Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Deobfuskácia kódu generovaného programom Obfuscator-LLVM

Bakalárska práca

2016
Martin Červeň

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Deobfuskácia kódu generovaného programom Obfuscator-LLVM

Bakalárska práca

Študijný program:	Informatika
Študijný odbor:	2508 Informatika
Školiace pracovisko:	Katedra informatiky
Školiteľ:	RNDr. Richard Ostertág, PhD.
Konzultant:	Mgr. Peter Košinár

Bratislava, 2016
Martin Červeň

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Martin Červeň
Študijný program: informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor: informatika
Typ záverečnej práce: bakalárska
Jazyk záverečnej práce: slovenský
Sekundárny jazyk: anglický

Názov: Deobfuskácia kódu generovaného programom Obfuscator-LLVM
Deobfuscation of Code Generated by Obfuscator-LLVM

Cieľ: Cieľom práce je implementovať deobfuskátor kódu generovaného programom Obfuscator-LLVM. Program načíta obfuskovaný kód a vygeneruje deobfuskovaný kód, pričom sa pokúsi čo do najväčšej miery odstrániť nasledovné metódy obfuskácie:

- Instructions Substitution
- Bogus Control Flow
- Control Flow Flattening

Vedúci: RNDr. Richard Ostertág, PhD.
Konzultant: Mgr. Peter Košinár
Katedra: FMFI.KI - Katedra informatiky
Vedúci katedry: doc. RNDr. Daniel Olejár, PhD.

Spôsob sprístupnenia elektronickej verzie práce:
bez obmedzenia

Dátum zadania: 26.10.2015

Dátum schválenia: 27.10.2015

doc. RNDr. Daniel Olejár, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

Poďakovanie: Rád by som sa poďakoval môjmu školiteľovi RNDr. Richardovi Ostertágovi, PhD. a môjmu konzultantovi Mgr. Petrovi Košinárovi za ich pomoc a podporu, za ich čas a vedomosti, s ktorými sa so mnou počas písania práce podelili.

Abstrakt

Táto bakalárska práca sa zaoberá obfuskačnými a deobfuskačnými technikami. V práci je spracovaný základný prehľad týchto techník. Hlavnou časťou je implementácia deobfuskátora na programy obfuskované programom Obfuscator-LLVM , konkrétne technikami vyrovnávania toku riadenia, falošného toku riadenia a substitúcie inštrukcií. Deobfuskátor využíva miasm framework a preto práca obsahuje aj prehľad funkcionality tohto frameworku.

Kľúčové slová: obfuskácia, deobfuskácia, Obfuscator-LLVM, miasm framework, deobfuskátor

Abstract

This bachelor thesis is dealing with obfuscation and deobfuscation techniques. This paper provides basic overview of these techniques. Main part is deobfuscator implementation for programs obfuscated by Obfuscator-LLVM, in particular by techniques of control flow flattening, bogus control flow and instruction substitution. Deobfuscator uses miasm framework and therefore this paper contains also overview of miasm framework's functionality.

Keywords: obfuscation, deobfuscation, Obfuscator-LLVM, miasm framework, deobfuscator

Obsah

Zoznam obrázkov

Úvod

Pojem obfuskácia vo všeobecnosti značí zneprehľadnovanie, zahmlievanie. V softvérovom inžinierstve sa pod obfuskáciou rozumie hlavne zneprehľadňovanie zdrojového kódu. Obfuskácia vo vývoji programov nachádza čoraz častejšie svoje miesto. Jej hlavným cieľom je ochrana duševného vlastníctva. Najčastejšie však obfuskáciu využívajú tvorcovia malvéru. Pre malvér je obfuskácia skoro až nevyhnutnou podmienkou prežitia. Bez použitia obfuskácie, by bol malvér ľahko identifikovateľný a aj jeho správanie by bolo jednoduché analyzovať. Deobfuskácia sa snaží sprehľadniť a zjednodušiť neprehľadný kód.

Hlavným motívom skúmania nových a vylepšovania už existujúcich obfuskačných techník je snaha, čo najlepšie ochrániť duševné vlastníctvo. Motiváciou na vytváranie a zdokonalovanie deobfuskačných techník je snaha čo najlepšie identifikovať a najefektívnejšie bojovať proti malvéru. Obfuskácia a deobfuskácia sú teda jedna pre druhú hnacím motorom vývoja a ich zlepšovania, lebo keď sa jedna oblasť zlepší, tak druhá sa musí tiež zlepšiť, aby mala význam.

Obfuskácia zvykne využívať vo svoj prospech aj nezverejňovanie obfuskačných algoritmov, čím sťaží deobfuskáciu, je však len otázkou času, kedy sú jednotlivé algoritmy odhalené. Obfuscator-LLVM je program na obfuskovanie programov počas kompilovania. Ide o open-source projekt a preto je zaujímavé skúmať, ako dobre dokáže obstáť obfuskácia, ktorá je detailne známa, v procese deobfuskácie.

Práca je logicky rozdelená na dve časti. Prvá časť uvedie čitateľa do problematiky obfuskácie a deobfuskácie vo všeobecnosti. Obfuskácii a jednotlivým technikám obfuskácie sa venujeme v kapitole 1. V kapitole 2 sa venujeme technikám využívaným pri deobfuskácii.

Druhá časť práce sa venuje obfuskačným technikám využitým v programe Obfuscator-LLVM a ich deobfuskácii. Konkrétne kapitola 3 popisuje Obfuscator-LLVM a kapitola 4 popisuje implementáciu deobfuskátora. Implementácia je postavená na miasm frameworku, ktorý je tiež popísaný v kapitole 4. Výsledkom a úspešnosti deobfuskátora vytvoreného v rámci tejto práce sa venujeme v poslednej kapitole.

Kapitola 1

Obfuskácia a obfuskačné techniky

Pôvod slovenského slova obfuskácia je z latinského obfuscāre, čo v preklade znamená zatemniť alebo stmaviť.

Definícia obfuskačnej transformácie, ktorá formálne popisuje spôsob, ktorý je aplikovaný počas procesu obfuskovania programového kódu, znie nasledovne [6]:

Definícia 1. *Nech $\tau(P) = P'$ je transformácia zdrojového programu P do cieľového programu P'. $\tau(P) = P'$ je obfuskačná transformácia, ak P a P' majú rovnaké pozorovateľné správanie. Na to, aby bola $\tau(P) = P'$ validná obfuskačná transformácia, musia byť dodržané nasledovné podmienky:*

- *Ak P neskončí alebo skončí s chybou, tak P' môže, ale nemusí skončiť.*
- *Inak P' musí skončiť a vytvoriť rovnaký výstup ako P.*

Pri tvorbe programov sa obfuskácia používa ako spôsob ochrany voči disassemblovaniu rôznymi disassemblermi. Ochrana programového kódu pomocou obfuskácie má za úlohu zabezpečiť ochranu duševného vlastníctva. Je preto dôležité, aby použitá obfuskačná tranformácia bola čo najkvalitnejšia a teda mala isté vlastnosti [5]:

- **účinnosť** - účinnosť τ určuje, nakoľko komplexný alebo nečitateľný bude kód po aplikovaní obfuskačnej transformácie τ.
- **pružnosť** - pružnosť τ určuje, ako dobre obstojí transformácia pod útokom automatického deobfuskátora. Častokrát sa používajú jednosmerné transformácie, ktoré odstraňujú časť informácií nepotrebných pre beh programu a teda takéto transformácie majú väčšiu pružnosť, pretože odstránené informácie sa ťažko rekonštruujú.
- **nenápadnosť** - nenápadnosť τ určuje, ako obfuskovaný kód splýva so zvyškom programového kódu, na ktorý nebola aplikovaná obfuskačná transformácia. Transformácia generujúca kód ľahko odlíšiteľný od kódu, na ktorý nebola aplikovaná, má menšiu nenápadnosť.

- **cena** - cena τ určuje, ako veľmi sa zvýšia nároky na pamäť a čas potrebný na beh obfuskovaneho programového kódu voči jeho neobfuskovanej verzií. Pri vytváraní obfuskačnej transformácie sa snažíme, aby mala minimálnu cenu.

1.1 Obfuskačné techniky a metódy

V procese obfuskácie sa využívajú viaceré obfuskačné techniky a metódy. Niektoré sú jednoduchšie a dajú sa pomerne jednoducho odstrániť. Pre dosiahnutie čo najlepšieho výsledku sa v praxi nepoužíva iba jedna obfuskačná technika, ale ich kombinácia, ktorá je dostatočne kvalitná podľa spomenutých kritérií a zároveň nezvyšuje nároky na pamäť a čas potrebný na vykonanie programu nad únosnú mieru.

1.1.1 Obfuskácia mien a formátovania

Obfuskácie mien je proces, v ktorom sa zamenia zmysluplné reťazce za nové identifikátory. Identifikátory často vedia napomôcť k rozpoznaniu štruktúr zdrojového kódu ako tried, metód, premenných a podobne. Pre proces premenovávania identifikátorov existujú isté obmedzenia. Tieto obmedzenia sa vzťahujú hlavne na mená tried zahrnutých v štandardných API, mená serializovatelných tried a mená tried, ku ktorým sa pristupuje pomocou reflexie alebo cez natívne metódy [12].

Zmena mien v kóde sa dá ľahko spoznať, najmä keď sú nové mená tvorené z náhodnej postupnosti znakov. Z tohto dôvodu sa niekdedy ako nové identifikátory používajú zmysluplné slová, ktoré napríklad znamenajú presný opak toho, čo vykonáva pomenovaná metóda. Medzi spôsoby výberu mien patrí využívanie rôznych kombinácií podobných znakov, napríklad znaky 'I', 'l' a '1'. Taktiež využívaný spôsob býva použiť v menách netlačiteľné znaky, prípadne namiesto ASCII znakov použiť unicode znaky. V prípade, že sú volené nové mená kratšie ako pôvodné, tak to vedie k zmenšeniu programového kódu a teda aj menším nárokom na pamäť.

Formátovanie sa v procese obfuskácie odstraňuje, aby bol kód menej prehľadný. Pri odstraňovaní formátovania sa odstraňujú aj komentáre, ktoré síce nemajú vplyv na vykonávanie programu, ale napomáhajú k pochopeniu funkcionality programu a programový kód je bez nich náročnejší na pochopenie [11]. Ako príklad odstránenia formátovania a zmeny identifikátorov uvádzame kód 1.1.

```
1 var S7a=function(a,b,c,d){b=P7a(new O7a(b),function(a){return a in this.H
    ?a:void 0},a);var f=a.U[b],g=a.W[b],h=a.V[b],l=a.T[b];try{var n=new f;
    c.controller=n;n.WW=c;n.IB=b;c.H=a;var p=g?new g(d):null;c.oe=p;var r=
    h?new h(n):null;c.Wg=r;a.R(v4a,n.IB);l(n,p,r);a.R(v4a,n.IB);return n}
    catch(v){c.controller=null;c.error=v;N7a(b,v);try{a.S.H(v)}catch(w){}
    return null}};
```

Kód 1.1: Príklad obfuskácie mien a formátovania.

1.1.2 Obfuskácia dátových štruktúr

Obfuskácia dátových štruktúr je proces meniaci spôsob a poradie ich uloženia v pamäti a prístupu k nim.

