FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

ZADÁNÍ DIPLOMOVÉ PRÁCE

Název:	Obfuskátor nad platformou LLVM
Student:	Bc. Roman Šíma
Vedoucí:	Ing. Tomáš Zahradnický, Ph.D.
Studijní program:	Informatika
Studijní obor:	Počítačová bezpečnost
Katedra:	Katedra počítačových systémů
Platnost zadání:	Do konce zimního semestru 2019/20

Pokyny pro vypracování

Seznamte se s taxonomií obfuskačních transformací [1] a frameworkem LLVM [2]. Navrhněte a implementujte obfuskační transformace jakožto LLVM moduly pro platformu Linux. Jako minimální sadu transformací implementujte převod funkcí do tabulkové formy a vkládání „mrtvého" nebo nepodstatného kódu, případně i další obfuskační transformace po dohodě s vedoucím. Zjistěte možnosti použití backendových LLVM modulů pro zápis kódu z interní reprezentace LLVM (LLVM IR) do formátu ELF architekruty x64_64. Použijte standardní metriky potence a resilence [1] ke srovnání implementovaných transformací s obdobnými obfuskátory.

Seznam odborné literatury

1. Collberg C. et al. A Taxonomy of Obfuscation Transformations. Technical Report #148. University of Auckland. New Zealand. https://researchspace.auckland.ac.nz/bitstream/handle/2292/3491/TR148.pdf

2. LLVM Team. The LLVM Compiler Infrastructure. University of Illinois at Urbana-Champaign. http://llvm.org/.

prof. Ing. Róbert Lórencz, CSc.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 18. února 2018

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Diplomová práce

Obfuskátor nad platformou LLVM

Bc. Roman Šíma

Katedra počítačových systémů
Vedoucí práce: Ing. Tomáš Zahradnický, Ph.D.

26. června 2018

Poděkování

Chtěl bych poděkovat rodině za podporu, a trpělivost, kterou se mnou během mého studia měli.

Dále bych chtěl poděkovat mému vedoucímu práce za ochotu a čas, které mi věnoval.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

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

V Praze dne 26. června 2018 .....

Odkaz na tuto práci

Šíma, Roman. *Obfuskátor nad platformou LLVM*. Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2018.

Abstrakt

Tento text se zabývá problematikou obfuskace, jakožto technické ochrany kódu. Dále je představen LLVM framework, jakožto možný nástroj pro tvorbu obfuskačních modulů kompilátoru. Vybrané obfuskační metody jsou dále implementovány a zhodnoceny, konkrétně se jedná o převod funkcí do tabulkové formy, vkládání mrtvého nebo nepodstatného kódu a prokládání funkcí.

Klíčová slova obfuskace, obfuskátor, tansformace, LLVM, kompilátor, reverzní analýza

Abstract

This text deals with obfuscation as a technical code protection. The LLVM framework is also introduced, as a possible tool for creating obfuscation plugins. The selected obfuscation methods are further implemented and evaluated, namely the table interpretation, inserting dead or irrelevant code and method interleaving.

Keywords obfuscation, obfuscator, transformation, LLVM, compiler, reverse analysis

Obsah

Seznam obrázků

Seznam tabulek

Úvod

V dnešní době není neobvyklé, že si firmy snaží ukradnout a okopírovat konkurenční produkty anebo jejich části. Může jít například o design, technologii, nebo nápad. Stále častěji se ale jedná o software anebo algoritmus provádějící nějakou specifickou a klíčovou úlohu. Důvodem je to, že zkopírování softwaru (nebo jeho části) je daleko rychlejší, než jeho případný vývoj. Jedním z příkladů je kauza, ve které společnost *Apple* žalovala společnost *Windows*[1] za to, že okopíroval některé části multimediálního přehrávače *QuickTime*, které přidal do svého multimediálního frameworku *Video for Windows* [1].

Existují různé právní nástroje k ochraně softwaru jako jsou patenty a licence. Takovýto software je chráněn autorským zákonem. Problém je v tom, že prokazování neoprávněného použití intelektuálního vlastnictví třetí osobou je velice obtížné a zdlouhavé. Vedle právní ochrany softwaru existuje i technická ochrana.

Technická ochrana je realizována pomocí široké škály principů, přístupů a technik určených k zabezpečení softwaru [2]. Uplatňuje se především proti nelegitimnímu používání nebo nelegálnímu kopírování. Technickou ochranou může být například aktivace softwaru pomocí sériového čísla, aktivace po internetu nebo ochrana v podobě hardwarového klíče [3]. Bohužel žádnou z těchto ochran nelze zcela zabránit reverzní analýze a následnému okopírování softwaru. Jednou z technik technické ochrany je i obfuskace [4].

Cílem obfuskace je znesnadnit analýzu softwaru, anebo některé jeho části. Většina obfuskátorů pracuje již s přeloženými soubory a nad nimi provádí obfuskace. My, na rozdíl od nich, budeme provádět obfuskaci již při překladu programu.

K tomu budeme využívat platformu LLVM [5]. Což je velice rozsáhlý framework určený pro tvorbu překladače, nebo jeho části. Mimo jiné umožňuje i tvorbu optimalizačních zásuvných modulů, což využijeme ke tvorbě obfuskačních modulů, jelikož oboje lze chápat jako jakousi transformaci kódu.

[1] žaloba zahrnovala i společnosti Intel a Canyon.

V poslední části představím vlastní implementaci modulů a zhodnotím jejich kvality.

Kapitola 1

Obfuskace kódu

Obfuskace [4][6] je proces, při kterém je aplikována posloupnost transformací $\mathcal{T} = \{\mathcal{T}_1, \ldots, \mathcal{T}_n\}$ na vstupní program P skládající se z prvků (třídy, metody, výrazy, proměnné atd.) $\{S_1, \ldots, S_m\}$. Výstupem obfuskační transformace je nový program $P' = \{\ldots, S'_j = \mathcal{T}_i(S_j), \ldots\}$ takový že:

- Program P' má stejnou funkcionalitu jako P. Jinak řečeno transformace zachovává sémantiku programu.
- Program P' je záměrně oproti původnímu P co nejvíce „znepřehledněn“. Analýza P' je z pohledu reverzního inženýra mnohem více časově náročná než analýza programu P.

Z předchozí definice plyne, že výstup programů P' a P musí být (pro ten samý vstup) stejný. Co se týče efektivnosti programů, tak zde žádný takový požadavek není. Dokonce pro mnoho transformací platí, že obfuskovaný program je pomalejší a větší.

Pro obfuskace dále platí, že je lze aplikovat na různé formy kódu. Ať už jde o zdrojový, strojový, anebo bytekód. Takováto obfuskace může být prováděna buďto „ručně“ anebo pomocí automatizovaných obfuskčních nástrojů. V této práci se dále budeme zabývat návrhem a implementací obfuskačního nástroje.

1.1 Motivace

Motivací pro obfuskaci kódu, je už jen samotná existence oboru reverzního inženýrství počítačového softwaru, který se zabývá odhalováním funkcionalit a vnitřních struktur programu.

Důležité je říci, že reverznímu inženýrství počítačového softwaru nelze nijak zabránit. Pokud bude reverzní inženýr mít dostatek času a motivace, tak je schopný program analyzovat bez ohledu na složitost ochrany. Jeho snažení však lze značně zkomplikovat a tím významně prodloužit čas analýzy, a to právě prostřednictvím obfuskačních technik.

Jak už bylo řečeno, obfuskovat lze i zdrojový kód. Uvažujme proto nyní webovou aplikaci napsanou v programovacím jazyku *javascript*. Nevýhodou *javascriptu* je to, že je distribuován ve zdrojové formě spolu s webovou stránkou. Každý uživatel tak může vidět veškerou funkcionalitu kódu. Jedním způsobem jak částečně chránit tento kód je obfuskovat ho. Dalším způsobem může být to, že přesuneme zpracování jeho hlavních funkcionalit na server. To však neznamená, že je kód zcela ochráněn. Například pokud budeme mít zařízený webhosting a nechceme, aby poskytovatel jednoduše viděl náš kód, je vhodné obfuskovat i serverovou část aplikace.

Jednou z nejvíce motivovaných skupin jsou společnosti vyvíjející počítačové hry. V minulosti byly hry často „prolomeny“ do několika týdnů nebo dokonce dnů po jejím vydání. To těmto společnostem značně snižovalo zisky, a proto začaly investovat a vyvíjet čím dál složitější ochrany kódu. Mezi tyto ochrany patří i obfuskace.

```
function mojeFunkce()
{
 window.alert("Obfuskovano")
}

mojeFunkce()
```

```
var _0xf7cf=["\x4F\x62\x66
\x75\x73\x6B\x6F\x76\x61
\x6E\x6F","\x61\x6C\x65\x72
\x74"];function mojeFunkce()
{window[_0xf7cf[1]]
(_0xf7cf[0])}mojeFunkce()
```

Listing 1.1: Ukázka obfuskace zdrojového kódu, konkrétně javascriptu.

1.2 Výhody/Nevýhody

Jak už bylo nastíněno dříve, hlavní výhodou obfuskace je ochrana kódu před reverzní analýzou. Některé programovací jazyky jsou navrženy tak, že lze „snadno“ provést převod ze spustitelného souboru zpět na zdrojový kód (dekompilace). Těmi jsou například *.NET* jazyky nebo *Java*. Pro tyto jazyky je obfuskace velice důležitý prvek při ochraně kódu. Existuje ale i mnoho nevýhod. Záleží na okolnostech a možnostech prostředí, kde bude program nasazen. Jednou z nevýhod může být velikost spustitelného souboru, kdy je do kódu přidán „mrtvý“ nebo nepodstatný kód, sloužící ke zmatení a demotivaci reverzního inženýra. Další nevýhodou může být zpomalení programu, což je v některých případech velice nežádoucí. Protože existuje mnoho typů obfuskací s různými vlastnostmi, lze vybrat vhodný kompromis mezi ochranou a rychlostí, nebo velikostí programu. Jako nevýhodu lze brát i fakt, že obfuskační techniky jsou hojně využívány výrobci škodlivých programů, což znesnadňuje práci právě reverzním inženýrům antivirových firem. Totiž čím déle nebude program označený jakožto škodlivý, tím déle může generovat zisky, popřípadě dělat činnost, pro kterou byl vyroben.

1.3 Klasifikace

Obfuskaci můžeme chápat jako jakýsi nástroj, kterým lze schovat určitou informaci o našem programu, ať už jde například o strukturu, funkcionalitu nebo nějaký řetězec. Proto obfuskační transformace primárně klasifikujeme [4] podle toho, na jaký druh informace v programu se zaměřují. Dělíme je do 4 základních kategorií:

Obfuskace lexikální struktury kódu (*Layout*) Dobrý programátor se snaží dodržovat určitá pravidla, která dělají pro člověka jim vytvořený kód přehledný a dobře čitelný. Jedná se například o pojmenování proměnných podle jejich významu nebo psaní komentářů k jednotlivým částem programu. Díky těmto informacím je pro reverzního inženýra snazší a rychlejší pochopit význam kódu. Cílem tohoto druhu obfuskace je odstranit tyto informace.

Obfuskace řízení toku kódu (*Control flow*) Tato transformace mění posloupnost vykonávaných příkazů s tím, že celková funkcionalita zůstává nezměněna. Jde třeba o sloučení a přeuspořádání kódu, o vkládání „mrtvých“ nebo nepodstatných bloků kódu anebo „rozbalení“ cyklu na sekvenční, po sobě jdoucí bloky kódu.

Obfuskace dat Tímto druhem obfuskace se rozumí zakrytí struktur dat používaných v programu. To zahrnuje techniky jako je slučování, rozdělování nebo přeuspořádání datových struktur. Jde například o rozdělení pole hodnot na více menších podpolí.

Preventivní obfuskace Na rozdíl od výše zmíněných se tento druh obfuskace nesnaží snížit čitelnost programu před reverzním inženýrem, ale spíše je používán proti automatickým nástrojům, které používá. Těmi jsou například nástroje pro statickou a dynamickou analýzu programu.

V předchozím textu jsme obfuskace klasifikovali podle toho, na jaký druh informací v programu jsou zacíleny. Dále můžeme obfuskace klasifikovat [4] podle toho, jaké druhy operací jsou nad těmito informacemi prováděny.

Slučovací obfuskace Tyto transformace slučují části programu, které spolu nesouvisí, nebo naopak rozděluje části, které spolu souvisí. Rozbíjejí abstrakci programu vytvořenou programátorem a vytváří novou, která s původní nemá nic společného. Příkladem je *inlining* a *outlining* funkcí.

Výpočetní obfuskace Tento druh obfuskací se snaží schovat původní tok kódu programem. Jedná se například o paralelizaci kódu.

Řadící obfuskace Tyto obfuskace mění pořadí provádění částí kódu. Příkladem je přeuspořádání cyklů smyčky. Jsou vhodné i pro datové struktury.

Například lze přeuspořádat pole hodnot. Důležité je říci, že transformace nesmí nijak změnit funkcionalitu programu.

1.4 Metriky

Předtím než začneme pracovat s obfuskacemi musíme definovat metriky, které popisují jaký vliv má daná transformace na program. Konkrétně budeme uvažovat [4] potenci, odolnost a cenu.

Potence udává, o kolik je pro reverzního inženýra těžší analyzovat obfuskovaný kód oproti původnímu programu. Rovněž lze říci, že potence vyjadřuje, jak moc se podařilo skrýt nějakou funkcionalitu programu. Tuto vlastnost budeme měřit pomocí metrik, které se používají pro vyjádření softwarové komplexity.

Softwarovou komplexitu lze podle [7] popsat jako: „*Pokud programy P a P' jsou shodné, kromě toho, že P' obsahuje více vlastnosti q než P, tak P' je více komplexní než program P.*“ Komplexitu programu lze měřit pomocí různých metrik. Pro většinu *control flow* obfuskací se používají [4] metriky:

Délka programu udává počet prvků v programu. Jedná se například o počet instrukcí nebo základních bloků.

Cyklomatická komplexita vyjadřuje počet nezávislých cest skrz program. S rostoucím počtem podmíněných výrazů tato komplexita roste.

Komplexita vnoření reprezentuje úroveň zanoření podmíněných výrazů.

Tyto metriky byly navrženy proto, aby pomohly programátorům vytvářet čitelnější, spolehlivější a udržovatelnější kód. Cílem programátorů je komplexitu co nejvíce minimalizovat. Našim cílem bude naopak jí maximalizovat, protože čím více je program *P'* komplexní oproti *P*, tím větší je potence transformace, která *P* na *P'* převádí, a o to více je program *P'* „hůře čitelnější“ než *P*.

Uvedli jsme, že potence udává, o kolik je těžší analyzovat obfuskovaný kód oproti původnímu. Tuto vlastnost nelze měřit pouze pomocí metrik softwarové komplexity. Obtížnost analýzy závisí i na schopnosti jednotlivých reverzních inženýrů porozumět obfuskovanému kódu. Proto budeme potenci vyjadřovat pomocí tří úrovní (*nízká*, *střední* a *vysoká*). Pro hodnocení implementovaných obfuskačních modulů budeme počítat komplexitu programu a na jejím základě diskutovat úroveň potence.

