proto, že umožňuje převádět do vnitřní reprezentace libovolný podporovaný kód a produkovat z něj kód v různých vysokoúrovňových jazycích.

Front-end Cílem tohoto modulu je načíst strojově specifický kód do vnitřní reprezentace. Mohli bychom jej dále rozdělit na *loader* (někdy také řazen do části zvané *preprocessing* [30]), *parser* a *sémantickou analýzu*.

Loader zjistí základní informace o spustitelném souboru, jako například architekturu, dostupnost ladicích informací, použití packeru či vstupní bod programu, a strojový kód načte do paměti. Parser strojový kód převádí do interní reprezentace, kterou strukturuje. Tím vznikají tzv. *základní bloky* (anglicky *basic block* [24]), reprezentující tok programu (cykly, funkce atd.). Jeden základní blok odpovídá několika málo příkazům ve vysokoúrovňovém jazyce. Sémantická analýza tuto reprezentaci dále upravuje, podstatná je zde zejména inference datových typů a rozeznání známých *idiomů* (způsobů, kterými jsou přeloženy určité vysokoúrovňové příkazy), které je možno převést do srozumitelnější podoby.

Middle-end Vnitřní reprezentace má typicky dvě úrovně, nižší, která je podobná jazyku symbolických instrukcí a vyšší, která se podobá vysokoúrovňovému jazyku. Tato část provádí převod mezi těmito dvěma úrovněmi, k čemuž opět využívá detekce idiomů nebo si pomáhá analýzou toku dat.

Podstatná je také detekce funkcí, které nebyly vytvořené uživatelem, ale pochází ze známých knihoven. Tato detekce je prováděna pomocí signatur a napomáhá čitelnosti výsledného kódu. Mimo knihovních funkcí je také nutno odstranit inicializační funkce. I v prostém programu typu *Hello, world!* (viz ukázka 2.3) s jednou funkcí ve zdrojovém kódu se může vyskytovat až 23 funkcí v kódu cílovém [22].

Výstupem této fáze je kód strukturovaný do bloků a funkcí, obsahující datové typy, včetně těch uživatelsky definovaných (struktury), který je uživatelem lépe čitelný neboť došlo také k navrácení různých optimalizací (bitové posuvy, cykly s podmínkou na konci).

Back-end Poslední část zpětného překladače je jazykově specifická, avšak umožňuje převádět interní reprezentaci do různých podporovaných jazyků. Pro převod využívá vytvořeného grafu volání a grafu toku řízení. Vnitřní kód je převáděn na ekvivalentní výrazy v daném jazyce. To do jaké míry jsou podporovány jazykově specifické konstrukce pak určují samotní vývojáři.

2.4 Debugger

Debugger (ladicí nástroj) je jedním z velmi užitečných nástrojů při zpětném inženýrství prováděném manuálně uživatelem. Ten spustí binární soubor a sleduje jeho běh, může tak analyzovat tok řízení a podstatné části, na které by se měl dále detailněji zaměřit.

Mimo to ladicí nástroj poskytuje širokou škálu možností pro dynamickou analýzu zdrojového kódu, které jsou velmi užitečné, zejména pokud jsou integrovány do vývojového prostředí (break pointy, zjištění obsahu paměťových buněk za běhu a další).

V případě dekompilace prováděné automaticky nemá debugger prakticky žádný význam, avšak informace pro něj určené mohou být velmi užitečné pro zpětný překladač, neboť poskytují detailní informace o původní struktuře zdrojového kódu.

2.5 Obrana proti zpětnému inženýrství

Je logické, že reverzní inženýrství není vždy žádoucí a například tvůrci proprietárního softwaru se snaží tomuto procesu zabránit nebo jej alespoň ztížit. Zatím neexistuje absolutní ochrana, avšak je možno tento proces zkomplikovat na takovou úroveň, že není časově ani ekonomicky zvládnutelný. Každá takováto snaha má ale také svou cenu, která se typicky projeví na velikosti nebo výkonu chráněné aplikace (zvláště v případě obfuskace kódu).

Základním prvkem obrany proti procesu reverzního inženýrství je mazání symbolických informací, mezi které patří názvy funkcí nebo globálních proměnných – tuto jednoduchou činnost typicky zvládá i překladač. Pokročilé postupy pak zahrnují nástroje popsané dále.

2.5.1 Obfuskátor kódu

Jedná se o nástroj, který se snaží snížit čitelnost kódu různými změnami jeho struktury, včetně přidávání neužitečných částí či skrývání řetězců. Výsledná podoba má stejnou funkcionalitu, avšak je mnohem hůře čitelná programátory.

Existují nástroje zvané *deobfuskátory*, které jsou výhodné zejména pro eliminaci neužitečného kódu. Na základě analýzy toku programu dokáží detekovat slepé uličky a tím pádem také kód, který nemá žádný reálný dopad na funkcionalitu systému. Zvládají i strukturovat kód do čitelnější podoby, tato činnost je však mnohem komplikovanější.

2.5.2 Packer

Packer je typ aplikace, která komprimuje nebo šifruje obsah spustitelného souboru. Existují dva typy těchto nástrojů

První z nich, využívající komprimaci, slouží k pouhému snížení velikosti binárního souboru. Při spuštění je poté uplatněn algoritmus, který obsah dekomprimuje. Tento proces není příliš složitý a je jednoduché jej zvrátit.

Druhý typ navíc využívá šifrování, jehož cílem je znesnadnit proces reverzního inženýrství. Aplikace je zašifrována a poté dešifrována až za běhu na základě klíče. Představuje to komplikaci pro statickou analýzu kódu, analýza kódu za běhu nijak netrpí. Získání zdrojového kódu může být složitější, neboť pokročilé nástroje (jako například Themida [18]), postupně dešifrují pouze aktuálně využívanou část aplikace. Ta tady v paměti není v žádný okamžik celá. V některých případech je však možno aplikaci dešifrovat a získat kód i pro statickou analýzu.

K rozšifrování a dekomprimaci slouží nástroje zvané *unpacker*.

2.5.3 Antidebugger

Součástí chráněné aplikace je speciální kód, který se snaží zamezit funkci ladicích nástrojů, případně detekovat jejich přítomnost a ukončit se [31]. Debuggery jsou totiž velmi často využívány při manuální dekompilaci, zejména pro získání dešifrovacího klíče.

Tyto snahy jsou však typicky platformně specifické, nepříliš spolehlivé a je možné je obejít.

2.6 Zpětný překladač společnosti AVG

Dekompilátor, který je středem zájmu v této práci, nese název *Retargetable Decompiler* [17]. Jedná se o univerzální nástroj, který není závislý na architektuře počítače ani formátu binárního souboru (a v oblasti rekonfigurovatelných zpětných překladačů ani neexistuje konkurence [29]). Využívá podobné struktury, jaká byla popsána v sekci 2.3.1. Během vývoje nástroje csim podporoval následující architektury – *x86* (respektive *IA-32*, neboť je podporována pouze 32 bitová verze, to platí i pro další zmíněné), *ARM*, *MIPS*, *PIC32* a *PowerPC*.

Vstupem je binární soubor, ve formátech *PE*, *ELF* či *COFF*, a výstupem kód v jazyce C nebo v jazyce podobném Pythonu. Kromě toho ale také dokáže generovat graf volání, výstup v jazyce symbolických instrukcí, graf toku řízení a další statistiky.

Jeho velkým plusem je schopnost detekovat knihovní funkce na základě jejich otisků, což výrazně zvyšuje čitelnost kódu. Dokáže také dobře napodobit původní strukturu kódu, tedy funkce, cykly nebo struktury. K tomu využívá také ladicí informace jak ve formátu *PDB*, tak i *DWARF* [30].

Součástí vyvíjeného zpětného překladače jsou i podpůrné nástroje, jako například nástroj pro dešifrování či extrahování binárního souboru (unpacker). Pro testování kvality zpětného překladu byl původně využíván nástroj *csimilarityCMP*, jehož náhradou je nástroj *csim*, vyvinutý v rámci této práce.

Dekompilaci si může čtenář vyzkoušet sám na webu [17] a s pomocí webového *API* (aplikační rozhraní) také integrovat zpětný překlad do svého vlastního nástroje. Zde je k dispozici krátká ukázka kódu před překladem, ukázka 2.3, a po zpětném překladu, ukázka 2.4. Složitější ukázky pak budou poskytnuty dále v kapitole 4.

```
#include <stdio.h>

int main() {
    printf("Hello, world!\n");
    return 0;
}
```

Ukázka 2.3: Program *Hello, world!* před překladem

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

// Address range: 0x407740 - 0x40775f
int main(int argc, char ** argv) {
    ___main(); // 0x407740
    puts("Hello, world!");
    return 0;
}
```

Ukázka 2.4: Program *Hello, world!* po zpětném překladu

Kapitola 3

Projekt LLVM

LLVM [16] je rozsáhlým aktivně vyvíjeným open-source projektem, který původně vznikl jakožto výzkumná práce na univerzitě v americké Illinois v roce 2000. Zahrnuje překladač *Clang* [4] a sadu nástrojů užitečných při psaní aplikací pro statickou analýzu nebo provádění optimalizací.

Mimo již zmíněné zahrnuje projekt LLVM i další nástroje a knihovny, jako je například debugger *LLDB*, *libc++*, což jsou optimalizované standardní knihovny jazyka C++ nebo *LLVM Core* provádějící jazykově nezávislé optimalizace nad *LLVM IR* (viz sekce 3.1).

3.1 LLVM IR

LLVM IR (*LLVM Intermediate Representation*) je interní reprezentace využívána překladačem Clang a dalšími nástroji (například i zpětným překladačem Retargetable Decompiler [29]). Reprezentace je podobná jazyku symbolických instrukcí, avšak abstrahuje od jakýchkoliv strojově specifických instrukcí. Namísto registrů také využívá proměnné.

I když je primárně určen pro použití při překladu programů z jazyků C a C++, je možno za pomoci knihoven LLVM převést do interní reprezentace celou řadu dalších jazyků, jako například Ruby nebo Python [11].

Celkem jsou poskytovány tři úrovně reprezentace – formát podobný jazyku symbolických instrukcí (viz ukázka 3.1), který je dobře čitelný, formát vhodný pro zpracování frontendovými nástroji a bajtkód, určený především pro serializaci.

```
@.str = private constant [13 x i8] c"Hello, world!\00"
declare i32 @printf(i8*, ...)

; Function Attrs: nounwind uwtable
define i32 @main(i32 %argc, i8** \%argv) {
  %1 = alloca i32
  %2 = alloca i32
  %3 = alloca i8**
  store i32 0, i32* %1
  store i32 %argc, i32* %2
  store i8** %argv, i8*** %3
  %4 = call i32 (i8*, ...)* @printf(i8*
        getelementptr inbounds ([13 x i8]* @.str, i32 0, i32 0))
  ret i32 0
}
```

Ukázka 3.1: Část LLVM IR odpovídající programu *Hello, world!* (viz ukázka 2.3)

3.2 Clang

Clang je především znám jako překladač jazyků C, C++ a Objective C, který jako svou interní reprezentaci využívá právě LLVM IR. V počátku byl vyvíjen zejména jako součást překladače *GCC* optimalizovaná pro překlad Objective C, avšak brzo se projekt transformoval v samostatný překladač.

3.2.1 Clang AST

Clang AST je reprezentace zdrojového kódu v podobě abstraktního syntaktického stromu (anglicky ***A**bstract **S**yntax **T**ree*), jehož struktura je velmi podobná C++ kódu a tudíž i dobře analyzovatelná.

Abstraktní syntaktický strom je pro činnost překladače velmi podstatný a je typicky výsledkem jeho syntaktické nebo sémantické analýzy. Reprezentuje zdrojový kód, kde každý uzel stromu představuje jednu určitou konstrukci daného jazyka. V této reprezentaci může být kód sémanticky i syntakticky ověřen, převeden do interní reprezentace a dále optimalizován.

Tento strom je díky kvalitnímu API možno efektivně procházet. Je složen z uzlů. Těmi základními jsou:

- `Type` – reprezentující datové typy,
- `Decl` – reprezentující veškeré deklarace, které jsou jeho podtřídami, mezi ty patří deklarace proměnných (`VarDecl`) či funkcí (`FuncDecl`),
- `Stmt` – reprezentující všechny příkazy a výrazy, dále je upřesňován podtřídami jako `Expr` (matematický výraz), `ForStmt` (cyklus `for`) nebo `ReturnStmt` (návrat z funkce).