Obfuskácia premenných a polí

Jednou z možností je zmena roly, ktorú majú dátové štruktúry v programe. Táto zmena sa dá docieliť zmenou miesta definície premennej, ktorá môže byť definovaná lokálne alebo globálne. Zmenou z lokálnej premennej na globálnu sa predĺži jej životnosť a môže prípadne nahradiť viacero lokálnych premenných rovnakého typu v rôznych metódach alebo častiach kódu, ktoré nebežia súčasne, čím sa sťaží porozumenia účelu premennej. Primitívne premenné je možné zmeniť v jazykoch, ako je napríklad Java, na objekty. Pri poliach je možné meniť ich rozmery a dimenzie, prípadne ich rozdeľovať a spájať. Pri vykonaní niektorej z uvedených zmien je potrebné upraviť spôsob, akým sa pristupuje k hodnotám, ktoré sú uložené v upravenom poli alebo poliach [12].

Zmena kódovania a obfuskácia reťazcov

Ďalšia využívaná metóda je založená na zmene spôsobu kódovania dát. V pamäti potom nie je uložená priama hodnota, ale tá sa nahradí výrazom, ktorý po vyhodnotení dáva požadovaný výsledok. Podobná je metóda, ktorá nahrádza statické premenné procedúrami. Procedúrami sa nahrádzaju najmä reťazce, pričom jedna procedúra môže generovať aj viac reťazcov a aktuálne generovaný reťazec určí vstupný parameter. Reťazce môžu byť v procese obfuskovania zakódované a takto uložené. Ich dekódovanie potom prebieha zväčša tesne pred použitím a po ňom sú opäť zakódované, aby ich nebolo možné získať pri statickej analýze, ale aby bolo treba program spustiť a zastaviť v správnom momente, keď sa reťazce nachádzajú v pamäti v dekódovanom stave. Uložené reťazce môžu byť rozdelené, podobne ako polia, na viac častí, ktoré sa spoja, keď to bude potrebné, alebo naopak viac reťazcov môže byť spojených do jedného a používa sa z tohto reťazca iba relevantná časť [5]. Obfuskácia reťazcov je ilustrovaná v kóde 1.2.

```
1 def generator(int x){
2   str = "helloworld"
3   if (x == 0): return str[0:5]; // vráti "hello"
4   if (x == 1): return str[5:10]; // vráti "world"
5 }
```

Kód 1.2: Príklad funkcie vracajúcej časť reťazca, ktorý bol vytvorený spojením dvoch reťazcov.

1.1.3 Obfuskácia toku riadenia

Tok riadenia programu je po aplikovaní transformácií vykonávajúcich obfuskáciu toku riadenia zmenený do podoby, ktorá je náročnejšia na skúmanie. Transformácie pri obfuskovaní menia zoskupenie, poradie a postup výpočtu toku riadenia [4].

Rozdelenie transformácií toku riadenia

Transformácie meniace tok riadenia môžeme rozdeliť do troch podkategórií. Prvou podkategóriou sú také transformácie, ktoré schovávajú pravý tok riadenia. Do druhej kategórie spadajú transformácie pridávajúce do kódu postupnosti inštrukcií z úrovne skompilovaného kódu, pre ktoré neexistujú zodpovedajúce konštrukty v programovacom jazyku vyššej úrovne. Do poslednej podkategórie spadajú transformácie odstraňujúce abstrakciu toku riadenia alebo pridávajúce falošnú [6].

Neprehľadný predikát

Viaceré metódy obfuskácie toku riadenia využívajú neprehľadné predikáty a premenné. Ich definícia znie nasledovne [4]:

Definícia 2. *Premenná V je neprehľadná v bode p v programe, ak V má vlastnosť q v p, ktorá je známa v čase obfuskácie. Píšeme V_p^q alebo V^q, ak je p známe z kontextu. Predikát P je neprehľadný v bode p, ak jeho výsledok je známy v čase obfuskácie. Píšeme P_p^F (P_p^T), ak P je vždy vyhodnotený ako False (True) v bode p. Rovnako p môže byť vynechané, ak je jasné z kontextu.*

Povedané neformálne, premenná V je neprehľadná, ak jej vlastnosť q je známa obfuskátoru, ale pre deobfuskátor je náročné ju odvodiť. Neprehľadný predikát je boolovský výraz, ktorý je vždy vyhodnotený rovnako, true alebo false, bez ohľadu na hodnoty premenných, ktoré obsahuje, ale je náročné túto vlastnosť identifikovať skúmaním výrazu [3].

Príkladom neprehľadného predikátu je $(x * (x + 1)\%2 == 0)$, kde x je prirodzené číslo. Ide o P^T neprehľadný predikát, lebo pre všetky prirodzené čísla x platí, že $(x * (x + 1))$ je párne číslo a teda zvyšok po delení číslom 2 je 0.

Inlining a outlining

Zoskupenie jednotlivých príkazov sa mení viacerými spôsobmi, využíva sa vkladanie tela metód namiesto ich volania podobne ako pri makrách. Niektoré obfuskačné transformácie robia opačný proces, ktorý vyberie časť kódu, urobí z neho novú metódu a nahradí ho volaním novovytvorenej metódy [5].

Úpravy cyklov

Pri cykloch, ktoré majú konštantný počet iterácií, sa daný cyklus rozroluje a jeho telo sa do kódu vloží jeden alebo viac krát, pričom sa upraví počet iterácií, aby bol zachovaný ich pôvodný počet. Cykly, ktoré nemajú vopred známy počet iterácií, je možné rozdeliť do viacerých cyklov, ktoré môžu byť aj vnorené, ale zachová sa celkový počet iterácií, cez ktorý iterujú. Niektoré cykly je možné preusporiadať napríklad tak, aby boli vykonávané opačne [4]. Cyklus je možné zneprehľadniť rozšírením podmienky, ktorá kontroluje počet iterácií takým spôsobom, že ostane zachovaný počet iterácií, ale podmienka bude komplexnejšia [6]. Príklad úpravy cyklov je v kóde 1.3.

```
// pôvodný cyklus
for(i = 1; i < n; i++){
  pole[i] = pole[i-1] * x;
}
// upravená verzia
for(int i = 1; i < n; i+=5)
  for(int j = 0; j < 5 && i+j < n; j++){
    pole[i+j] = pole[i+j-1] * x;
  }
```

Kód 1.3: Príklad vytvorenia vnoreného cyklu.

Vytváranie kópií

Ďalší spôsob menenia zoskupenia je vytvoriť viaceré kópie jednej metódy a volania pôvodnej metódy nahradiť na rôznych miestach volaniami rôznych kópií. Takto sa vytvorí dojem, že ide o volanie rôznych metód, aj keď to nie je pravda. Pre zvýšenie tohto dojmu a sťaženie analýzy sa na jednotlivé kópie pôvodnej metódy aplikujú ďalšie obfuskačné techniky, ktoré sú pre každú kópiu rôzne [5].

Menenie usporiadania

Programátori zyknú časti kódu, ktoré spolu súvisia, koncentrovať k sebe. V procese obfuskácie sa usporiadanie jednotlivých častí programu znáhodňuje. Niektoré časti je môžné preusporadúvať bez väčších obmedzení, napríklad metódy v rámci jednej triedy. Pri menení usporiadania príkazov v základnom bloku je nutné vykonať analýzu závislosti na dátach, ktorá určí, ktoré usporiadania je možné vykonať bez zmeny významu [4].

Pridávanie irelevantného a mŕtveho kódu

Pre schovanie pravého toku riadenia býva do programového kódu pridávaný kód, ktorý nemá vplyv na správanie programu z pohľadu používateľa. Najčastejšie ide o inštruk-

cie, ktoré majú navzájom opačný účinok alebo volania funkcií, ktorých výsledok sa nepoužije. Ďalšia možnosť ako schovať tok riadenia programu, je pridanie kódu, ktorý sa nikdy nevykoná. Takýto kód sa môže nachádzať za podmienkou, ktorá vždy nasmeruje tok riadenia do správnej vetvy alebo za inštrukciou, ktorá spôsobí výnimku. Pridávanie kódu, ktorý sa nikdy nevykoná, zväčšuje nároky na pamäť, keďže celý súbor s programom je väčší, nároky na čas vykonávania sú porovnateľné s neobfuskovaným programom. Pridávanie irelevantných inštrukcií má vplyv na zväčšenie ako pamätových nárokov, tak aj nárokov na čas vykonávania programu [6, 12]. Mŕtvy kód je znázornený v kóde 1.4.

```
try{
  pôvodný kód ...
  x/0 //delenie nulou spôsobí výnimku
  mŕtvy kód // táto časť sa nikdy nevykoná
} catch() {
  pokračovanie kódu
}
```

Kód 1.4: Príklad mŕtveho kódu za výnimkou.

Tabuľková interpretácia

Veľmi efektívnou, ale zároveň náročnou na zdroje, je transformácia nazývaná tabuľková interpretácia. V tejto transformácii sa časť kódu (napríklad Java bytecode) prekonvertuje do kódu iného virtuálneho stroja (napríklad Python). K obfuskovanému programu je potom nutné pripojiť interpreter, ktorý bude vykonávať časti kódu ktoré boli prekonvertované [4, 5].

Odstraňovanie volania knižničných funkcií

Volania knižničných funkcií sú vo viacerých programovacích jazykoch na základe mena, pričom tieto mená nemôžu byť obfuskované. Knižničné funkcie použité v programe preto môžu vypovedať veľa o fungovaní programu. Použitie funkcií zo štandardných knižníc sa schováva vytvorením kópie danej knižnice alebo knižničnej funkcie tak, aby nebolo zrejmé, že ide o túto knižnicu alebo funkciu [4].

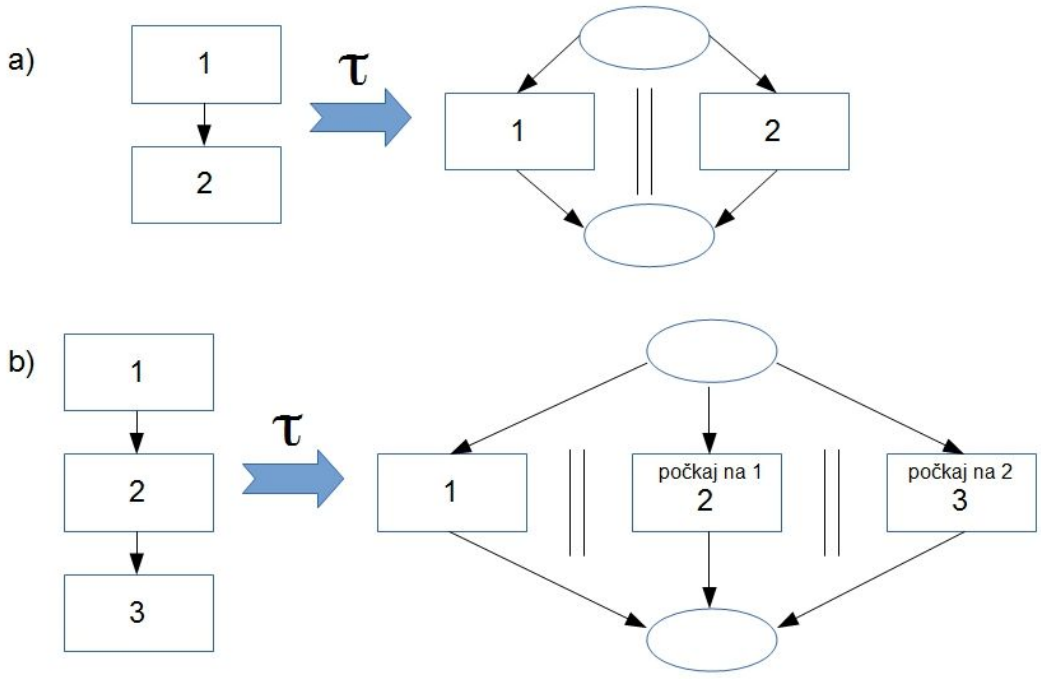
Programátorské zvyklosti

Podobná situácia, ktorá môže veľa odhaliť o fungovaní programu, je spôsob, akým sa niektoré časti kódu zvyknú programovať. Takéto vzory je možné nájsť a použiť ich ako odrazový mostík pre pochopenie programu. Preto ich nahradenie za menej nápadné konštrukcie napomôže ku sťaženiu analýzy fungovania programu. V minulosti, keď Java ešte neobsahovala štandardnú triedu LinkedList s operáciami (insert, delete,

enumerate,...) pre spájaný zoznam, tak bolo zvykom programovať spájaný zoznam ako list objektov, ktoré na seba odkazovali premennou next. Iterovanie cez takýto zoznam je ľahko identifikovateľný vzor [4].