Odolnost vyjadřuje, jak moc je transformace odolná vůči automatizovaným nástrojům. Podobně jako obfuskátor lze totiž vyrobit i deobfuskátor,

Rozsah	Polynomiální složitost	Exponenciální složitost
Lokální	triviální	slabá
Globální	slabá	silná
Mezi-procedurální	silná	úplná
Mezi-procesová	úplná	úplná

Tabulka 1.1: Definice úrovní odolnosti obfuskačních transformací podle [4]. **Lokální** = pokud transformace postihne pouze jeden základní blok funkce. **Globální** = pokud se transformace dotýká celé funkce. **Mezi-procedurální** = pokud transformace ovlivňuje tok informací mezi funkcemi. **Mezi-procesová** = pokud transformace ovlivňuje interakci mezi jednotlivými procesy.

který automaticky anebo s lidskou pomocí odstraňuje na kódu provedené obfuskační transformace. Podle [4] budeme odolnost transformace chápat jako kombinaci dvou veličin. První z nich je čas vynaložený reverzním inženýrem na výrobu deobfuskátoru. Takovýto čas je subjektivní, proto ho budeme určovat podle rozsahu obfuskační transformace:

Lokální pokud transformace postihne pouze jeden základní blok funkce.

Globální pokud se transformace dotýká celé funkce.

Mezi-procedurální pokud transformace ovlivňuje tok informací mezi funkcemi.

Mezi-procesová pokud transformace ovlivňuje interakci mezi jednotlivými procesy.

Druhou veličinou je časová a paměťová složitost deobfuskátoru. Tuto složitost budeme klasifikovat jako *polynomiální* nebo *exponenciální*.

Odolnost obfuskačních transformací budeme vyjadřovat pomocí pěti úrovní (*triviální*, *slabá*, *silná*, *úplná* a *jednosměrná*). Tyto úrovně definujeme pomocí tabulky (1.1). Speciálním případem je *jednosměrná* transformace. Ta vyjadřuje situaci, kde byla transformací původního programu trvale odebrána nějaká informace. Příkladem může být odstranění komentářů.

Cena představuje dodatečnou režii na vykonání obfuskovaného programu. Cenu obfuskačních transformací budeme klasifikovat pomocí čtyř úrovní (*zdarma*, *levná*, *drahá* a *nákladná*). Podle [4] budeme definovat tyto úrovně jako:

$$cena = \begin{cases} \textit{zdarma} & \text{Pokud obfuskovaný program vyžaduje o } \mathcal{O}(1) \\ & \text{více zdrojů než původní program.} \\ \textit{levná} & \text{Pokud obfuskovaný program vyžaduje o } \mathcal{O}(n) \\ & \text{více zdrojů než původní program.} \\ \textit{drahá} & \text{Pokud obfuskovaný program vyžaduje o } \mathcal{O}(n^p) \\ & \text{více zdrojů než původní program.} \\ \textit{nákladná} & \text{Pokud obfuskovaný program vyžaduje exponen-} \\ & \text{ciálně více zdrojů než původní program.} \end{cases}$$

V této části jsme definovali tři základní metriky. Pomocí těchto metrik budeme dále posuzovat kvality jednotlivých obfuskačních transformací.

1.5 Obfuskační techniky

Teď, když víme co to obfuskace jsou a jak je měřit, si některé typy obfuskací ukážeme a nastíníme jejich přibližný dopad na výsledný kód. Dříve než budeme pokračovat, si ještě představíme pojem základního bloku (*basic block* nebo zkráceně *BB*). Posloupnost instrukcí označíme za základní blok, pokud se tento blok vyznačuje tím, že se do něj vstupuje vždy od první instrukce a naopak se z něj vystupuje přes poslední instrukci [8] (nebereme v úvahu výjimky). Poslední instrukcí je většinou nějaký druh skoku nebo návratová instrukce. Z těchto základních bloků jsou tvořeny funkce. Funkce lze pak reprezentovat jako hierarchickou posloupnost těchto bloků. Ta nám dává bližší představu, jak se bude daná funkce vykonávat.

1.5.1 Vkládání *opaque* predikátů

Predikáty jsou výrazy, jejichž výsledkem je buďto pravda (*true*) anebo nepravda (*false*). Predikát se nazývá *opaque* predikát (*OP*), pokud jeho výsledek je pro obfuskátor znám již při transformaci programu, a pro reverzního inženýra je obtížné ho vyhodnotit [9]. Cílem *OP* je „schovat se“ mezi původní kód, udělat program více komplexní a jeho analýzu těžší. Přidáním *OP* zvyšujeme cyklomatickou komplexitu programu a tedy i potenci.

Při návrhu *OP* je důležité brát ohled na jejich odolnost vůči deobfuskaci. Pokud budeme do programu vkládat na první pohled „zřejmé“ predikáty, tak je jedno kolik jich vložíme, protože budou snadno odstranitelné. Důležitou roli hraje jejich rozmanitost. Vkládáním jednoho typu predikátů můžeme „předčasně“ prozradit jejich přítomnost.

Cena této transformace je závislá na složitosti podmíněného výrazu a na počtu vkládaných predikátů.

```
int x = 2;
x <<= rand()%10;

if(x&1 != 0){
    //nikdy se neprovede
}else{
    //vzdy se provede
}
```

Listing 1.2: Ukázka jednoduchého *OP* v jazyce C.

1.5.2 Změna názvu proměnných

Tato obfuskační transformace je užitečná zejména u interpretovaných programovacích jazyků, kde je kód mnohdy distribuován ve zdrojové formě. Proměnné jsou většinou pojmenovávány podle jejich účelu anebo podle toho, jaký druh dat obsahují. Změnou názvu tuto informaci odstraníme. Potence [4] transformace je *střední*, odolnost *jednosměrná* a cena je *zdarma*.

1.5.3 Změna kódování

Jedná se o přeškálování hodnoty proměnné, kde například $i' = i * konstanta$. Cílem je převést hodnotu na výraz nebo výrazy, ze kterých není ihned patrná jejich hodnota [3]. U této transformace bychom si měli dávat pozor na velikost datových typů. Mohlo by se totiž stát, že dojde k přetečení.

Jednotlivé metriky [4] závisí na tom, jak propracovaná je změna kódování. Existuje zde totiž kompromis mezi potencí a odolností ne jedné straně, a cenou na druhé. Jinak řečeno jednoduché změny kódování budou mít pouze malý vliv na výpočetní čas a paměť, ale bude snadné je deobfuskovat.

```
int k = 5;
for(int i=0; i<k; i++)
{
    int a = Pole[i];
    ...
}
```

```
int k = 500;
for(int i=0; i<k; i+=100)
{
    int a = Pole[i/100];
    ...
}
```

Listing 1.3: Ukázka změny kódování, příklad datové obfuskace.

1.5.4 Inlining funkcí

Inlining nahrazuje volání funkce obsahem volané funkce. Tato technika je využívána v optimalizačních fázích při překladu programu. Některé velice krátké funkce se kvůli režii na přesun do a z funkce nevyplatí vůbec volat, a tak jsou všechna volání nahrazena funkcí samotnou [3]. Inlining se ale

z obfuskačního hlediska používá z jiného důvodu. Funkce sama o sobě nese určitou míru abstrakce a představuje nějakou dílčí funkcionalitu. Tím, že danou funkci inlinujeme, ztrácíme informaci, a také bude v případě reverzního inženýra nutné analyzovat stejnou funkci tolikrát, kolikrát byla inlinována.

Podle [4] je tato transformace hodnocena jako vysoce odolná. Dokonce pokud jsou všechny volání funkce nahrazeny její obsahem, a funkce samotná je odstraněna, tak je odolnost *jednosměrná*. Potence je *střední* a cena transformace *levná*. Obfuskovaný program nevyžaduje žádný výpočetní čas navíc, ale jeho velikost je závislá na počtu volání inlinované funkce a na její velikosti.

1.5.5 Outlining funkcí

Druhou obfuskační metodou je outlining, což je přesný opak inliningu. Vezmeme část kódu z funkce a uděláme z něho novou funkci, na místo původního kódu je vložené volání na novou funkci [3]. Na rozdíl od inliningu vytváříme umělou abstrakci kódu, která nemá s původním programem nic společného.

Potenci transformace hodnotíme [4] jako *střední*. Co se týče odolnosti, tak ta je *silná*. Tato transformace je *zdarma*. Potenci lze zvýšit tím, že obě předchozí metody spojíme dohromady. Nejdříve provedeme inlining, pak outlining.

```
void funkce_A()
{
    a_1;
    funkce_B();
    a_2;
}
void funkce_B()
{
    b_1;
    b_2;
}
```

```
void funkce_A()
{
    a_1;
    b_1;
    b_2;
    a_2;
}
```

```
void funkce_C()
{
    a_1;
    b_1;
}
void funkce_D()
{
    b_2;
    a_2;
}
```

Listing 1.4: Ukázka inliningu a následného outliningu funkcí. Vlevo je původní kód. Uprostřed je do *funkce_A* inlinován obsah *funkce_B*. Vpravo je obsah *funkce_A* (prostřední) outlinovaný do *funkce_C* a *funkce_D*. *a_1*, *a_2*, *b_1* a *b_2* představují proměnné funkcí.

1.5.6 Eliminace volání knihovních funkcí

Každý program používá nějaké standardní aplikační rozhraní poskytované buďto přímo s programovacím jazykem nebo operačním systémem. Programátor často používá funkce, jejichž sémantika je známa. Jsou jimi například funkce pro práci s řetězci nebo alokace paměti. Nabízí se řešení, změnit názvy těchto funkcí, čímž bychom odstranili informaci o významu dané funkce. Jelikož ale jde o funkce z nějaké externí knihovny, názvy změnit vždy nelze. Tato

technika nahradí některé nejběžněji používané funkce vlastními verzemi těchto funkcí [4]. Programátor obfuskačního modulu si předem vytipuje a připraví nejběžněji používané funkce pro daný programovací jazyk a při transformaci kódu je nahradí svými verzemi. Popřípadě lze knihovní funkci inlinovat, což také zajistí ztrátu informace. Například v programovací jazyku *C* se často objevují funkce jako jsou *strlen*, *strcmp*, *strcpy*, *memcpy* nebo *memset*. Jejich nahrazením se ztíží případná analýza programu.

Potence [4] této transformace je *střední* a odolnost *silná*. Cena závisí na tom, jakým způsobem budeme funkci nahrazovat. Pokud nahrazujeme původní knihovní funkci vlastní verzí, tak záleží na konkrétní implementaci.

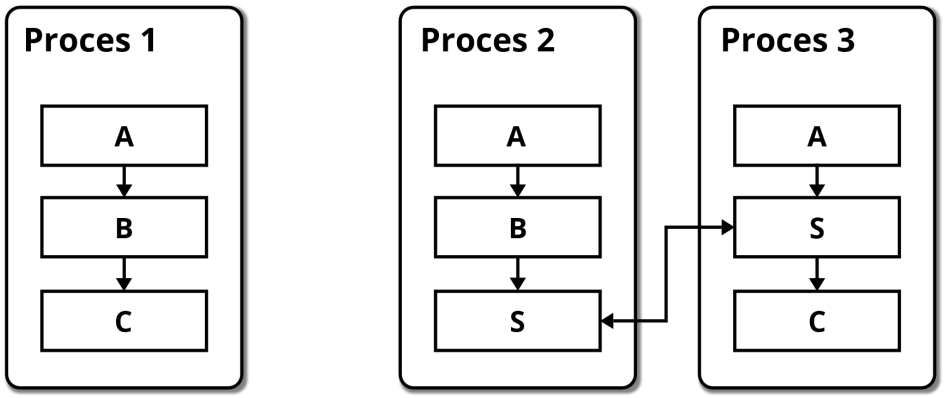
1.5.7 Paralelizace kódu

Velice efektivní je technika paralelizace kódu, kde je program rozdělen na části a je vykonáván paralelně. Podobně jako inlining i tato technika může být obsažena v optimalizačních fázích kompilátoru, který ji využívá pro snížení výpočetního času programu. Obfuskační nástroje ji využívají kvůli zvýšení složitosti programu, protože sekvenční tok programem je rozdělen na několik částí, které se mohou vykonávat současně. Takto obfuskovaný kód je náročný jak na statickou, tak dynamickou analýzu. Lze uvažovat paralelizaci buďto pomocí vláken nebo přesunutí některých částí do jiného procesu viz Obr. 1.1. Záleží ovšem na programovacím jazyku a prostředí. U této techniky může být velice těžké rozdělit program na více disjunktních částí, které na sobě nejsou závislé. Popřípadě lze použít různé synchronizační prvky. Tyto části mohou obsahovat proměnné, které jsou závislé na vykonání předchozí části. U vláken tak můžeme poskytovat tyto hodnoty, pomocí globálních proměnných. U procesů potom pomocí nějakého mechanizmu komunikace mezi procesy, jako jsou roury, sdílená paměť nebo zasálání zpráv. Těchto technik využívají vývojáři škodlivých programů, kde do paralelizovaných částí programů ještě přidávají dodatečný kód pro detekci zda zbytek procesů daného programu stále běží. Pokud byl nějaký proces ukončen, tak ho jiný po chvíli opět spustí.

Potence a odolnost transformace je závislá na tom, jestli použijeme vlákna nebo procesy. Obecně se ale dá říci, že potence je *vysoká* a odolnost *silná*. Takže tato transformace je velice kvalitní, co se týče zmatení člověka, bohužel ale cena odpovídá kvalitě, a proto je *nákladná*.

1.5.8 Vykonávání kódu v obsluze výjimek

Většina programovacích jazyků a operačních systémů umožňuje vyvolání a následnou obsluhu výjimek. Existují hardwarové a softwarové výjimky. Hardwarové jsou vyvolávány procesorem, pokud dojde například k dělení nulou nebo přístupu do paměťové stránky, která není mapována. Softwarové jsou vyvolány programem nebo operačním systémem. Obecně systém výjimek slouží buďto jako možnost opravit kód nebo stav, který tuto událost předcházel, anebo k



Obrázek 1.1: Schéma paralelizace kódu. Vlevo je původní kód a vpravo paralelizovaný. A,B,C představují *BB* původního programu. S je základní blok obsahující synchronizaci.

popisu chyby, která nastala. Jak už bylo řečeno, některé programovací jazyku umožňují definovat vlastní obsluhu a typy výjimek. Z pohledu obfuskace to znamená, že lze přerušit klasický tok kódem a vykonat kód uložený v obsluze výjimek. Některé části programu, tak přesuneme do registrovaných obsluh výjimek a v původním kódu nám stačí výjimku vyvolat.

Potence a odolnost závisí na tom, jak nápadně, popřípadě nenápadně, je výjimka vyvolána. Tato transformaci je relativně *levná*. Vykonávání programu je prodlouženo o režii na zpracování výjimky operačním systémem. Záleží také ovšem na tom, jak často je kód vykonáván pomocí obsluhy výjimek.

1.5.9 Zabalení obsahu spustitelného souboru

Jde o techniku, kde je náš spustitelný soubor zabalen. Nyní je chápán jako binární blok dat. Poté je vytvořen nový spustitelný soubor, který obsahuje tento blok dat a jakousi rozbalovací rutinu, která nahraje původní spustitelný kód do paměťového prostoru procesu a spustí jej. Nástroje které provádí tuto transformaci se nazývají „packery“. Tato metoda je oblíbená mezi vývojáři škodlivých programů, protože tím ztěžují práci antivirovým firmám. Je odolná proti statické analýze, ale naopak pomocí dynamické analýzy lze nalézt původní vstupní bod. Cena takovéto transformace závisí na komplexnosti rozbalovací rutiny.

