Univerzita Komenského
Fakulta Matematiky, Fyziky a Informatiky
Katedra Informatiky

Obfuskácia zdrojového kódu

Diplomová práca

Ľubomír Karaba

Diplomový vedúci: Mgr. Miroslav Demeter

Bratislava Máj 2007

Obfuskácia zdrojového kódu

Diplomová práca

Ľubomír Karaba

Univerzita Komenského v Bratislave
Fakulta Matematiky, Fyziky a Informatiky
Katedra informatiky

Informatika

Diplomový vedúci: Mgr. Miroslav Demeter

Bratislava 2007

Čestne prehlasujem, že túto diplomovú prácu som vypracoval samostatne, len s použitím uvedenej literatúry.

Bratislava, máj 2007 Ľubomír Karaba

Abstrakt

Obfuskácia - zahmlievanie zdrojového kódu programov je uznávaná ako lacný prístup k ochrane programov a ich algoritmov ako duševného vlastníctva. Nikdy nemôže poskytnúť absolútnu ochranu, cieľom je však transformovať pôvodný zdrojový kód takým spôsobom, že investície potrebné na jeho pochopenie, prípadnú zmenu, alebo znovupoužitie sa neoplatia. Obfuskačné transformácie zdrojového kódu za účelom jeho zneprehľadnenia môžu byť aplikované vo všetkých životných štádiách programu - buď priamo v zdrojovom kóde, v skompilovanom medzikóde, alebo v strojovom kóde. Medzikód je typický najmä pri platformovo nezávislých programovacích jazykoch, na konkrétnych operačných systémoch sa potom tie isté skompilované programy spúšťajú pomocou interpretera. Príkladom medzikódu je *bytecode* používaný v Jave.

Kľúčové slová: obfuskácia, spätné inžinierstvo, ochrana softvéru, obfuskácia toku riadenia

Predhovor

Cieľom diplomovej práce je zoznámiť čitateľa s pojmami obfuskácia, deobfuskácia, s myšlienkami, ktoré boli vymyslené a implementované, s ich bezpečnosťou. Naviac chceme vyvinúť vlastnú metódu (resp. postup), ktorá by odhaľovala slabiny v doterajších riešeniach a navrhovala alternatívne riešenia. Ciele sa dajú zoradiť do hlavných skupín:

1. Opísať kontext ochrany softvéru, spätné inžinierstvo, zmieniť sa o alternatívnych spôsoboch ochrany softvéru.

2. Opísať definíciu obfuskácie, miery kvality obfuskácie, doteraz navrhnuté riešenia, spraviť výpočet diel, ktoré priniesli nové a zaujímavé myšlienky.

3. Odhaliť nedostatky v obfuskačných a deobfuskačných prácach a popísať vlastnú metódu. Cieľom nie je implementovať vlastný automatický nástroj, len podrobne popísať algoritmus, ktorý by bol implementovaný automatickým nástrojom.

4. Spraviť rozbor kvality našej obfuskácie, jej prípadné nedostatky a možnosti vylepšenia.

Obsah

Úvod

V úvodnej kapitole sa budeme venovať ochrane softvéru. Najskôr v 1.1 definujeme pojem spätné inžinierstvo, čo je jeho cieľom a tiež popíšeme, aké škody môže napáchať. Boj proti spätnému inžinierstvu sa stal motiváciou pre rozvoj viacerých techník, ktoré si rozoberieme v sekcii 1.2.

Jedným z účinných prostriedkov je aj obfuskácia, doterajším výsledkom v tejto oblasti je venovaná celá druhá kapitola. V nej najskôr v sekcii 2.1 definujeme metriky kvality obfuskácie, aby sme mohli jednotlivé postupy porovnávať. Obfuskačné transformácie sa delia do troch hlavných tried. Pre triedy všeobecne ako aj pre jednotlivé transformácie popíšeme ich delenie a vzťah k definovaným metrikám v sekcii 2.2. Pokiaľ bude možné, budeme sa snažiť upozorniť aj na možnosti automatickej deobfuskácie týchto techník. Delenie na statické a dynamické a súhrn týchto automatických deobfuskačných techník sa nachádza v 2.3. Na záver tejto kapitoly v 2.4 príde ešte zoznam prác, ktoré buď priniesli niečo nové do obfuskácie alebo do deobfuskácie, priniesli implementácie vybraných postupov, empirické výsledky alebo teoretické dôkazy o zložitosti.