Paralelizovanie kódu

Poslednou transformáciou, ktorú si spomenieme, je paralelizovanie kódu. Časti kódu, ktoré sú nezávislé na dátach, môžu byť jednoducho paralelizované. Pri častiach kódu pracujúcich s rovnakými dátami je nutné pridať synchronizáciu [13]. Príklad paralelizovania kódu sa nachádza na obrázku 1.1.



Obr. 1.1: Príklad paralelizovania kódu s nezávislými dátami hore: a) a častí závislých na spoločných dátach dole: b).

Kapitola 2

Deobfuskácia

Predchádzajúca kapitola nám umožnila nahliadnuť do spôsobov obfuskácie. V tejto kapitole sa budeme venovať opačnému procesu, ktorý sa nazýva deobfuskácia. Deobfuskácia je prevod programu náročného na porozumenie do pochopiteľnejšej formy. Nie je to teda v pravom zmysle opačný proces k obfuskácií, lebo jej cieľom nie je vrátiť program do pôvodnej podoby, ale do zrozumiteľnejšej. Pri niektorých obfuskačných transformáciách dochádza k odstráneniu informácií nepotrebných pre samotný chod programu a preto prevod do pôvodnej podoby ani nie je možný.

V procese deobfuskácie je na začiatku potrebné zistiť, aké konkrétne obfuskačné techniky boli aplikované na program, ktorý chceme deobfuskovať. Po dôkladnom preštudovaní použitých obfuskačných transformácií je možné začať s výberom techník, ktoré budú použité v procese deobfuskácie. Základné rozdelenie deobfuskačných techník je na dve kategórie. Prvou kategóriou sú statické a druhou dynamické deobfuskačné techniky. Tieto techniky sa v procese deobfuskácie kombinujú pre dosiahnutie lepších výsledkov.

Pri statických deobfuskačných technikách sa s programom pracuje bez jeho spúšťania iba na základe analýzy kódu. Dynamické obfuskačné techniky sú založené na analýze bežiaceho programu. Obfuskované programy sú väčšinou spúšťané počas analýzy v debuggeroch, sandboxoch alebo kontrolovaných prostrediach, aby bolo možné s nimi jednoduchšie interagovať a ľahšie zaznamenávať informácie o ich behu.

2.1 Deobfuskačné techniky

2.1.1 Identifikácia a vyhodnotenie neprehľadných predikátov

Najťažšia časť v procese deobfuskácie je identifikácia a vyhodnotenie neprehľadných predikátov. Toto sú dva procesy, ktoré sú kvôli silno rozšírenému využívaniu neprehľadných predikátov obfuskačnými transformáciami veľmi potrebné [4].

2.1.2 Globálna analýza toku dát

Globálna analýza toku dát je optimalizačná technika, ktorá sa využíva aj pri vyhodnocovaní neprehľadných predikátov. V procese deobfuskácie ju pri jednoduchších neprehľadných predikátoch môžeme využiť na nahradenie premenných konštantami. Ide o prípad, kedy premenná nachádzajúca sa v predikáte nebola od posledného priradenia konštanty menená.

2.1.3 Dokazovanie viet

Pri vytváraní neprehľadných predikátov sa často využívajú matematické vety. V takomto prípade sa môžu deobfuskátory pokúsiť využiť program na dokazovanie viet. Úspešnosť tejto metódy závisí od vývoja týchto programov a od zložitosti vety, ktorú treba dokázať. S vetami, ktoré sa dajú dokázať pomocou matematickej indukcie, sa vedia programy na dokazovanie viet vysporiadať. Keďže dokazovanie viet je náročné a taktiež pre sťaženie deobfuskácie sa využívajú aj vety, ktoré sú ťažko dokázateľné alebo ich dôkaz neexistuje, tak je efektívnejšie túto metódu deobfuskácie použiť až keď jednoduchšie metódy nepriniesli výsledky [4].

2.1.4 Hľadanie vzorov

V procese deobfuskácie môžeme ťažiť z vedomosti o tom, akým spôsobom vytvára obfuskátor neprehľadné predikáty. Vedomosť o tomto procese môžeme získať štúdovaním obfuskátora, či už dekompiláciou alebo skúmaním obfuskovaného kódu. Na základe zistených informácií vytvoríme pravidlá pre hľadanie vzorov popisujúce často používané neprehľadné predikáty. Táto technika sa dá efektívne použiť na jednoduché lokálne predikáty [4].

Hľadanie vzorov je možné využiť aj pri staticky linkovaných knižničných funkciách. Disasembler a dekompilátor IDA Pro využíva na identifikovanie štandardných knižničných funkcií FLIRT (Fast Library Identification and Recognition Technology). Táto technológia identifikuje sekvencie kódu ako knižničné funkcie na základe vzorov [7, 13]. Ak bola na tieto funkcie aplikovaná nejaká obfuskačná transformácia, tak hľadanie pomocou vzorov pre neobfuskované verzie nebude fungovať. Pokiaľ neboli obfuskované, ale boli pridané, aby sťažili celkovú orientáciu v kóde, tak táto technika pomôže.

2.1.5 Program slicing

Rozdeľovanie programu na časti, takzvané slices, sa využíva pri odstraňovaní mŕtveho kódu alebo v prípade rozdelenia a disperzií súvisiacich častí kódu. Vďaka tejto technike je možné identifikovať časti kódu ovplyvňujúce hodnotu v premennej. Vieme teda získať

z programu algoritmus počítajúci hodnotu neprehľadného predikátu, čo nám pomôže v pokračovaní deobfuskácie [4].

Existuje aj dynamické rozdeľovanie programu na časti, ktoré ovplyvňujú hodnotu premennej. V dynamickom variante sa vyberajú na rozdiel od statického variantu nie všetky časti ovplyvňujúce premennú, ale iba časti kódu, ktoré ovplyvnili hodnotu premennej v konkrétnom behu programu [1].

2.1.6 Čiastočné vypočítanie

Čiastočné vypočítanie je optimalizačná technika založená na špecializácii. Čiastočný evaluátor je algoritmus, ktorý z programu a časti vstupných dát vytvorí špecializovaný program. Tento špecializovaný program po spustení so zvyšnými vstupnými dátami dospeje k rovnakému výsledku ako pôvodný program s celým vstupom [8].

V prípade deobfuskácie sa čiastočný evaluátor používa na rozdelenie programu na dve časti, na statickú časť, ktorá môže byť vypočítaná dopredu a dynamickú, ktorá sa vykonáva pri spustení. Dynamická časť zodpovedá originálnemu, neobfuskovanému programu, zatiaľ čo statická časť zodpovedá pridanému kódu v procese obfuskácie. Keď sa podarí identifikovať tento kód, tak môže byť vyhodnotený a následne odstránený [4].

2.1.7 Dosiahnuteľnosť

Definícia 3. *Statická dosiahnuteľnosť: Časť kódu C je staticky nedosiahnuteľná v programe P, ak statická analýza určí, že neexistuje cesta vykonávania z entrypointu programu P do časti C [3].*

Definícia 4. *Dynamická dosiahnuteľnosť: Časť kódu C v programe P je dynamicky dosiahnuteľná pre sadu vstupov I, ak tok riadenia dosiahne C počas vykonávania P na vstupe I [3].*

Nedosiahnuteľný kód nemôže byť vykonaný počas behu programu. Preto je dobré tieto časti identifikovať pred začatím ďalšej analýzy, lebo nemajú na beh programu žiaden vplyv a je teda zbytočné snažiť sa pochopiť, aké akcie vykonávajú tieto časti kódu. Niekedy môže byť kód programu obfuskovaný takým spôsobom, že nie je možné presne určiť statickou dosiahnuteľnosťou, ktoré časti sú dosiahnuteľné a ktoré nie sú. V takomto prípade sa dá využiť dynamická dosiahnuteľnosť. Keďže dynamická dosiahnuteľnosť sa zisťuje na základe konkrétnych vstupov, je potrebné zopakovať test dosiahnuteľnosti na čo najväčšom počte vstupov, ktoré pokrývajú čo najväčšie spektrum. Po vykonaní týchto testov sa výsledky skombinujú do jedného.

2.1.8 Štatistická analýza

Štatistická analýza sa využíva pri sledovaní hodnôt, ktoré nadobúdajú predikáty počas behu programu. V prípade, že niektorý predikát pri dostatočne veľkom počte testovacích behov nadobúdal rovnakú hodnotu, je vysoko pravdepodobné, že ide o neprehľadný predikát. Takýto predikát treba následne preskúmať a až po uistení sa, že je to naozaj neprehľadný predikát vykonať akcie na jeho odstránenie. Môže totiž ísť o špecifickú podmienku, akou je napríklad kontrola priestupného roku.

Štatistická analýza sa dá využiť aj ako kontrola. Ak máme program, v ktorom sme identifikovali a odstránili neprehľadný predikát, tak pomocou štatistickej analýzy môžeme skontrolovať na sade testovacích behov výstupy pôvodného programu a programu s odstráneným neprehľadným predikátom. Rovnaké výstupy pre nás znamenajú, že sme označili neprehľadný predikát správne [4].

Kapitola 3

Obfuscator-LLVM

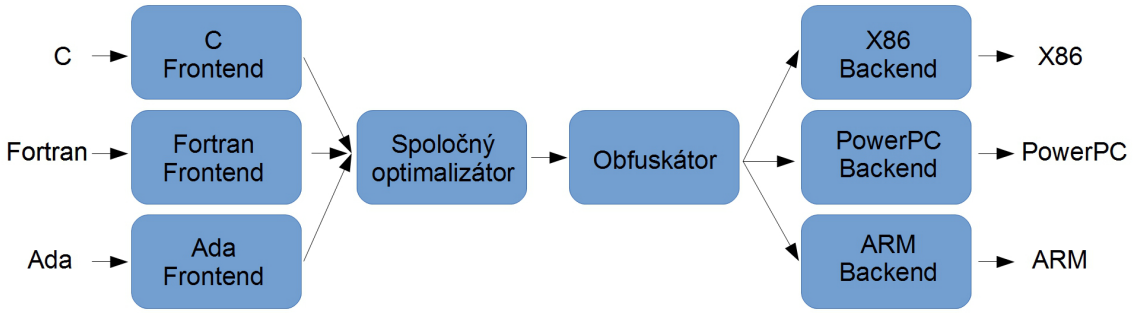
V tejto kapitole rozoberieme spôsob fungovania programu Obfuscator-LLVM, na ktorý budeme implementovať deobfuskátor. Obfuscator-LLVM (O-LLVM) je open-source projekt iniciovaný v júni v roku 2010 skupinou zaoberajúcou sa informačnou bezpečnosťou, ktorá pôsobí na University of Applied Sciences and Arts v západnom Švajčiarsku. Cieľom tohto projektu je poskytnúť open-source odnož kompilačných programov LLVM schopnú poskytnúť zvýšenú bezpečnosť softvéru pomocou obfuskácie kódu. LLVM v procese kompilácie preloží kód do internej reprezentácie. Vďaka tejto vlastnosti LLVM podporuje viaceré programovacie jazyky (C, C++, Objective-C, Ada a Fortran) a rôzne platformy (x86, x86-64, PowerPC, PowerPC-64, ARM, Thumb, SPARC, Alpha, CellSPU, MIPS, MSP430, SystemZ a XCore). Samotná obfuskácia v programe O-LLVM prebieha na kóde v internej reprezentácii LLVM a preto je kompatibilná so všetkými programovacími jazykmi a cieľovými platformami, ktoré sú aktuálne podporované kompilačnými programami LLVM [9].