1.6 Existující nástroje

Existují obfuskátory, jak komerčních, tak s volně dostupným kódem (*open-source*). Většina z nich se zaměřuje na specifický programovací jazyk nebo

platformu. Proto nelze provést přímé srovnání. My se dále budeme zabývat třídou obfuskátorů založených na LLVM frameworku. Konkrétně si představíme nástroj s názvem *Obfuscator-LLVM*.

1.6.1 Obfuscator-LLVM

Jde o projekt [10], který byl zahájen roku 2010 výzkumníky švýcarské univerzity[2]. Hlavním cílem projektu je poskytnout *open-source* řešení, zajišťující ochranu softwaru pomocí obfuskace kódu.

Obfuskátor je založen na LLVM frameworku, díky čemuž je kompatibilní se všemi programovacími jazyky a cílovými architekturami, jako aktuální verze LLVM. Tento nástroj poskytuje tři obfuskační transformace:

Substituce instrukcí hlavní myšlenka této obfuskační techniky spočívá v nahrazení standardních binárních operací za funkčně ekvivalentní, ale více složitější posloupnost instrukcí.

Vkládání „mrtvého“ kódu do původního kódu jsou vkládány nové *BB* spolu s *OP*. Ta větev podmíněného výrazu která se vždy provede odkazuje na původní kód, ta která se nikdy neprovede odkazuje na „mrtvý“ kód.

Převod funkcí do tabulkové formy převádí hierarchii základních bloků funkce na množinu *BB*, které jsou postupně „volány“ pomocí nově přidané kontrolní logiky. Tato transformace odstraňuje informaci o tom, jak se postupně prochází funkcí přes jednotlivé *BB*.

Tyto obfuskační techniky lze aplikovat na celý program, anebo na jednotlivé funkce. Pomocí anotací můžeme ve zdrojovém kódu určit, které funkce a jakými transformacemi obfuskovat.

```
int pocitani() __attribute((__annotate__(("sub"))));

int pocitani() {
    ...
}
```

Listing 1.5: Ukázka anotace funkcí v programovacím jazyku *C*. Z celého programu je pouze na funkci *pocitani* aplikována obfuskace (substituce instrukcí).

[2]Univerzita aplikované vědy a západního umění v *Yverdon-les-Bains*.

1.7 Zhodnocení

V této kapitole jsme definovali obfuskaci jako určitou transformaci kódu, jejíž účelem je skrýt funkcionalitu programu, popřípadě zmást reverzního inženýra. Uvedli jsme důvody proč obfuskovat a jaké výhody a nevýhody z těchto transformací plynou.

Řekli jsme, že obfuskace lze chápat jako jakýsi nástroj, kterým lze schovat určitou informaci o programu. Proto jsme obfuskace klasifikovali za prvé podle toho na jaký druh informací se zaměřují, a za druhé podle toho jaké druhy operací se nad těmito informacemi provádí.

Dále jsme definovali tři metriky pomocí kterých budeme obfuskace měřit. Jedná se o potenci, která udává o kolik je těžší analyzovat obfuskovaný kód oproti původnímu. Dále odolnost, ta vyjadřuje jak je transformace odolná vůči automatizovaných nástrojům. A nakonec cena. Ta udává o kolik je obfuskovaný program časově popřípadě paměťově náročnější než-li původní program.

Představili jsme si některé druhy obfuskačních technik. Tyto techniky v dalších kapitolách ještě doplníme o vkládání „mrtvého“ nebo nepodstatného kódu, převod funkcí do tabulkové formy a prokládání funkcí. Na závěr jsme si představili obfuskátor postavený na LLVM frameworku.

Dále budeme hovořit o platformě LLVM, kterou využijeme k tvorbě vlastních obfuskačních modulů.

Kapitola 2

Platforma LLVM

Projekt LLVM [5], vznikl v roce 2000 na *University of Illinois*. Šlo o výzkum technik pro překlad libovolného statického nebo dynamického programovacího jazyka. Do dnešní doby se projekt rozrostl a nyní zastřešuje a rozvíjí řadu nízkoúrovňových nástrojů určených pro překlad kódu. Jedná se o otevřenou platformu se spousty přispěvateli, jako jsou firmy, vědci ale i jednotlivci.

LLVM je vyvíjen v programovacím jazyku *C++*, jakožto balík knihoven, které jsou navrženy tak, aby byly co nejvíce modulární. Hlavní myšlenka je taková, že překladač se skládá z různých komponent, které jsou zaměnitelné popřípadě doplnitelné jinými. To umožňuje například jednoduché portace pro jiné cílové architektury, nebo vývoj nových programovacích jazyků [11].

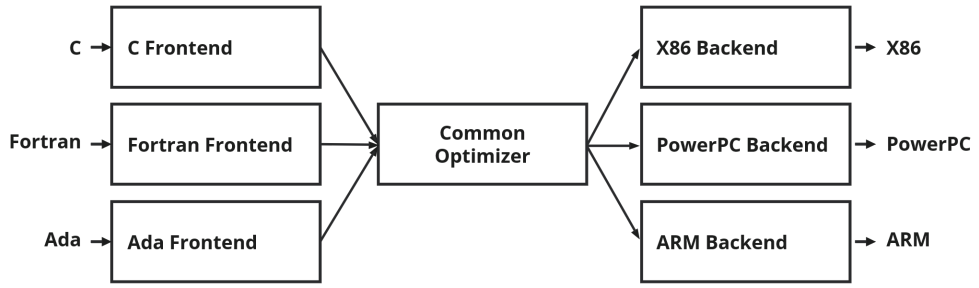
Jedním z projektů LLVM je *Clang* [12]. Slouží jakožto přední části překladače pro programovací jazyky jako jsou *C*, *C++*, *Objective-C* a další. Výhodou je, že jeho příkazy jsou kompatibilní s populárním *GNU GCC* překladačem.

2.1 Základní struktura překladače

Jak už bylo řečeno, překladač je „složen“ z mnoha komponent. My budeme uvažovat model, kde je rozdělen na 3 části, a to přední (tzv. *frontend*), střední a zadní (tzv. *backend*) část [13]. Každá z těchto částí je zodpovědná za nějakou transformaci kódu. Jako vstup pro překladač uvažujeme zdrojový kód nějakého programovacího jazyka a výstupem nejčastěji bývá strojový kód cílové architektury. Většina překladačů obsahuje více různých předních a zadních částí viz Obr. 2.1.

2.1.1 Přední část (front-end)

Účelem této části je syntaktická analýza. Dále je tato část zodpovědná za převod zdrojového kódu do reprezentace, se kterou může překladač dále pracovat. Každý programovací jazyk má svoji vlastní přední část překladače.



Obrázek 2.1: Třífázový návrh překladače [5].

V LLVM se pro překlad *C* programů používá *Clang*. Tento převod obsahuje několik fází, které popíšeme dále [13].

lexikální analýza se stará o rozdělení vstupního textu na jednotlivé symboly takzvané „tokeny“. Důvodem je příprava pro syntaktickou analýzu. Výstupem této fáze je sekvence těchto symbolů.

syntaktická analýza zajišťuje transformaci z lineární sekvence symbolů z předešlé fáze na tzv. abstraktní stromovou strukturu (*AST*[3]). V této fázi se kontroluje syntaxe podle pravidel daného programovacího jazyka.

sémantická analýza prochází stromovou strukturu a kontroluje sémantiku podle daného jazyka. Jedná se například o typovou kontrolu nebo kontrolu platnosti proměnných.

Nakonec je *AST* představující vstupní kód převeden do vnitřní reprezentace (IR) a předán střední části překladače.

2.1.2 Střední část (middle-end)

Přední část překladače pouze kontroluje a převádí kód do IR. Cílem této části je provést analýzy a následné optimalizace kódu. Tato část se někdy nazývá optimalizační. Není závislá ani na programovacím jazyku ani na cílové architektuře (nebo jen minimálně). Je to díky tomu, že optimalizace se provádí na vnitřní reprezentaci kódu. Některé tyto reprezentace mají formu *SSA*[4] a tříadresového kódu.

SSA neboli *Static Single Assignment* je vlastnost, kde každé proměnné je přiřazena hodnota pouze jednou. Zároveň musí být definována dříve, než je použita.

[3] Abstraktní syntaktický strom
[4] Static Single Assignment

Tříadresový kód je taková forma kódu, že každá instrukce obsahuje nejvýše 3 operandy. Většinou se jedná o přiřazení výsledku nějaké binární operace do proměnné.

2.1.3 Zadní část (back-end)

Následně je vnitřní reprezentace převedena do platformně závislého kódu. Nyní, když máme tento kód, můžeme provést optimalizace specifické pro danou architekturu. Výstupním kódem je buďto kód symbolických instrukcí nebo strojový kód. Každá cílová architektura má svojí vlastní zadní část překladače.

Nakonec bychom pomocí *linkeru* program sestavili. V našem textu se budeme zabývat střední částí překladače, protože obfuskace jsou v jistém smyslu podobné optimalizacím. Oboje představují transformace kódu.

2.2 Vnitřní reprezentace v LLVM

Vnitřní reprezentace LLVM IR (*Intermediate Representation*) je typově bezpečný nízkoúrovňový programovací jazyk postavený na *SSA*. Svoji formou je podobný jazyku symbolických instrukcí, ale sám o sobě obsahuje spoustu přidaných informací, takže pomocí něj lze reprezentovat jakýkoliv vysokoúrovňový programovací jazyk bez ztráty informace. V případě, že chceme vytvořit překladač pro nový programovací jazyk, stačí vytvořit pouze přední část překladače, který převádí daný jazyk do vnitřní reprezentace. O zbytek se už dále postará samotné LLVM [14].

Vnitřní reprezentace může být použita ve třech různých podobách. Buďto jakožto výše zmíněná textová podoba programovacího jazyka s koncovkou souboru *.ll*. Nebo v podobě binárního kódu, který je efektivní například při práci s JIT (*Just-In-Time*) překladačem nebo interpretem. Tyto soubory se označují koncovkou *.bc*. Obě tyto podoby jsou ukládány na pevný disk jako soubory. Poslední podobou je vnitřní reprezentace uložená v paměti. Všechny tyto podoby jsou z funkčního hlediska ekvivalentní [14].

Dále stručně shrneme textovou podobu vnitřní reprezentace, jelikož je velice úzce spojená s programovým rozhraním, které budeme používat pro výrobu optimalizačních (obfuskačních) modulů. To zahrnuje identifikátory, typy, funkce, instrukce a další [14].

2.2.1 Identifikátory

Identifikátory lze rozdělit na globální a lokální. Globální jsou funkce a globální proměnné. Názvy těchto identifikátorů začínají znakem *@*. Lokální začínají znakem *%* a představují názvy registrů (v této části překladu by se dalo říci proměnné). Názvy identifikátorů jsou buďto převzaty ze zdrojového programu (*@nějakáProměnná*), vygenerovány při překladu na vnitřní reprezentaci po-

mocí číselného suffixu (*%prom*, *%prom0*), popřípadě může jít o konstantu, která je dána hodnotou bez názvu.

LLVM IR, podobně jako ostatní jazyky, obsahuje některá klíčová slova. Jsou to například názvy typů nebo binárních operátorů. Výhodou je to, že všechny proměnné začínají buďto @ nebo %, takže s nimi nedojde ke konfliktu.

```
@globalni = global i64 0, align 8
; ... zacatek funkce ...
%lokalni = load i64, i64* @globalni, align 8
%lokalni0 = load i64, i64* @globalni, align 8
```

Listing 2.1: Ukázka identifikátorů v LLVM IR.

2.2.2 Základní typy

Jednou z důležitých vlastností LLVM IR jsou typy. Každá proměnná má svůj typ. Důvodem pro zavedení typů je to, že některé optimalizace je možné provést přímo bez další analýzy kódu. Zároveň lze říci, že takto typově „orientovaný“ jazyk je čitelnější. Některé typy jsou závislé na cílové architektuře. Obecně LLVM IR obsahuje běžné základní typy, které jsou viděny u ostatních jazyků. Jsou jimi například celočíselné typy (*i16*, *i32*, *i64*, ...), kde číslo za *i* říká, kolika bitové číslo může uchovat. Například *i1* je analogický pro *bool*. LLVM obsahuje i typy pro reprezentaci čísel s plovoucí řádovou čárkou (*float*, *double*, ...).

```
@.str = constant [8 x i8] c"retezec\00", align 1
%0 = alloca i16, align 2      ; 16-bitove cislo
store i16 65535, i16* %0, align 2
```

Listing 2.2: Ukázka základních datových typů v LLVM IR.

Podobně jako programovací jazyk *C* má i LLVM IR ukazatele na typy. To je realizováno přidáním za daný typ hvězdičku (*<type*>*). Mimo klasických typů obsahuje i některé specifické jako je vektor, pole, token, návěští nebo struktura.

2.2.3 Instrukce

Program ve vnitřní reprezentaci obsahuje strukturovaný kód. Modul obsahuje funkce, jednotlivé funkce obsahují množinu základních bloků a nakonec základní bloky obsahují instrukce. Instrukční sadu LLVM IR lze rozdělit do několika kategorií podle uplatnění.

Koncové instrukce jak bylo řečeno dříve, každý základní blok má jednu vstupní a jednu výstupní instrukci. Koncové instrukce jsou poslední instrukce v těchto blocích. Jedná se například o návratovou instrukci (*ret*)

nebo různé druhy instrukcí definující tok kódem (*br*, *switch*, ...). Speciální koncovou instrukcí je *invoke*. Jde o volání funkce s možností obsluhy výjimek.

Binární instrukce většina výpočtů v programu je provedeno právě pomocí těchto instrukcí. Obsahují dva vstupní operandy a jeden výstupní, kde všechny musí být stejného typu. Mezi tyto instrukce patří klasické sčítaní, odčítaní, násobení, dělení, a to jak celých čísel, tak i čísel s plovoucí řádovou čárkou. Další binární instrukcí je například zbytek po dělení (*urem*).

Bitové instrukce další podporou jsou instrukce pro manipulaci s jednotlivými bity. Podobně jako binární instrukce i bitové instrukce vyžadují dva vstupní a jeden výstupní operand stejného typu. Jedná se o logické operace jako jsou *and*, *or*, nebo *xor*, a bitové posuny *shl*, *lshr* a *ashr*.

```
define i32 @main() #0 {
  %0 = load i32, i32* @g, align 4
  %add = add nsw i32 %0, 5                ; g += 5
  %sub = sub nsw i32 %add, 2              ; g -= 2
  %mul = mul nsw i32 %sub, 3              ; g *= 3
  %lshr = ashr i32 %mul, 1                ; g >>= 1
  store i32 %lshr, i32* @g, align 4
  ret i32 0                               ; return 0
}
```

Listing 2.3: Ukázka bitových a binárních operací v LLVM IR.

Paměťové instrukce LLVM IR obsahuje ukazatele na typy, musí s nimi umět i pracovat. Obecně tyto instrukce pracují s pamětí zásobníku, v rámci dané funkce. Hlavní instrukcí je *alloca*, která alokuje část paměti (podle daného datového typu) na zásobníku. Vrátí ukazatel příslušného typu. Dalšími instrukcemi jsou *load* (načítání hodnot ze zásobníku) a *store* (ukládání hodnot na zásobník).

Strukturové instrukce v instrukční sadě jsou i některé instrukce pro práci s vektory, poli a strukturami. Umožňují základní operace jako je vkládání (*insertelement*, ...) a přístup k (*extractelement*, ...) jednotlivým prvkům.

```
%a = alloca [3 x i32], align 4
%i = getelementptr inbounds [3 x i32], [3 x i32]* %a,
    ↪ i64 0, i64 1
store i32 5, i32* %i, align 4
```

Listing 2.4: Ukázka paměťových a strukturových operací v LLVM IR.