```
cdefs.h:55:54> /usr/include/stdio.h:920:13 funlockfile 'void (FILE *)' extern
| |-ParmVarDecl 0x3a3d290 <col:26, col:32> col:32 __stream 'FILE *'
| `-NoThrowAttr 0x3a3d3c0 </usr/include/x86_64-linux-gnu/sys/cdefs.h:55:35>
`-FunctionDecl 0x3a3d550 <hello.c:3:1, line:6:1> line:3:5 main 'int (int, char **)'
  |-ParmVarDecl 0x3a3d410 <col:10, col:14> col:14 argc 'int'
  |-ParmVarDecl 0x3a3d480 <col:20, col:27> col:27 argv 'char **'
  `-CompoundStmt 0x3a3d780 <col:33, line:6:1>
    |-CallExpr 0x3a3d6e0 <line:4:2, col:23> 'int'
    | |-ImplicitCastExpr 0x3a3d6c8 <col:2> 'int (*)(const char *, .)' <FunctionToPointerDecay>
    | | `-DeclRefExpr 0x3a3d600 <col:2> 'int (const char *)' Function 0x3a2f220 'printf' 'int (const char *)'
    | `-ImplicitCastExpr 0x3a3d728 <col:9> 'const char *' <BitCast>
    |   `-ImplicitCastExpr 0x3a3d710 <col:9> 'char *' <ArrayToPointerDecay>
    |     `-StringLiteral 0x3a3d668 <col:9> 'char [13]' lvalue "Hello, world!"
    `-ReturnStmt 0x3a3d760 <line:5:2, col:9>
      `-IntegerLiteral 0x3a3d740 <col:9> 'int' 0
```

Ukázka 3.2: Výsek *Clang AST* odpovídající programu *Hello, world!* (viz ukázka 2.3)

3.2.2 Rozhraní Clang

Clang, jakožto knihovna, poskytuje rozhraní [3] umožňující psát nástroje vyžadující syntaktické či sémantické informace o zdrojovém kódu. Celkem existují tři dostupná rozhraní:

- LibClang,
- Clang Plugins,
- LibTooling.

LibClang *LibClang* je rozhraní napsané v C, ale využitelné i v C++ či Pythonu (za pomocí *C bindings*). Rozhraní je spíše stabilní a snaží se zachovávat zpětnou kompatibilitu napříč novými verzemi.

Poskytuje přístup k abstraktnímu syntaktickému stromu na vysoké úrovni s velkým množstvím abstrakcí. Není tudíž vhodné v případě, že chceme mít plnou kontrolu nad Clang AST.

Clang Plugins Rozhraní *Clang Plugins* umožňující úpravy a analýzu abstraktního syntaktického stromu během překladu a to v podobě přídavných akcí, které jsou vykonány *front-endem* překladače.

LibTooling Rozhraní *LibTooling* je vytvořené v C++ a poskytuje úplnou kontrolu nad AST. Lze jej využít k psaní nástrojů nezávislých na překladači.

Problémem je, že se nehledí na zachovávání zpětné kompatibility a rozhraní se napříč novými verzemi často mění. Rozhraní také neposkytuje prakticky žádnou abstrakci od implementace a vyžaduje tedy jeho větší znalost programátorem.

3.2.3 Clang versus GCC

Výhodou GCC, kterou se však projekt LLVM snaží eliminovat, je širší podpora jazyků a také cílových platforem.

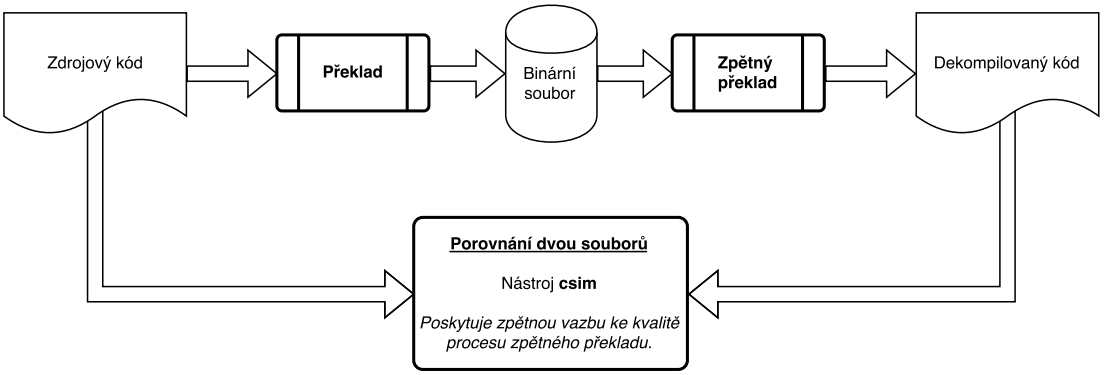
Naopak výhod Clangu je celá řada [5]. Clang a jím nabízené knihovny, zejména v souvislosti s abstraktním syntaktickým stromem, jsou velmi dobře použitelné a čitelné. Většina nástrojů navíc nabízí veřejná a dokumentovaná API, umožňující statickou analýzu zdrojových kódů. Tato výhoda je patrná zejména v nástroji csim. Clang také nabízí snáze pochopitelná a detailnější chybová hlášení.

Co se týče rychlosti, existují oblasti, kde Clang vyniká, ale i oblasti, kde GCC stále dominuje [15]. Vývojáři překladače Clang však tuto mezeru s každým novým vydáním minimalizují.

Kapitola 4

Problematika porovnání dekompilovaných souborů

Cílem této bakalářské práce (viz obrázek 4.1) je vytvořit nástroj, který porovnává dva soubory – původní a dekompilovaný. Výstupem takového porovnání je nejen celková podobnost, ale i podobnost dle různých kritérií. Ty poskytují vývojářům zpětnou vazbu k provedeným změnám a informují o oblastech, v nichž zpětný překladač vyniká a ve kterých nikoliv.



Obrázek 4.1: Znázornění cíle této bakalářské práce

Porovnání zahrnující dekompilované soubory má svá specifika, která značně komplikují tento proces. Zatímco existují nástroje pro porovnání dvou obecných zdrojových kódů nebo dvou binárních souborů, autorovi se nepodařilo nalézt žádný nástroj snažící se o podobné porovnání, o které se snaží nástroj csim. A to ani nástroj, který by obdobným způsobem porovnával abstraktní syntaktické stromy obecných zdrojových souborů.

Samozřejmě šlo nalézt nástroje, které porovnávají dva soubory. Jedním z nich byl *BinDiff* [19], který porovnává binární soubory. Ten bude také detailněji popsán dále, neboť byl využíván jako referenční. Nástrojů pro porovnání zdrojových kódů na úrovni textu existuje více, můžeme zmínit například webový *DiffNow* [7], avšak porovnání je opravdu prováděno pouze na textové úrovni – chybějící nebo přebývající slova či řádky atd. I když DiffNow umožňuje specifikovat porovnání zdrojových kódů jazyka C/C++ a poté nabízí různá vylepšení (například uvažuje `return EXIT_FAILURE` za ekvivalentní k `return 1`), tak je takovéto porovnání v případě originálního a dekompilovaného souboru naprosto nevhodné. I proto se dále budeme bavit spíše o hypotetických nástrojích, porovnávajících abstraktní syntaktické stromy dvou obecných kódů a rozdílech, při porovnání původního a dekompilovaného kódu.

Také fakt, že získání izomorfního kódu (totožná struktura původního a dekompilovaného kódu) není ze své podstaty možné, značně ovlivňuje proces porovnání těchto souborů. Maximální podobnost u klasických nástrojů znamená, že jsou soubory totožné – jejich struktura, parametry funkcí, graf volání a další. Maximální podobnost u nástroje csim však spíše vyjadřuje, že zpětný překladač vynaložil maximální úsilí pří získávání původní podoby kódu vzhledem k informacím, které měl k dispozici. Z toho logicky vyplývá, že musí tolerovat různé nedostatky, které zpětný překladač nemohl ovlivnit.

4.1 Porovnání dekompilovaného souboru s původním

Soubory lze na úrovni zdrojových kódů porovnávat z více hledisek. Tím stěžejním, a také nejkomplikovanějším, je porovnání funkcí. Jádrem tohoto porovnání je proces promítnutí funkcí jednoho souboru na jejich protějšky v druhém souboru. Pokud jsou dostupné symboly (a je známo, že nedošlo k přejmenování funkcí), můžeme je k promítnutí využít. Tím je celý proces značně zjednodušen. Pokud se na ně však nemůžeme spolehnout, je nutno provést promítnutí na základě podobnosti atributů funkcí.

Atributů je mnoho a porovnávaná množina se může napříč různými nástroji lišit. Ty pro porovnání dvou obecných zdrojových kódů by například mohly porovnávat typy parametrů a proměnných, použité příkazy, typy cyklů a další – na žádnou z těchto informací se nelze při porovnávání dekompilovaného souboru s původním spolehnout.

Nástroje pro porovnání binárních souborů využívají typy instrukcí či délky těl funkcí – což se překladem a následnou dekompilací taktéž silně mění. Inference datových typů po překladu je velmi složitá a původní podoby ani nelze dosáhnout. Typy cyklů jsou překladači často měněny, to samé platí i pro strukturu podmíněných výrazů. Délka funkcí se pak často liší i více než desetinásobně.

Velmi slibně může znít idea promítání funkcí skrze *graf volání* [1]. I ten však nemusí být zachován, zejména kvůli již zmíněné optimalizaci zvané *inlining*. A to už vůbec nebereme v potaz možné chyby zpětného překladače, kdy by i jedna nedetekovaná funkce zmařila výsledky celé analýzy. Přesto je to jeden z nejspolehlivějších ukazatelů, na kterém lze řešení jednoznačně založit, jak bude ukázáno později.

Různých již zmíněných nedostatků, které musí nástroj csim tolerovat, bylo detekováno větší množství. Při návrhu nástroje csim a následné implementaci na ně byl brán zřetel. Níže můžeme nalézt krátký výčet některých těchto problémů.

- inference celočíselných datových typů na typy s definovanou šířkou
  `int` ⇒ `int32_t`
  - s tím souvisí vkládání hlavičkového souboru `stdint.h` definujícího tyto typy
- záměna proměnných za ukazatele na proměnnou
  `int` ⇒ `int *`
- náhrada návratových typů `void` za `int`
  `void function()` ⇒ `int32_t function()`
- neschopnost detekovat typy cyklů
- neschopnost rekonstruovat strukturu složených podmíněných příkazů

[1] Graf volání je orientovaný graf, jehož každý uzel představuje funkci a každá hrana popisuje vztah dvou funkcí, kde jedna stojí v pozici volajícího a druhá v pozici volaného.

- rozdíly v grafech volání jinak funkčně ekvivalentních programů

Problematika se dále komplikuje s rostoucí počtem prováděných optimalizací. Nástroj musí být schopen porovnat i soubory, které spolu na první pohled nemají nic společného a objektivně vyjádřit jejich podobnost.

4.2 Porovnání dvou dekompilovaných souborů

I když se práce primárně zaměřuje na porovnání původního programu s jeho dekompilovanou variantou, je řešení rozšířeno i na porovnání dvou dekompilovaných souborů. Hlavní využití se, vzhledem k zaměření zpětného překladače, nalézá v porovnání dvou dekompilovaných vzorků, z nichž jeden je škodlivým softwarem (malwarem). Samotná podobnost poté představuje pravděpodobnost, že druhý soubor je taktéž stejnou rodinou malwaru. Podobně bychom mohli tento nástroj využít i pro porovnání dvou vzorků virů (s časovým rozestupem) a detekci změn, které byly provedeny.

Protože před procesem zpětného překladu neexistuje stoprocentní ochrana, zaměřují se tvůrci malware spíše prodloužení doby, po kterou je nově vyvinutý virus, nebo jeho pozměněná verze, schopen unikat detekci antivirovými systémy [24]. Čím delší tato doba je, tím více počítačů je infikováno. Detekce virů je typicky prováděna porovnáním otisku souboru s databází otisků virů. Cílem je tedy změnit otisk nové verze viru natolik, že se již nepodobá svému předchůdci. K tomuto je možno využít například následujících dvou postupů – *polymorfismu* a *metamorfismu* [31].

Polymorfismus *Polymorfismus* je proces šifrování zdrojového kódu, který je poté opětovně dešifrován za běhu, a vkládání neužitečných částí před a za zdrojový kód. Hlavním nedostatkem této metody je fakt, že samotný šifrovací algoritmus (který na nejvyšší úrovni logicky šifrovaný není a ani být nemůže) lze využít jako otisk k detekci viru.

Metamorfismus *Metamorfismus* je složitější a mnohem více nebezpečný, neboť dokáže déle unikat odhalení. Kód při *replikaci* (infikování dalšího počítače) sám mění svou strukturu – to zahrnuje spojování či rozdělování funkcí, vkládání neužitečného kódu, prohazování pořadí instrukcí nebo celých funkcí, převod podmínek na ekvivalentní a další menší či větší změny. Této metody využívá mnoho rozšířených malwarů a takto upravené viry je obtížnější detekovat.

Nástroj csim, který je psán speciálně za účelem objektivního porovnání dvou souborů, které se mohou značně lišit, zde může dosahovat zajímavých výsledků, zejména v módu porovnání dvou dekompilovaných souborů (kde lze využít optimalizací jinak nepoužitelných při porovnání původního a dekompilovaného souboru).

4.3 Ukázkové porovnání

Aby si čtenář lépe dokázal představit zmíněné problémy, je přiložena následující ukázka zobrazující podobu dekompilovaného kódu a jeho vztah k původnímu, který se, jak již bylo zmíněno, značně liší se stoupající úrovní optimalizací.

Zdrojový kód byl přeložen překladačem *GCC* na operačním systému *Windows* pro architekturu *Intel x86*. Nebylo využito vkládání ladicích informací a došlo k odstranění symbolických jmen (názvy, které vidíte v ukázce 4.2 a v ukázce 4.3 byly vytvořeny zpětným překladačem). Spouštěné optimalizace se liší napříč ukázkami.

4.3.1 Původní kód

Zdrojovým kódem je poněkud komplikovaný program typu „Hello, world!“ (ukázka 4.1) obsahující funkce, struktury a cykly. Ten je přeložen a poté dekompilován zpětným překladačem Retargetable Decompiler verze `2.1.2` zveřejněné 27. ledna 2016 (v ukázkách chybí komentáře, obsahující například adresový rozsah funkce, z důvodu snadnější čitelnosti a vyšší kompaktnosti).

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

struct Delimiter{
    char character;
};

char* string[] = {"Hello,", "world!"};

void print(int offset) {
    printf(string[offset]);
}

int main(int argc, char **argv) {
    struct Delimiter *delChar = malloc(sizeof(struct Delimiter));
    delChar->character = '␣';

    for (int i = 0; i < 2; i++) {
        print(i);
        printf("%c", delChar->character);
    }
    printf("\n");

    return 0;
}
```

Ukázka 4.1: Zdrojový kód složitějšího programu typu *Hello, world!*

4.3.2 Dekompilovaný kód

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

int32_t function_401560(int32_t a1);

char * g1[2] = {
    "Hello,",
    "world!"
};


int32_t function_401560(int32_t a1) {
    return printf(g1[a1]);
}
```

```
int main(int argc, char ** argv) {
    ___main();
    char * mem = malloc(1);
    *mem = 32;
    for (int32_t i = 0; i < 2; i++) {
        function_401560(i);
        putchar((int32_t)*mem);
    }
    putchar(10);
    return 0;
}
```

Ukázka 4.2: Dekompilovaný kód, bez optimalizací při překladu

V předcházející ukázce 4.2 nebylo využito žádných přepínačů optimalizace (ekvivalentní přepínači `O0`). Lze vidět, že kódy jsou si strukturálně velmi podobné. I přesto si však můžeme všimnout některých problémů zmíněných dříve – struktura `Delimiter` úplně chybí, datový typ `int` byl nahrazen za `char*`, návratový typ funkce `print` již není `void`, ale `int32_t`. V hlavní funkci `main` došlo k přidání proměnné a volání `printf(\n)` bylo nahrazeno `putchar(10)`, kde 10 je ASCII kód znaku nového řádku.

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

int main(int argc, char ** argv) {
    ___main();
    printf("Hello,");
    putchar(32);
    printf("world!");
    putchar(32);
    putchar(10);
    return 0;
}
```

Ukázka 4.3: Dekompilovaný kód, s nejvyšší úrovní optimalizací při překladu

V poslední ukázce 4.3 bylo využito maximální optimalizační úrovně `O3`. Můžeme si všimnout, že struktura kódu je nyní naprosto odlišná a nebýt řetězců, nebylo by v žádném případě možné poznat, že dekompilovaný kód může být podobný původnímu.

Nedostatky dekompilovaného kódu zde však nejsou způsobeny nekvalitním zpětným překladem, ale optimalizacemi a dalšími fázemi překladače. Kód informace o původní struktuře neobsahoval ani v jazyce symbolických instrukcí (a většinou již ani v optimalizovaném vnitřním kódu překladače) a tudíž nelze lépe rekonstruovat.

Kapitola 5

Původní stav

Zpětný překladač *Retargetable Decompiler* je ve vývoji již šest let a pro testování byl v minulosti napsán obdobný nástroj. Součástí zadání bylo se s tímto nástrojem seznámit a analyzovat jeho problémy, kterých by se nové řešení mělo vyvarovat. Původní nástroj nese název *csimilarityCMP*.

Dva zdrojové soubory porovnává na úrovni abstraktního syntaktického stromu. Podporovaná kritéria podobnosti jsou čtyři – porovnává řetězce, globální proměnné, funkce a graf volání.

5.1 Klady

Velkým kladem z hlediska řešení této práce byla možnost v začátcích nahlédnout do zdrojových kódů pracujících s knihovnami Clang a LLVM. Obdobné nástroje totiž nejsou příliš časté. Bylo však nutno vynaložit velké úsilí k pochopení některých částí kódu a žádnou z nich v novém nástroji, vzhledem k později zmíněným nedostatkům a jinému přístupu k řešení, nebylo možno využít.

K propracovaným částem řešení patřilo porovnání datových typů, kdy byl brán v úvahu nejen typ samotný, ale i jeho šířka. A také porovnání grafu volání, které bylo prováděno velmi kvalitně, avšak nevhodně pro tento konkrétní účel (viz dále).