3.1 Architektúra O-LLVM

O-LLVM má architektúru skoro identickú s architektúrou, akú má originálny LLVM. Architektúra LLVM pozostáva z troch hlavných častí:

- viacero frontendov - pre každý podporovaný programovací jazyk jeden
- jeden spoločný optimalizátor
- viacero backendov - pre každú podporovanú platformu jeden

Každý z frontendov urobí preklad z programovacieho jazyka, pre ktorý je určený, do internej reprezentácie LLVM. Kód v internej reprezentácii je následne optimalizovaný. Po optimalizácii je urobená kompilácia pre požadovanú platformu. Výhoda takéhoto dizajnu je jeho jednoduchá rozšíriteľnosť. Na pridanie podpory pre ďalší programovací



Obr. 3.1: Znázornenie umiestnenia obfuskátora do LLVM.

jazyk stačí vytvoriť nový frontend. Na pridanie podpory ďalšej platformy stačí obdobne vytvoriť nový backend.

Z výhody ľahkej rozšíriteľnosti ťaží aj O-LLVM, ktorého funkcionalita je vložená medzi optimalizáciu a backendy, ako to môžeme vidieť na obrázku 3.1. V podstate boli na výber dve možné umiestnenia. Prvé miesto je za frontendy pred optimalizáciu a druhé miesto, na ktorom sa aj nachádza, za optimalizáciu pred backendy. Prvé spomenuté miesto by bolo trochu kontraprodutktívne, lebo práve proces optimalizácie by mohol odstrániť niektoré obfuskácie vytvorené obfuskátorom.

3.2 Obfuskačné techniky O-LLVM

V O-LLVM sú implementované nasledovné tri obfuskačné techniky:

- substitúcia inštrukcií
- falošný tok riadenia
- vyrovnávanie toku riadenia

Spôsob, ako sú tieto techniky v O-LLVM implementované, si rozoberieme v nasledujúcej časti. Príklady obfuskovaných programov pomocou falošného toku riadenia a vyrovnávania toku riadenia sa nachádzajú v kapitole 5.

3.2.1 Substitúcia inštrukcií (Instruction Substitution)

Cieľom tejto obfuskačnej techniky je zámena inštrukcií binárnych operátorov (sčítanie, odčítanie a boolovské operátory) za funkcionálne ekvivalentné inštrukcie. Keď je na výber z viacero ekvivalentných možností, tak je výber náhodný. Sčítanie $a = b + c$ sa nahradí jednou z nasledovných možností:

- $a = b - (-c)$
- $a = -(-b + (-c))$
- $r = rand(); a = b + r; a = a + c; a = a - r$
- $r = rand(); a = b - r; a = a + b; a = a + r$

Odčítanie $a = b - c$ sa nahradí jednou z nasledovných možností:

- $a = b + (-c)$
- $r = rand(); a = b + r; a = a - c; a = a - r$
- $r = rand(); a = b - r; a = a - c; a = a + r$

Logické operácie:

- AND: $a = b\&c$ sa nahradí $a = (b \oplus \neg c)\&b$
- OR: $a = b|c$ sa nahradí $a = (b\&c)|(b \oplus c)$
- XOR: $a = a \oplus b$ sa nahradí $a = (\neg a\&b)|(a\&\neg b)$

Substitúcia inštrukcií je dosť priamočiara technika obfuskácie a teda nepridáva veľa bezpečnosti. Substituované inštrukcie sa dajú odstrániť opätovnou optimalizáciou. Použitie náhodných hodnôt aspoň spôsobuje rozdiely v produkovaných binárnych súboroch.

3.2.2 Falošný tok riadenia (Bogus Control Flow)

Pred tým, ako sa oboznámime s obfuskačnou technikou falošného toku riadenia, si definujeme pojemy základný blok a graf toku riadenia.

Definícia 5. *Základný blok je maximálna množina usporiadaných inštrukcií, v ktorej vykonávanie začína prvou inštrukciou a končí poslednou inštrukciou. Táto množina neobsahuje žiadnu vetviacu inštrukciu. Výnimkou je posledná inštrukcia, ktorá môže byť prípadne aj vetviaca [14].*

Definícia 6. *Graf toku riadenia je orientovaný graf, ktorého vrcholy reprezentujú základné bloky a hrany reprezentujú skoky v toku riadenia [2].*

V informatike sa graf toku riadenia využíva na reprezentáciu všetkých ciest, ktorými môže program počas vykonávania prejsť. Graf toku riadenia obsahuje dva typy špeciálnych blokov. Prvým špeciálnym blokom je vstupný blok, ktorým tok riadenia vstupuje do grafu a druhým je výstupný blok, ktorým tok riadenia opúšťa graf. Vstupný blok môže byť maximálne jeden, zatiaľ čo výstupných blokov môže byť viac.

Technika obfuskácie s názvom falošný tok riadenia upravuje graf toku riadenia spôsobom ako ukazuje kód 3.1.

```
s pravdepodobnosťou p1 pre každú funkciu: // p1 je štandardne 100%
  s pravdepodobnosťou p2 pre každý základný blok: // p2 je štandardne 30%
    pridaj blok s neprehľadným predikátom
    doplň do základného bloku neprehľadný predikát
    pridaj kópiu základného bloku a doplň ju náhodnými inštrukciami
    odstráň hranu rodič -> základný blok
    pridaj hrany:
      rodič -> blok s neprehľadným predikátom
      blok s neprehľadným predikátom -> základný blok (True)
      blok s neprehľadným predikátom -> kópia základného bloku (False)
      kópia základného bloku -> základný blok
      základný blok -> kópia základného bloku (False)
      základný blok -> pôvodná výstupná hrana (True)
```

Kód 3.1: Pseudokód algoritmu vykonávajúceho obfuskáciu falošného toku riadenia.

Neprehľadný predikát v novo pridanom základnom bloku je vždy vyhodnotený tak, že je vykonaný podmienečný skok do pôvodného základného bloku. Štandardné nastavenie programu O-LLVM je také, že pokiaľ je aktivovaná technika falošného toku riadenia, tak je obfuskovaná každá funkcia (pravdepodobnosť obfuskovania funkcie je 100 pecent) jedným prechodom a pravdepodobnosť, že bude nejaký základný blok obfuskovaný, je 30 percent. Tieto hodnoty je možné zmeniť. Vzhľadom na to, že v jednom prechode nemusí byť obfuskovaný každý základný blok, tak je možné urobiť viac prechodov. V ďalších prechodoch sa nerozlišuje medzi pôvodnými základnými blokmi programu a blokmi vytvorenými v predošlých prechodoch a môžu byť obfuskované aj základné bloky vytvorené obfuskátorom.

3.2.3 Vyrovnávanie toku riadenia (Control Flow Flattening)

Cieľom tejto obfuskačnej techniky je úplne vyrovnať graf toku riadenia programu.

V procese obfuskovania sa najprv rozdelí obfuskovaná metóda na základné bloky. Tieto bloky, pôvodne nachádzajúce sa v grafe v rôznych úrovniach vnorenia, sa umiestnia vedľa seba do jednej úrovne vnorenia. Následne sú tieto bloky zabalené do vetviacej štruktúry (inštrukcia switch). Každý blok je samostatný case prípad a selekcia je zabalená do cyklu. Správny tok riadenia je zabezpečený pridaním riadiacej premennej, ktorá je na konci každého základného bloku aktualizovaná a riadi selekciu ďalšieho základného bloku, prípadne ukončenie cyklu [10].

3.2.4 Obfuskačné anotácie funkcií

Anotácie umožňujú určiť, ktoré metódy programu majú byť akým spôsobom obfuskované. Stačí pridať atribúty, v ktorých treba uviesť požadované obfuskácie. Metóda

bez uvedených atribútov bude obfuskovaná podľa všeobecných nastavení zadaných pri kompilácii.

Pre každú metódu programu je možné uviesť jednu alebo aj viac anotácií. Je tiež možné použiť opačný príznak, ktorý vypne uvedený spôsob obfuskácie pre metódu, pri ktorej je nastavený. Možnosti anotácií funkcií v O-LLVM sú: "sub"(pre substitúciu inštrukcií), "nosub"(pre zakázanie substitúcie inštrukcií), "fla"(pre vyrovnanie toku riadenia), "nofla"(pre zakázanie vyrovnávania toku riadenia), "bcf"(pre falošný tok riadenia), "nobcf"(pre zakázanie falošného toku riadenia). Napríklad, ak by sme nechceli, aby bola na niektorú metódu programu aplikovaná obfuskačná technika falošného toku riadenia, tak by sme použili anotáciu "nobcf". Pre ilustráciu uvádzame príklad v kóde 3.2.

```
1 int foo() __attribute((__annotate__(("fla"))));
2 int foo() {
3   return 47;
4 }
```

Kód 3.2: Anotácia pre obfuskovanie metódou vyrovnávania toku riadenia.

Ostatné parametre obfuskácie nie je možné meniť pomocou anotácií, dajú sa zmeniť iba globálne pre celý program.

Kapitola 4

Implementácia deobfuskátora

V tejto kapitole popíšeme spôsob, ktorý sme využili pri implementácii deobfuskátora na programy obfuskované programom Obfuscator-LLVM. Pri implementácii sme využili miasm framework. Je to open-source framework určený pre reverzných inžinierov napísaný v programovacom jazyku Python. Ešte pred tým, ako popíšeme implementáciu nášho deobfuskátora, ukážeme možnosti, ktoré ponúka miasm framework.

4.1 Miasm framework

Počas implementácie deobfuskátora sme sa stretávali s problémom, že miasm framework nemá dokumentáciu a teda zisťovanie jednotlivých funkcií tohto frameworku nebolo vždy triviálne. Miasm framework nie je veľmi rozšírený a preto najčastejším zdrojom informácií bol zdrojový kód tohto frameworku a pár príkladov, ktoré sú jeho súčasťou. Ďalším zdrojom, čo sa dá použiť, bol oficiálny blog http://www.miasm.re/blog/. Z tohto dôvodu v tejto práci uvedieme ucelenejší prehľad možností miasm-frameworku doplnený o ukážky, ktoré sme vytvorili.

Veľkou výhodou miasm frameworku je implementácia viacerých architektúr procesorov: X86, ARM, MIPS, SH4 a MSP430. V kombinácii s podporou otvárania, upravovania a generovania súborov vo formátoch pre operačný systém Microsoft Windows, Portable Executable (PE), ako aj pre operačné systémy Linux, Executable and Linkable (ELF), ponúka širokú škálu použitia.

4.1.1 Asemblovanie a disasemblovanie

Základ pre prácu so spustiteľnými súbormi je vedieť interpretovať jednotlivé inštrukcie zapísané postupnosťou bajtov do čitateľnejšej podoby, vedieť urobiť ich úpravu a opätovné prevedenie do postupnosti bajtov. Miasm framework má triedu Container, ktorá automaticky zistí, či ide o PE alebo Elf súbor na základe hlavičky a údajov v nej, pričom na parsovanie používa knižnicu ElfEsteem. Pri analýze súboru automaticky zdeteguje

jeho architektúru a tiež adresu entry-pointu. Detekcia architektúry procesora je len informačná a pre ďalšiu prácu je nutné importovať danú architektúru z miasm do kódu príslušným príkazom alebo pomocou abstrakcie využitím triedy Machine.