Instrukce pro konverzi podobně jako ve vyšších programovacích jazycích, tak i tady je možná konverze z některých datových typů na jiné. Tyto instrukce vyžadují pouze jeden vstupní a jeden výstupní operand. Například lze převést větší celočíselný typ na menší pomocí příkazu *trunc* nebo naopak z menší na větší pomocí *sext* nebo *zext*. Popřípadě lze provádět bitové konverze pomocí *bitcast* instrukce.

Ostatní instrukce v poslední kategorii „skončily“ všechny zbylé. Jde o instrukce, které nelze lehce zařadit, ale i o specifické instrukce pro LLVM IR. Je to například klasické volání funkcí (*call*), instrukce vyhodnocující podmínky (*icmp*), anebo *phi* instrukce.

```
%0 = load i32, i32* %a, align 4
%conv = trunc i32 %0 to i16
%call = call i32 @_funkce(i16 signext %conv)
```

Listing 2.5: Ukázka konverze datových typů v LLVM IR.

2.2.4 Vestavěné funkce

Do LLVM IR je přidán pojem vestavěné funkce (*Intrinsic Functions*). Tyto funkce mají známý název a sémantiku, jako je například *llvm.memcpy* nebo *llvm.sqrt*. Jde o jakýsi rozšiřující mechanizmus jazyka LLVM. Názvy těchto funkcí musí začínat vždy prefixem *<llvm.*>*. Tento prefix je rezervovaný, takže žádný název uživatelské funkce nesmí takto začínat. Všechny tyto funkce jsou externí a nelze tak definovat jejich tělo [14]. Jediné co umožňují je jejich zavolání. Některé vestavěné funkce jsou přetěžované, takže lze volat stejnou funkci s jiným datovým typem. Jelikož je celá platforma LLVM psána pomocí programovacího jazyka *C*, tak mnoho vestavěných funkcí je ze standardní *C* knihovny. Jedná se například o funkce pro práci s pamětí (*llvm.memset.**, *llvm.memmove*, ...) nebo o matematické funkce jako je (*llvm.sin.**, *llvm.pow.**, *llvm.ceil.** nebo *llvm.log.**).

```
declare void @llvm.va_start(i8*)
declare float @llvm.pow.f32(float %v, float %pow)
```

Listing 2.6: Příklad vnitřních funkcí v LLVM IR.

2.3 Základní nástroje LLVM IR

Platforma LLVM poskytuje mnoho nástrojů pro práci s vnitřní reprezentací kódu. Tyto programy se spouští pomocí příkazové řádky daného operačního systému. Některé programy umožňují provést určité části překladu (například *llc* nebo *opt*). Většina z jich akceptuje jak textovou (*.ll*), tak binární (*.bc*) podobu. Dále jsou tyto programy představeny [15].

llvm-as jedná se o LLVM *assembler*, který převádí textovou podobu vnitřní reprezentace (*.ll*) na její binární podobu (*.bc*). Jde pouze o převod v rámci vnitřní reprezentace. Binární soubor je stále platformně nezávislý.

```
llvm-as soubor.ll -o soubor.bc
```

Listing 2.7: Příklad LLVM assembleru.

llvm-dis je komplementární nástroj k *llvm-as*. Naopak převádí binární (*.bc*) podobu vnitřní reprezentace na textovou (*.ll*). Jedná se o takzvaný LLVM *disasembler*.

```
llvm-dis soubor.bc -o soubor.ll
```

Listing 2.8: Příklad LLVM disassembleru.

llc tento nástroj zajišťuje překlad z vnitřní reprezentace (ať už textové nebo binární) do jazyka symbolických adres dané architektury. Jde o zadní část překladače. Pomocí přepínače *-march* lze definovat pro kterou architekturu/y je výstupní soubor určen.

```
llc soubor.ll -march=x86 -o soubor_x86.s
llc soubor.bc -march=arm -o soubor_arm.s
```

Listing 2.9: Příklad použití *llc* překladače.

lli umožňuje spouštění kódu přímo z binární reprezentace LLVM buďto pomocí *JIT* překladače nebo interpretu.

```
clang -emit-llvm -c soubor.c -o soubor.bc
lli soubor.bc
```

Listing 2.10: Příklad LLVM JIT překladače.

llvm-link na vstupu načítá několik binárních LLVM souborů (*.bc*) a spojí je do jednoho. Nejedná se o sestavovací program. Výsledkem je jeden soubor opět ve vnitřní reprezentaci (buďto *.ll* nebo *.bc*).

llvm-diff velice užitečný nástroj pro porovnávání LLVM struktur. Je určený pro vývojáře optimalizačních modulů. V kombinaci s programem *opt* lze porovnávat a zkoumat jednotlivé fáze optimalizace.

llvm-stress je nástroj určený pro generování náhodných LLVM souborů (*.ll*), určených pro testovací účely.

opt jeden z nejdůležitějších nástrojů pro práci s vnitřní reprezentací. Umožňuje analyzovat a následně optimalizovat kód vnitřní reprezentace. Lze načítat vlastní optimalizační moduly v podobě sdílených knihoven. Výstup optimalizace je opět zapsán do vnitřní reprezentace.

```
clang -emit-llvm -c soubor.c -o soubor.bc
opt -load p.so -p1 soubor.bc -o vystup_p1.bc
opt -load d.so -d1 soubor.bc -o vystup_d1.bc
opt -load d.so -d1 -d2 -d3 soubor.bc -o vystup_d123.bc
```

Listing 2.11: Příklad optimalizace v LLVM pomocí opt.

2.4 Optimalizační moduly

Motivací pro představení problematiky optimalizačních modulů je to, že se jedná podobně jako u obfuskací o transformace kódu. Toho budeme využívat v implementační části této práce. Jednotlivé optimalizační moduly jsou spouštěny jakožto jednotlivé průchody nad LLVM IR kódem. Aplikační rozhraní LLVM pro tvorbu těchto modulů je zajištěno díky objektově orientované hierarchii tříd, a to pomocí programovacího jazyka *C++*.

Pro implementaci vlastního optimalizačního modulu musíme zdědit jednu z předem připravených abstraktních tříd (*ModulePass*, *LoopPass*, ...). Všechny jsou podtřídy třídy *Pass*. Každá tato podtřída je cílena na optimalizaci specifické části kódu nebo pro určitou fázi překladu [16]. Pro práci se vstupním programem musíme implementovat patřičnou virtuální metodu z abstraktní třídy (například *runOnModule*). Tyto metody nám v parametru předávají reprezentaci části kódu, který chceme optimalizovat. Pokud chceme provádět některé operace ještě předtím nebo potom, jsou tu připravené virtuální metody *doInitialization* a *doFinalization*. Všechny tyto metody vrací *bool* hodnotu, a to *true* pokud došlo ke změně LLVM IR kódu, nebo *false* v opačných případech.

ModulePass modul je obecná struktura, nad kterou lze provádět optimalizace. Pokud budeme dědit od této třídy, tak to znamená, že chceme implementovat optimalizace nad celým programem. Abychom nad ním mohli provádět transformace, musíme v našem kódu přetížit metodu *runOnModule(Module &M)*. V tomto případě, reference *M* reprezentuje celý modul vstupního programu.

CallGraphSCCPass tato třída je vhodná pro procházení programu z hlediska volání funkcí. Slouží pro optimalizaci vykonávání funkcí. Třída obsahuje virtuální metodu *runOnSCC(CallGraphSCC &SCC)*. Reference *SCC* představuje graf volání funkcí v programu.

FunctionPass umožňuje optimalizace těl jednotlivých funkcí. Většinou pracuje s množinou základních bloků. Pořadí jednotlivých funkcí, které vždy obdržíme není nijak zaručeno. V tomto typu je zakázáno měnit ostatní funkce a mazat jakékoliv funkce nebo globální proměnné. Pro použití musíme přetížit metodu *runOnFunction(Function &F)*. Reference *F* reprezentuje jednotlivé funkce modulu. Tato funkce je volána pro každou funkci zvlášť.

LoopPass jedná se o speciální třídu sloužící pro optimalizaci cyklů. Jako příklad může být *Loop Unrolling* nebo *Loop Fission*. Pokud jsou cykly vnořené, tak jsou optimalizace vykonávány od vnitřních po vnější. Abychom mohli tyto transformace provádět, musíme přetížit metodu *runOnLoop(Loop *L, LPPassManager &LPM)*.

BasicBlockPass je určena pro ty nejemnější optimalizace na jednotlivých instrukcích. Příkladem můžou být různé *peephole* optimalizace. Podobně jako u *FunctionPass* je zakázáno mazat nebo měnit ostatní základní bloky. Pro použití musíme přetížit metodu *runOnBasicBlock(BasicBlock &BB)*. V tomto případě, reference *BB* reprezentuje jeden základní blok funkce. Funkce je volána pro všechny základní bloky všech funkcích.

MachineFunctionPass jde o podtřídu *FunctionPass*. Provádí se taky na jednotlivých funkcí, ale s tím rozdílem, že optimalizace jsou volány až v zadní části překladače na již platformně závislém kódu. Takovéto pluginy se musí registrovat funkcí *TargetMachine::addPassesToEmitFile* a obecně nemohou být spouštěny pomocí *opt* nástroje. Platí tu stejná omezení jako u *FunctionPass* a navíc nelze vytvářet a přidávat žádné nové instrukce a základní bloky. Tato třída obsahuje metodu *runOnMachineFunction(MachineFunction &MF)*. Obdobně *MF* reprezentuje jednotlivé funkce.

ImmutablePass je velice speciální třída. U této třídy nemáme v úmyslu žádné optimalizace. Slouží k získání informací o cílové architektuře nebo o nastavení překladače.

Jednotlivé moduly musí mezi sebou také správně interagovat. Jedná se například o pořadí aplikování těchto modulů. Lze vyžádat, které optimalizace mají být provedeny před touto, popřípadě které zneplatnit anebo zachovat. Zneplatnit znamená to, že jsme provedli nějaké transformace kódu, takže jsme opět mohli do kódu zavést jakousi „neoptimálnost“. Tím pádem musí být předchozí optimalizace provedena znovu [16]. Pro určení těchto pořadí musí modul přetížit metodu *getAnalysisUsage(AnalysisUsage &Info)*.

2.5 Aplikační rozhraní pro práci s LLVM IR

V této sekci představíme některé základní třídy a rozhraní používané pro práci s vnitřní reprezentací kódu. V tomto případě uvažujeme že LLVM IR je uložen v paměti [17]. LLVM poskytuje objektově orientované aplikační rozhraní napsané v jazyce *C++*. Proto jakýkoliv prvek LLVM IR je v paměti reprezentován jakožto objekt. Jde například o instrukce, základní bloky, ale třeba i argumenty funkcí nebo tabulku symbolů. Většina těchto objektů má určitou příslušnost k jinému objektu. Říkáme, že objekt *A* náleží objektu *B* a žádnému jinému. Platí například, že pokud instance určité instrukce patří do nějakého základního bloku, nemůže patřit do jiného (ale může být použita v jiné instrukci jiného bloku jako operand).

Třída *Value* je nadtřídou pro mnoho základních prvků, se kterými se při implementaci setkáme. Každý z těchto prvků má nějaký typ a popřípadě název. Veledůležitými třídami jsou také *User* a *Use*, které představují takzvané *def-use* a *use-def* řetězy.

def-use nám říká, „kdo“ všechno používá jistý prvek (instrukci, funkci, ..., obecně nějaká podtřída třídy *User*). Těchto „uživatelů“ může být i více. A každý prvek si drží seznam těchto „uživatelů“. Tomuto seznamu se říká *def-use* řetěz. Například pokud budeme chtít smazat určitou instrukci, musíme nejdřív zjistit a následně nahradit nebo opravit její výskyty
v kódu. Poté, až když nemá žádné reference na jiné části kódu, ji můžeme vymazat.

```
%0 = load i32, i32* %a, align 4
%1 = load i32, i32* %b, align 4
%add = add i32 %0, %1          ;Users: %add, %sub
%sub = sub i32 %c, %1
```

Listing 2.12: Ukázka def-use řetězu.

use-def je vlastně to samé jako *def-use*, ale z opačného pohledu. Máme jistý prvek a chceme zjistit jaké jiné prvky používá. Například máme nějakou binární operaci a my chceme dostat reference na její operandy. V tomto případě jsou operandy nějaké instrukce definovány v kódu předtím.

```
%0 = load i32, i32* %a, align 4
%1 = load i32, i32* %b, align 4
%add = add i32 %0, %1          ;Uses: %0, %1
%sub = sub i32 %c, %1
```

Listing 2.13: Ukázka use-def řetězu.

LLVM IR obsahuje různé druhy instrukcí reprezentované různými třídami. Všechny tyto druhy jsou podtřídy třídy *Instruction*. V některých případech se nám může hodit vědět, o jaký druh instrukce se jedná. Pro tyto případy LLVM nabízí funkce *isa<>*, *cast<>* a *dyn_cast<>*.

V některých případech je vhodné uživatelsky specifikovat určitou vlastnost optimalizačního modulu. To lze například pomocí parametru příkazové řádky. Pro práci s těmito parametry má LLVM vestavěnou podporu v podobě („llvm/Support/CommandLine.h“). Umožňuje registrovat parametr, jeho datový typ, výchozí hodnotu a popřípadě i popis, který je zobrazen v nápovědě (*-help*).

```
static cl::opt<unsigned> numOfFuncs("mi-num"),
    cl::init(1), cl::desc("Number of functions."),
    cl::value_desc("uint"));
```

Listing 2.14: Ukázka registrace parametru příkazové řádky.

2.6 LLVM moduly pro zápis kódu

V předchozím textu jsme ukázali, jak se překládá libovolný zdrojový kód do vnitřní reprezentace, a jak jsou navrženy optimalizace pro tento kód. Nyní se podíváme na možnosti převodu LLVM IR kódu do platformně závislého kódu, na zadní část překladače. Podobně jako přední i zadní část překladače má několik fází. Mnoho z těchto fází používá stejné metody pro různé cílové architektury. LLVM přišel s tím, že platformně závislé části lze abstrahovat do *.td* (*target descriptor*) souborů, které popisují specifické vlastnosti pro danou platformu. Jedná se například o množinu registrů, instrukční sadu daného procesoru nebo o volací konvence funkcí. Pro implementaci překladače pro novou cílovou architekturu nám tak stačí specifikovat tyto vlastnosti, zbytek pak může být převzat z jiného modulu pro jinou architekturu. Zadní část překladače obsahuje fáze, jako jsou převod LLVM IR na orientovaný acyklický graf (*DAG*), legalizace instrukcí, optimalizace, výběr a plánování instrukcí, alokace registrů a nakonec generování platformně závislého kódu [18].

Převod na DAG nejdříve je LLVM IR převeden na *DAG*. Tato reprezentace kódu je vhodnější pro následné fáze překladu. Kód zůstává stále nezávislý na cílové architektuře.

Legalizace instrukcí v této fázi jsou některé instrukce nahrazeny instrukcemi podporovanými cílovou architekturou. Například *x86* architektura nepodporuje *sdiv* instrukci. Proto je v této fázi nahrazena *sdivrem* instrukcí [18].

Optimalizace v předchozí částech jsme kód převedli na *DAG* a nahradili některé instrukce již platformně závislými. Proto je vhodné provést ve-

stavěné optimalizace této reprezentace kódu. Lze provádět i vlastní optimalizace pomocí dříve představené *MachineFunctionPass* třídy.