5.2 Zápory

Mezi velké problémy tohoto řešení patřila jeho struktura. Zdrojové kódy neměly jednotný styl, nebyly dostatečně komentované a velmi často obsahovaly duplicitní kód. Udržovatelnost by byla problematická, neboť čitelnost a především pochopitelnost kódu byla velmi nízká. Patrné byly části kódu pocházející z různých internetových zdrojů, jejichž integrace do okolního kódu nebyla nejlepší.

Z hlediska rozšiřitelnosti nástroj také nebyl ideální. Přidání porovnání dle nových kritérií by zahrnovalo také přidání dalších duplicitních pasáží kódu a úpravu velkého množství souborů. Byla by taktéž nutná znalost knihoven Clang, jelikož zde neexistovala žádná úroveň abstrakce od těchto knihoven a samotné porovnání bylo prováděno v těsné souvislosti se získáváním dat z abstraktního syntaktického stromu. Ten byl v důsledku procházen vícenásobně, což z časového hlediska nebylo efektivní.

Přestože promítání funkcí bylo řešeno velmi pokročilým způsobem, jeho časová složitost byla v případě delších souborů obrovská (a v případě porovnání opravdu dlouhých souborů,

například s dvaceti tisíci řádky, algoritmus neskončil ani za více než 15 minut). Řešení bylo samozřejmě možno ukončit dříve, zadáním časového limitu, ale u takto velkých souborů stále nebyly dosažené výsledky příliš uspokojivé. Také nalezení optimálního časového limitu není jednoduché a určitě není úkolem pro automatizované testy.

Co bylo také dalším nedostatkem promítání funkcí byla chybějící detekce inliningu. Výsledkem byla nejen nízká podobnost, ale i to, že funkce, které ve skutečnosti měli svůj protějšek ve funkci, do které byly inlinovány, byly prohlášeny za unikátní.

Nevyhovující částí stávajícího řešení byly i poskytované výstupy. Ty byly špatně zpracovatelné strojem a i pro člověka byl problém textový výstup analyzovat. Pro představu je přiložen jeho ukázkový výstup (ukázka 5.1) a pro srovnání je také v příloze k nahlédnutí výstup nového nástroje pro totožné vstupní soubory.

Posledním, ale nijak méně významným, nedostatkem byla absence automatizovaných testů. To představuje problém ze dvou pohledů. Jednak není možno prokázat, že nástroj opravdu děla to, co má. Ale také to komplikuje provádění změn, nelze totiž otestovat, že nezpůsobily problémy na jiných místech. Právě to může být ve zdrojových kódech s problematickou strukturou velmi časté.

Všechny zmíněné problémy se nově vyvíjený nástroj *csim* pokouší odstranit.

```
Call graph:
    printf|printf    ::
    malloc|malloc    ::
      main|main      ::            malloc(1)|malloc        (1)
     print(1)|function_401560   (1)       printf(2)|printf      (1)
     print|function_401560::          printf(1)|printf      (1)
      NULL|___main   ::
      NULL|putchar   ::

Program call graph similarity: 76%

Functions & parameters comparison:
  Function: main
        Return type                    : 100 %
        Count of parameters            : OK
        Type of parameters             : 100 %
  Function: print
        Return type                    :   0 %
        Count of parameters            : OK
        Type of parameters             : 100 %
  Count of wrong recognized functions  : 0

"world!"
"Hello,"
String similarity : 100 %

Variable comparation:  50%
   char*[2] string() x  (Not found)         :   0 %
   int offset(print) x  (Not found)         :   0 %
      int argc(main) x int argc(main)       : 100 %
   char** argv(main) x char** argv(main)    : 100 %
struct* delChar(main) x  (Not found)         :   0 %
         int i(main) x int i(main)          : 100 %
```

Ukázka 5.1: Výstup původního nástroje pro ukázky 4.1 a 4.2

Kapitola 6

Návrh nového řešení

Na základě setkání s vývojáři zpětného překladače *Retargetable Decompiler* byly vytyčeny základní požadavky, jak funkcionální, strukturální tak i výkonnostní, které musí finální nástroj splňovat.

Podstatný pro návrh je fakt, že se jedná o velmi specificky zaměřený software pro porovnání dvou zdrojových souborů a k tomuto účelu je nutno přihlédnout. Jednak můžeme využít optimalizací, které by u porovnání obecných souborů použít nešly, ale také musíme tolerovat nepřesnosti, které zavádí problematika zpětného překladu.

6.1 Požadavky na řešení

Hlavním požadavkem je vytvoření kvalitnějšího řešení, než je to stávající. Slovo *kvalitnější* v tomto případě znamená, že nástroj bude podporovat porovnání podle více kritérií, bude poskytovat detailnější a užitečnější výstupy ve více formátech (určených jak pro člověka, tak pro zpracování strojem), bude rychlejší než stávající řešení, bude snadno rozšiřitelný a udržovatelný, a bude multiplatformní (podstatné je stejné chování na Windows i UNIX).

Kromě toho je kladen velký důraz na kvalitu zdrojových kódů. Ty musí mít jednotný formát, být vhodně komentované a využívat dobrých programátorských praktik, aby byla implementace efektivní, dobře čitelná a kódy snadno upravitelné.

Samotné řešení je určeno pro okamžité nasazení do provozu a z toho důvodu je nutno jej důkladně otestovat. A to jak na úrovni jednotlivých modulů, tak jako celek. Samozřejmě včetně testů po zkušebním integrování nástroje do automatizovaných nočních testů v nichž spočívá jeho hlavní využití.

Vzhledem k rozsahu projektu a používaným knihovnám je nutno projekt implementovat za použití principů objektově orientovaného programování.

6.2 Kritéria podobnosti

V rámci nástroje je zdrojový kód porovnáván z mnoha hledisek. Ty hlavní, které jsou zastřešeny vlastní analýzou (s výjimkou analýzy podobnosti grafu volání, která je součástí porovnání funkcí, neboť je na něm značně závislá), jsou následující:

- funkce,
- graf volání,

- hlavičkové soubory,
- řetězcové literály,
- složené datové typy,
- globální proměnné a inicializátory.

6.3 Struktura řešení

Výsledkem této bakalářské práce je nástroj *csim* a knihovna, kterou využívá.

Knihovna, s názvem *csiml*, úzce spolupracuje s rozhraním poskytovaným knihovnami překladače *Clang*. Ty je nutno využít neboť porovnání není prováděno nad textovou podobou zdrojového kódu, ale nad jeho reprezentací ve formě abstraktního syntaktického stromu (dále jako *AST*), který *front-endové* nástroje překladače vytváří. Pojem *abstraktní syntaxe* vyjadřuje, že strom neobsahuje veškeré konstrukce, ale pouze ty podstatné (chybí například závorky, které jsou ve stromu implicitní). Tím se tedy abstraktní syntaktický strom liší od derivačního stromu (anglicky *concrete syntax tree*).

Aby bylo rozšiřování nástroje snazší (tedy nebyla vyžadována extenzivní znalost knihoven Clang) a nedocházelo k opakovanému procházení AST, je nutno informace z něj načíst do vnitřních struktur nástroje, které jsou součástí třídy `FileInfo`. Základem všech analýz je třída `Similarity`, definující rozhraní, pomocí kterého analýzy komunikují se zbytkem nástroje. Obdobně je vytvořena také bázová třída `Result`, obsahující výsledky analýz a provádějící jejich transformaci na výstup v textovém formátu a formátu *JSON*.

6.4 Výběr rozhraní Clang

V sekci 3.2.2 byly zmíněny tři rozhraní, pomocí kterých lze přistupovat k abstraktnímu syntaktickému stromu. Před samotným návrhem bylo nutno jedno z nich zvolit.

Vybráno bylo rozhraní *LibTooling*, které poskytuje největší svobodu při manipulaci s abstraktním syntaktickým stromem. Rozhraní se ale často mění, neexistuje zde prakticky žádná abstrakce od implementace a je tedy velmi rozsáhlé. I přes tyto problémy však stále představuje nejlepší volbu, právě díky možnosti získat prakticky libovolné informace o zdrojovém souboru, a tím představuje velký potenciál pro budoucí rozšiřování.

6.5 Nástroj csim

Samotný nástroj csim slouží pouze pro interakci s uživatelem a poskytování výstupu na základě spouštění analýz podobnosti. Interakce s uživatelem je kompletně založena na přepínačích, neboť se jedná o konzolovou aplikaci. Ty musí umožnit předání dvou zdrojových kódů pro porovnání, které mohou být zadány souborem, ale i řetězcovým literálem (což je užitečné zejména pro testování). Ve výchozím nastavení jsou vždy spouštěny všechny dostupné analýzy, avšak musí být možno spustit pouze některé. Vzhledem k požadavku na snadnou rozšiřitelnost musí být tyto přepínače nastavovány dynamicky, podle aktuálního seznamu porovnání nabízených knihovnou csiml.

Jelikož je řešení založeno na použití knihoven překladače, je podstatné, aby nástroj měl k dispozici všechny využívané hlavičkové soubory. Musí tedy existovat možnost zadání

adresářů pro jejich vyhledání. Bez hlavičkových souborů nemusí být funkce a další konstrukce v nich definované, správně rozpoznány, což vede k syntaktickým chybám majícím velký vliv na podobu konstruovaného stromu (pokud například použiji funkci `printf()` bez uvedení hlavičkového souboru `stdio.h`, nebude tato funkce správně rozpoznána a bude ve výsledném AST chybět).

Kromě již zmíněných, nástroj musí podporovat také přepínače pro výpis nápovědy, výběr výstupního formátu, poskytnutí detailního výstupu pro ladění (tzv. *verbose*) a tichý mód, určený zejména pro automatizované testy.

Průběh analýz je možno ovlivnit třemi přepínači. `--decompiled` aktivuje optimalizace určené pro porovnání dvou dekompilovaných souborů (primárně je nástroj určen pro porovnání původního a dekompilovaného kódu). `--strip-symbols` způsobí, že k symbolickým jménům se nástroj chová stejně, jako by k dispozici nebyly (vhodné pro porovnání dvou zdrojových kódů, u nichž víme, že došlo k jejich přejmenování). Posledním je `--no-inlining`, který vypne prováděnou detekci inliningu při promítání funkcí jednoho souboru na funkce v druhém (a s tím související spojování funkcí).

6.6 Knihovna csiml

Zjednodušený třídní diagram knihovny *csiml* lze nalézt na obrázku 6.1. Třídy v ní obsažené můžeme rozdělit do tří význačných částí:

- vytváření abstraktního syntaktického stromu a jeho procházení,
- načítání informací do vnitřních struktur a jejich normalizace,
- provádění porovnání včetně zprostředkování výstupu.

Vytváření AST a jeho procházení

Vytváření abstraktního syntaktického stromu je prováděno *front-endovými* nástroji překladače *Clang*. Je nutno nastavit možnosti překladu a poskytnout cesty k vyhledání hlavičkových souborů knihoven používaných v analyzovaných zdrojových kódech.

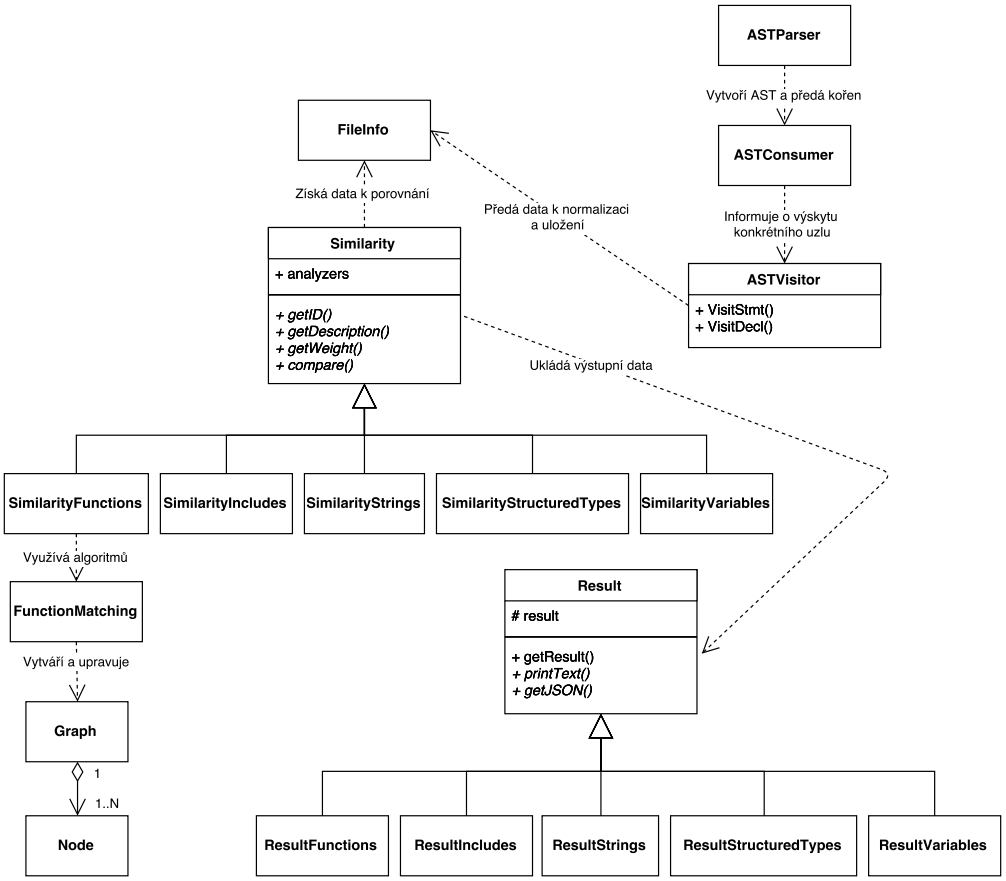
Abychom mohli analyzovat vstupy ve formě řetězcových literálů (primárně určeno pro použití *unit testy*), což překladač nezvládá, je nutno pro ně vytvořit virtuální soubory (vytvoření reálných souborů by nebylo vhodné vzhledem k předpokládanému paralelnímu používání).

Poté, co je strom vytvořen, je za pomoci třídy `ASTConsumer` získán kořenový uzel a je zahájeno rekurzivní procházení. Strom je tvořen z uzlů jistých typů, se kterými jsou spjaty metody třídy zvané `ASTVisitor` a ty jsou volány, pokud je daný uzel nalezen. Tyto metody jsou virtuální a jejich předefinováním můžeme ze stromu extrahovat informace. Pro nás jsou zajímavé zejména dvě metody – `VisitStmt()` a `VisitDecl()`.

`VisitDecl()` je volána, pokud je ve stromu nalezena deklarace funkce, proměnné nebo struktury. Poté, co je rozhodnuto o typu prvku, je ukazatel na jeho deklaraci předán odpovídající metodě třídy `FileInfo`, která z něj extrahuje informace potřebné pro porovnání.

`VisitStmt()` je volána, pokud je nalezen výskyt jakéhokoliv výrazu, ať už matematické rovnice, řetězcového či číselného literálu nebo volání funkce. Podstatné informace jsou taktéž zpracovávány třídou `FileInfo`.