S jednotlivými inštrukciami je možné pracovať na troch úrovniach. Prvá je postupnosť bajtov, druhá je asembler a posledná je interná reprezentácia miasm. Na zmenenie jednej inštrukcie alebo jej operandov sa dá využiť reprezentácia v asembleri. Väčšie úpravy je jednoduchšie robiť v internej reprezentácii, s ktorou pracujú viaceré súčasti miasm frameworku. Nie je teda nutné implementovať rovnakú funkcionalitu ešte raz nad reprezentáciou v asembleri, ale oplatí sa využiť možnosti, ktoré miasm framework ponúka. Medzi týmito možnosťami je aj možnosť transformovať internú reprezentáciu naspäť až do postupnosti bajtov v podobe záplat, ktoré je možné aplikovať na pôvodný súbor alebo jeho kópiu. Kód 4.1 ukazuje príklad disasemblovania kódu s využitím miasm frameworku.

```
# Container je wrapper pre ELF, PE, ...
from miasm2.analysis.binary import Container
# Machine je wrapper pre manažovanie viacerých architektúr
from miasm2.analysis.machine import Machine
cont = Container.from_stream(open("dump.bin"))
machine = Machine(cont.arch) # architektúra extrahovaná z hlavičky
mdis = machine.dis_engine(cont.bin_stream) # disasembler
blocks = mdis.dis_multibloc(cont.entry_point) # disasembluje kód
# Zapíše graf toku riadenia do dot súboru
open("cfg.dot", "w").write(blocks.dot())
```

Kód 4.1: Príklad kódu v jazyku Python, ktorým vykonávame disasemblovanie s využitím miasm frameworku.

4.1.2 Emulácia

Emulácia kódu prebieha v sandboxe vo zvolenej architektúre. Na emuláciu používa miasm knižnicu LibTCC alebo niektorý z programov LLVM, GCC alebo Python. Python sa používa na emuláciu kódu v internej reprezentácii. Emuláciu je možné vykonať iba na časti spustiteľného súboru alebo na celom súbore. Miasm má implementovanú možnosť python callbackov, ktoré napomáhajú interakcii s vykonávaním emulácie alebo na simuláciu volaní knižničných funkcií.

Na vytvorenie sandboxu je potrebné určiť architektúru a knižnicu alebo program, pomocou ktorého sa emulácia vykonáva. Následne je potrebné pre vytvorený virtuálny stroj inicializovať zásobník, do ktorého sa vloží hodnota, pri ktorej budeme vedieť, že je zásobník prázdny. Prázdny zásobník znamená, že sa skončila emulácia kódu a je potrebné virtuálny stroj zastaviť. Toto dosiahneme vytvorením metódy, ktorá zastaví beh virtuálneho stroja a pridaním breakpointu na nami vloženú hodnotu na zásobníku,

pričom tento breakpoint pri aktivovaní zavolá našu metódu a tá signalizuje zastavenie. Ide o breakpoint viažúci sa nie na inštrukciu, ale na adresu. Pred spustením emulácie je potrebné vytvoriť stránku pamäte, naplniť ju disasemblovaným kódom a určiť adresu v pamäti, na ktorú má byť táto stránka nahraná. Na to, aby sme vedeli, aké zmeny sa diali počas emulácie, je možné aktivovať logovanie zmien v registroch a tiež v pamäti. Príklad emulácie s využitím miasm frameworku uvádzame v kóde 4.2.

```
def stop_vm(jitter): # Metóda, ktorá zastaví emulátor
    jitter.run = False
    jitter.pc = 0
    return True

myjitter = Machine("x86_32").jitter() # Vytvorí x86 32bit sandbox
myjitter.init_stack() # Pridá pamäť pre zásobník a nastaví ESP
data = open(filename).read() # Prečíta kód
# Pridá pamäť pre kód
run_addr = 0x40000000
myjitter.vm.add_memory_page(run_addr, PAGE_READ | PAGE_WRITE, data)
# Aktivácia logov
myjitter.jit.log_regs = True
myjitter.jit.log_mn = True
myjitter.push_uint32_t(0x1337beef) # Push špeciálnej adresy na zásobník
# Pridanie breakpointu viažúceho sa na adresu na zastavenie emulácie
myjitter.add_breakpoint(0x1337beef, stop_vm)
# Inicializácia a spustenie emulátora
myjitter.init_run(run_addr)
myjitter.continue_run()
```

Kód 4.2: Príklad spustenia emulácie kódu v miasm frameworku.

Keď je sandbox pripravený a inicializovaný, stačí ho spustiť. Počas behu je možné s ním interagovať. Napríklad nechať si zobraziť hodnotu registra alebo aj meniť hodnoty uložené v registroch ako môžeme vidieť v kóde 4.3.

```
# Zobrazí hodnotu v registri EAX v šestnástkovej sústave
hex(jitter.cpu.EAX)
# Zmení hodnotu v registri EAX na 0x40000000
myjitter.cpu.EAX = 0x40000000
```

Kód 4.3: Interakcia počas emulácie.

4.1.3 Symbolické vykonávanie

Symbolické vykonávanie predstavuje veľkú pomoc, keď nechceme program spustiť, ale chceme zistiť, ktoré časti kódu ovplyvňujú nejaké miesto v pamäti alebo hodnotu v registri.

Symbolické vykonávanie sa v miasm frameworku inicializuje nasledovne:

- Na začiatku treba importovať architektúru a disasemblovať kód do internej reprezentácie miasm.
- Potom je potrebné vytvoriť istú množinu symbolov, ktoré reprezentujú registre a naplniť ich počiatočnými hodnotami. Počiatočné hodnoty môžeme zobrať predvolené alebo ak vieme, že predošlá časť kódu nechala napríklad v registri EAX hodnotu 0x55, tak môžeme iniciovať register EAX na túto hodnotu.
- Následne je možné spustiť emuláciu. Emulácia sa vykonáva vždy na jednom základnom bloku. Pre emulovanie viacerých blokov stačí urobiť cyklus, v ktorom sa budú postupne bloky emulovať.

4.1.4 Zjednodušovanie

Miasm framework má implementované zjednodušovanie výrazov a zjednodušovanie grafu toku riadenia. Pri obidvoch typoch zjednodušovania sa využívajú takzvané žolíky ("jokers"). Zjednodušovanie výrazov sme použili aj pri implementácii nášho deobfuskátora a preto sa mu budeme venovať neskôr. Teraz sa pozrime na zjednodušovanie grafov toku riadenia.

Zjednodušovanie grafu toku riadenia sa realizuje hľadaním vzorov podgrafov, ktoré sú vytvárané zo žolíkov. Na vytvorenie vzoru je potrebné špecifikovať jednotlivé vrcholy a hrany, ktoré ich spájajú. Žolík zodpovedá vrcholu grafu a má viacero parametrov, ktoré konkretizujú, s akými blokmi grafu toku riadenia bude vytvárať zhodu. Prvým parametrom je meno, pomocou ktorého ho budeme referencovať. Nasledujú dva parametre určujúce, koľko hrán môže vchádzať a koľko vychádzať z daného vrchola. Ak je niektorý z nich nastavený na False, tak zodpovedajúcich hrán (vstupných alebo výstupných) môže byť iba toľko, koľko uvedieme pri vytváraní vzoru, v opačnom prípade počet hrán nie je obmedzený. Posledným parametrom môžeme konkretizovať obsah bloku, ktorý je reprezentovaný daným vrcholom, či už je to počet riadkov alebo aj konkrétne inštrukcie v ňom.

Po vytvorení potrebných žolíkov určíme, ktoré žolíky majú byť spolu pospájané hranami. Keď máme vytvorený graf zo žolíkov, ďalší krok je vytvorenie metódy, ktorá vykoná zamýšľané zmeny na grafe toku riadenia. Na záver nám stačí pridať do prechodov triedy DiGraphSimplifier metódu, ktorú sme vytvorili a aplikovať ju na graf toku riadenia programu. Celý postup zjednodušovania grafov s využitím miasm frameworku je ilustrovaný príkladom v kóde 4.4.

```
1 # Blok obsahujúci dve inštrukcie MOV a CMP
2 kontrola = MatchGraphJoker(name="kontrola",
3                 filt=lambda block: (len(block.lines) == 2 and
4                         block.lines[0].name == "MOV" and
5                         block.lines[1].name == "CMP"))
```

```
# Blok, do ktorého môžu vstupovať a vystupovať iba hrany, ktoré určíme
telo = MatchGraphJoker(name="telo", restrict_in=True, restrict_out=True)
# Posledný blok môže mať neobmedené množstvo vychádzajúcich hrán
koniec = MatchGraphJoker(name="koniec", restrict_out=False)
# Vytvoríme MatchGraph tým, že určíme hrany medzi blokmi (A >> B znamená
    hranu z A do B)
matcher = kontrola >> telo >> koniec
# Vizualizáciou overíme, či sme vytvorili správny vzor
open("to_match.dot", "w").write(matcher.dot())

def block_merge(dgs, graph):
    for sol in matcher.match(graph):
    # Miesto na úpravu grafu pre nájdené výskyty vzoru, ktorý sme hľadali
        ...

from miasm2.core.graph import DiGraphSimplifier
dgs = DiGraphSimplifier()
dgs.enable_passes([block_merge]) # Pridanie funkcie na zjednodušovanie
blocks_after = dgs(blocks) # aplikovanie na graf
open("cfg_after.dot", "w").write(blocks_after.dot())
```

Kód 4.4: Zjednodušovanie grafov v miasm frameworku.

4.2 Deobfuskátor

Samotný deobfuskátor sme implementovali v programovacom jazyku Python. Voľba programovacieho jazyka bola priamočiara, keďže sme sa rozhodli stavať na miasm frameworku.

Deobfuskátor je rozdelený do viacerých samostatných skriptov, pre jeho jednoduchšiu rozšíriteľnosť alebo prípadnú integráciu do iných programov. Okrem deobfuskačných skriptov sme vytvorili aj jeden skript na komunikáciu s používateľom, ktorého úlohou je spúšťanie deobfuskačných skriptov na základe vstupu od používateľa.

Všetky súčasti deobfuskátora sú implementované pre 32 bitovú architektúru x86. Upraviť jednotivé skripty na inú architektúru by malo byť uskutočniteľné vymenením tried, ktoré manipulujú s dátami.

4.2.1 Deobfuskácia falošného toku riadenia

Základom na odstránenie tejto obfuskačnej techniky je identifikovať a vyhodnotiť neprehľadný predikát. O-LLVM používa jeden konkrétny neprehľadný predikát, ktorý vyzerá nasledovne: $(y < 10 \,||\, x * (x-1) \,\%\, 2 == 0)$. Hodnota tohto predikátu je pre prirodzené x vždy True.