Tretia kapitola je už venovaná nášmu návrhu postupu, ktorý má viacero krokov a každý z nich je popísaný v samostatnej sekcii. V 3.1 je sumár týchto krokov. Prvý krok v (3.2) je prípravný, zjednodušuje ostatné kroky a zbiera potrebné informácie o programe. Ďalej nasledujú kroky, ktoré transformujú program na nečitateľný v sekciách 3.3 až 3.9. Technické vylepšenia algoritmu, ktoré však nepatria do funkčnosti, bez ktorej sa nezaobíde, sú popísané v 3.10. Hľadaný kompromis medzi bezpečnosťou a výkonnosťou algoritmu je možné konfigurovať. Tieto konfigurovateľné obmedzenia sú popísané v 3.11. Analýze vzťahu k definovaným metrikám sa venuje sekcia 3.12.

Kapitola 1

Ochrana softvéru proti spätnému inžinierstvu

1.1 Spätné inžinierstvo

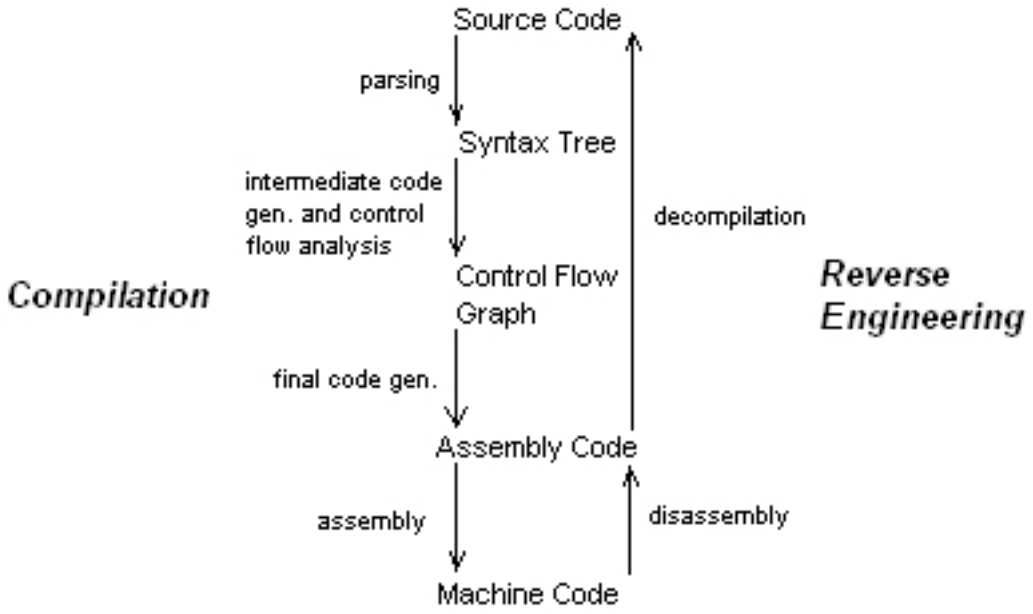
Najskôr si definujeme dva pojmy, ktoré sa budú vyskytovať v celom texte:

Definícia 1.1. *Spätným inžinierom (v našom prípade) alebo tiež útočníkom nazveme osobu, ktorej cieľom je nelegálne využiť existujúci kód na získanie konkurenčnej výhody.*

Poznámka 1.1. Mimo našej práce nemusí byť práca spätného inžiniera vždy nelegálna, môže mať napríklad snahu vylepšiť len pre svoje potreby existujúci program, ku ktorému nemá zdrojový kód, môže to byť rovno autor programu, ktorý sa z ľubovoľných príčin nevie dostať k zdrojákom.