Výběr a plánování instrukcí v této části jsou jednotlivé instrukce reprezentovány jakožto LLVM IR uzly grafu. Pro tyto uzly jsou vybrány příslušné instrukce pro danou architekturu a následně jsou převedeny na uzly reprezentující platformně závislé instrukce. Uzly samotné nám nic neříkají o pořadí vykonávání instrukcí, proto je kód dále poskládán do tří-adresové reprezentace.

Alokace registrů zde jsou „virtuálním“ registrům (proměnným) co nejefektivněji přiřazeny reálné fyzické registry daného procesoru.

Generování kódu nakonec je tato reprezentace kódu převedena na daný výstupní formát. Do kódu jsou doplněny některé další náležtosti jako je například prology a epilogy funkcí. Výstup může být jak v textové (jazyk symbolických instrukcí), tak v binární (objektová forma *.o*) podobě. Binární podoba splňuje příslušný formát pro spustitelný soubor (obsahuje hlavičku souboru například *PE*, *ELF*, nebo *COFF*).

Výstup ze zadní části překladače je strojový kód dané architektury, který musí být ještě sestaven pomocí *linkeru*, aby byl spustitelný. Ale většina překladačů ve výchozím nastavení toto dělá na pozadí za nás. Jak už bylo řečeno dříve do LLVM přispívá mnoho firem i jednotlivých odborníků. Proto není překvapením, že LLVM podporuje velké množství cílových architektur jako jsou například *ARM*, *x86*, *x86_64*, *PowerPC*, *Nvida PTX*, *SystemZ* a další.

2.7 Zhodnocení

V této kapitole jsme představili projekt LLVM, který nyní zastřešuje mnoho dalších projektů jako je například *Clang*. Uvedli jsme si základní strukturu překladače, jeho jednotlivé části a fáze. Dále jsme stručně probrali vnitřní reprezentaci (LLVM IR) kódu a shrnuli základní nástroje pro práci s touto reprezentací. Ukázali jsme také aplikační rozhraní pro tvorbu optimalizačních modulů a pro základní práci s prvky vnitřní reprezentace. Nakonec jsme přestavili možnosti generování kódu z LLVM IR do strojového kódu dané architektury.

Kapitola 3

Návrh

V této kapitole si představíme a popíšeme obfuskační techniky, které budeme dále implementovat. Konkrétně jde o převod funkcí do tabulkové formy, prokládání funkcí a vkládání „mrtvého“ nebo nepodstatného kódu. Tyto techniky budou dále implementovány jakožto obfuskační moduly nad platformou LLVM.

3.1 Převod funkcí do tabulkové formy

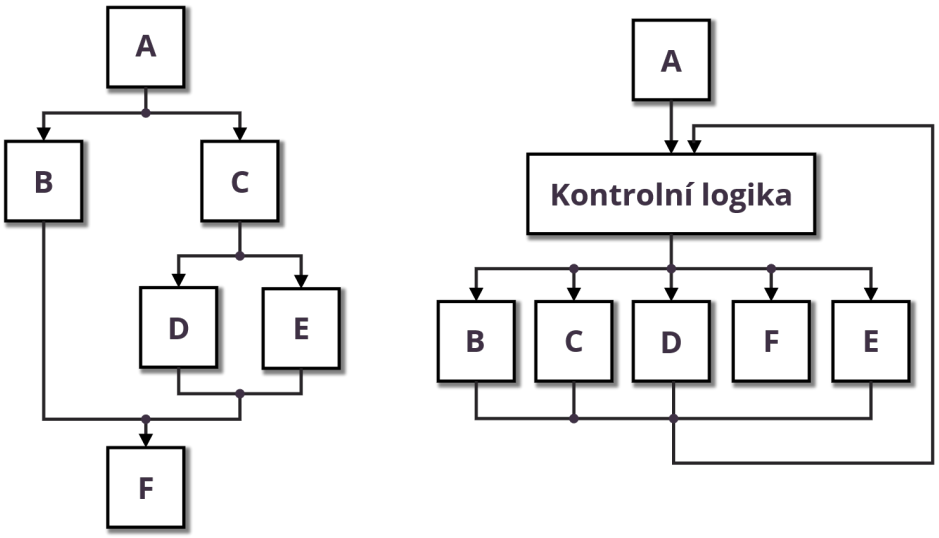
Jak víme, funkci lze reprezentovat jako hierarchickou posloupnost základních bloků, což nám dává informaci o tom, jak se kódem postupně prochází. Cílem této transformace je schovat tuto informaci a ztížit tak statickou analýzu kódu.

Transformace převádí hierarchii základních bloků funkce na množinu *BB*, které jsou postupně „volány“ pomocí nově přidané kontrolní logiky. Každý z *BB* skáče zpět na blok obsahující kontrolní logiku, a každý *BB* je odpovědný za aktualizaci proměnné používané kontrolní logikou. Tato proměnná nese informaci o tom, jaký další *BB* má být vykonán [19].

Kontrolní logiku lze realizovat například pomocí *switch* instrukce, kde každá větev této instrukce obsahuje jeden BB, a proměnná kontrolní logiky nese hodnotu další větve.

My budeme realizovat kontrolní logiku pomocí nepřímého skoku [3]. Adresy základních bloků funkce jsou uloženy do tabulky a proměnná kontrolní logiky obsahuje index do této tabulky. Výhodou nepřímého skoku oproti *switch* instrukci je to, že z pohledu reverzního inženýra může být adresa nepřímého skoku „jakákoliv“, kdežto u *switch* prvku je omezena pouze na větve této instrukce.

Řekli jsme, že každý základní blok je odpovědný za aktualizaci kontrolní proměnné. To však není vždy jednoduché realizovat. Pokud bude například *BB* obsahovat podmíněný skok, tak musíme nejdříve vyhodnotit na jaký *BB* se bude dále skákat, a podle toho přiřadit index do kontrolní proměnné. Pro takovéto situace je v LLVM připravena instrukce *SelectInst*. Tato in-



Obrázek 3.1: Převod funkce do tabulkové formy. Vlevo je původní funkce. A-F jsou základní bloky původní funkce. Vpravo je funkce převedena do tabulkové formy.

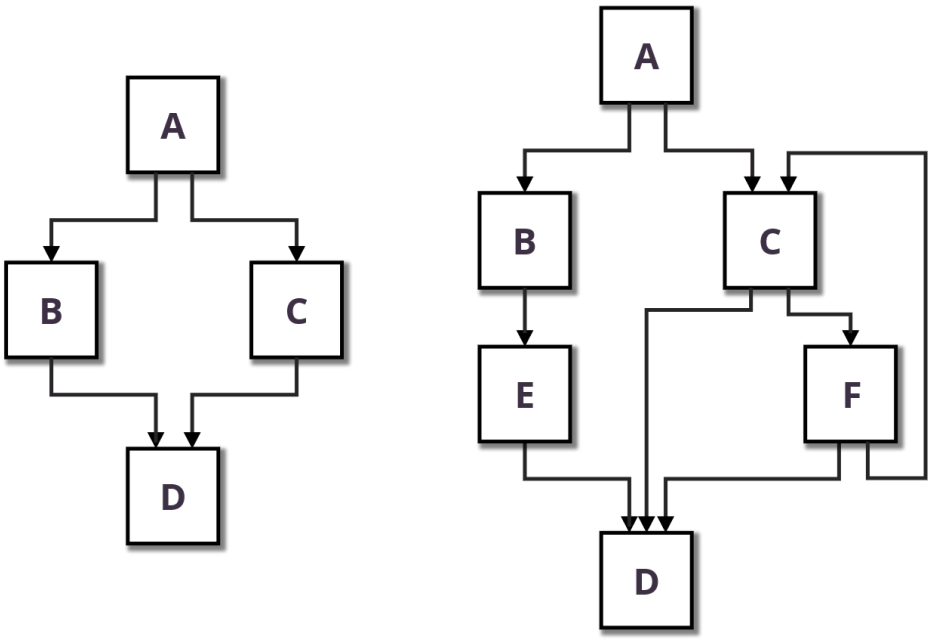
strukce na základě podmínky vrací jednu ze dvou předem připravených hodnot.
V našem případě se bude jednat o index do tabulky, který následně uložíme do kontrolní proměnné.

Podle [4] je potence této transformace *vysoká* a odolnost *silná*. Tato obfuskace je velice efektivní, ale bohužel velmi *nákladná*. Především kvůli tomu, že pro každý vykonávaný *BB* musíme připočítat režii na provedení kontrolní logiky.

Pro tuto transformaci dále zavedeme dělení *BB* na menší části a maskování hodnoty indexů ukládaných do kontrolní proměnné. Tím ještě o něco zvýšíme komplexitu programu a ztížíme statickou analýzu. Pro další ztížení analýzy lze například vkládat *OP* anebo přiřadit adresy do tabulky, na které se nebude nikdy skákat [19].

3.2 Vkládání „mrtvého“ nebo nepodstatného kódu

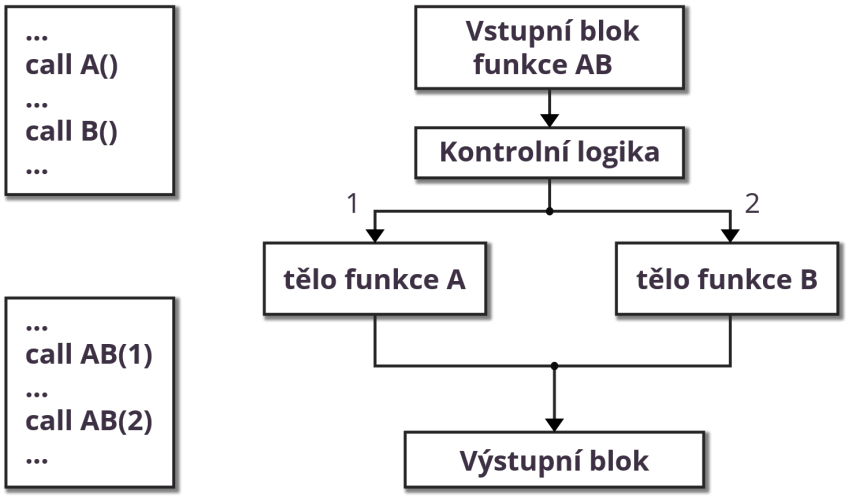
Vkládání „mrtvého“ nebo nepodstatného kódu je jedna ze základních obfuskací pro řízení toku kódem, kde cílem výsledné transformace je zmást popřípadě demotivovat reverzního inženýra. Jde o to, že do kódu jsou přidány



Obrázek 3.2: Vkládání „mrtvého“ a nepodstatného kódu. Vlevo je původní funkce. Vpravo je funkce po obfuskaci. A,B,C,D představují základní bloky původní funkce. E reprezentuje blok nepodstatného kódu a F blok „mrtvého“ kódu.

nadbytečné části kódu. Tyto části kódu se buďto vůbec nevykonají nebo vykonají, ale s původním programem nemají nic společného. Kód, který se nikdy nevykoná se nazývá „mrtvý“ kód.

Nepodstatný nebo „mrtvý“ kód je do programu vkládán spolu s *OP*. Při konstrukci *OP* je předem známo, na kterou část kódu se bude skákat. Do té větve, která se vždy provede bude vložen původní *BB*, naopak do té druhé budeme vkládat *BB* obsahující „mrtvý“ kód. Do programu lze vkládat i malé části nepodstatného kódu. Na rozdíl od „mrtvého“ kódu, nepodstatný nelze náhodně generovat, ten musíme připravit dopředu. Jedná se o kód, který se sice provede, ale výsledek nemá žádný vliv jak na chování programu, tak ani na data. Opět jde o snahu zvětšit námahu reverznímu inženýrovi při analýze. Nejde pouze o to, že bude muset analyzovat více kódu, ale i o to, že bude muset přemýšlet, jak tato část kódu souvisí s ostatním kódem. Příkladem nepodstatného kódu můžou být i samotné *OP*, které nemají na vykonávání programu žádný vliv.



Obrázek 3.3: Prokládání funkcí. Část vlevo nahoře reprezentuje původní kód a část vlevo dole obfuskovaný kód. Vpravo je funkce AB, do které jsou sloučeny funkce A a B. V obfuskovaném kódu jsou původní volání nahrazena voláními na novou funkci AB.

Cena těchto transformací se odvíjí od jejich četnosti a od stupně zanoření ve funkcích. Naopak potence a odolnost od kvality vkládaných *OP*, a také od jejich rozmanitosti. Pokud by totiž byl v programu pouze jeden typ *OP*, tak by byl brzy odhalen a následně eliminován. Proto zavedeme několik druhů *OP*. Jednotlivé druhy budou náhodně vkládány spolu s „mrtvým“ nebo nepodstatným kódem. Podobně jako u *OP*, zavedeme více způsobů, jak nový kód vkládat. Tím lze také oddálit případné odhalení.

Potence a odolnost tady spolu úzce souvisí, protože čím více *OP* budeme vkládat, tím je cyklometrická komplexita programu větší, ale zároveň se zvyšuje i šance na odhalení *OP*. Cena zase závisí na počtu vkládaných *OP*, ale do jisté míry i na tom, jestli vkládáme více nepodstatného nebo více „mrtvého“ kódu. Proto zavedeme pro tuto transformaci škálovatelné parametry, kterými lze určit vlastnosti výsledného kódu. Za prvé, parametr reprezentující poměr mezi nově vkládaným kódem a původním kódem, a za druhé parametr určující poměr mezi „mrtvým“ a nepodstatným kódem.

3.3 Prokládání funkcí

Tato obfuskace slučuje obsah několika různých funkcí do jedné, a nahrazuje volání původních funkcí, voláním výsledné funkce. Výslednou funkci budeme dále nazývat interpretem funkcí, jelikož interpretuje chování některé

z původních funkcí. Každá původní funkce sama o sobě nese určitou míru abstrakce a představuje nějakou dílčí funkcionalitu. Tím, že sloučíme více funkcí do jedné výsledné, tuto abstrakci „rozbijeme“.

Pro vykonávání původních funkcionalit musíme v interpret funkci zavést kontrolní logiku, která bude určovat, jaká část kódu se má vykonat. Tuto kontrolní logiku bude reprezentovat *switch* prvek. Také je potřeba rozlišit jednotlivá volání interpret funkce podle toho, jakou původní funkcionalitu chceme provést. To budeme realizovat tak, že původním funkcím přiřadíme unikátní identifikátory. Ty budou předávány jako dodatečný argument interpret funkce, a následně zpracovány kontrolní logikou.

Do interpret funkce jsou spolu s identifikátory předávány i původní argumenty funkcí. Problémem je to, že sloučené funkce nemusí mít stejný počet ani typ parametrů. Toto lze v LLVM řešit pomocí variabilních argumentů. Podobný problémem je i s různými typy návratových hodnot. Řešením může být například slučování funkcí se stejnou návratovou hodnotou, anebo vracení hodnot pomocí univerzálního ukazatele.

Dále pro tuto transformaci zavedeme škálovatelný parametr, kterým lze určit počet interpret funkcí. Tím způsobíme, že původní množina funkcí je rozdělena mezi více interpret funkcí.

Potence a odolnost této transformace závisí na složitosti kontrolní logiky a na tom, jak jsou „původní“ funkce od sebe odlišitelné.

Kapitola 4

Implementace

V této kapitole se zaměříme na implementaci námi navržených transformací. Popíšeme jednotlivé fáze obfuskačních modulů a ukážeme, jaký dopad mají transformace na výsledný kód. Dále popíšeme problémy, na které jsme při tvorbě modulů narazili. Pro implementaci obfuskačních technik budeme používat již zmíněný LLVM framework.