Tento neprehľadný predikát si môžeme rozdeliť na dve časti, časť pred znamienkom logického *alebo* a za ním. Prvá časť ($y < 10$) neovplyvní výsledok. Druhá časť ($x * (x-1) \% 2 == 0$) platí, keď hodnota ($x * (x-1)$) je deliteľná 2. Párne číslo krát nepárne číslo je párne číslo pre všetky prirodzené čísla a teda druhá časť predikátu bude platiť vždy. Na nájdenie a nahradenie tejto časti neprehľadného predikátu sme využili zjednodušovanie výrazov v miasm frameworku ako ukazuje kód 4.5.

```
def simp_opaque_bcf(e_s, e):
  global neprehladny
  # žolíky na jednotlivé časti predikátu
  jok1 = ExprId("jok1")
  jok2 = ExprId("jok2")
  jok3 = ExprId("jok3")
  # Snažíme sa nájsť (a * b) % 2
  to_match = ((jok1 * jok2)[0:32] & ExprInt32(1))
  result = MatchExpr(e,to_match,[jok1,jok2,jok3])
  if (result is False) or (result == {}):
    return e # Nenašli sme zhodu, vraciame pôvodný výraz
  # Skontrolujeme, že b == (a - 1)
  mult_term1 = expr_simp(result[jok1][0:32])
  mult_term2 = expr_simp(result[jok2][0:32])
  if mult_term2 != (mult_term1 + ExprInt(uint32(-1))):
    return e # Nezhoduje sa, vraciame pôvodný výraz
  # flag, ktorý nám signalizuje, že sme zjednodušili výraz
  neprehladny = 1
  # Nahradíme (a * (a - 1)) % 2 za 0
  return ExprInt32(0)

# Pridáme nami vytvorené zjednodušovanie do miasm frameworku
simplifications = {ExprOp :[simp_opaque_bcf]}
expr_simp.enable_passes(simplifications)
```

Kód 4.5: Zjednodušovanie neprehľadného predikátu falošného toku riadenia pomocou miasm frameworku.

Keď vieme identifikovať a aj vyhodnotiť neprehľadný predikát, môžeme začať prechádzať všetky vetvy toku riadenia pomocou symbolického vykonávania. Miasm framework bude prechádzať iba vetvami, ktoré sú dosiahnuteľné, lebo už vie zjednodušiť neprehľadný predikát. Ukážka inicializácie symbolického vykonávanie je v kóde 4.6.

```
bin_file = open(fname).read() # Načítanie súboru
bin_stream = bin_stream_str(bin_file)
mdis = dis_x86_32(bin_stream) # disasemblovanie blokov na adrese offset
disasm = mdis.dis_multibloc(offset._arg)
# vytvorenie a naplnenie objektu internej reprezenácie
ir = ir_a_x86_32(mdis.symbol_pool)
for bbl in disasm:
```

```
    ir.add_bloc(bbl)
# Inicializácia symbolov registrami požadovanej architektúry procesora
symbols_init = {}
for i, r in enumerate(all_regs_ids):
    symbols_init[r] = all_regs_ids_init[i]
# Vytvorenie stroja na symbolické vykonávanie
symb = symbexec(ir, symbols_init)
# Vykonanie bloku na adrese address a zjednodušenie výrazu určujúceho ďal
    ší blok/bloky
block = ir.get_bloc(address)
nxt_addr = symb.emulbloc(block)
simp_addr = expr_simp(nxt_addr)
```

Kód 4.6: Inicializácia symbolického vykonávania a vykonanie jedného bloku.

Symbolické vykonávanie usmerňujeme pomocou zásobníka, do ktorého si ukladáme adresu nasledujúceho bloku/blokov. Pokračujeme, kým nie je zásobník prázdny. Po symbolickom vykonaní každého bloku si uložíme na zásobník tiež informácie o tom, či obsahoval daný blok neprehľadný predikát, aktuálne hodnoty v registroch a v pamäti a adresu tohto bloku. Pokiaľ daný blok nie je z pôvodného programu, ale bol pridaný v procese obfuskácie, tak si ukladáme adresu najbližšieho rodiča, ktorý je z pôvodného programu.

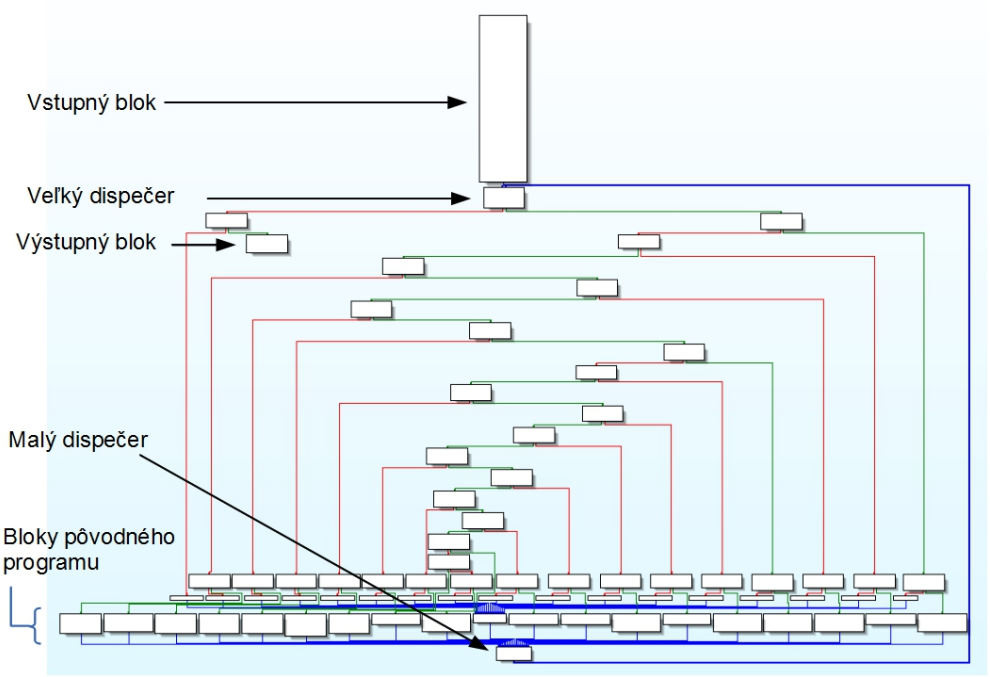
Informácie uložené na zásobníku nám pomáhajú identifikovať bloky z pôvodného programu. Blok, ktorý nebol obfuskovaný, neobsahuje neprehľadný predikát a teda blok bez neprehľadného predikátu je z pôvodného programu. Blok, ktorý bol obfuskovaný, obsahuje neprehľadný predikát a jeho rodič musí tiež obsahovať neprehľadný predikát. Keď natrafíme na blok obsahujúci neprehľadný predikát, tak pokiaľ jeho rodič neobsahuje neprehľadný predikát, tak ide o blok iba s neprehľadným predikátom pridaný v procese obfuskácie. Ak rodič obsahuje neprehľadný predikát, tak ide o blok pôvodného programu.

V procese obfuskácie technikou falošný tok riadenia pridáva O-LLVM bloky s jednou inštrukciou. Ide o inštrukciu skoku na ďalší blok. Takéto bloky tiež vynechávame. Adresy blokov patriacich pôvodnému neobfuskovanému programu sú exportované miasm frameworkom ako graf vo formáte DOT (graph description language).

4.2.2 Deobfuskácia vyrovnávania toku riadenia

V rámci vyrovnávania toku riadenia obfuskátor pridal za vstupný blok pôvodného programu sadu vetviacich inštrukcii, z ktorých prvú nazývame veľký dispečer. Bloky pôvodného programu pokračujú blokom malý dispečer, ktorý vracia riadenie veľkému dispečerovi.

Na rekonštrukciu pôvodného toku riadenia potrebujeme zistiť, ktoré bloky patrili pôvodnému neobfuskovanému programu a v akom poradí za sebou nasledujú. Oba prob-



Obr. 4.1: Graf vyrovnaného toku riadenia s vyznačenými relevantnými blokmi.

lémy môžeme riešiť pomocou symbolického vykonávania. Relevantné bloky sú vstupný blok, výstupný blok a rodičia bloku malý dispečer, ako to môžeme vidieť na obrázku 4.1.

Veľkému dispečerovi zodpovedá blok nasledujúci za vstupným blokom a malému dispečerovi zodpovedá jeho druhý rodič. Spolu s adresami dispečerov si pri prvom prechode funkciou, ktorej graf toku riadenia sa snažíme zistiť, uložíme adresy relevantných blokov. Počas druhého prechodu si značíme ku každému relevantnému bloku to, ktoré relevantné bloky nasledovali za ním. Výsledný graf toku riadenia je exportovaný miasm frameworkom vo formáte DOT (graph description language).

4.2.3 Deobfuskácia substitúcie inštrukcií

O-LLVM má zoznam, ktoré inštrukcie nahrádza a tiež akými inštrukciami ich nahrádza. Stačí teda v obfuskovanom programe hľadať obfuskované verzie inštrukcií. Tieto inštrukcie hľadáme a nahrádzame podobným spôsobom ako pri neprehľadnom predikáte z obfuskačnej techniky falošného toku riadenia. Pre každú dvojicu, pozostávajúcu z obfuskovanej a neobfuskovanej verzie výrazu, sme vytvorili jednu metódu pre zjednodušovanie výrazov v miasm frameworku. Keď identifikujeme postupnosť zodpovedajúcu niektorému obfuskovanému variantu inštrukcie, tak ju nahradíme pôvodnou inštrukciou. Deobfuskovanú verziu je možné zobraziť jednotlivo po základných blokoch podľa ich ofsetu.

Kapitola 5

Výsledky

Implementovaný deobfuskátor sme testovali na programoch, ktoré sme napísali v programovacích jazykoch C a C++. Programy sme obfuskovali pomocou O-LLVM na operačnom systéme Linux, takže vstupom pre deobfuskátor boli ELF súbory. Uvádzame výsledky pre nasledujúci program v kóde 5.1.

```
unsigned int funkcia(unsigned int n, unsigned int b, unsigned int c){
  unsigned int mod = n % 6;
  unsigned int result = 0;
  if (mod == 0) result = b + c ;
  else if (mod == 1) result = b + c ;
  else if (mod == 2) result = b - c;
  else if (mod == 3) result = b - c;
  else if (mod == 4) result = b & c;
  else if (mod == 5) result = b ^ c;
  else result = b | c;
  return result;
}

int main(){
  funkcia(47, 48, 49);
  return 0;
}
```

Kód 5.1: Zdrojový kód testovacieho programu.

Graf toku riadenia funkcie "funkcia" z tohto programu v pôvodnej neobfuskovanej verzií sa nachádza na obrázku 5.1. Graf toku riadenia tej istej funkcie po aplikovaní obfuskačnej techniky vyrovnávanie toku riadenia sa nachádza na obrázku 5.2. Na obrázku 5.4 sa nachádza graf toku riadenia funkcie "funkcia" po aplikovaní obfuskačnej techniky falošného toku riadenia jedným prechodom na všetky základné bloky. Substitúcia inštrukcii nemení tok riadenia a preto graf po aplikovaní tejto obfuskačnej techniky neuvádzame.

5.1 Vyrovnávanie toku riadenia

Na obrázku 5.2 môžeme vidieť, že v porovnaní s pôvodným grafom toku riadenia (obrázok 5.1), vyrovnaný graf toku riadenia obsahuje oveľa viac základných blokov a na pôvodný graf sa vôbec nepodobá. Polohu základných blokov pôvodného programu vie nami implementovaný deobfuskátor identifikovať aj s informáciou o ich poradí. O tom svedčí graf toku riadenia na obrázku 5.3, kde môžeme vidieť 7 vetiev pôvodného programu a adresy základných blokov patriacich pôvodnému programu.