Definícia 1.2. *Spätným inžinierstvom sa nazýva postup, pri ktorom sa zo spustiteľného binárneho súboru získa pôvodný zdrojový kód. Naviac zahŕňa kroky, počas ktorých sa spätný inžinier snaží získať informácie o funkcionalite programu. Ďalšou situáciou, kedy spätný inžinier je na strane dobra je tá, keď sa jedná o človeka, ktorý sa snaží rozpoznať škodlivý softvér ako napríklad vírus, červ a podobne.*

Jednotlivé etapy tohto postupu sú znázornené na obr. 1.1 spolu s krokmi kompilovania zdrojového kódu do spustiteľnej podoby. Medzi zdrojovým kódom a skompilovaným binárnym spustiteľným súborom sa často nachádza ešte medzikód, ktorý má štruktúru bližšiu výsledným strojovým inštrukciám. Z hľadiska obfuskácie je aj tento medzikód významný, ako si povieme neskôr, keď budeme rozoberať doteraz preskúmané možnosti obfuskácie.



Obrázok 1.1: Etapy kompilácie a spätného inžinierstva

Spravme si zoznam prípadov, keď spätné inžinierstvo slúži ako možnosť dostať sa k chráneným algoritmom a ich pochopenie vedie k zneužitiu:

- Spätný inžinier môže získať cudziu prácu, ktorá obsahuje netriviálne algoritmy a postupy, ktorých vývoj stál množstvo času a prostriedkov. Pochopením a použitím týchto algoritmov vo svojich programoch získa výhodu. Tento postup sa často nazýva pojmom krádež duševného vlastníctva.

- Ďalšie riziko hrozí, ak zdrojový kód obsahuje techniky, ktoré majú softvér, ktorý nie je z kategórie open source alebo free, chrániť proti nelegálnemu kopírovaniu a používaniu. Bežným príkladom je krabicový softvér (softvér, ktorý nebol vyrobený na objednávku nejakej organizácie) a získanie algoritmu, podľa ktorého sa vytvárajú registračné kľúče podľa zadaného mena používateľa, prípadne odstránenie tej časti kódu, ktorá zabezpečuje, aby si aplikácia pri inštalácii pýtala tento registračný kľúč.

- Chránené proti nelegálnemu kopírovaniu a prehrávaniu môžu byť nielen spustiteľné programy, ale taktiež filmy, hudba, prípadne iné multimédiá. V nich sa môže ukrývať informácia o autorských právach, o oprávnenom používateľovi. Odhalenie tejto informácie by pomohlo jej odstráneniu.

- Nakoniec spomeňme distribuované aplikácie, ktoré sa objavili v nedávnej dobe a ktoré

riešia jeden problém obrovského rozmeru tak, že rozkúskujú problém na množstvo menších, pošlú dáta jednotlivým uzlom, tie vypočítajú výsledok a pošlú späť koordinátorovi. Keby sa podarilo zmeniť algoritmus v jednom uzle, aby posielal nazad nesprávne výsledky, mohla by sa jeho chyba šíriť a premietnuť do konečného výsledku, ktorý by bol úplne nesprávny. Tu má však obfuskácia trošku ľahšiu úlohu, pretože jednotlivé počítajúce uzly môžu produkovať zakódované výsledky.

Špeciálne sa problematika spätného inžinierstva stala aktuálnou pri vzniku moderných, objektovo orientovaných jazykov typu Java, ktoré sú platformovo nezávislé, čo znamená, že zdrojový kód sa najskôr upraví do formy *bytecode*, ktorý je prenositeľný a spustiteľný pomocou virtuálneho stroja *JVM - Java Virtual Machine* na ľubovoľnej platforme. Tento *bytecode* (assembly code na obrázku 1.1) obsahuje kompletnú informáciu o zdrojovom kóde a nie je ťažké ho dekompilovať. Java, ako najvhodnejší a najznámejší príklad jazykov, ktorých zdrojové programy sú prenášané v čiastočne preložených formách, sa stala jazykom, v ktorom budeme uvažovať, a v ktorom budú uvádzané príklady v tejto práci.

1.2 Metódy ochrany proti spätnému inžinierstvu

Proti rizikám spätného inžinierstva bolo navrhnutých viacero spôsobov ochrany, každá technika má svoje výhody aj nedostatky, ktoré budú v ďalšom texte popísané. Takéto rozdelenie techník ochrany softvéru bolo spomenuté napríklad v [6].