Menší komplikací je v tomto případě fakt, že AST je procházen jako celek, tedy včetně prvků pocházejících z hlavičkových souborů. Je tedy nutno rozhodnout, které prvky jsou pro



Obrázek 6.1: Zjednodušený třídní diagram knihovny *csiml*

nás zajímavé a které nikoliv. To je složitější zejména v případě, že analyzujeme řetězcový vstup. V rámci tohoto procesu musí také dojít k uložení názvů použitých hlavičkových souborů (pozor na to, že ne vždy jsou vkládány jen na začátku souboru).

Načítání informací z AST a jejich normalizace

Po rozhodnutí o typu uzlu je volána odpovídající funkce uživatelské třídy `FileInfo`, která uzel zpracuje a v odpovídajícím formátu načte informace v něm obsažené do patřičných struktur.

Pro porovnání, která jsou knihovnou csiml nabízena, je důležité získat informace uvedené dále. Pojmy jako *bázový typ* a *signatura* budou vysvětleny později.

- globální proměnné a inicializátory
  - název
  - typ
  - bázový typ
  - inicializátor (vektor hodnot)
- hlavičkové soubory
  - název

- řetězcové literály
  - text
- strukturované typy
  - název
  - typ (struktura nebo union)
  - položky
    - název
    - typ
    - bázový typ
  - signatura
- funkce
  - jméno
  - návratový typ
  - bázový návratový typ
  - parametry
    - typ
    - bázový typ
  - lokální proměnné
    - typ
    - bázový typ
  - signatura
  - počet cyklů jednotlivých typů
  - počet podmínek
  - volané funkce z testovaného souboru
  - názvy volaných funkcí z knihoven
  - složitost funkce

Vzhledem k tomu, že inference typů proměnných či parametrů je jedním z nejsložitějších problémů zpětného překladu, je nutno tolerovat různé nedostatky. Toho lze dosáhnout využíváním bázových typů, které sdružují zaměnitelné typy pod jeden hlavní. Klasické typy jsou samozřejmě také porovnávány a celková podobnost je dána váženým průměrem obou porovnání.

Další vytvářenou položkou jsou signatury, ty jsou užitečné pro rychlé porovnání volaných funkcí, přiřazení typů strukturám nebo pro detailní výpisy. V prováděných porovnáních totiž není možno využívat jmen funkcí či struktur, neboť jejich zobrazení na sebe je prováděno až na základě výsledků analýz (a ne vždy mají původní jména). Signatura se skládá z bázových typů, u funkcí je dána návratovým typem a typem parametrů, u strukturovaných typů pak typy položek v nich obsažených.

Složitost funkce není získávána přímo z abstraktního syntaktického stromu, ale je vypočtena jakožto součet počtu významných položek, jako jsou parametry, cykly či volání

funkcí. Pokud bylo specifikováno porovnání dvou dekompilovaných souborů jsou zde taktéž zahrnuty lokální proměnné (při porovnání původního a dekompilovaného by toto možné nebylo, neboť počty se liší i více než desetinásobně).

U lokálních proměnných a parametrů je taktéž zbytečné ukládat názvy, neboť je k porovnání nevyužíváme a pro výstup analýz nejsou natolik podstatné (navíc často dochází ke smazání této informace překladačem). Názvy funkcí či struktur naopak využít lze (pokud máme k dispozici symbolická jména a ladicí informace – názvy struktur totiž nejsou součástí symbolických jmen, zde jsou zahrnuty pouze názvy funkcí a globálních proměnných) a jsou velmi podstatné pro srozumitelnost výstupů nástroje.

Zmíněna byla dříve také normalizace informací, která se primárně skládá z vytváření bázových typů. Mimo to je ale při ukládání přihlédnuto k dalším problémům zpětného překladu a dochází tedy například k umazání hlavičkového souboru `stdint.h`, neboť ten je přidán kvůli používání typů s definovanou šířkou (`int32_t`). Také návratový typ `void` je zaměňován za `int` nebo jsou umazávány volání inicializačních funkcí (například `___main`).

Porovnávání

Třída `Similarity` je bázovou třídou všech analýz a kromě již zmíněné definice rozhraní, také zastřešuje registraci nových porovnání. Odtud je poté nástroj csim schopen načíst informace o aktuálně nabízených analýzách. Zdrojové soubory této třídy jsou také jediné, které je třeba editovat při přidání nového typu porovnání – samozřejmě kromě souborů samotné analýzy. Třída `Result` je bázovou třídou definující rozhraní pro získání výsledku ve dvou formátech a výstupy povinné pro všechny analýzy (například celkový výsledek).

Každou analýzu poté tvoří třída odvozená z třídy `Similarity`, implementující samotné porovnání, a třída dědící z `Result`, která obsahuje výstupní informace a provádí jejich transformaci do výstupního formátu.

Vzhledem k tomu, že některé analýzy jsou podstatnější než jiné a nástroj má nabízet také celkový výsledek všech analýz, je nutno jejich výsledky váhovat. Váha je dána číslem na škále od 1 do 10, které je dále násobeno počtem porovnávaných položek (deset stejných funkcí o podobnosti vypovídá více, než jeden odlišný řetězec). Aby však nedošlo k přílišnému ovlivnění výsledku jednou analýzou, je váhový strop nastaven na hodnotu 100.

Porovnání funkcí a grafu volání Porovnání funkcí je pro podobnost dvou souborů naprosto stěžejní. I proto má maximální váhu 10 a je nutno mu věnovat patřičnou pozornost. Základem porovnání funkcí je správné zobrazení původní funkce na dekompilovanou. V případě, že jsou k dispozici symboly, jsou na sebe promítnuty pouze na základě svých názvů (s následnou detekcí inlinigu). Pokud však k dispozici nejsou, porovnání je mnohem složitější, a návrh i výběr nejlepšího řešení byl jednou z největších výzev v rámci této práce.

Přístupů k porovnání funkcí bylo navrženo více, celkem byly implementovány tři generace analýzy, ze kterých byl na základě testování vybrán ten nejlepší algoritmus.

První generace vzájemně přiřazovala funkce pouze na základě jejich podobnosti (výpočet zmíněn dále). I když tento algoritmus dosahoval dobrých výsledků a byl velmi rychlý, mohlo lehce dojít ke zkreslení analýzy vzájemným přiřazením podobných funkcí, které však neodpovídalo realitě. I proto byla druhá generace navíc založena na přiřazování funkcí průchodem grafu volání. Tato analýza byla o poznání přesnější, avšak problémy, jakými jsou *inlining* nebo špatná detekce volání funkce, mohli způsobit úplné selhání této analýzy již na počátku.

Třetí generace, která byla později zvolena jako nejlepší, kombinuje obě předchozí generace a navíc zavádí složitost funkcí a detekci *inliningu* nebo naopak rozdělení funkce (pravděpodobné při polymorfismu malwaru). Princip této metody je popsán v dalších odstavcích.

Algoritmus promítání funkcí Nejprve dojde k vytvoření reprezentace grafu volání pro oba soubory zvlášť, kde každý uzel obsahuje množinu funkcí z daného souboru, odpovídající uzel z grafu volání druhého souboru, bezprostřední následníky a předchůdce, a následníky a předchůdce do definované hloubky (ve výchozím nastavení 3). Na počátku obsahuje uzel funkci, které odpovídá.

Poté dojde k vzájemnému porovnání všech funkcí (pokud nebyly k dispozici symboly, jinak jsou funkce promítnuty pouze na základě svého jména), jehož výsledky jsou seřazeny podle dosažené podobnosti (je nutno použít stabilní algoritmus řazení, neboť pořadí také vypovídá o podobnosti). Z takto seřazené posloupnosti jsou nejprve vybírány ty dvojice funkcí, u kterých je pravděpodobnost správného přiřazení nejvyšší, a těmi jsou grafy volání postupně populovány. Při vkládání každé takové dvojice je zkontrolováno, že neporušuje strukturu grafu volání původního souboru a to tak, že pokud přímý následník, respektive předchůdce, již má přiřazenu svou dvojici, tak se tento uzel musí vyskytovat v množině následníků, respektive přechůdců, plánované dvojice. Takto se algoritmus dokáže vypořádat i s inliningem.

Na počátku jsou pro přiřazení vybírány funkce s větší než průměrnou složitostí, které splňují další limity, týkající se parametrů a dalších atributů funkcí. U složitějších funkcí je totiž menší pravděpodobnost přiřazení velmi podobných funkcí, které však mají úplně rozdílnou polohu v grafu volání. V dalších iteracích jsou limity snižovány a dochází také k přiřazování funkcí jednoduchých, které však musí respektovat dříve přiřazené dvojice složitých funkcí a nemělo by tedy docházet k nepřesnostem.

Poté, co jsou všechna možná přiřazení provedena, dochází k finální fázi porovnání, která z nepřiřazených funkcí vybírá takové, u nichž existuje podezření na inlining (jejich poloha v grafu volání jednoho souboru v druhém grafu neexistuje). Ty zkouší spojit (dochází k sjednocení funkcí, které má uzel přiřazen) a porovnat s funkcí, nebo funkcemi, v druhém uzlu. Pokud je nová podobnost vyšší, je inlining potvrzen a uzly grafu spojeny. Opětovně je taková detekce prováděna i opačným směrem, kde místo inliningu detekujeme rozdělení funkcí (vytknutí části kódu). Tato detekce je opakována a dokáže se tedy vyrovnat i s vícenásobným inliningem.

Porovnání dvou funkcí v první fázi je založeno na porovnání pomocí osmi různě váhovaných kritérií (dle jejich důležitosti):

- návratový typ,
- počet a bázové typy parametrů,
- typy lokálních proměnných, včetně bázových,
- signatury volaných funkcí ze souboru,
- názvy volaných knihovních funkcí,
- řetězcové literály,

- počet cyklů,
- počet podmínek,
- složitost funkce.

Ve funkcích jsou porovnávány pouze typy lokálních proměnných, ne počty, důvod je prostý – zpětný překladač obvykle zavádí velké množství nových proměnných (detekce původního počtu je problematická) a pokud by docházelo i k porovnání počtu výskytů, tak by byly výsledky značně zkresleny. Obdobně je porovnáván pouze celkový počet cyklů, nikoliv počty jednotlivých typů cyklů, neboť překladač jejich typy často mění (oblíbenou variantou překladače je cyklus typu `do { } while (...)`). Stejně tak je testován pouze počet podmínek, protože jejich struktura značně podléhá optimalizacím, a navíc by bylo nutno podmínky normalizovat a složitě porovnávat, což by se neblaze podepsalo na časové náročnosti řešení.

Porovnání grafu volání je prováděno až po dokončení promítání funkcí a zjišťuje, zda promítnuté funkce volají funkce, které na sebe byly taktéž promítnuty. Výsledná podobnost se poté dále snižuje o funkce, které žádný svůj protějšek v druhém souboru neměly.

Informace poskytované na výstupu jsou podobnost grafu volání a souhrnná podobnost každého z osmi atributů pro porovnání funkcí. Detailně jsou zde také uvedeny promítnuté funkce a jejich podobnost, včetně informací o detekovaném *inliningu* nebo případně rozdělení funkcí.

Porovnání řetězců Porovnávány jsou veškeré řetězcové literály v souborech, ať již globální či lokální, včetně například formátovacích řetězců ve funkcích jako `printf()`. Je zde zapotřebí brát v potaz fakt, že překladač může převádět funkci `printf()` na `puts()` [34], pokud končí znakem nového řádku a neobsahuje formátovací značky, tím dojde k odstranění tohoto znaku – ten je tedy vždy odstraněn již při ukládání do vnitřních struktur. Obdobně se také převádí volání `printf()` na `putchar()`, pokud je vypisován jediný znak. Taktéž se může stát, že zpětný překladač *inlinuje* některé řetězce, proto je nutno vytvářet množinu namísto multimnožiny.

Poskytovanými výstupy jsou společné řetězce a řetězce unikátní pro každý ze dvou souborů. Výsledná procentuální shoda je vypočtena na základě vzorce 6.1 pro Jaccardův koeficient podobnosti [27], který vyjadřuje podobnost dvou množin.

$$
\begin{aligned}
J(A,B) &= \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|} \\
J(A,B) &\in \langle 0, 1 \rangle \\
J(A,B) &= 1, \text{pokud } A \text{ i } B \text{ jsou prázdné}
\end{aligned}
\tag{6.1}
$$

Porovnání použitých hlavičkových souborů Porovnání je založeno na principech popsaných výše v odstavci *Porovnání řetězců* (opět totiž porovnáváme dvě množiny s názvy souborů). Poskytované výstupy jsou totožné.

Při interpretaci výsledků je třeba brát v potaz, že jsou uvedeny pouze ty hlavičkové soubory, z nichž byla použita nějaká funkce, struktura nebo datový typ (a to i pokud byly použity nepřímo, tedy načteny skrze jiný hlavičkový soubor). Pomocí tohoto přístupu lze eliminovat odlišnosti v chování na operačních systémech Windows a Linux. Taktéž se

vyhneme problémům způsobeným vkládáním nadbytečných direktiv `#include` programátorem. Chybí také ty hlavičkové soubory, kdy prvek z nich použitý byl odstraněn preprocesorem a tedy chybí v abstraktním syntaktickém stromu.

Pokud ukládáme pouze opravdu použité hlavičkové soubory, tak má analýza mnohem větší vypovídací hodnotu. A to zejména pro testování detekce známých knihovních funkcí pomocí jejich signatur, kterou zpětný překladač provádí.

Porovnání strukturovaných typů Dochází k porovnání dvou strukturovaných typů – *struktur* a *typu union*. Typ union v době vytváření nebyl zpětným překladačem podporován, neboť jeho analýza je složitá, dílčí výsledek tady nebyl uvažován (s výjimkou, kdy se typ union vyskytl). Vůbec zahrnuto nebylo porovnání výčtů (anglicky *enumerations*), které zpětný překladač taktéž nepodporoval a navíc je informace o jejich použití při překladu často ztracena.

Porovnání struktur, podobně jako u funkcí, zahrnuje promítání těch z prvního a druhého souboru na sebe. V případě, že jsou k dispozici symboly, je toto promítání provedeno skrze názvy. V opačném případě jsou porovnány počty členů a jejich typy – obyčejné i bázové. Na základě vypočtené podobnosti jsou struktury promítnuty.

Součástí výstupů je celkový výsledek analýzy, výsledek porovnání struktur a typu union, a zobrazení jednotlivých strukturovaných typů na sebe včetně informace o procentuální shodě typů členů (normálních i bázových).

Porovnání globálních proměnných a inicializátorů Vzhledem k tomu, že zpětný překladač vkládá velké množství proměnných číselných typů, je nevhodné je porovnávat. Tato analýza provádí pouze porovnání strukturovaných typů definovaných v hlavičkových souborech nebo uživatelem (včetně polí, a to i těch číselných).

V každém souboru je spočítán počet výskytů globální proměnné s tímto typem. Výskyty jsou poté porovnány a součástí výstupu je celková podobnost a také informace o tom, kolikrát se který konkrétní typ vyskytoval v jednotlivých souborech.

Mimo to dochází také k porovnání inicializátorů, které se v souboru vyskytly. Porovnávány jsou jednotlivé prvky inicializátorů a to včetně jejich pořadí. Z abstraktního syntaktického stromu můžeme získat čtyři typy položek – číslo s pevnou řádovou čárkou, číslo s plovoucí řádovou čárkou, řetězec a znak. U čísel nedokážeme rozhodnout o jejich přesném typu a může tedy dojít ke špatnému rozeznání hodnoty, avšak toto rozeznání je totožné pro oba soubory a pro porovnání podobnosti to nepředstavuje problém (a součástí výstupů tato informace není).