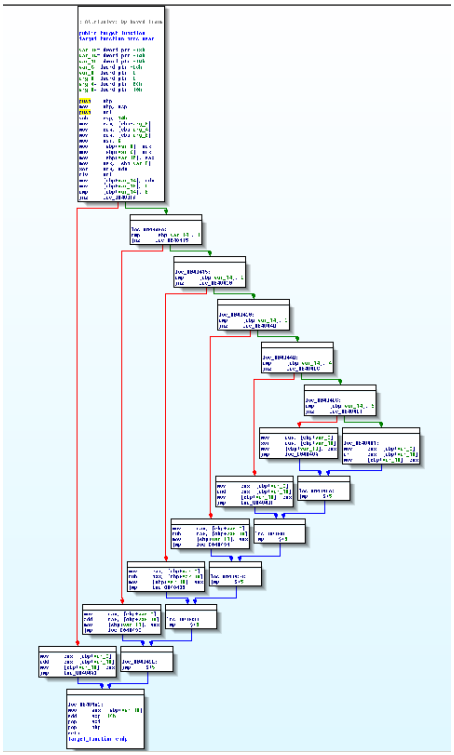
5.2 Falošný tok riadenia

Podobne ako pri technike obfuskácie vyrovnávania toku riadenia, aj technika falošného toku riadenia pridáva veľké množstvo nových základných blokov (obrázok 5.4) a odstraňuje podobnosť s pôvodným grafom (obrázok 5.1). Nami implementovaný deobfuskátor vie vďaka vedomosti o neprehľadnom predikáte a miestach pridávania nových blokov v O-LLVM zrekonštruovať graf toku riadenia obfuskovanej funkcie. Zrekonštruovaný graf toku riadenia je na obrázku 5.5. Obsahuje rovnako ako pôvodný graf toku riadenia 7 vetiev a všetky základné bloky patriace pôvodnému programu. Celkový počet základných blokov grafu toku riadenia je o niečo väčší ako v grafe neobfuskovaného programu. Dôvodom je, že O-LLVM v procese obfuskácie rozdelí základné bloky končiace podmienečným skokom na dva základné bloky, samotný blok bez podmienečného skoku je jeden nový základný blok a druhý je blok s podmienečným skokom. Tieto bloky ponechávame rozdelené a oba sa nachádzajú v deobfuskovanom grafe toku riadenia.

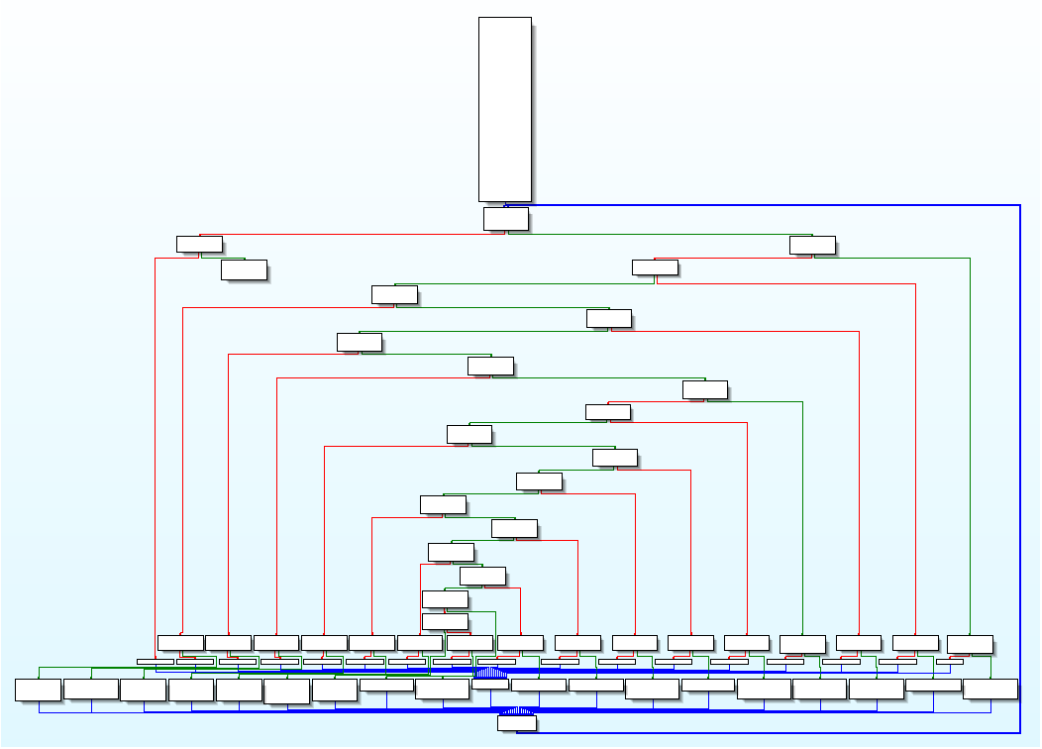
5.3 Substitúcia inštrukcií

Zo substituovaných inštrukcií nami implementovaný deobfuskátor dokáže nájsť a nahradiť na pôvodnú formu obfuskované varianty inštrukcií XOR a OR. Na obrázku 5.6 a) sa nachádza základný blok neobfuskovaného programu obsahujúci XOR dvoch premenných. Časť b) znázorňuje obfuskovaný variant toho základného bloku. Časti c) a d) sú v internej reprezentácií miasm frameworku. Časť c) znázorňuje obfuskovaný výpočet a časť d) zjednodušenie vytvorené nami implementovaným deobfuskátorom. Zjednodušený výraz zodpovedá pôvodnej verzii, ktorá je XOR dvoch premenných.

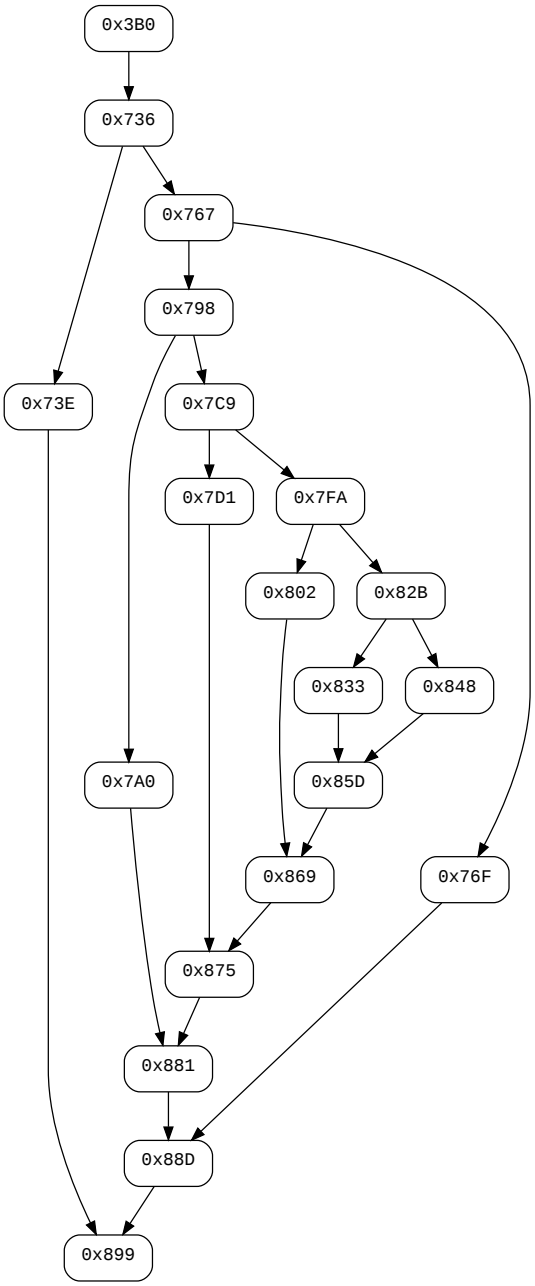
Na obrázku 5.7 a) sa nachádza základný blok neobfuskovaného programu obsahujúci duhú inštrukciu, ktorej obfuskovaný variant vieme deobfuskovať a to OR dvoch premenných. Časť b) znázorňuje obfuskovaný variant toho základného bloku. Časti c) a d) sú v internej reprezentácií miasm frameworku. Časť c) znázorňuje obfuskovaný výpočet a časť d) zjednodušenie vytvorené nami implementovaným deobfuskátorom. Zjednodušený výraz zodpovedá pôvodnej verzii, ktorá je OR dvoch premenných.



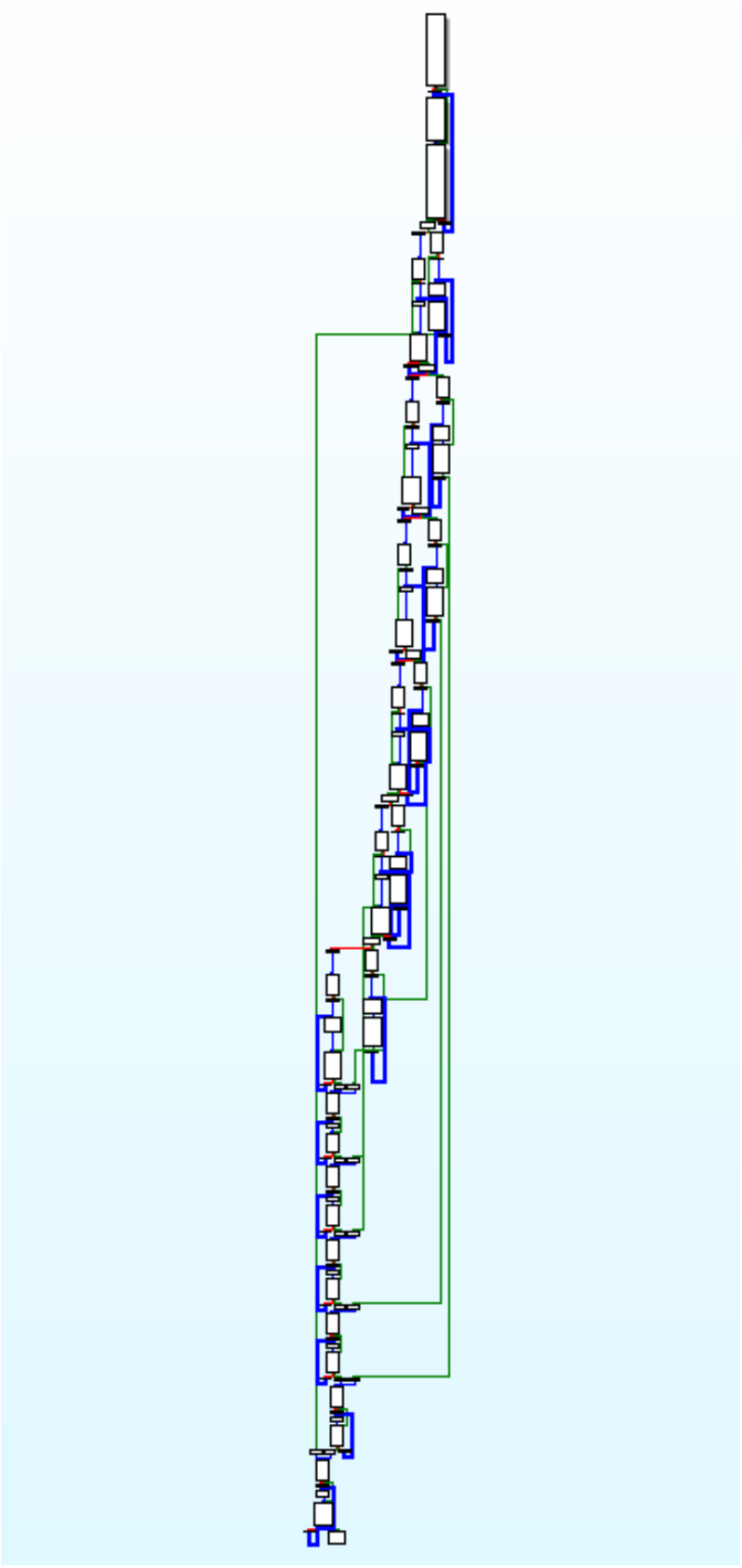
Obr. 5.1: Graf toku riadenia testovacieho programu.