4.1 Převod funkcí do tabulkové formy

Na začátku implementace musíme zvolit jednu ze tříd, která nám umožní vytvořit obfuskační modul. Jelikož u této techniky probíhají veškeré transformace na úrovni celých funkcích, zvolili jsme třídu *FunctionPass*. Jednotlivé funkce jsou tedy transformovány nezávisle na ostatních.

```
class TableInterpretation : public FunctionPass {
public:
    bool runOnFunction(Function &F) override {
        if (!isFunctionEligible(&F))
            return false;
        convertToTableForm(&F);
        repairFunction(&F);
        return true;
    }
    // ...
};
```

Listing 4.1: Základní podoba implementace modulu pro převod funkcí do tabulkové formy. Ve funkci *runOnFunction* jsou pomocí pseudokódu nastíněny jednotlivé fáze obfuskačního modulu.

Při obfuskaci bereme ohled na to, které funkce budou transformovány. Do *runOnFunction* funkce jsou totiž předávány i deklarace funkcí. Ty nemají žádné základní bloky. Dále do transformace nezahrnujeme funkce, které ob-

sahují méně než-li dvě instrukce skoku. Pro takovéto funkce je transformace zbytečná. Všechny ostatní jsou převedeny do tabulkové formy.

Nejdříve rozdělíme *BB* na menší části a následně identifikujeme ty, které budeme transformovat. Jde o základní bloky zakončené skokem na jiný blok. V LLVM IR existuje mnoho koncových instrukcí, ale pro účely této transformace jsou vhodné pouze *BranchInst* instrukce reprezentující jak podmíněné, tak nepodmíněné skoky. Identifikované *BB* ukládáme do tabulky v podobě *ConstantVector* třídy a zároveň si uchováváme indexy těchto *BB*.

Dále je do funkce vkládána kontrolní logika. Ta je realizována pomocí proměnné *next* a nepřímého skoku. Do proměnné je ukládán index následujícího *BB*, který má být vykonán.

Když máme identifikované *BB* a do funkce je vložena kontrolní logika, následuje samotný převod hierarchie bloků na tabulkovou formu. Do každého bloku je vkládán výraz, kterým aktualizujeme proměnnou *next*. Pro *BB* obsahující nepodmíněný skok pouze uložíme index následujícího *BB* do proměnné *next*. U *BB* s nepodmíněnými skoky, musíme nejdříve vyhodnotit podmínku, abychom věděli, který index do proměnné *next* uložit. Pro tyto účely je v LLVM IR instrukce *SelectInst*.

Pro předávaný index jsme zavedli maskování v podobě *xor* operace. Do proměnné *next* je vkládána již maskovaná hodnota a při načtení proměnné v kontrolní logice je hodnota zpátky demaskována. Pro každou funkci je vygenerována náhodná maskovací hodnota.

Na závěr této transformace musíme funkci upravit tak, aby odpovídala pravidlům LLVM IR. Jelikož je LLVM IR založena na SSA, musí být každá proměnná vnitřní reprezentace před samotným použitím definována. Převodem funkce do tabulkové formy je toto pravidlo porušeno. Všechny proměnné vyskytující se ve více základních blocích musíme přesunout do vstupního bloku. Pro přesun používáme funkci *DemoteRegToStack*, která ho provede za nás.

4.2 Vkládání „mrtvého“ nebo nepodstatného kódu

Stejně jako u předchozího modulu i zde si musíme vybrat vhodnou třídu, která nám umožní vytvořit obfuskační modul. Pokud budeme uvažovat vkládání „mrtvého“ kódu, tak to je vlastně přidávání nových základních bloků. Proto nejvhodnější třídou je *FunctionPass*. Jednotlivé transformace budou prováděny nad celými funkcemi. Po zdědění třídy implementujeme její funkce, a to konkrétně *doInitialization*, kterou použijeme k inicializaci proměnných ještě před samotnými transformacemi a *runOnFunction*, ve které jsou jednotlivé transformace prováděny.

```
class DeadCode : public FunctionPass {
public:
    bool runOnFunction(Function &F) override {
        splitBBs(&F);
        set<BasicBlock*> toObf = analyzeFunction(&F);
        if (toObf.empty())
            return false;
        insertDeadOrIrrelevantCode(&F);
        return true;
    }
    // ...
};
```

Listing 4.2: Základní podoba implementace modulu pro vkládání „mrtvého“ nebo nepodstatného kódu. Ve funkci *runOnFunction* jsou pomocí pseudokódu nastíněny jednotlivé fáze obfuskačního modulu.

Nástroje pro reverzní analýzu mohou provádět některé drobné optimalizace. Jedná se například o různé druhy *peephole* optimalizací, které dokáží vyhodnotit a zjednodušit lokální části kódu. Proto budeme do vkládaných částí kódu zavádět globální proměnné, které se budou vyskytovat skrze celý program. Tím můžeme v některých případech zabránit zjednodušení nebo dokonce eliminaci kódu. Tyto globální proměnné nijak neovlivňují výsledný kód, takže vícevláknový přístup nemusíme řešit.

Dalším důležitým prvkem transformace je náhodnost. Tím nejenže zamezíme vkládání částí kódů podle určitých vzorů, ale dosáhneme i toho, že každá transformace bude unikátní.

Pro tyto účely jsme použili *Mersenne Twister* psudo-náhodný generátor v podobě *std::mt19937* třídy ze standardní *c++* knihovny *<random>*. U operací, které jsou založeny na náhodnosti (například volba transformace nebo volba vkládaného nepodstatného kódu), jsme také zavedli hlídání rovnoměrného rozdělení v podobě *std::uniform_int_distribution* tak, aby se jednotlivé typy vyskytovaly přibližně stejně. Toto je vidět spíše u větších (ve smyslu počtu základních bloků) funkcí. Na malé funkce to skoro nemá vliv.

Protože uvažujeme transformace na úrovni jednotlivých základních bloků, musíme zařídit, aby nové části kódu nebyly vkládány pouze mezi původní základní bloky. Proto nejprve před samotnou transformací rozdělíme původní bloky na menší podbloky, což nám umožní vkládat náš kód i dovnitř původních bloků. Pro toto jsme vhodně využili funkci *splitBasicBlock* třídy *BasicBlock*, která rozděluje základní blok na dvě části a do prvního bloku na konec vkládá skok na druhý blok. Při rozdělování si musíme dát pozor, abychom zachovali *phi* instrukce v původním bloku.

Dále jsme pro tuto transformaci zavedli následující parametrizace:

-dc-new-ratio <uint> udává procentuální poměr mezi nově přidanými („mrtvými“ nebo nepodstatnými) a původními základními bloky. Čím je hodnota blíže k 100, tím se transformace uplatňují na větší počet původních *BB*.

-dc-irr-ratio <uint> udává procentuální poměr mezi „mrtvým“ a nepodstatným kódem, který je do programu vkládán. Tento parametr vyjadřuje, který z těchto dvou druhů transformací se má více uplatňovat. Čím více se hodnota blíží k 100, tím více se vkládá „mrtvého“ kódu a naopak.

Pro samotné vkládání kódu musíme nejdříve určit, na kterých základních blocích funkce budeme provádět transformace. Počet těchto bloků je dán rovnicí (4.1).

$$n = \frac{N * r}{100}, \tag{4.1}$$

kde n je počet bloků k obfuskaci, N je počet původních bloků ve funkci a r je poměr daný parametrem *dc-new-ratio*. Když máme určený počet, náhodně vybereme některé bloky.

Při vkládání kódu je třeba dát pozor na tzv. *landingpad* instrukce, které se používají při odchytávání výjimek v kódu. Tyto instrukce musí být v *BB* umístěny jako první, takže nelze vkládat kód před tyto instrukce.

Další instrukcí, na kterou si dát pozor, je *invoke*. Ta slouží k volání funkcí s registrovanou obsluhou výjimek. Jde o koncovou instrukci, která se vyskytuje až na konci základního bloku. Po této instrukci nelze v daném *BB* vkládat další instrukce.

Důležitou je i *phi* instrukce. Pokud se více základních bloků sbíhá do jednoho, pak můžeme chtít dále pracovat s některou hodnotou z předešlých bloků. Jelikož ale nevíme, který blok bude vykonán, tak nelze rozhodnout, kterou hodnotu z jakého bloku použít. Kvůli tomuto LLVM IR obsahuje instrukci *phi*, která je vkládána na počátek bloku, kde se ostatní bloky scházejí. V této instrukci je vždy „zaregistrována“ dvojice (blok, proměnná), která reprezentuje hodnotu z daného bloku. Podle toho který blok byl vykonáván předtím, tak podle toho budeme uvažovat proměnnou (z této dvojice) pro další práci. Tato schopnost má však i svoje úskalí. Sice vkládáme „mrtvý“ kód do původního programu, ale i tento kód musí být správně strukturovaný podle pravidel LLVM IR. V *phi* instrukci mohou být obsaženy pouze přímo předcházejících bloky. Pokud tedy vkládáme nový blok, musíme aktualizovat příslušné *phi* instrukce následujícího bloku.

Generování „mrtvého“ kódu

Dříve než představíme jednotlivé způsoby vkládání kódu, musíme představit kód, který se bude vkládat. V případě „mrtvého“ kódu se bude jednat o sekvenci binárních operací. Nejprve ale musíme zajistit prvky, které budou vkládány jako operandy do jednotlivých operací. Množina těchto prvků obsahuje výše zmíněné globální proměnné a náhodně generované konstanty. Navíc všechny binární operace budou do sebe zřetězené, takže výsledek z první operace bude operandem pro druhou. Nakonec výsledek ukládáme do jedné z globálních proměnných. Pro generování tohoto kódu jsme použili instrukce *add*, *sub*, *mul*, *xor*, *or*, *and*, *shl* a *lshr*. Jednotlivé instrukce jsou vybírány náhodně.

```
%g_8 = load i32, i32* @glob.8
%g_2 = load i32, i32* @glob.2
%lshr_0 = lshr i32 %g_8, %g_2
%xor_0 = xor i32 %lshr_0, 25
%mul_0 = mul i32 %xor_0, 91
store i32 %mul_0, i32* @glob_.6
```

Listing 4.3: Ukázka „mrtvého“ kódu v LLVM IR.

Generování nepodstatného kódu

Nevýhodou u tohoto kódu je to, že se bude moci vykonávat, takže nelze kód jednoduše generovat jako v předešlém případě. Proto jsme připravili pár krátkých kusů kódu. Jedná se o jednoduché geometrické výpočty, konkrétně pro *objem krychle*, *povrch krychle*, *objem kvádru*, *povrch kvádru*, *obsah lichoběžníku* a *obvod lichoběžníku*. To že jsme vybrali tyto vzorce, nemá žádný hlubší důvod. Podobně jako v předchozím případě, jsou parametry vybírány z množiny globálních proměnných a celočíselných konstant. Výsledek výpočtu je uložen zpět do náhodně vybrané globální proměnné. Jelikož by mohlo dojít k přetečení hodnoty některé z globálních proměnných, je číslo vždy před použitím omezeno na určitý počet bitů (například *int b = (g & 1023)*, tak *b* je omezeno na maximální hodnotu 1023).

```
%g_3 = load i32, i32* @glob_.3
%a = and i32 %g_3, 16383      ;omezeni hodnoty
%c = and i32 82, 16383        ;omezeni hodnoty
%h = and i32 35, 16383        ;omezeni hodnoty
%0 = add i32 %a, %c           ;a+c
%1 = ashr i32 %0, 1           ;(a+c)/2
%s = mul i32 %1, %h           ;s = ((a+c)/2)*h
store i32 %s, i32* @glob_.4
```

Listing 4.4: Ukázka nepodstatného kódu v LLVM IR. Konkrétně výpočet obsahu lichoběžníku.

Způsoby vkládání kódu

Dále si představíme způsoby, jak je kód vkládán do programu. Pokud by existoval pouze jediný způsob, jak kód vkládat, tak by ho reverzní inženýr po krátkém čase odhalil a následně kód odstranil. Způsobem vkládání kódu je myšleno to, jak nový blok navazuje na (popřípadě předchází) původní bloky. Při vkládání musíme brát ohled na sémantiku LLVM IR. Kód musí být platný, i když se nikdy nevykoná. Dále jsou jednotlivé způsoby uvedeny tak, jak jsme je pojmenovali ve zdrojových kódech:

ADD_NEXT jde o nejednodušší typ transformace. Za určitý původní blok je přidán nový, obsahující jeden z typů nepodstatného kódu (geometrické výpočty). Pokud je k obfuskaci vybrán vstupní nebo výstupní blok nějaké funkce, tak je na něj automaticky použita právě tato transformace.

DEAD_BEFORE tento typ zahrnuje i vkládání *OP*. Před původním základním blokem je vložen podmíněný výraz, který určuje jestli se má vykonat originální nebo nově generovaný „mrtvý“ blok. Tento podmíněný výraz je vytvořen tak, aby byla provedena vždy větev s původním kódem. Jelikož i „mrtvý“ kód musím mít určitou strukturu, tak na konec tohoto bloku je přidán skok na původní blok.

IRR_BEFORE je stejný jako *DEAD_BEFORE* akorát na místo náhodně generovaného „mrtvého“ bloku je přidán nějaký typ nepodstatného. V tomto případě nezáleží na výsledku podmíněného výrazu, jelikož se původní kód vykoná vždy.

DEAD_INSIDE tento typ transformace na rozdíl od *DEAD_BEFORE* vkládá nový kód do původního bloku. Nejdříve se původní blok rozdělí na dvě části, kde na konci první části je skok na druhou část. My tento skok nahradíme podmíněným výrazem, který jakožto druhou větev obsahuje „mrtvý“ blok (první zůstává druhá část původního bloku). Podobně jako u *DEAD_BEFORE* je na konci „mrtvého“ kódu vložen skok na druhou část původního bloku.

IRR_INSIDE je stejný jako *DEAD_INSIDE*, ale opět se jedná o nepodstatný kód namísto „mrtvého“.

LOOP_BACK jde o velice podobnou transformaci jako *DEAD_INSIDE*, ale s tím rozdílem, že na konci „mrtvého“ kódu je vložen skok na začátek první části původního bloku namísto na druhou část.

Vkládané *OP*

Podobně jako u vkládání kódu, i tady uvažujeme více typů, a to ze stejného důvodu (k odhalení dojde později). Všechny parametry pro dané výrazy jsou

opět generovány náhodně v určitém rozsahu tak, aby nedošlo k přetečení. Výrazy jsou tedy různé, nemělo by tak docházet k častému opakování. Jednotlivé typy výrazů jsme opět pojmenovali tak, jako ve zdrojových kódech:

MUL jedná se o výraz $(2a)\&1 \neq 1$, kde a je z rozsahu 1 až 100000. Je postaven na vlastnosti sudých čísel.

SHIFT jedná se o výraz $(2^a)\&1 \neq 1$, kde a je z rozsahu 1 až 30. Je postaven na stejném principu jako předchozí výraz.

PRIME jedná se o výraz $2^{p-1} \equiv 1 \pmod{p}$, kde $p \in \{5, 7, 11, 13, 17, 19, 23, 29\}$. Výraz je založen na Malé Fermatově větě.

AND jedná se o výraz $(2^a - 1)\&1 = 1$, kde a je z rozsahu 1 až 30. Jde o to, že výraz $2^a - 1$ bude v binární reprezentaci vypadat takto (01^{a-1}), což vždy splní náš výraz výše.