Kapitola 7

Implementace

Následující kapitola popisuje implementaci navrženého řešení nástroje *csim*, knihovny *csiml* a problémy, které se během ní vyskytly. Tento proces bude opět popsán na třídách, které řešení obsahuje.

V závislosti na výběru knihovny *LibTooling* bylo možno k implementaci zvolit pouze jazyk *C++*. Vzhledem k požadavkům na vytvoření kvalitního a moderního kódu bylo nutno seznamit se se standardy *C++11* [1] a *C++14* [2]. Na splnění tohoto kritéria bylo navíc důsledně dohlíženo v rámci revizí zdrojových kódů každého *commitu*[1]. V rámci vývoje této práce bylo využíváno verzovacího systému *git* [10].

Pro správnou funkčnost nástroje je doporučeno používat knihovny překladače *Clang* verze `3.6.2`. Již dříve totiž bylo zmíněno, že rozhraní *LibTooling* se může napříč verzemi měnit. Během vývoje práce se několikrát stalo, že nalezené řešení problému nebylo možno v této verzi využít (buď pocházelo ze starší nebo novější verze).

Používané nástroje Mimo již zmíněného používání knihoven překladače Clang a verzovacího systému git byly využívány i další nástroje. Všechny zmíněné nástroje jsou *open-source*.

Doxygen [8] je určen ke generování dokumentace, to je prováděno z komentářů psaných programátorem. Ty musí dodržovat určitý styl a atributy, což zdrojové kódy tohoto nástroje splňují. Výsledná dokumentace je přiložena na DVD.

CMake [6] je multiplatformní sada nástrojů pro překlad a testování softwaru. V této práci byla využívána právě pro překlad vytvářených zdrojových kódů. Výhodou CMake oproti klasickému *Makefile* je robustnost řešení.

7.1 Nástroj csim

Samotný nástroj se skládá z jednoho zdrojového souboru `csim.cpp`. Ten zpracovává vstupní parametry a spouští vyžádané analýzy, jejichž výsledky sesbírá a váženým průměrem vypočte celkový výsledek porovnání. Poté na standardní výstup vypíše detaily analýz v textovém formátu nebo formátu JSON, určeném pro zpracování dalšími nástroji spouštějícími testy a poskytujícími výstupy prostřednictvím webových služeb.

K vytváření výstupu ve formátu JSON byla využita knihovna `JsonCpp` [14], která nabízí intuitivní vkládání dat a také sama řeší indentaci při jejich výpisu.

[1] Úprava kódu nahraná na server prostřednictvím verzovacího systému

7.2 Knihovna csiml

Implementaci knihovny csiml bude nejvhodnější popsat na základě tříd (viz obrázek 6.1 v kapitole 6), které obsahuje. Ty vhodně reflektují řešený problém a jeho dekompozici.

Mimo dále popsaných tříd řešení obsahuje také soubory `csiml_typedef.cpp/h`, definující pomocné funkce, struktury a výčty (`enum class`) používané knihovnou.

ASTParser

Jak již bylo zmíněno v návrhu, jedná se o provedení *front-endových* akcí překladače. Je nutno nastavit cílovou architekturu, nastavit *stream* pro výpis chybových hlášek, zvolit standard jazyka, který je překládán, a definovat cesty k hlavičkovým souborům.

Právě hlavičkové soubory představovali při řešení výrazný problém. Vyhledávání hlavičkových souborů nelze provádět automaticky, jako to dělá překladač, a ani by to nebylo žádoucí, neboť jedním z požadavků bylo stejné chování jak na operačním systému Windows, tak na systémech založených na UNIX. Prostřednictvím parametrů je třeba předat adresáře, v nichž mají být vyhledány. Právě předání vhodných a úplných cest je kritické pro správné porovnání. Abstraktní syntaktický strom je totiž vytvářen překladačem a v případě, že dojde k chybě, nemusí být určitý podstrom vytvořen a dochází ke ztrátě informací. Nástroj navíc musí počítat s tím, že bude analyzovat programy určené pro Linux na operačním systému Windows a naopak. To souvisí primárně s hlavičkovými soubory, ale i dalšími problémy, jako třeba rozeznání datového typu `FILE`, neboť struktura, která jej reprezentuje, je platformně specifická.

Jelikož nástroj umožňuje také analýzu řetězcových literálů, zejména pro testování, je nutno pro ně vytvořit virtuální soubory obsahující tento řetězec. K tomu rozhraní knihoven Clang nabízí třídu `FileManager` s metodou `getVirtualFile()`.

ASTConsumer

Úkolem této třídy je poté, co je vyvolána během „překladu“, zahájit a řídit procházení abstraktního syntaktického stromu od kořenového uzlu pomocí třídy `ASTVisitor`.

ASTVisitor

Základem této třídy, která dědí od třídy `clang::RecursiveASTVisitor` knihovny LibTooling, jsou dvě virtuální funkce `VisitStmt()` a `VisitDecl()` (využita je také `VisitType()`, která však slouží pouze pro získání použitých hlavičkových souborů), které jsou vyvolány, pokud překladač narazí na výraz, respektive deklaraci.

Na základě dynamických přetypování nabízených knihovnou LLVM je poté rozhodnuto, které podtřídy je uzel instancí a zavolána odpovídající metoda třídy `FileInfo`, která z něj získá informace požadované analýzami.

Součástí této třídy je také metoda `isInParsedFile()`, která rozhoduje, zda daný uzel leží ve vstupním souboru nebo jím použitých hlavičkových souborech. Pokud je vstupem soubor, pak se pomocí rozhraní LibTooling dotážeme na název souboru, pokud je vstupem řetězec, pak stačí pouze ověřit, že funkce nepochází z reálného souboru, ale z virtuálního. Toto rozhodnutí je podstatné, neboť je zbytečné analyzovat a ukládat informace z použitých hlavičkových souborů, jejichž obsah je u obou porovnávaných souborů totožný. Pokud detekujeme použití prvku z hlavičkového souboru, je také nutno jméno souboru uložit.

FileInfo

Třída `FileInfo` je jedním z klíčových prvků řešení. Probíhá v ní načítání informací z abstraktního syntaktického stromu do interních struktur a normalizace ukládaných hodnot. Ukládané informace byly zmíněny při návrhu (viz sekce 6.6).

Podle volané metody jsou provedeny náležité úkony nutné k extrakci informací o daném uzlu z jeho deklarace, a většinou je také nutno normalizovat datové typy vzhledem k důvodům zmíněným dříve. O tuto normalizaci se stará metoda `getAsString()`, která datový typ uzlu vrací v podobě řetězce definujícího interní reprezentaci typu. Tyto typy jsou ukládány ve dvou úrovních, buďto jako kompletní typ nebo jako bázový typ, který dále nerozlišuje podtypy typů s plovoucí a pevnou řádovou čárkou a taktéž jsou zanedbány ukazatele a velikosti polí. Při získávání typů struktur, bylo nutno zavést maximální hloubku (defaultně 2). Pokud totiž obsahují samy sebe nebo cyklickou závislost na jiné struktury, došlo by k zacyklení. Typ jakožto řetězec lze také získat pro ukazatele na funkci i pro pole (včetně jejich velikosti).

V rámci této třídy jsou také prováděny podpůrné akce pro analýzy, mezi které patří například generování signatur funkcí. Vzhledem k tomu, že struktura pro uložení funkcí obsahuje i ukazatele na volané funkce, je podstatné, aby v destruktoru třídy došlo ke smazání těchto ukazatelů, neboť se mezi nimi může vyskytnout cyklická závislost. Ta způsobí, že tzv. *smart pointers* (chytré ukazatele, které sledují počet referencí na sebe sama a v případě, že dosáhne počtu nula, dojde k volání destruktoru), používané v implementaci, neuvolní jimi alokovanou paměť. To platí i pro další části řešení.

Similarity

Bázová třída `Similarity` definuje rozhraní, které je stejné pro všechny třídy provádějící analýzy. Každá taková třída musí obsahovat čtyři metody – spuštění porovnání – `compare()`, získání popisu analýzy – `getDescription()`, získání identifikátoru analýzy – `getID()` a získání váhy analýzy – `getWeight()`. Podstatná je i metoda `createAnalyses()`, která vytváří instance všech existujících analýz (ta se jako jediná mění při přidání nové analýzy).

Porovnání funkcí (SimilarityFunctions) Implementace analýzy je obsažena ve třídě `SimilarityFunctions`. Porovnání funkcí je stěžejní částí řešení a jejich analýza má také největší možnou váhu (kombinovanou s počtem porovnávaných funkcí). Detekce podobnosti dvou funkcí je prováděna celkem z osmi hledisek o různých vahách. Každé porovnání dvou funkcí obdrží 0–100 bodů. Porovnávané atributy jsou:

- návratový typ – porovnání návratového typu a bázového návratového typu,
- parametry – porovnání bázových typů a počtu parametrů,
- volání funkcí ze standardních hlavičkových souborů – na základě jejich názvu,
- volání funkcí z daného souboru – na základě jejich signatur,
- typy lokálních proměnných – jak úplné typy, tak bázové,
- řetězcové literály,
- cykly – pouze celkový počet, nelze rozlišovat typy neboť se velmi mění,
- podmínky – opět pouze počet (i tak může být značně zkresleno, proto má nižší váhu).

V případě, že jsou dostupné symboly (a nebylo přepínačem zvoleno, aby od nich nástroj upustil) je přirazení funkcí provedeno na základě jejich jména. O to se stará metoda `compareWithSymbols()`. V opačném případě je volána metoda `compareWithoutSymbols()`. Rozhodnutí o tom, zda jsou jména dostupná, provádí metoda `areSymbolsAvailable()`, hledající funkce s názvem odpovídajícím regulárnímu výrazu `(function|sub)_[0-9a-f]+` – tento název je funkcím přidělen zpětným překladačem a hexadecimální číslo označuje paměťovou adresu začátku funkce.

Obě metody pro porovnání využívají algoritmů třídy `FunctionMatching`. V případě, že nejsou k dispozici symbolická jména (nebo nebylo zvoleno, aby nedocházelo k detekci inliningu), je vytvořena reprezentace grafu volání (třída `CallGraph`), skládajícího se z uzlů (třída `Node`). Metoda s názvem `FunctionMatching::compareWithoutSymbols()` provede porovnání funkcí a vzájemné přiřazení uzlů obou grafů založené na jejich složitosti a poloze v grafu volání. Metoda `FunctionMatching::compareWithSymbols` provede to samé, ovšem na základě jmen. Poté je volána metoda `postProcessing()`, detekující inlining. Využívá metod `detectSimpleInlining()` a `detectAdvancedInlining()`. *Simple* se zde myslí takový případ, kdy inlinovaná funkce nevolá žádnou další, zatímco *Advanced* ano.

Zajímavou metodou je také `mergeFunctions()`, provádějící sjednocení funkcí – ta sjednocuje například řetězce či volání funkcí, ale některé atributy, jako parametry, bere pouze z jedné, hlavní, funkce.

Z finálních párů jsou poté sesbírány jejich bodové ohodnocení, které jsou zprůměrovány a uloženy jako výstup analýzy. Je také vypočtena podobnost grafu volání metodou `compareGraphs()` a celkový výsledek této analýzy.

Porovnání řetězců (SimilarityStrings) Porovnání řetězců je procesem, kdy je na množinách (kontejner `std::set`) řetězců proveden průnik a rozdíl (využíváno je funkcí spojených se standardními kontejnery jazyka C++).

Výstupem jsou shodné řetězce a řetězce unikátní pro každý soubor. Je také vypočtena procentuální shoda dána vzorcem 6.1.

Porovnání použitých hlavičkových souborů (SimilarityIncludes) Jak již bylo zmíněno, jedná se pouze o porovnání množin řetězců obsahujících názvy hlavičkových souborů. Proto je algoritmus takřka totožný s porovnáním řetězců. Akce specifické pro porovnání hlavičkových souborů jsou prováděny spíše ve třídě `ASTVisitor`, neboť zde musí dojít k filtrování hlavičkových souborů jen na ty použité. Zde probíhá pouze odstranění knihovny `stdint.h`.

Porovnání strukturovaných datových typů (SimilarityStructuredTypes) Porovnání strukturovaných datových typů – přesněji struktur a typu *union*, je do jisté míry podobné porovnání funkcí. Taktéž je nutno struktury na sebe přiřadit a k tomu je možno využít symbolických jmen. O jejich dostupnosti rozhoduje opět metoda `areSymbolsAvailable()` tentokrát používající regulární výraz `(struct|union)_[0-9]+`, kde číslo za podtržítkem označuje pořadí v souboru. Třída pak obsahuje metody `createMatchesWithSymbols()` a `createMatchesWithoutSymbols()`.

Porovnání globálních proměnných a inicializátorů (SimilarityVariables) Porovnání globálních proměnných a jejich inicializátorů provádí třída `SimilarityVariables`. Porovnání typů je založeno na porovnání počtu výskytů daných typů, ty jsou ukládány do standardního kontejneru `std::map`. Ukládány jsou všechny globální proměnné kromě těch s bázovým datovým typem `integral` a `float` (typy definované knihovnou csiml).

Inicializátory jsou porovnávány položku po položce, oproti původnímu plánu porovnávat signaturu v podobě řetězce, což je přesnější a toleruje chyby.

Result

Bázová třída `Result` reprezentuje rozhraní, které poskytuje výsledky analýz.

Povinné metody jsou tři – výpis výsledku v textovém formátu – `printText()`, získání výsledku ve formátu JSON – `getJSON()` a získání procentuálního výsledku analýzy – `getResult()`. Výstup ve formátu JSON je samozřejmě také vytvářen knihovnou `JsonCpp` a to pomocí asociačního pole.

Každá podtřída, která odpovídá jedné analýze (např. `ResultFunctions` koresponduje s `SimilarityFunctions` atd.), obsahuje atributy popisující její výsledky na dostatečně detailní úrovni. Ty jsou poté na vyžádání poskytnuty v odpovídajícím formátu. Textový formát je přímo vytištěn na zadaný *stream*, zatímco výstup ve formátu JSON vrací pouze svou část. Tyto části jsou sesbírány v hlavním souboru `csim.cpp` a poté vypsány, aby byla zachována validita formátu `JSON` a správná indentace.

Kapitola 8

Testování

Vzhledem k nasazení řešení do provozu je nutno jej náležitě otestovat. Jakékoliv změny provedené po nasazení by znehodnotily testy provedené dříve a tím by utrpěla detekce regresí. Je nutno ověřit nejen to, že nástrojem poskytované výsledky jsou správné, ale také to, že jsou poskytovány v rozumném čase a nedochází k problémům, jako jsou pády nebo neuvolňování paměti, což by by při nočních testech běžících řádově na desítkách tisíc vzorků mohlo způsobit velké problémy.

8.1 Návrh

Nejkomplikovanější částí pro testování bylo promítání funkcí, které bylo prováděno zejména na vzorcích malware, ty měly typicky desítky tisíc řádků kódu a stovky funkcí. Zde bylo možno řádně otestovat kvalitu vyvinutých algoritmů. Protože by ruční porovnání správnosti promítnutí bylo časově velmi náročné, bylo využito již existujícího nástroje *BinDiff*. Výstupy tohoto nástroje a nástroje csim byly následně porovnávány.