1.2.1 Enkrypcia

Spočíva v tom, že sa skompilovaný zdrojový kód zakóduje podľa nejakého kľúča. Takýto kód nie je spustiteľný a treba mať kľúč na jeho rozšifrovanie a následné spustenie. Samozrejme, že tento kľúč nesmie byť známy, toto sa dá docieliť napríklad špeciálnym hardvérom, v ktorom je tento kľúč implementovaný. Tento koprocesor bude zakryptovaný kód odkodúvať a posielať inštrukcie priamo procesoru, ktorý ich vykoná. Zdrojový kód teda vidí len privátna pamäť koprocesora, do ktorej užívateľ nemá právo nahliadnuť. Táto technika má svoje muchy napríklad v tom, že každý počítač, na ktorom má byť program spustený musí byť vybavený ďalším kusom hardvéru, s čím sa koncoví užívatelia určite nestotožnia, pretože ich to bude stáť investície a námahu navyše, pričom nepocítia žiadny efekt. Z toho sa dá usudzovať, že táto bezpečná technika sa používa a bude používať len pri veľmi špecifických a na bezpečnosť citlivých projektoch. Táto technika ochrany sa spomína napríklad v [6].

1.2.2 Spúšťanie na strane servera

Ďalším bezpečným spôsobom, ako ochrániť duševné vlastníctvo autora je spúšťanie aplikácií cez sieť na strane servera. To znamená, že na klientskom počítači sa nachádzajú len tie úseky kódu, ktoré nie je potrebné chrániť, zložité chránené algoritmy sa nachádzajú na serveri, klient im pošle svoju požiadavku na výpočet a server vráti odpoveď. Keď uvažujeme server, ktorý by bol dokonale zabezpečený, aj táto metóda je maximálne bezpečná. Problémy su však inde: klientský počítač musí byť pripojený na sieť, v ktorej sa nachádza server s chránenými algoritmami, čiže ani nie je zaručené, že sa požiadavka na server dostane. Aj výkon aplikácie je obmedzený rýchlosťou a dostupnosťou siete, cez ktorú sa pristupuje k serveru. Naviac môže dôjsť k preťaženiu servera, ak bude od neho naraz veľa klientských počítačov požadovať výsledky.

1.2.3 Obfuskácia

Pod týmto názvom, ktorého čisto slovenským ekvivalentom môže byť buď zahmlievanie alebo zatemnenie, sa ukrývajú také transformácie zdrojového alebo skompilovaného kódu, ktoré ponechajú funkcionalitu programu nezmenenú, ale snažia sa ho spraviť nepochopiteľný pre čitateľa. Collberg a spol. v článku [6] definoval obfuskáciu nasledovne:

Definícia 1.3 (Zahmlievajúcej transformácie)**.** *Nech $\tau : P \mapsto P'$ je transformácia zdrojového programu do cieľového programu. Potom $\tau : P \mapsto P'$ je zahmlievajúca transformácia ak P a P' majú rovnaké pozorovateľné správanie. Formálnejšie, ak $\tau : P \mapsto P'$ má byť zahmlievajúca transformácia, musia byť splnené nasledujúce podmienky:*

- *Ak P neskončí alebo skončí s chybou, tak P' môže, ale nemusí skončiť.*
- *Inak musí skončiť aj P' a musí dávať rovnaké výsledky ako P, okrem tohto môže mať aj bočné efekty ako je napríklad produkovanie súborov.*

Samozrejme, nedá sa zaručiť, že žiadny človek v kombinácii s automatickými nástrojmi nedokáže zistiť, ako daný kód pracuje. Dôležité je však to, aby sa čas a prostriedky, ktoré sa vynaložia na analýzu zdrojového kódu neoplatili a aby prípadný vlastný vývoj softvéru bol návratnejšou investíciou. V súčasnosti existuje viacero obfuskátorov - programov, ktorých vstupom je zdrojový kód a podporné údaje, ktoré určujú napríklad, o koľko najviac sa môže zhoršiť výkon programu pri jeho transformácii. Zahmlievajúce transformácie sa potom cyklicky aplikujú dovtedy, kým nie je prekročené toto obmedzenie na povolené zníženie výkonu. Výstupom je transformovaný kód s rovnakým pozorovateľným správaním, ktorý

je však nečitateľný a mätúci, prípadne produkuje aj zbytočné výstupy, ktorých existencia tiež vedie k pomýleniu osoby, ktorá číta tento zdrojový kód.