MOD_3 jedná se o výraz $a^3 - a \equiv 0 \pmod{3}$, kde a je z rozsahu 2 až 1000. Je postaven na dělitelnosti třemi [9].

MOD_4 jedná se o výraz $a^2(a+1)^2 \equiv 0 \pmod{4}$, kde a je z rozsahu 2 až 1000. Je postaven na dělitelnosti čtyřmi [4].

4.3 Prokládání funkcí

Jelikož při této transformaci vytváříme nové funkce, musíme pro tvorbu tohoto modulu použít třídu *ModulePass*. Transformace jsou zde prováděny nad celým modulem. Po zdědění třídy implementujeme funkci *runOnModule*, ve které jsou transformace prováděny.

```
class MethodInterleaving : public ModulePass {
public:
    bool runOnModule(Module &M) override {
        set<Function*> toObf = analyzeModuleFunctions(&M);
        if (toObf.empty())
            return false;
        createInterleavedFunction(&M);
        replaceFunctionCalls();
        return true;
    }
    // ...
};
```

Listing 4.5: Základní podoba implementace modulu pro prokládání funkcí. Ve funkci *runOnModule* jsou pomocí pseudokódu nastíněny jednotlivé fáze obfuskačního modulu.

Na začátku transformace zaznamenáme všechna volání funkcí, které se v programu vyskytují. Každé volání funkce (jakožto prvek LLVM IR) je reprezentováno třídou *CallInst* nebo *InvokeInst*. Rozdíl mezi těmito dvěma je v tom, že *InvokeInst* je volání s registrovanou obsluhou výjimek. Pokud je ve volané funkci vyvolána výjimka, je zavolána její příslušná obsluha. Spolu s voláními jsou zaznamenány i samotné funkce. Všem těmto funkcím je zároveň přiřazen unikátní identifikátor. Pokud žádné volání funkcí nalezeno nebylo, je modul ukončen, a vstupní program zůstává nezměněn.

Počet interpret funkcí, které budou slučovat množiny původních funkcí je uživatelsky určen pomocí následujícího parametru:

-mi-num <uint> udává počet interpret funkcí. Pokud je hodnota parametru větší než polovina původních funkcí, je tento parametr omezen tak, aby každá interpret funkce slučovala minimálně dvě původní. Výchozí hodnota parametru je 1.

Vytvoření interpret funkce

Když máme analyzovaný program, můžeme přejít k vytváření interpret funkcí. Počet interpretů je dán parametrem *mi-num*. Pokud jich je více než jeden, musíme rozdělit množinu analyzovaných funkcí. Rozdělování probíhá rovnoměrně, takže každý interpret bude obsahovat přibližně stejný počet funkcí.

Při vytváření interpret funkcí nastává problém, pokud původní funkce předávají různý počet nebo typ argumentů. Jelikož LLVM disponuje řadou vestavěných funkcí, tak jsme tento problém vyřešili pomocí variabilního předávání argumentů. Tato schopnost je převzatá z programovacího jazyka *C*. Konkrétně se jedná o vestavěné funkce *llvm.va_start* a *llvm.va_end*, které předávání umožňují.

Dalším problémem při tvorbě interpret funkce je to, že každá původní funkce může vracet různé typy hodnot. Proto návratové hodnoty původních funkcí ukládáme do paměti, odkud si je po návratu z interpret funkce vyzvedáváme.

Každá interpret funkce obsahuje kontrolní logiku v podobě *switch* instrukce, kde jednotlivé větve této instrukce obsahují původní funkce. Původní funkce jsou do těchto větví *inlinovány* pomocí LLVM funkce *InlineFunction*.

Nahrazení volání funkcí

Když máme vytvořené příslušné interpret funkce, zbývá nahradit původní volání funkcí voláním interpretu (popř. interpretů) s příslušnými parametry. Nejdříve před každým voláním vytvoříme proměnnou *uid*, do které uložíme unikátní identifikátor příslušné volané funkce. Následně vytvoříme nové volání interpret funkce, kde je jako argument předána *uid* proměnná a všechny argumenty původní funkce. Dále je zajištěno vyzvednutí návratové hodnoty z paměti. Nakonec je původní volání nahrazeno voláním interpret funkce.

Kapitola 5

Testování

V této kapitole změříme a zhodnotíme námi implementované techniky. Pro hodnocení obfuskačních modulů budeme používat metriky představené v úvodní kapitole. Jde o potenci, odolnost a cenu. Nakonec porovnáme naše implementované řešení s *Obfuscator-LLVM*, který je také postaven na LLVM frameworku.

5.1 Postup měření

Potenci budeme měřit pomocí metrik softwarové komplexity. Konkrétně budeme uvažovat délku a cyklomatickou komplexitu programu. K měření použijeme nástroj *obfuscation-metrics* [20], který umožňuje měřit kód v LLVM IR formě. Dále budeme diskutovat, jak obfuskace „znepřehledňuje“ program. Na základě „znepřehlednění“ programu a změřených metrik následně určíme úroveň potence.

Odolnost transformace jsme definovali jako kombinaci dvou veličin. První z nich je snaha vynaložená na výrobu deobfuskátoru, druhá je výpočetní složitost deobfuskátoru. Jelikož snaha na výrobu deobfuskátoru je subjektivní, nahrazujeme jí rozsahem obfuskační transformace. Platí, že čím větší rozsah obfuskace, tím je nutné vynaložit větší snahu na výrobu deobfuskátoru. Výpočetní složitost deobfuskátoru nejsme schopni určit. Pro hodnocení odolnosti budeme používat úrovně definované v kapitole 1.4.

Cena obfuskace představuje dodatečnou režii na vykonání obfuskovaného programu. My budeme měřit čas výpočtu obfuskovaného programu v závislosti na vstupu. Výsledek měření budeme hodnotit podle úrovní definovaných v kapitole 1.4.

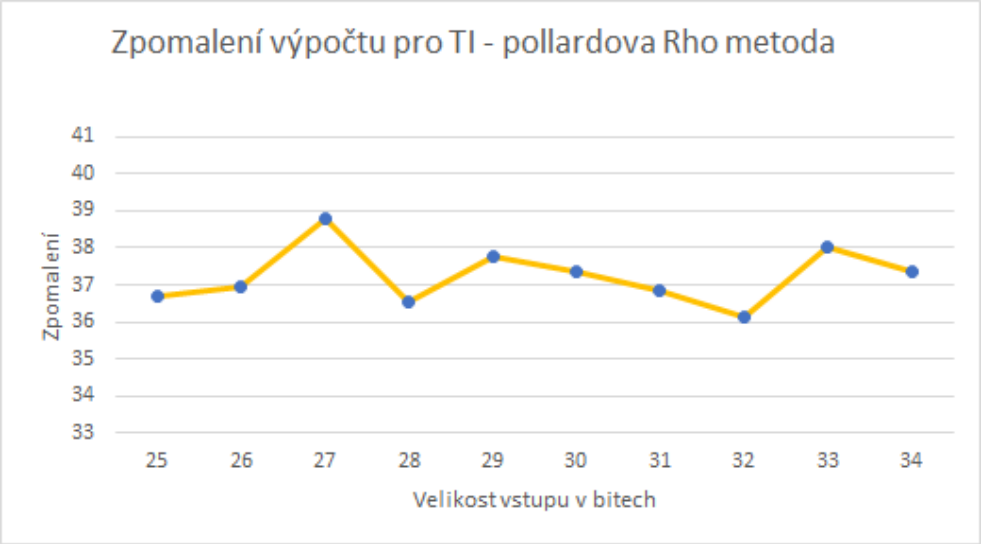
Veškeré měření budeme provádět na předem připravených programech. Konkrétně se jedná o řadící algoritmy a algoritmus pro řešení diskrétního logaritmu (*Pollardova Rho* metoda). Tyto programy jsou přiloženy v příloze práce.

Obfuskace	Parametr	DLP	bubbleSort	quickSort	mergeSort
TI	—	3.70	3.15	3.61	3.28
DC	dc-new-ratio=10	2.21	2.37	2.23	2.35
	dc-new-ratio=20	2.75	3.08	2.77	3.07
	dc-new-ratio=30	3.44	3.72	3.35	3.56
	dc-new-ratio=40	4.07	4.33	3.85	4.35
	dc-new-ratio=50	4.96	4.95	4.78	5.03
	dc-new-ratio=60	5.71	5.75	5.62	5.62
	dc-new-ratio=70	6.09	6.19	6.31	6.74
	dc-new-ratio=80	6.32	7.75	7.06	7.66
	dc-new-ratio=90	6.73	8.34	7.53	8.14
	dc-new-ratio=100	7.18	8.75	8.38	8.82
MI	mi-num=1	2.17	2.59	2.64	2.50
	mi-num=2	2.18	2.65	2.68	2.53
	mi-num=3	2.20	2.73	2.72	2.57

Tabulka 5.1: Průměrná změna délky programu. Hodnoty v tabulce udávají poměr mezi obfuskovaným a původním programem. TI = převod do tabulkové formy, DC = vkládání „mrtvého“ kódu, MI = prokládání funkcí, DLP = zkušební program pro výpočet diskrétního logaritmu.

Obfuskace	Parametr	DLP	bubbleSort	quickSort	mergeSort
TI	—	9.06	3.54	3.75	3.82
DC	dc-new-ratio=10	1.90	1.37	1.36	1.44
	dc-new-ratio=20	2.53	1.75	1.64	1.72
	dc-new-ratio=30	3.38	1.91	1.92	2.11
	dc-new-ratio=40	4.12	2.28	2.19	2.53
	dc-new-ratio=50	5.26	2.45	2.47	2.96
	dc-new-ratio=60	6.33	2.84	2.79	3.15
	dc-new-ratio=70	6.54	2.97	3.12	3.54
	dc-new-ratio=80	7.09	3.46	3.35	4.01
	dc-new-ratio=90	7.73	3.77	3.60	4.36
	dc-new-ratio=100	8.04	4.10	3.91	4.92
MI	mi-num=1	1.95	2.02	1.98	1.96
	mi-num=2	1.98	2.04	2.04	2.01
	mi-num=3	2.01	2.06	2.07	2.03

Tabulka 5.2: Průměrná změna cyklomatické komplexity programu. Hodnoty v tabulce udávají poměr mezi obfuskovaným a původním programem. TI = převod do tabulkové formy, DC = vkládání „mrtvého“ kódu, MI = prokládání funkcí, DLP = zkušební program pro výpočet diskrétního logaritmu.



Obrázek 5.1: Průměrné zpomalení výpočtu pro TI obfuskaci. TI = převod funkcí do tabulkové formy. Jednotlivé hodnoty představují poměr mezi výpočetním časem obfuskovaného programu a výpočetním časem původního programu. Program řeší problém diskrétního logaritmu.

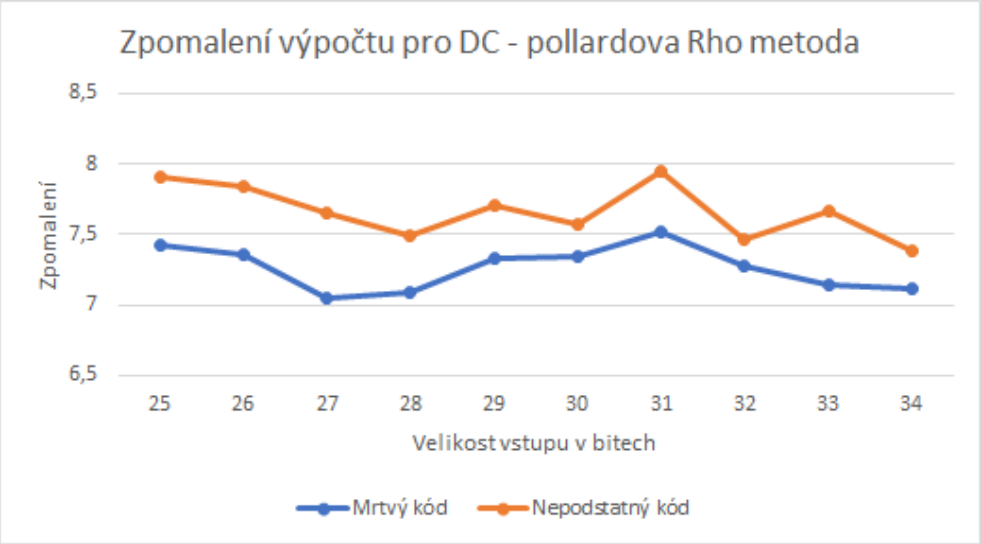
5.2 Převod funkcí do tabulkové formy

Z tabulky 5.2 lze vidět, že tato transformace má vliv na komplexitu programu. To je dáno především tím, že před samotnou transformací jsou *BB* rozděleny na menší části.

Pro reverzního inženýra je „znepřehlednění“ obfuskovaného programu velké. Transformací skrýváme informaci o tom, jaký vztah mají jednotlivé *BB* mezi sebou, a jak jsou postupně vykonávány [19]. Analýzu jsme ještě ztížili o maskování předávaného indexu. Proto hodnotíme potenci obfuskace jako *střední*.

Odolnost této transformace hodnotíme jako *slabá*, jelikož nejsou vkládány žádné *OP*. Na druhou stranu do výsledného programu je vkládána proměnná *next* vyskytující se skrze celou funkcí. Proměnná je zároveň maskovaná pomocí *xor* operace.

Z grafu 5.1 lze vidět, že zpomalení obfuskovaného program oproti původnímu je značné. Nicméně s rostoucím vstupem je zpomalení konstantní, takže cenu obfuskace hodnotíme jako *zdarma*. Zpomalení výpočtu je způsobeno tím, že pro provedení každého *BB* je nutno vykonat i kontrolní logiku.



Obrázek 5.2: Průměrné zpomalení výpočtu pro DC obfuskaci. DC = vkládání „mrtvého“ nebo nepodstatného kódu. Jednotlivé hodnoty představují poměr mezi výpočetním časem obfuskovaného programu a výpočetním časem původního programu. Oranžová čára uvažuje vkládání pouze nepodstatného kód. Naopak modrá vkládání pouze „mrtvého“ kódu. Program řeší problém diskrétního logaritmu.

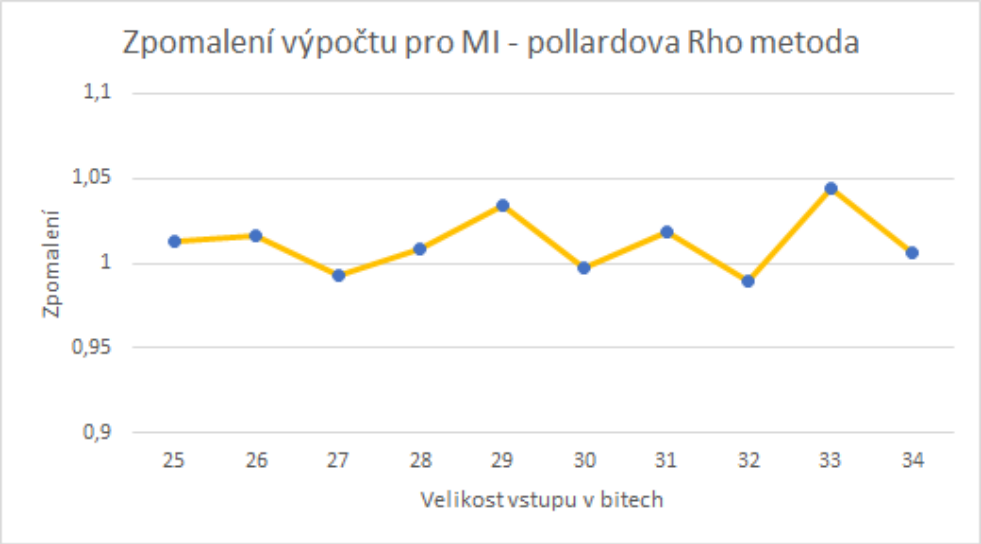
5.3 Vkládání „mrtvého“ nebo nepodstatného kódu

Odolnost vkládání „mrtvého“ kódu je dána především kvalitou *OP*. Ty jsou vkládány pouze na úrovni *BB*, takže odolnost *OP* hodnotíme jako *triviální*. Tato transformace vkládá i nepodstatný kód. Odolnost nepodstatného kódu je dána hlavně rozmanitostí jeho typů. Kód je vkládán skrze celou funkci. Jelikož transformace zahrnuje oba typy kódu, hodnotíme odolnost jako *slabou*.