Aby bylo možno během vývoje odhalit *regrese* (chyby zanesené novými změnami do kódu, který dříve fungoval) a předvést správnost, tak byly napsány i sady automatizovaných testů. Ty testovaly jednotlivé komponenty, nástroj samotný a i přímo zpětný překladač. Testy byly velmi užitečné pro rychlý průběh testování změn.

Celkem byly používány čtyři typy testů:

- jednotkové testy,
- regresní testy,
- regresní testy využívající dekompilaci,
- noční testy.

I přes existenci automatizovaných testů probíhalo také rozsáhlé manuální testovaní různých okrajových případů s cílem odhalit chyby, a to na speciálně vytvářených souborech.

8.2 Jednotkové testy

Úkolem *jednotkových testů* (anglicky *unit tests*) je testovat konkrétní, menší, části kódu (funkce či třídy). Většinou je píše přímo vývojář a napomáhají implementaci.

V této práci je využíváno open-source frameworku *Google Test* [12], který je využíván i zbytkem nástrojů testovaného zpětného překladače. Již pro něj tedy existovala podpora, včetně skriptů pro generování pokrytí kódu testy.

Celkem řešení obsahuje více než 70 testů rozdělených do 11 kategorií, které testují vytváření abstraktního stromu, jednotlivé analýzy a jimi poskytované výstupy (zejména správnost procentuálního vyjádření podobnosti). Bylo pokryto přibližně 70% všech řádků kódu (některé řádky nejsou testovatelné jednotkovými testy a analýza podobnosti hlavičkových souborů lze provádět pouze v rámci regresních testů).

Jak vypadá kód jednotkových testů a následný výstup lze najít v ukázkách 8.1 a 8.2.

```
TEST_F(ASTParserTests, GlobalIntsCorrectlyRecognized) {
    auto fileInfo = parse(R"(int␣c;)");

    ASSERT_EQ(1, fileInfo->globalVars.size());
    EXPECT_EQ("int", fileInfo->globalVars[0]->type);
    EXPECT_EQ("c", fileInfo->globalVars[0]->name);
}
```

Ukázka 8.1: Ukázka jednotkového testu

```
[==========] Running 71 tests from 11 test cases.
[----------] Global test environment set-up.
...
[----------] 19 tests from ASTParserTests
...
[ RUN      ] ASTParserTests.GlobalIntsCorrectlyRecognized
[       OK ] ASTParserTests.GlobalIntsCorrectlyRecognized (0 ms)
...
[----------] 19 tests from ASTParserTests (5 ms total)
...
[----------] Global test environment tear-down
[==========] 71 tests from 11 test cases ran. (40 ms total)
[  PASSED  ] 71 tests.
```

Ukázka 8.2: Výstup testovacího frameworku Google Test

8.3 Regresní testy

Zřejmým nedostatkem jednotkových testů je neschopnost testovat program jakožto celek. Na testování spolupráce jednotlivých modulů se zaměřují regresní testy. Ty také, jak již název napovídá, slouží k detekci regresí.

V případě nástroje csim je testována extrakce všech informací v testovaných souborech, jejich normalizace a správnost všech výsledků jednotlivých analýz (zejména promítání funkcí či struktur a také detekce použitých hlavičkových souborů).

Testy jsou napsány v jazyce Python, v existujícím frameworku vyvinutém pro zpětný překladač Retargetable Decompiler.

Regresní testy nástroje csim Vstupem tohoto typu testů jsou přímo dva soubory, obsahující mnoho různých konstrukcí. Testy se skládají z více než 130 jednotlivých případů, kde dochází k ověření správnosti zpracování parametrů, načítání informací, promítání funkcí i procentuálního výsledku analýz. Testováno je především porovnání dvou obecných, ale i dekompilovaných, zdrojových kódů.

Regresní testy využívající dekompilaci Nástroj je samozřejmě testován i na reálných dekompilovaných vzorcích, které jsou také využívány při nočních testech. Vstupem je tedy pouze jeden soubor. Ten je přeložen (s různými nastaveními – architektura, formát, úroveň optimalizace, zachování symbolických jmen), zpracován zpětným překladačem a výsledek je porovnán s původním souborem. Jako vstup byly vybrány ty soubory, které zpětný překladač již zvládal rozeznávat správně.

Díky těmto testům, prováděným na různorodých vzorcích, je možno odhalit množství problémů. Vzhledem k jejich počtu (více než 680 jednotlivých testů) je možno ověřit i časovou náročnost, ta se však naplno projeví spíše v nočních testech (kde je tento test automatizován).

Jak vypadá kód těchto testů a následný výstup lze vidět v ukázkách 8.3 a 8.4.

```
from regression_tests import *

# Files decompiled using O0 option
class Test_O0(Test):
    settings = TestSettings(
        input= [..., 'printf.c', 'ackermann.c', 'fibo.c',
            'stack.c', 'strlen.c', ...],
        arch=ALL,
        format=ALL,
        compiler=ALL,
        compiler_opts='-O0',
        debug_info=True,
        strip=False,
        args='--check-results'
    )

    # ARM, PE, GCC is worse than other combinations
    @skip_for(arch='arm', format='pe', compiler='gcc')
    # http://xxx.vutbr.cz/yyy/issues/1590
    @skip_for(input='ackermann.c', arch='x86', format='pe')
    def test_c_files_are_sufficiently_similar(self):
        self.assertGreater(
                self.decomp.csim_output['overall_result'], 80)
```

Ukázka 8.3: Ukázka regresních testů využívajících dekompilaci

```
Running tests in /regression-tests/csim
for commit 32614fd3
...
csim.integration.Test_O0 (ack.c -a mips -f elf -c clang -C -O0 -g
        --check-results)                    [  OK  ]  (1.15s)
csim.integration.Test_O0 (ack.c -a powerpc -f elf -c clang -C -O0 -g
        --check-results)                    [  OK  ]  (1.18s)
...
SUCCESS (686/686)
```

Ukázka 8.4: Výstup regresních testů

8.4 Noční testy

Noční testy jsou spouštěny mimo pracovní dobu, tedy v době kdy je výpočetní síla nevyužita. Na desítkách tisíc vzorků testují jak jednotlivé nástroje využívané zpětným překladačem, tak jej samotný. Účelem testů je odhalit možné regrese zavedené během posledních změn nebo pokrok, který byl daný den učiněn.

Právě k tomu slouží i nástroj csim, ten byl již před dokončením práce úspěšně integrován do těchto testů. Jak výstup takových testů vypadá lze vidět na obrázku 8.1, který je popsán níže.

x86/pe	gcc/O0	gcc/O1	gcc/O2	gcc/O3	clang/O0	clang/O1	clang/O2	clang/O3
IR syntax result (%)	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
C generation result (%)	100.0	100.0	100.0	99.8	100.0	100.0	100.0	99.1
C syntax result (%)	92.3	94.3	93.9	91.5	97.3	95.3	94.6	93.7
[-] C similarity result (%)	53.2	51.8	50.2	48.7	52.8	49.6	48.5	47.7
- Functions (%)	57.4	56.5	55.0	52.1	58.7	56.8	54.4	52.2
- Functions: call graph (%)	59.8	59.4	54.9	50.8	59.7	57.7	49.4	48.7
- Functions: calls (%)	94.8	94.9	91.6	90.3	93.0	92.3	87.0	86.6
- Functions: library calls (%)	59.5	57.3	50.7	48.7	64.9	56.6	50.3	50.5
- Functions: local variables (%)	57.2	54.3	52.3	51.4	53.3	51.0	43.4	43.3
- Functions: parameters (%)	79.4	79.7	78.4	78.1	77.0	75.8	75.9	75.5
- Functions: return types (%)	93.6	93.9	96.1	97.2	92.0	92.5	97.1	97.3
- Functions: conditions (%)	64.9	63.3	63.8	60.7	66.4	63.2	60.1	54.0
- Functions: loops (%)	95.3	92.8	87.9	83.0	93.7	93.2	83.1	82.4
- Functions: strings (%)	80.5	74.4	68.5	66.6	66.2	61.4	54.9	55.8
- Global variables (%)	41.5	41.3	41.0	40.9	39.0	41.5	46.4	46.6
- Global variables: init (%)	82.6	82.7	82.4	82.4	81.8	82.1	81.5	81.4
- Global variables: var (%)	42.6	42.0	41.8	41.8	39.2	42.6	47.9	48.3
- Includes (%)	47.2	47.4	47.8	47.2	47.4	47.2	48.4	48.6
- Strings (%)	62.1	60.2	59.1	59.1	58.9	55.0	56.2	57.3
- Structured types (%)	74.6	74.8	74.0	73.7	72.8	72.5	73.3	73.7

Obrázek 8.1: Ukázkový výstup nočních testů po vylepšení (zelená) detekce inliningu

Testovanou kombinací zde byla architektura *x86* a formát souboru *pe* (popsány dále). První tři řádky jsou výstupy jiných interních nástrojů společnosti AVG, zbylé jsou již výstupy nástroje csim. Popis toho, co představují, je uveden v následujícím seznamu.

- C Similarity Result – celková podobnost v procentech (vážený průměr dílčích výsledků analýz)
- Functions – celkový výsledek analýzy podobnosti funkcí a grafu volání (opět váhováno)
  - call graph – podobnost grafu volání
  - calls – podobnost volání uživatelských funkcí (dle signatur)
  - library calls – podobnost volání knihovních funkcí (dle názvu)
  - local variables – podobnost typů (i bázových) lokálních proměnných (nikoliv počty)
  - parameters – podobnost parametrů (bázové typy i počty)
  - return types – podobnost návratových typů (jak úplné, tak bázové)
  - conditions – podobnost podmínek (pouze počty)
  - loops – podobnost cyklů (pouze počty, nerozlišují se typy)
  - strings – podobnost pouze řetězců ve funkcích, nikoliv globálních
- Global variables – celkový výsledek podobnosti globálních proměnných a inicializátorů
  - init – podobnost inicializátorů (hodnoty a jejich pořadí)
  - var – podobnost globálních proměnných (počty výskytů bázových typů)
- Includes – podobnost hlavičkových souborů (pouze těch opravdu použitých)
- Strings – podobnost řetězců (lokálních i globálních, včetně formátovacích značek)
- Structured types – analýza podobnosti strukturovaných typů (struktur a typu union)

Kapitola 9

Vytvořený nástroj

V této kapitole je popsáno a zhodnoceno finální řešení a jeho testování. Ukázkové výstupy nástroje lze nalézt v příloze.

9.1 Parametry a spouštění

První dva parametry jsou povinné a představují vstupní soubory nebo řetězce (uvozené parametrem `-s`) obsahující analyzovaný zdrojový kód. Další parametry jsou volitelné a jejich popis lze nalézt v tabulce 9.1.

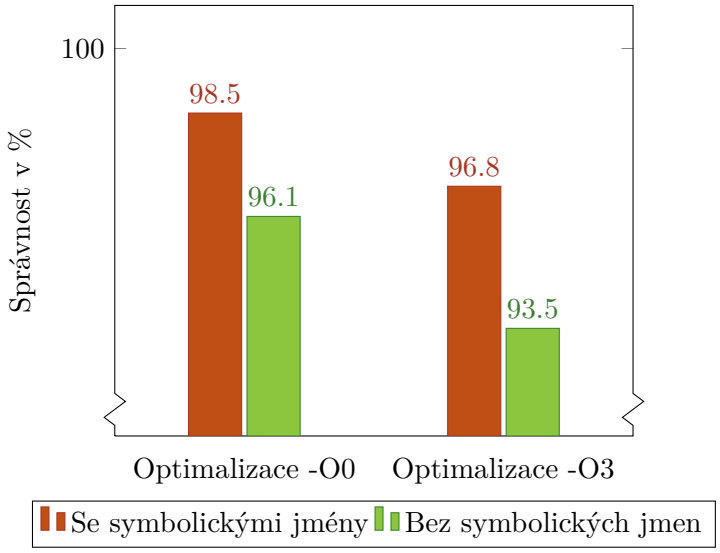
Tabulka 9.1: Volitelné přepínače

Přepínač	**Popis**
`--help (-h)`	Vypíše nápovědu, dynamicky vytvářeno pro usnadnění rozšiřování
`--verbose (-v)`	Zapíná detailní výpisy během provádění, včetně chyb a vnitřních struktur
`--silent`	Potlačí veškeré výpisy, užitečné při strojovém zpracování
`--string (-s)`	Uváděno před řetězec, označuje přímý textový vstup, vhodné zejména pro testování
`--include (-I)`	1–N cest k hlavičkovým souborům testovaného zdrojového souboru (standardní načítány automaticky)
`--analyses (-A)`	1–N vybraných analýz, názvy uvedeny v nápovědě, defaultně spouštěny všechny
`--format (-f)`	Výběr formátu výstupu – `JSON` nebo `text`
`--strip-symbols`	Při porovnávání nejsou využity symbolická jména
`--decompiled`	Aktivuje využívání optimalizací pro porovnání dvou dekompilovaných souborů
`--no-inlining`	Vypne detekci inliningu

9.2 Zhodnocení

Řešení bylo po odstranění všech chyb a náležitém testování nasazeno do provozu jakožto součást nočních testů zpětného překladače. Druhým možným využitím byla detekce malware, i dosažení tohoto cíle bylo podrobeno testům (viz sekce 9.2.4).

Jedním z požadavků byly také kvalitní a dobře komentované zdrojové kódy. Jak již bylo zmíněno, na splnění tohoto kritéria bylo v průběhu práce důsledně dohlíženo pravidelnými inspekcemi zdrojových kódů (tzv. *code review*). Všechny takto odhalené nedostatky byly průběžně odstraňovány.



Obrázek 9.1: Správnost promítání funkcí pro kombinaci GCC/x86/pe.

Klíčovým prvkem řešení bylo promítání funkcí, to bylo podrobeno zvláště důkladným testům, dosažené výsledky lze nalézt na obrázku 9.1. Správnost byla zjišťována na 15 náhodně vybraných souborech z nočních testů. Nedílnou součástí byla i detekce inliningu, jako důkaz její kvality slouží také obrázek 8.1, kde lze vidět, že procentuální podobnost se se zvyšující úrovní optimalizací snižuje, ovšem ne o tolik. Důvodem je právě zavedení detekce inliningu (které se projevilo zeleným podbarvením, čím jasněji zelená, tím větší zlepšení). Před jejím zavedením se se stoupající úrovní optimalizací snižovala podobnost vždy o 2–5%.

Algoritmus i přes veškerou snahu stále není ideální. Jedním z jeho velkých problémů je nesprávné přiřazení funkcí, které si však jsou velmi podobné a mají totožnou pozici v grafu volání (a mají s nějakou funkcí větší podobnost, než s tou, která jim opravdu odpovídá – důsledek zpětného překladu).

Promítání na základě symbolických jmen také není stoprocentní, neboť se zde uvažuje i detekce inliningu. Algoritmus jej nezvládl rozeznat v jednom případě. Také se stalo, že do dekompilovaného souboru bylo přidáno více než 10 funkcí, které v původním souboru neexistovali. Z nich některé měli název `function_ADRESA`, což spustilo porovnání bez symbolických jmen (kde dochází k problémům popsaným výše). Jednou se také stalo, že zpětný překladač změnil název funkce (což by se stávat nemělo a analýza založená na porovnávání symbolických jmen toto neuvažuje).