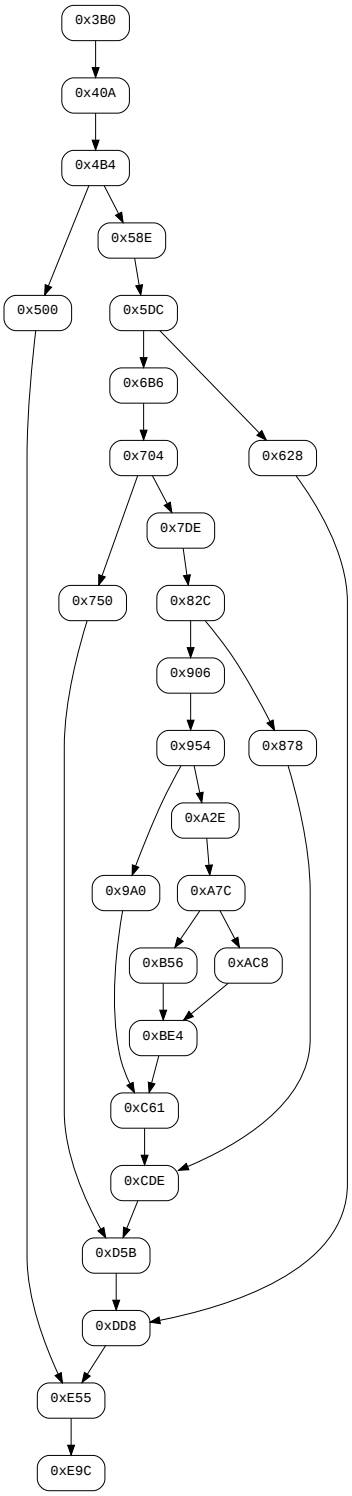
Obr. 5.2: Graf toku riadenia po jeho vyrovnaní.



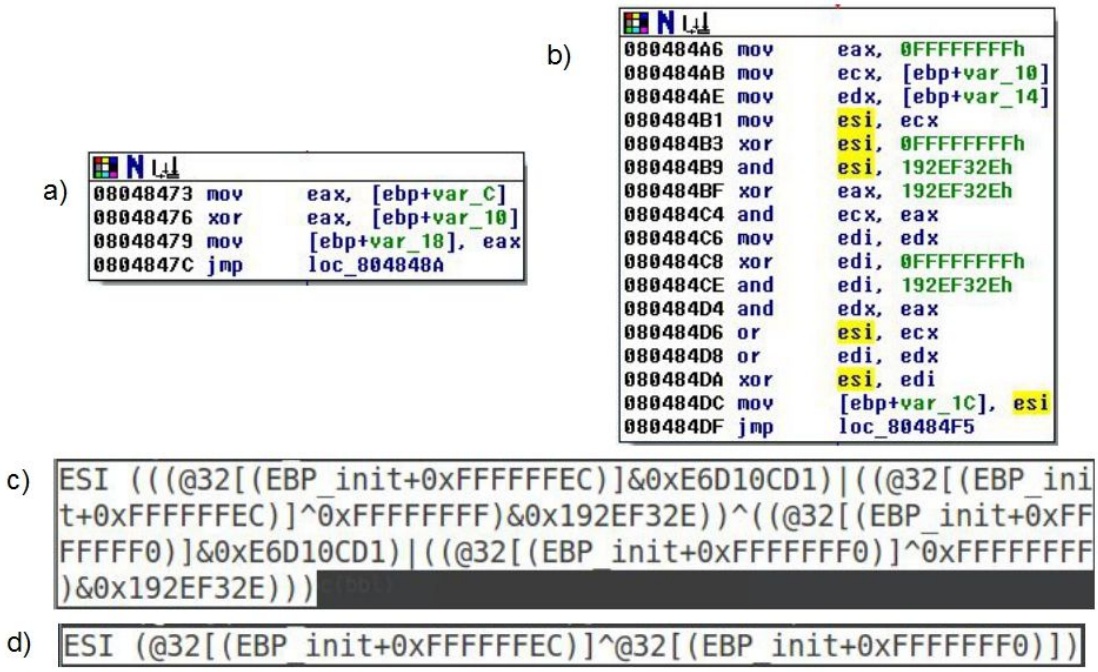
Obr. 5.3: Graf toku riadenia po deobfuskovaní obfuskačnej techniky vyrovnávajúcej tok riadenia.



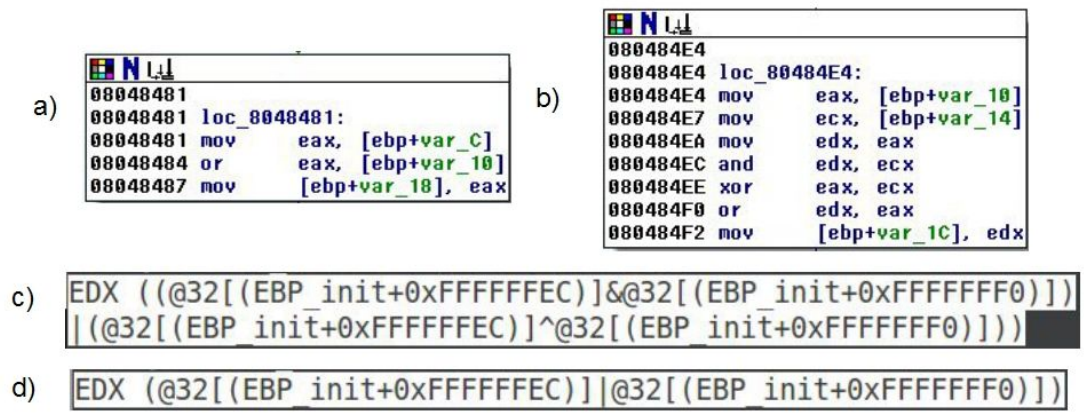
Obr. 5.4: Graf toku riadenia po aplikovaní obfuskačnej techniky falošného toku riadenia.



Obr. 5.5: Graf toku riadenia po deobfuskovaní obfuskačnej techniky falošného toku riadenia.



Obr. 5.6: Deobfuskácia substitúcie inštrukcie XOR. a) Originálny základný blok. b) Obfuskovaný základný blok c) Reprezentácia obfuskovaného výpočtu v internej reprezentácii miasm frameworku. d) Zjednodušený výraz v internej reprezentácii miasm frameworku.



Obr. 5.7: Deobfuskácia substitúcie inštrukcie OR. a) Originálny základný blok. b) Obfuskovaný základný blok c) Reprezentácia obfuskovaného výpočtu v internej reprezentácii miasm frameworku. d) Zjednodušený výraz v internej reprezentácii miasm frameworku.

Záver

Obfuskácia si našla svoje umiestnenie vo vývoji softvéru. Na jednej strane si firmy chránia pomocou obfuskácie duševné vlastníctvo. Na druhej strane tvorcovia malvéru využívajú obfuskáciu na to, aby sťažili jeho odhalenie.

Pre lepšie oboznámenie sa s problematikou deobfuskácie sme si naštudovali a spracovali základný prehľad obfuskačných a aj deobfuskačných techník. Vďaka tomuto prehľadu sme dokázali lepšie pochopiť aj pozadie jednotlivých obfuskačných techník implementovaných v O-LLVM; nielen to, ako sú implementované, ale vieme napríklad posúdiť aj ich kvalitu a pomohlo nám to aj pri výbere spôsobu použitého na ich deobfuskáciu.

Implementáciu deobfuskátora sme stavali na miasm frameworku. Tento framework je veľmi slabo zdokumentovaný. Preto sme pre nás aj pre prípadných ďalších záujemcov spísali menší prehľad jeho funkcionality aj s príkladmi využitia.

Cieľom práce bolo implementovať deobfuskátor odstraňujúci tri obfuskačné techniky (vyrovnávanie toku riadenia, falošný tok riadenia a substitúcia inštrukcií), ktoré sú implementované v O-LLVM. Funkčnosť deobfuskátora sme testovali na nami vytvorených programoch, ktoré sme obfuskovali pomocou O-LLVM.

Pri obfuskačných technikách ovplyvňujúcich tok riadenia deobfuskátor identifikoval všetky pôvodné bloky patriace pôvodnému neobfuskovanému programu. Pri filtrovaní pôvodných blokov z obfuskovaného programu technikou falošného toku riadenia sa v deobfuskovanom grafe toku riadenia nachádza viac základných blokov ako mal pôvodný program. Dôvodom je, že O-LLVM rozdeľuje v procese obfuskácie základné bloky končiace podmienečným skokom na dva. Takéto bloky necháva deobfuskátor rozdelené. Zo substituovaných inštrukcii deobfuskátor správne identifikuje a nahrádza pôvodnými inštrukciami obfuskované varianty inštrukcií XOR a OR.

Literatúra

[1] Hiralal Agrawal and Joseph R Horgan. Dynamic program slicing. In *ACM SIGPLAN Notices*, volume 25, pages 246–256. ACM, 1990.

[2] Frances E Allen. Control flow analysis. In *ACM Sigplan Notices*, volume 5, pages 1–19. ACM, 1970.

[3] Srinivasan Chandrasekharan and Saumya Debray. Improving reverse engineering of obfuscated code.

[4] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. Technical report, Department of Computer Science, The University of Auckland, New Zealand, 1997.

[5] Christian Collberg, Clark Thomborson, and Douglas Low. Breaking abstractions and unstructuring data structures. In *Computer Languages, 1998. Proceedings. 1998 International Conference on*, pages 28–38. IEEE, 1998.

[6] Christian Collberg, Clark Thomborson, and Douglas Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 184–196. ACM, 1998.

[7] Chris Eagle. *The IDA pro book: the unofficial guide to the world's most popular disassembler*. No Starch Press, 2011.

[8] Neil D Jones, Carsten K Gomard, and Peter Sestoft. *Partial evaluation and automatic program generation*. Peter Sestoft, 1993.

[9] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. Obfuscator-LLVM – software protection for the masses. In Brecht Wyseur, editor, *Proceedings of the IEEE/ACM 1st International Workshop on Software Protection, SPRO'15, Firenze, Italy, May 19th, 2015*, pages 3–9. IEEE, 2015.

[10] Tımea László and Ákos Kiss. Obfuscating C++ programs via control flow flattening. *Annales Universitatis Scientarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Computatorica*, 30:3–19, 2009.

[11] Douglas Low. Java control flow obfuscation. *Master of Science Thesis, Department of Computer Science, The University of Auckland*, 1998.

[12] Marius Popa. Techniques of program code obfuscation for secure software. *Journal of Mobile, Embedded and Distributed Systems*, 3(4):205–219, 2011.

[13] Sebastian Schrittwieser, Stefan Katzenbeisser, Johannes Kinder, Georg Merzdovnik, and Edgar Weippl. Protecting software through obfuscation: Can it keep pace with progress in code analysis? *ACM Computing Surveys (CSUR)*, 49(1):4, 2016.

[14] Javad Yousefi, Yasser Sedaghat, and Mohammadreza Rezaee. Masking wrong-successor control flow errors employing data redundancy. In *Computer and Knowledge Engineering (ICCKE), 2015 5th International Conference on*, pages 201–205. IEEE, 2015.

Príloha

CD

Zdrojové súbory deobfuskátora sa nachádzajú na priloženom CD.