Pro tuto transformaci jsme provedli měření v závislosti na parametru *dc-new-ratio*. Z naměřených hodnot lze usuzovat, že transformace má vliv jak na délku, tak i na komplexitu programu. Délka programu je závislá na množství nově vkládaného kódu, komplexita potom na množství vkládaných *OP*.

Vkládáním „mrtvého“ nebo nepodstatného kódu zvyšujeme čas potřebný pro analýzu programu. Původní funkcionalita je schována mezi nově přidané části kódu, které musí reverzní inženýr analyzovat. S ohledem na měření, hodnotíme potenci obfuskace jako *střední*.

U této obfuskace jsme mimo *dc-new-ratio* zavedli i parametr *dc-irr-ratio*. Ten udává poměr mezi „mrtvým“ a nepodstatným kódem. Pokud se hodnota parametru blíží ke 100, je vkládán více „mrtvý“ kód. Naopak pokud se hodnota blíží k 0, tak je vkládán více nepodstatný kód. V grafu 5.2 lze vidět, jaký



Obrázek 5.3: Průměrné zpomalení výpočtu pro MI obfuskaci. MI = prokládání funkcí. Jednotlivé hodnoty představují poměr mezi výpočetním časem obfuskovaného a původního programu. V tomto případě jsou programy „stejně“ rychlé. Program řeší problém diskrétního logaritmu.

vliv má parametr *dc-irr-ratio* na zpomalení obfuskovaného programu. Z grafu je zároveň vidět, že zpomalení je s rostoucím vstupem konstantní. Cenu této obfuskace hodnotíme jako *zdarma*.

5.4 Prokládání funkcí

Měření pro tuto transformaci bylo provedeno v závislosti na parametru *minum*. Podle naměřených hodnot lze usuzovat, že parametr nemá vliv ani na délku, ani na komplexitu programu. Hodnoty u samotné obfuskace nepředstavují žádné výrazné navýšení metrik.

Pro reverzního inženýra je „znepřehlednění“ obfuskovaného programu dáno tím, že původní funkce jsou sloučeny do jedné výsledné. Míra „znepřehlednění“ programu také závisí na složitosti kontrolní logiky. My jsme tuto logiku implementovali pomocí *switch* instrukce. Vzhledem k naměřeným hodnotám a implementaci kontrolní logiky hodnotíme potenci jako *nízkou*.

Odolnost hodnotíme jako *slabou*. Transformace sice zahrnuje několik funkcí najednou, ale kontrolní logika je jednoduchá. Pro zvýšení potence a odolnosti je nutné implementovat složitější kontrolní logiku například v podobě *OP*. Dále by bylo vhodné prokládat funkce na úrovní *BB*.

Cena obfuskace je *zdarma*. Z grafu 5.3 vidíme, že zpomalení obfuskovaného programu oproti původnímu je nulové, a zároveň se zvyšujícím se vstupem i konstantní.

5.5 Porovnání s *Obfuscator-LLVM*

V úvodní kapitole jsme představili *Obfuscator-LLVM* a jeho obfuskační techniky. Tento nástroj umožňuje vkládání „mrtvého“ kódu, převod funkce do tabulkové formy a substituci instrukcí. Tyto obfuskace budeme dále porovnávat s námi implementovanými moduly.

Vkládání „mrtvého“ kódu: Transformace probíhá na úrovni základních bloků. Uvažovaný *BB* je zkopírován a doplněn o náhodně generované instrukce. Tento zkopírovaný blok představuje „mrtvý“ kód. Dále je před původní blok vložen nový *BB* obsahující *OP*. Podmíněný skok je vždy vyhodnocen ve prospěch původního *BB*. V této implementaci je vkládán pouze jeden typ *OP*. Konkrétně jde o podmíněný výraz $(x * (x + 1))\%2 == 0$. Podobně i kód je vkládán jedním způsobem.

Náš obfuskační modul oproti tomuto obsahuje více způsobů jak kód vkládat, a používá více různých typů *OP*. Zároveň vkládá i nepodstatný kód. U obou obfuskačních modulů lze nastavit množství vkládaného kódu.

Převod funkce do tabulkové formy: V této implementaci je kontrolní logika realizována pomocí *switch* prvku, který rozhoduje, jaký další *BB* má být vykonán. Hodnota následujícího *BB* je předávána v proměnné.

Náš obfuskátor implementuje kontrolní logiku pomocí nepřímého skoku. Informace o následujícím *BB* je předávána také v proměnné, ale tato hodnota je navíc maskována. Zároveň jsme zavedli rozdělování bloků na menší části.

Tento obfuskátor oproti našemu dále implementuje substituci instrukcí. Ta nahrazuje standardní binární operace za složitější, funkčně ekvivalentní, výrazy.

Co se týče vkládání „mrtvého“ kódu a převodu funkcí do tabulkové formy, tak tvrdíme, že naše implementace jsou o něco propracovanější.

Na druhou stranu, výhodou tohoto obfuskátoru oproti našemu, je možnost vybrat, která funkce bude obfuskována. To je realizováno pomocí anotace funkcí ve zdrojovém kódu.

5.6 Zhodnocení

U implementovaných modulů jsme hodnotili potenci, odolnost a nakonec cenu. Pro zvýšení potence a odolnosti u všech obfuskací by bylo vhodné implementovat složitější *OP*. Dále jsme porovnali implementaci našeho modulu s implementací *Obfuscator-LLVM*. Pro vkládání „mrtvého“ kódu a pro převod funkcí do tabulkové formy tvrdíme, že naše implementace jsou propracovanější. Naopak *Obfuscator-LLVM* obsahuje některé techniky, které v našem obfuskátoru nejsou.

Závěr

Cílem této práce bylo seznámit se s taxonomii obfuskačních transformací a LLVM frameworkem. Na tomto základě jsme měli navrhnout a implementovat obfuskační transformace jakožto LLVM moduly. Dále jsme měli tyto moduly vyhodnotit a výsledky porovnat s obdobnými obfuskátory.

V úvodu práce jsme se zabývali pojmem obfuskace, jakožto určitou transformací kódu. Řekli jsme také, proč takovéto transformace provádíme a jaké výhody, popřípadě nevýhody z nich plynou. Pro měření kvality těchto transformací jsme definovali standardní metriky. Některé základní typy obfuskací jsme ukázali a zhodnotili.

Následně jsme si představili projekt LLVM a uvedli jeho hlavní myšlenku (modulárnost). Potom stručně popsali základní strukturu tohoto překladače, a uvedli jednotlivé fáze překladu. Jelikož obfuskace jsou podobné optimalizacím, dále jsme se věnovali střední části překladače, konkrétně vnitřní reprezentaci kódu (LLVM IR). Představili jsme základní nástroje pro práci s vnitřní reprezentací a uvedli strukturu a základní prvky tohoto jazyka. Důležitou částí byla i tvorba optimalizačních modulů, kde bylo ukázáno jak používat aplikační rozhraní, které LLVM poskytuje. Na konci této kapitoly jsme stručně představili LLVM moduly pro zápis výsledného strojově závislého kódu.

Navrhli jsme tři obfuskační transformace. První z nich, vkládání „mrtvého“ kódu, přidává do původního programu nadbytečný kód, a tím prodlužuje případnou analýzu. Druhá z nich, převod funkcí do tabulkové formy, převádí hierarchii základních bloků na množinu bloků, která je volána nově přidanou kontrolní logikou. Poslední z nich, prokládání funkcí, slučuje funkcionalitu původních funkcí do nově vytvořené interpret funkce.

V implementační části jsme představili implementované techniky a uvedli problémy spojené s implementací. Dále jsme pomocí standardních metrik vyhodnotili a diskutovali námi implementované moduly. Na konec jsme porovnali náš obfuskátor s nástrojem *Obfuscator-LLVM*.

Příloha A

Seznam použitých zkratek

API Application Programming Interface

AST Abstract syntax tree

BB Basic Block

COFF Common Object File Format

DAG Directed acyclic graph

ELF Executable and Linkable Format

JIT Just In Time

OP Opaque Predicate

PE Portable Executable

RISC Reduced instruction set

SSA Static single assignment

Příloha B

Návod na použití

Zde jsou popsány jednotlivé kroky k sestavení projektu ze zdrojových kódů. Na překlad potřebujeme *C++* překladač. Dále používám *Ninja* (obdoba *unix*ového nástroje *make*) a *cmake* nástroje k automatizovanému překladu.

```
> sudo apt-get install cmake ninja-build
```

1. Nejprve potřebujeme zdrojové kódy pro *LLVM* a *Clang*.

   ```
   > git clone http://llvm.org/git/llvm.git
   > cd llvm/tools
   > git clone http://llvm.org/git/clang.git
   ```

2. Dále si připravíme adresáře, do kterých budou zdrojové soubory překládány.

   ```
   > cd ../..
   > mkdir build install
   ```

3. Začleníme obfuskační moduly do zdrojových souborů.

   ```
   + llvm/lib/Transforms/Obfuscator
   ```

4. Přidáme *add_subdirectory(Obfuscator)* do *llvm/lib/Transforms/CMakeList.txt*.

5. Nakonec přeložíme zdrojové kódy (překlad může trvat déle).

   ```
   > cd build
   > cmake ../llvm/ -G Ninja -DCMAKE_BUILD_TYPE="Release" -
       ↪ DCMAKE_INSTALL_PREFIX="../install"
   > ninja && ninja install
   ```

B.1 Použití obfuskátoru

Pro obfuskaci programu je nejprve nutné, převést zdrojový kód na LLVM IR. Následně jsou pomocí nástroje *opt* na LLVM IR kódu provedeny obfuskace. Na konec obfuskovaný kód přeložíme do spustitelné formy.

Nástroj *opt* načítá obfuskační modul (*Obfuscator.so*), a podle zadaných parametrů provádí jednotlivé obfuskační transformace. Dále jsou uvedeny parametry pro jednotlivé transformace:

-ti převod funkcí do tabulkové formy.

-dc vkládání „mrtvého“ nebo nepodstatného kódu.

-mi prokládání funkcí.

Pořadí obfuskačních transformací je dáno pořadím předávaných argumentů. Jedinou podmínkou pro pořadí argumentů je, aby převod funkcí do tabulkové formy následoval prokládání funkcí. Důvodem je, že funkce převedena do tabulkové formy obsahuje nepřímý skok, který odkazuje přímo na adresy *BB*. Při prokládání funkcí by byla tato funkce *inlinována* i s příslušnými adresami původních *BB*.

```
> clang++ -c -emit-llvm file.cpp -o file.bc
> opt -load Obfuscator.so -mi -mi-num=2 file.bc -o file_mi.bc
> opt -load Obfuscator.so -dc -ti file.bc -o file_dc_ti.bc
> clang++ file_mi.bc -o file_mi
> clang++ file_dc_ti.bc -o file_dc_ti
```

Literatura

[1] NewYork Times: Court Order on Microsoft. Dostupné z: https://www.nytimes.com/1995/03/03/business/court-order-on-microsoft.html

[2] Main, A.; van Oorschot, P.: Software Protection and Application Security: Understanding the Battleground. Technická zpráva, Cloakware Corporation, Ottawa, Canada, Computer Science, Carleton University, Ottawa, Canada, 2003.

[3] Eilam, E.: *Reversing: Secrets of Reverse Engineering*. Wiley, 2011.

[4] Collberg, C.; Low, D.; Thomborson, C.: A Taxonomy of Obfuscating Transformations. Technická zpráva, University of Auckland, 1997.

[5] Brown, A.; Wilson, G. (editoři): *The Architecture of Open Source Applications*, kapitola LLVM. lulu.com, 2014, s. 155–170. Dostupné z: http://www.aosabook.org/en/llvm.html

[6] Collberg, C.; Thomborson, C.: Watermarking, tamper-proffing, and obfuscation: tools for software protection. Technická zpráva, Department of Computer Science, University of Arizona, Department of Computer Science, University of Auckland, 2000.

[7] Collberg, C.; Thomborson, C.; Low, D.: Breaking Abstraction and Unstructuring Data Structures. Technická zpráva, Department of Computer Science, University of Auckland, 1998.

[8] Aho, V. A.; Sethi, R.; Lam, S. M.; aj.: *Compilers: Principles, Techniques, & Tools*. Pearson Education, Inc., 2006.

[9] Preda, M. D.; Giacobazzi, R.; Madou, M.; aj.: Opaque Predicates Detection by Abstract Interpretation. Technická zpráva, University of Verona, Ghent University, 2006.

[10] Junod, P.; Rinaldini, J.; Wehrli, J.; aj.: Obfuscator-LLVM – Software Protection for the Masses. In *Proceedings of the IEEE/ACM 1st International Workshop on Software Protection, SPRO'15, Firenze, Italy, May 19th, 2015*, editace B. Wyseur, IEEE, 2015, s. 3–9, doi:10.1109/SPRO.2015.10.

[11] Lattner, C.: Introduction to the LLVM Compiler System. Technická zpráva, llvm.org, 2008. Dostupné z: `http://llvm.org/pubs/2008-10-04-ACAT-LLVM-Intro.pdf`

[12] Apple Inc.: C language family frontend for LLVM. Dostupné z: `http://clang.llvm.org/`

[13] Lopes, B. C.; Auler, R.: *Getting Started with LLVM Core Libraries*. Packt Publishing, 2014.

[14] Lattner, C.; Adve, V.: *LLVM Language Reference Manual*. Dostupné z: `http://llvm.org/docs/LangRef.html`

[15] LLVM Team: *LLVM Command Guide*. Dostupné z: `http://llvm.org/docs/CommandGuide/index.html`

[16] Lattner, C.; Laskey, J.: *Writing an LLVM Pass*. Dostupné z: `http://llvm.org/docs/WritingAnLLVMPass.html`

[17] LLVM Team: *LLVM Programmer's Manual*. Dostupné z: `http://llvm.org/docs/ProgrammersManual.html`

[18] Sarda, S.: *LLVM Essentials*. Packt Publishing, 2015.

[19] Dang, B.; Gazet, A.; Bachaalany, E.; aj.: *Practical Reverse Engineering: x86, x64, ARM, Windows Kernel, Reversing Tools, and Obfuscation*. Wiley, 2014.

[20] Mueller, B.: Bitcode Metrics. Dostupné z: `https://github.com/b-mueller/obfuscation-metrics`

Příloha C

Obsah přiloženého CD


DP_Sima_Roman_2018


src — zdrojový kód obfuskátoru


samples — adresář s ukázkami obfuskace


text — zdrojové kódy textu práce

DP_Sima_Roman_2018.pdf