9.2.1 Porovnání s csimilarityCMP

V porovnání s původním nástrojem nový obstál velmi dobře. Nejenže poskytuje přehlednější výsledky (které jsou zároveň lehce integrovatelné do výsledků nočních testů díky dostupnosti výstupu ve formátu JSON), ale poskytuje je také mnohem rychleji. U jednodušších vzorků s velmi podobnými grafy volání jsou výsledky poskytovány v přibližně třetinovém čase. U složitějších vzorků je tato výhoda patrnější a nástroj csim je mnohanásobně rychlejší.

Přesné procentuální vyjádření neexistuje, vzhledem k tomu, že původní nástroj dlouhé soubory ani nezvládá zpracovat. Často je nutno běh nástroje csimilarityCMP časově omezit, získané výsledky pak stále nejsou natolik uspokojivé, jako v případě nového nástroje (způsob analýzy správnosti bude popsán dále).

I když procentuální vyjádření zrychlení nelze získat, lze porovnat celkové časy běhu nočních testů, které byly prováděny na totožných vzorcích. V případě využití nástroje csim trvaly noční testy o 20 minut méně (přesněji 6h 38m oproti 6h 57m) než při využití původního řešení. Tento údaj však vzhledem k paralelnímu běhu není až tak podstatný. Mnohem důležitější jsou konkrétní časy běhu nástrojů v jednotlivých sadách testů (sdružují se dle typu architektury a formátu). V tomto případě je nový nástroj 2–5x rychlejší v závislosti na uvažované sadě, jak je ukázáno v tabulce 9.2. Původní řešení navíc bylo nutno v 71 případech ukončit (v případě nového nástroje k tomu nikdy nedošlo), jelikož doba výpočtu byla delší než 300 sekund (a to sady neobsahují příliš dlouhé testovací vzorky).

Tabulka 9.2: Srovnání doby běhu

Kombinace	**csimilarityCMP**	**csim**
arm/elf	33m 13s	16m 36s
arm/pe	59m 20s	23m 46s
x86/pe	33m 13s	17m 43s
powerpc/elf	40m 09s	16m 38s
mips/ihex	1h 27m 21s	16m 38s

Rozdíl v procentuální podobnosti, získané nočními testy při použití obou nástrojů (prováděno na stejné testovací množině), je znatelný, avšak těžko porovnatelný. Některé výsledky jsou vyšší, další však nižší (a ne vždy testují úplně to samé). Původní nástroj navíc nenabízí žádný celkový výsledek.

Nový nástroj nabízí porovnání dle více kritérií, které navíc sdružuje do analýz a nabízí smysluplné celkové výsledky každé z nich (a také výsledek celého procesu porovnání). Všechny tyto celkové výsledky jsou váhovány dle důležitosti a počtu porovnávaných prvků, aby bylo zajištěno, že získaná podobnost bude opravdu objektivní.

Csim také navíc nabízí detekci inliningu. To výrazně zvyšuje přesnost porovnání zdrojových kódů přeložených s vyšší úrovní optimalizace (zde je inlining častější).

Správnost všech výstupních informací byla testována jak regresními testy (zejména z pohledu obsahu a celkové podobnosti), tak jednotkovými testy (zejména na procentuální podobnost jednotlivých kritérií či analýz). Obdobné testy původní řešení postrádalo.

9.2.2 Regresní testy

Regresní testy dozajista splnily svůj účel, neboť často dokázaly odhalit problémy, které byly způsobeny změnami nástroje csim. Byly také velmi podstatné pro ověření, že se nástroj chová stejně na systémech Windows i UNIX.

Mimo to však dokázaly odhalit i regrese samotného zpětného překladače a také jeho dosud neznámé problémy. Ty byly zjištěny zejména při prvotním výběru testovací množiny souborů a analýze správnosti porovnání (tyto problémy zahrnovaly chybějící těla funkcí, nadbytečné funkce, špatné přiřazení strukturovaného typu parametrům, chybějící řetězce a další). Nalezené problémy byly reportovány vývojářům.

9.2.3 Noční testy

Testování v rámci nočních testů (viz sekce 8.4) naposledy probíhalo na 81 137 vzorcích a trvalo 6 hodin a 43 minut (dříve uvedená hodnota byla nižší, neboť v té době bylo testováno pouze 77 225 vzorků). Z toho byl nástroj spouštěn v 63 270 případech (některé testy vychází přímo z binárních souborů a tudíž není k dispozici původní kód), v žádném

případě nedošlo k pádu nebo násilnému ukončení z důvodu zacyklení nebo přílišné časové náročnosti (nástroj byl totiž testován i na vzorcích malware s desítkami tisíc řádků).

V závislosti na kombinaci architektury a formátu, nástroj csim zabírá přibližně 6–18% celkového času testování a 3–12% spotřebované paměti. Z tohoto času i paměti více než polovinu spotřebuje vytváření abstraktního syntaktického stromu *front-endovými* nástroji překladače Clang, to je patrné zejména u delších souborů.

V tabulkách 9.3 a 9.4 lze nalézt vybrané procentuální výsledky nástroje, vyjadřující celkovou podobnost (`C Similarity Result`) originálního a dekompilovaného souboru v závislosti na překladači, optimalizační úrovni, architektuře a typu souboru. V odstavcích níže lze také nalézt přehled zkratek z těchto tabulek, které představují testovanou architekturu a formát spustitelného souboru. Se stoupající optimalizační úrovní by měla zpravidla podobnost klesat, neboť optimalizace mají velký vliv na množství informací uchovaných o původním zdrojovém kódu (může se však stát, že bude i o něco málo vyšší, neboť v případě většího množství inlinovaných funkcí je porovnáváno méně jejich signatur, ve kterých se typy často liší).

Tabulka 9.3: Podobnost pro překladač GCC

Architektura	**Formát**	**GCC O0**	**GCC O1**	**GCC 02**	**GCC O3**
x86	pe	53,2%	51,8%	50,2%	48,7%
x86	elf	43,5%	42,4%	42,0%	41,0%
arm	pe	45,4%	41,5%	40,0%	38,7%
arm	elf	44,1%	43,1%	40,5%	39,9%
powerpc	elf	51,0%	49,8%	48,8%	47,2%

Tabulka 9.4: Podobnost pro překladač Clang

Architektura	**Formát**	**Clang O0**	**Clang O1**	**Clang O2**	**Clang O3**
x86	pe	52,8%	49,6%	48,5%	47,7%
x86	elf	45,4%	42,9%	41,1%	40,8%
arm	elf	44,1%	42,2%	39,6%	39,8%

Architektury *x86* je skupina instrukčních sad se zpětnou kompabilitou založená na procesorech Intel. Zpětný překladač dokáže dekompilovat jeho 32 bitovou variantu (přesněji zvanou *IA-32*), která vznikla v roce 1985. V současnosti se jedná o nejpoužívanější architekturu, která je dominantní na osobních počítačích (dnes však převažuje její 64 bitová verze, kterou zpětný překladač zatím nepodporuje). *ARM* (anglicky *Advanced RISC Machine*), je architektura s redukovanou instrukční sadou (tzv. *RISC*, opakem je obsáhlejší instrukční sada – *CISC*, používaná architekturou *x86*). *PowerPC* je taktéž *RISC* architektura používaná zejména vestavěnými systémy nebo vysoce výkonnými procesory.

Formáty spustitelných souborů *PE* (anglicky *Portable Executable*) je formát používaný operačním systémem Windows pro spustitelné soubory nebo DLL knihovny. *ELF* (anglicky *Executable and Linkable Format*) je formát s obdobným využitím, avšak používaný systémy UNIX a dalšími na nich založených.

9.2.4 Detekce malware

Součástí bylo i ověření možného přínosu nástroje csim pro detekci malware. Jak již bylo zmíněno, viry se postupem času transformují za účelem uniknout detekci antivirovými programy – tento jev nazýváme *metamorfismus*.

Společnost AVG Technologies s.r.o. poskytla několik vzorků různých rodin virů za delší časové období. Přesněji se jednalo o viry *Zeus*, *CryptoWall*, *Locky* a *Teslacrypt* určené pro operační systém Windows.

Zeus je zástupcem malware typu trojský kůň. Používá se i pro získání citlivých údajů založeném na detekci stisknutých kláves (tzv. *keylogger*), ale může také šířit jiné viry.

Zbylé tři rodiny mají společný základ, šifrují soubory na disku, a za peníze (*bitcoiny* či předplacené karty) poté prodávají postiženým uživatelům dešifrovací klíče. Tomuto typu virů se říká *ransomware*. Jejich největší hrozba je právě v síle použitého šifrování. I poté, co uživatel malware úspěšně odstraní ze systému, soubory stále zůstávají zašifrované. Šance uživatelů na získání původních souborů bez zaplacení jsou minimální, i podle FBI [33] jsou šifrovací algoritmy natolik účinné, že většinou není jiné cesty než tvůrcům zaplatit (ti tímto způsobem vydělávají i několik miliónů dolarů ročně).

> „Ransomware je velmi účinný. Abych byl upřímný, většinou lidem radíme, aby tvůrcům prostě zaplatili.“ [1] [33] (*Joseph Bonavolonta*)

Z poskytnutých vzorků bylo vybráno vždy několik, které byly porovnány. Jak se jejich podobnost mění v čase lze vidět na několika příkladech v tabulce 9.5. Pokud byly vzorky zachyceny s rozdílem několika málo dní, pak jejich podobnost dosahuje i více než 95%. Pokud však porovnáváme vzorky, mezi jejichž zachycením uplynula delší časová perioda, tak se podobnost rychle snižuje. Typicky však neklesne pod 50%, neboť jádro škodlivého softwaru zůstává stále velmi podobné. Veškeré výsledky nad tuto hranici lze tedy vždy považovat za minimálně vysoce podezřelé.

Tabulka 9.5: Měnící se podobnost malware v čase

Vzorek A	**Vzorek B**	**Podobnost**
CryptoWall (29. 2. 2016)	CryptoWall (18. 1. 2016)	51,05%
CryptoWall (8. 2. 2016)	CryptoWall (18. 1. 2016)	83,83%
Locky (1. 3. 2016)	Locky (16. 2. 2016)	74,40%
Locky (1. 3. 2016)	Locky (25. 2. 2016)	82,95%
Locky (18. 2. 2016)	Locky (16. 2. 2016)	95,11%
Zeus (25. 2. 2016)	Zeus (15. 2. 2016)	60,42%
Teslacrypt (2. 3. 2016)	Teslacrypt (14. 1. 2016)	60,46%
Teslacrypt (2. 3. 2016)	Teslacrypt (1. 2. 2016)	62,41%
Teslacrypt (2. 3. 2016)	Teslacrypt (15. 2. 2016)	67,98%

Pro ilustraci lze v tabulce 9.6 vidět porovnání rodin virů mezi sebou – zde by podobnost neměla dosahovat příliš vysokých čísel (výsledek přibližně do 20% je přípustný, neboť nějakou podobnost v souborech o desítkách tisíc řádků lze vždy vypozorovat).

Tabulka 9.6: Měnící se podobnost malware v čase

Vzorek A	**Vzorek B**	**Podobnost**
Zeus (25. 2. 2016)	Locky (16. 2. 2016)	14.71%
Teslacrypt (2. 3. 2016)	CryptoWall (8. 2. 2016)	8,35%
Zeus (15. 2. 2016)	CryptoWall (29. 2. 2016)	14,84%
Zeus (25. 2. 2016)	pic32mx-gcc-4.5.2 (překladač)	10,75%

[1] Joseph Bonavolonta je agentem FBI řídící její *CYBER and Counterintelligence Program* a k tomuto prohlášení došlo na na *Cyber Security Summit 2015*. Citát je volně přeložen z angličtiny.

9.2.5 BinDiff

BinDiff [19] je nástroj společnosti *Google*, který porovnává dva soubory na binární úrovni. Jedná se o plugin populárního disassembleru *IDA Pro* [13].

Pro zhodnocení kvality promítání funkcí na své protějšky byl tento nástroj zvolen jako referenční. Zpočátku vyvinuté metody, založené hlavně na porovnání podobnosti, dosahovaly zhruba 70% shody s výstupem tohoto nástroje. Podobnost dosažená poslední verzí algoritmu byla vyšší, ale stále nebyla stoprocentní. Avšak při manuálním srovnání přiřazení provedených oběma nástroji bylo zjištěno, že nástroj csim mnoho rozdílných porovnání zvládl lépe. Nástroj csim také navíc provádí detekci inliningu.

9.3 Budoucí vývoj

Vzhledem k tomu, že byl nástroj již od začátku psán s ohledem na budoucí rozšiřitelnost, tak tato činnost není příliš náročná. Pouze pokud by nová analýza vyžadovala informace, které aktuálně nejsou dostupné ve třídě `FileInfo`, bylo by nutno je získat za pomoci rozhraní LibTooling. Toto získávání je však do větší míry předpřipraveno.

Do budoucna by šlo do nástroje jistě zahrnout i další konstrukce, ke kterým bude vývoj zpětného překladače směřovat. Jedním z největších plánovaných rozšíření je poskytování dekompilovaného kódu v jazyce C++, což by znamenalo i zahrnutí objektově orientovaného přístupu (třídy atd.). Tyto informace je taktéž možno z knihovny LibTooling získat (třídy uvozené `CXX`) a jejich zahrnutí by již vyžadovalo rozsáhlejší, ale ne problematické, rozšíření.

Aktuální algoritmus porovnání funkcí dosahuje velmi dobrých výsledků, přesto by bylo možné jej dále vylepšovat. Zejména z ohledu zlepšení přesnosti promítání podobných funkcí se stejnou polohou v grafu volání.

V případě, že by nástroj bylo v plánu více využívat pro detekci malwaru, šly by určité algoritmy dále optimalizovat. Například stejné řetězce (pokud pomineme ty obecné, jako jsou například formátovací značky funkce `printf()`) dozajista vypovídají o podobnosti malware více, než jaká je jejich váha při testování kvality zpětného překladu (i proto při polymorfismu často dochází ke snahám o skrytí řetězců – například do polí obsahujících ASCII hodnoty znaků řetězce).

Kapitola 10

Závěr

V úvodu této práce byl čtenář seznámen se základy zpětného inženýrství, zejména problémy, kterým čelí zpětný překlad binárního souboru. Byl také seznámen s projektem *LLVM*, do kterého patří i překladač *Clang*, jehož knihovny jsou využívány v této práci.

Cílem této bakalářské práce bylo navrhnout, vyvinout, otestovat a nasadit do provozu nástroj *csim*, sloužící pro porovnání dvou zdrojových souborů, určený primárně pro testování kvality zpětného překladače *Retargetable Decompiler* společnosti *AVG Technologies s.r.o.* Nástroj tak musel být schopný nahradit stávající – tedy být obsáhlejší, přesnější a rychlejší.

Jak je patrné z předchozí kapitoly, tyto vytyčené cíle se podařilo splnit. Nástroj je až několikanásobně rychlejší než původní, a to i přesto že analyzuje soubory dle více kritérií. Tyto kritéria navíc sdružuje do analýz, u kterých nabízí i celkový výsledek a také výslednou podobnost souborů (což původní nástroj neuměl). Poskytuje také lépe čitelné výstupy ve více formátech (včetně formátu pro strojové zpracování). Jeho součástí jsou i sady automatizovaných testů, testující proces porovnání i jednotlivé poskytované výsledky.

Nástroj již v nočních testech, ve kterých se nachází jeho primární využití, nahradil původní. Provoz byl již zprvu bezproblémový. Bude tak nadále využíván při vývoji zpětného překladače.

Mimo to byl nástroj vyvíjen s ohledem na dobré programátorské praktiky a je tedy snadno rozšiřitelný, což byl také jeden z požadavků zadání. Nástroj tak může reagovat na průběh vývoje zpětného překladače přidáváním dalších testovacích kritérií (a změny lze snadno ověřit díky dostupnosti automatizovaných testů).

I když bylo primárním účelem nástroje testování, algoritmy byly psány obecněji a byly analyzovány také pro možný přínos při odhalování škodlivého softwaru, s čímž souvisí podpora detekce optimalizace zvané *inlining* nebo naopak detekce rozdělení funkcí. I v tomto ohledu nástroj ukázal jistý potenciál. Navíc tyto algoritmy měly také příznivý vliv na primární účel – testování.

Celkově byly splněny všechny požadavky kladené na výsledky této práce, ty byly navíc ještě rozšířeny. I přesto by bylo možné nástroj dále vylepšovat, zejména v oblasti detekce škodlivého softwaru a správnosti promítání podobných funkcí se stejnou polohou v grafu volání. Nástroj by také bylo možno rozšířit o analýzy konstrukcí specifických pro jazyk C++, jehož podpora zpětným překladačem je do budoucna plánována.

Literatura

[1] C++11 Language Extensions - general Features. [online]. [cit. 2016-04-27]. Dostupné z: https://isocpp.org/wiki/faq/cpp11-language.

[2] C++14 Language Extensions. [online]. [cit. 2016-04-27]. Dostupné z: https://isocpp.org/wiki/faq/cpp14-language.

[3] Choosing the Right Interface for Your Application. [online]. [cit. 2016-01-20]. Dostupné z: http://clang.llvm.org/docs/Tooling.html.

[4] Clang: A C language family frontend for LLVM. [online]. [cit. 2016-01-20]. Dostupné z: http://clang.llvm.org.

[5] Clang vs Other Open Source Compilers. [online]. [cit. 2016-01-20]. Dostupné z: http://clang.llvm.org/comparison.html.

[6] CMake. [online]. [cit. 2016-04-27]. Dostupné z: https://cmake.org/.

[7] DiffNow - compare files online. [online]. [cit. 2016-04-28]. Dostupné z: https://www.diffnow.com/.

[8] Doxygen: Main Page. [online]. [cit. 2016-04-27]. Dostupné z: http://www.doxygen.org/.

[9] GCC Optimization options. [online]. [cit. 2016-04-10]. Dostupné z: http://www.gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html.

[10] Git. [online]. [cit. 2016-04-27]. Dostupné z: https://git-scm.com/.

[11] Google Code Archive: py2llvm. [online]. [cit. 2016-05-10]. Dostupné z: https://code.google.com/archive/p/py2llvm.

[12] Google Test. [online]. [cit. 2016-04-27]. Dostupné z: https://github.com/google/googletest.

[13] IDA: About. [online]. [cit. 2016-04-27]. Dostupné z: https://www.hex-rays.com/products/ida/.

[14] jsoncpp: A C++ library for for interacting with JSON. [online]. [cit. 2016-04-29]. Dostupné z: https://github.com/open-source-parsers/jsoncpp.

[15] LLVM Clang 3.7 vs. GCC Compiler Benchmarks on Linux. [online]. [cit. 2016-04-27]. Dostupné z: http://www.phoronix.com/scan.php?page=Misc&item=clang-37-gcc52.

[16] The LLVM Compiler Infrastructure. [online]. [cit. 2016-01-20]. Dostupné z: http://llvm.org.

[17] Retargetable Decompiler. [online]. [cit. 2016-01-20]. Dostupné z: https://retdec.com.

[18] Themida: Advanced Windows Software Protection System. [online]. [cit. 2016-05-10]. Dostupné z: http://www.oreans.com/themida.php.

[19] zynamics.com - BinDiff. [online]. [cit. 2016-04-27]. Dostupné z: https://www.zynamics.com/bindiff.html.

[20] Aho, A. V.; Lam, M. S.; Sethi, R.; aj.: *Compilers - Principles, Techniques, and Tools*. Addison-Wesley, 1986, ISBN 0201101947, 303–310 s.

[21] Chikofsky, E.; Cross, J.: Reverse engineering and design recovery: a taxonomy. *IEEE Software*, ročník 7, č. 1, 1990: s. 13–17, ISSN 0740-7459.

[22] Cifuentes, C.; Gough, J. K.: Decompilation of Binary Programs. *Software: Practice and Experience*, ročník 25, č. 7, 1995: s. 811–829, ISSN 0038-0644.

[23] Eagle, C.: *The IDA Pro Book*. San Francisco: No Starch Press, druhé vydání, 2011, ISBN 1593272898, 127–166 s.

[24] Eilam, E.; Chikofsky, E.: *Reversing: secrets of reverse engineering*. Indianapolis: Wiley Publishing, Inc., 2005, ISBN 9780764574818.

[25] Gosling, J.; McGilton, H.: *The Java language Environment*. Sun Microsystems, 1996.

[26] Hamilton, J.; Danicic, S.: An Evaluation of Current Java Bytecode Decompilers. *University of London, UK*, 2008.

[27] Jaccard, P.: The Distribution of the Flora in the Alpine Zone. *New Phytologist*, ročník 11, 1912: s. 37–50, ISSN 1469-8137.

[28] Johnson, M.: Intermediate Representation. Sylabus, 2010.

[29] Křoustek, J.: Retargetable Analysis of Machine Code. 2015: str. 190.

[30] Křoustek, J.; Matula, P.; Zemek, P.; aj.: A Novel Approach to Online Retargetable Machine-Code Decompilation. *Journal of Network and Innovative Computing*, ročník 2, č. 1, 2014: s. 224–232.

[31] Li, X.; Loh, P. K. K.; Tan, F.: Mechanisms of Polymorphic and Metamorphic Viruses. *European Intelligence and Security Informatics Conference*, 2011.

[32] Meduna, A.: *Elements of Compiler Design*. Auerbach Publications, 2007, ISBN 1439815674.

[33] Roberts, P.: FBI's Advice on Ransomware? Just Pay the Ransom. *The Security Ledger*, 2015, [online]. [cit. 2016-04-27]. Dostupné z: https://securityledger.com/2015/10/fbis-advice-on-cryptolocker-just-pay-the-ransom/.

[34] Seiderer, P.: About GCC printf optimization. 2005, [online]. [cit. 2016-04-19]. Dostupné z: http://www.ciselant.de/projects/gcc_printf/gcc_printf.html.

Přílohy

Seznam příloh

Příloha A

Ukázkové výstupy

V sekci 4.2 je čtenáři pro představu poskytnuta ukázka zdrojového kódu před překladem (ukázka 4.1) a po zpětném překladu (ukázka 4.2). Zde jsou taktéž poskytnuty výstupy, které nástroj *csim* produkuje při porovnání těchto dvou souborů. A to jak v textovém formátu, tak ve formátu JSON.

A.1 Výstup v textovém formátu

```
=== Global variables similarity ===
Types recognized: 1
    Type: integral[]
        File 1: 1x
        File 2: 1x
Initializers in  first file: 1
Initializers in second file: 1
Variable    similarity: 100.00%
Initializer similarity: 100.00%
=== Total        100.00% ===

=== Function similarity ===
Functions in first  file: 2
Functions in second file: 2
Matches made: 2
    print -> function_401560 (100/100 points)
        Return type     : 15/15 points
        Parameters      : 15/15 points
        Calls           : 15/15 points
        Calls to library: 15/15 points
        Local variables : 10/10 points
        Strings         : 10/10 points
        Conditions (ifs): 10/10 points
        Loops           : 10/10 points
    main -> main (74/100 points)
        Return type     : 15/15 points
        Parameters      : 15/15 points
        Calls           : 15/15 points
        Calls to library: 5/15 points
        Local variables : 4/10 points
        Strings         : 0/10 points
        Conditions (ifs): 10/10 points
        Loops           : 10/10 points
```

```
Unmatched functions in first  file: 0
Unmatched functions in second file: 0
String                detection: 50.00%
Return type           detection: 100.00%
Parameter type        detection: 100.00%
Calls to lib funcs    detection: 66.67%
Calls to functions    detection: 100.00%
Local variables type  detection: 70.00%
Loop similarity       detection: 100.00%
Condition similarity  detection: 100.00%
Call graph similarity detection: 100.00%
===       Total        90.25%      ===

=== String similarity ===
Strings in first  file: 4
Strings in second file: 2
Strings in both files: 2
    Hello,
    world!
Unique strings in first file: 2
    \n
    %c
Unique strings in second file: 0
=== Total        50.00% ===

=== Includes similarity ===
Includes in first  file: 2
Includes in second file: 2
Includes in both files: 2
    stdio.h
    stdlib.h
Unique includes in first file: 0
Unique includes in second file: 0
=== Total        100.00% ===

=== Structured types similarity ===
Structs in first  file: 1
Structs in second file: 0
Struct matches made: 0
Unmatched structs in first  file: 1
    Delimiter
Unmatched structs in second file: 0
Unions in first  file: 0
Unions in second file: 0
Union matches made: 0
Unmatched unions in first  file: 0
Unmatched unions in second file: 0
Structure comparison result: 0.00%
Union   comparison   result: 100.00%
===       Total        0.00%      ===

=== Overall result 69.92% ===
```

A.2 Výstup ve formátu JSON


```
{
   "function" : {
      "call_graph_result" : 100,
      "call_result" : 100,
      "condition_result" : 100,
      "first_file_num" : 2,
      "lib_call_result" : 66.666666666666671,
      "local_var_result" : 70,
      "loop_result" : 100,
      "matches" : [
         {
            "call_points" : 15,
            "condition_points" : 10,
            "first_file_function" : "print",
            "lib_call_points" : 15,
            "local_points" : 10,
            "loop_points" : 10,
            "param_points" : 15,
            "points" : 100,
            "return_points" : 15,
            "second_file_function" : "function_401560",
            "string_points" : 10
         },
         {
            "call_points" : 15,
            "condition_points" : 10,
            "first_file_function" : "main",
            "lib_call_points" : 5,
            "local_points" : 4,
            "loop_points" : 10,
            "param_points" : 15,
            "points" : 74,
            "return_points" : 15,
            "second_file_function" : "main",
            "string_points" : 0
         }
      ],
      "matches_num" : 2,
      "param_result" : 100,
      "result" : 90.25,
      "return_result" : 100,
      "second_file_num" : 2,
      "string_result" : 50,
      "unmatched_first_file" : [],
      "unmatched_first_file_num" : 0,
      "unmatched_second_file" : [],
      "unmatched_second_file_num" : 0
   },
   "include" : {
      "common" : [ "stdio.h", "stdlib.h" ],
      "common_num" : 2,
      "first_file_num" : 2,
      "result" : 100,
      "second_file_num" : 2,
      "unique_first_file" : [],
      "unique_first_file_num" : 0,
      "unique_second_file" : [],
```

```
      "unique_second_file_num" : 0
   },
   "overall_result" : 69.920634920634924,
   "string" : {
      "common" : [ "Hello,", "world!" ],
      "common_num" : 2,
      "first_file_num" : 4,
      "result" : 50,
      "second_file_num" : 2,
      "unique_first_file" : [ "", "%c" ],
      "unique_first_file_num" : 2,
      "unique_second_file" : [],
      "unique_second_file_num" : 0
   },
   "struct" : {
      "result" : 0,
      "struct_first_file_num" : 1,
      "struct_matches" : [],
      "struct_matches_num" : 0,
      "struct_result" : 0,
      "struct_second_file_num" : 0,
      "struct_unmatched_first_file" : [ "Delimiter" ],
      "struct_unmatched_first_file_num" : 1,
      "struct_unmatched_second_file" : [],
      "struct_unmatched_second_file_num" : 0,
      "union_first_file_num" : 0,
      "union_matches" : [],
      "union_matches_num" : 0,
      "union_result" : 100,
      "union_second_file_num" : 0,
      "union_unmatched_first_file" : [],
      "union_unmatched_first_file_num" : 0,
      "union_unmatched_second_file" : [],
      "union_unmatched_second_file_num" : 0
   },
   "var" : {
      "detailed_result" : [
         {
            "first_file_count" : 1,
            "second_file_count" : 1,
            "type" : "integral[]"
         }
      ],
      "first_file_initializer_num" : 1,
      "initializer_result" : 100,
      "result" : 100,
      "second_file_initializer_num" : 1,
      "variable_result" : 100
   }
}
```

Příloha B

Obsah DVD

Přiložené DVD obsahuje jak soubory vytvořené autorem, tak soubory zpětného překladače (ten je přiložen v binární podobě, aby si jej uživatel mohl vyzkoušet).

Vytvořené autorem:

- `/manual.txt` – popis obsahu DVD a používání nástroje csim i zpětného překladače Retargetable Decompiler,
- `/Makefile` – `makefile` umožňující spouštění nástrojů a testů i překlad,
- `/thesis.pdf` – tato práce ve formátu PDF,
- `/documentation.html` – *Doxygen* dokumentace nástroje csim a knihovny csiml,
- `/thesis/` – zdrojové texty práce v LaTeXu,
- `/sample/` – zdrojové kódy příkladu použitého v této práci,
- `/source/decompiler/decdev/testing/csim/` – zdrojové kódy nástroje csim,
- `/source/decompiler/decdev/testing/csiml/` – zdrojové kódy knihovny csiml,
- `/source/decompiler/decdev/testing/csiml/tests/` – zdrojové kódy jednotkových testů,
- `/source/testsuite/regression-tests/csim/` – regresní testy využívající dekompilaci,
- `/source/testsuite/regression-tests/tools/csim/` – regresní testy nástroje.

Knihovny či soubory zpětného překladače:

- `/source/` – knihovny potřebné pro zpětný překladač i nástroj csim,
- `/source/decompiler/` – zpětný překladač, včetně nástroje csim,
- `/source/decompiler/decompiler/bin/` – přeložená verze zpětného překladače a nástroje csim (pro 64bit Linux),
- `/source/decompiler/scripts/regression_tests/` – skripty frameworku regresních testů společnosti AVG,
- `/source/decompiler/libs/` – knihovny používané nástrojem csim (včetně licencí).

Příloha C

Manuál

Aby uživatel mohl spouštět veškeré nástroje a testy jednoduše, je v kořenové složce DVD přiložen soubor `Makefile`. Pomocí `make TARGET` lze vyzkoušet jednotlivé části vyvinuté v rámci této práce, ale i samotný zpětný překladač *Retargetable Decompiler*. Podporované `TARGET` jsou:

- `regr` – spustí regresní testy využívající dekompilaci,
- `regr-tool` – spustí regresní testy nástroje,
- `unit` – spustí jednotkové testy,
- `build` – přeloží nástroj csim a knihovnu csiml,
- `thesis` – přeloží zdrojové soubory této práce v LaTeXu,
- `decompiler` – spustí zpětný překladač,
- `csim` – spustí nástroj csim.

V případě použití `decompiler` a `csim` je zapotřebí také předat argumenty. Toho docílíme nastavením proměnné `ARGS`. Příklady spuštění jsou uvedeny níže, oba nástroje podporují argument `-h` pro výpis nápovědy.

```
make decompiler ARGS="path/binary_file"
make csim       ARGS="path/file1.c path/file2.c"
```