Niektoré techniky obfuskácie sa nevyužívajú obfuskátormi, ale sú používané priamo programátorom, aby bola odstránená akákoľvek pravidelnosť a tým aby bola znížená šanca na automatické deobfuskovanie. Obfuskátory sa líšia svojou účinnosťou podľa toho, ktoré metódy z množstva existujúcich implementujú. Cieľom je zmiasť spätného inžiniera, prípadne automatický deobfuskátor, ktorí sa snažia získať pôvodný zdrojový kód z obfuskovaného.

Význam obfuskácie zdôrazňujú príbuzné postupy, ako zaručiť ochranu proti softvérovému pirátstvu. Tieto postupy sa priamo spoliehajú na obfuskáciu, bez nej nemajú význam, pretože po ich objavení sa dajú odstrániť a ponechajú funkčný kód alebo súbor bez vloženej ochrany. Jedná sa o:

- **Watermarking** znamená vloženie informácie o autorovi alebo o legálnom užívateľovi do zdrojového kódu, mediálneho súboru a podobne tak, aby nebolo ovplyvnené správanie sa programu prípadne výzor mediálneho súboru pri prehrávaní. Snahou watermarkingu je vložiť túto informáciu tak, aby sa nedala odhaliť a v prípade odhalenia sa nedala odstrániť. Vďaka tejto technike sa dá odhaliť pôvodný autor diela, prípadne pri nelegálnom kopírovaní sa dá zistiť, ktorý legálny užívateľ poskytol svoju kópiu na skopírovanie.

- Pri **tamperproofingu** sa jedná o to, aby bol zdrojový kód pri ľubovoľnej zmene nespustiteľný a poskytuje možnosť zistiť, kde nastala zmena, prípadne túto zmenu odstrániť, keďže zdrojový kód môže byť v niektorých prípadoch zmenený aj neúmyselne. Táto technika sa priamo spolieha na obfuskáciu, bez nej nemá význam.

Obfuskácia je zo spomenutých techník najvhodnejšia pre prípady, keď je treba použiť rýchlu a lacnú stratégiu ochrany proti softvérovému pirátstvu, ktorá je aplikovateľná bez špeciálnych podmienok, akými boli v predchádzajúcich metódach existencia pripojenia na vzdialený server ponúkajúci služby alebo existencia špeciálneho kryptovacieho a dekryptovacieho hardvéru.

Momentálne je známych mnoho metód, ktoré sa líšia v tom, ako dokážu zmiasť človeka - spätného inžiniera alebo automatický deobfuskátor, o koľko sa zníži výkon aplikácie skompilovanej z transformovaného kódu, ako ľahko sa dá zistiť, že kód je vôbec obfuskovaný. Menej známe sú výsledky, ktoré hovoria o tom, aké vlastnosti ktorá metóda obfuskácie má vzhľadom na predchádzajúce metriky. Preto v nasledujúcej kapitole najskôr uvediem štandardne udávané metriky, podľa ktorých sa posudzuje účinnosť zahmlievania.

Na záver je vhodné poznamenať, že metódy obfuskácie môžu byť použité aj na záporné účely, napríklad zahmlený Javascriptový program na webovej stránke môže sústavne popri svojej užitočnej práci vyhadzovať pop-up okná, taktiež sa obfuskovanie kódu využíva pri tvorbe počítačových vírusov, pri tvorbe spamu a podobných škodlivých kódoch. V nedávnej minulosti sa do popredia dostal program *Skype*, ktorý poskytuje komunikačné služby cez internet. Tento softvér používa veľké množstvo obfuskácie, dekrypcie v pamäti, obsahuje stovky kontrolných súčtov a ďalších techník proti spätnému inžinierstvu. Tento program vie rafinovane prejsť cez Firewally, NAT, Proxy servery, dáta posiela cez vybrané uzly - počítače s nainštalovaným *Skype* softvérom a toto správanie sa nedá vypnúť. Prišla preto prirodzená otázka, či nie je *Skype* nebezpečný. Napriek týmto zložitým zahmlievacím technikám sa podarilo časť funkcionality a protokol odhaliť. Bezpečnostná analýza *Skype* v skratke sa nachádza v prezentácii [9].

Kapitola 2

Obfuskácia

V tejto kapitole si popíšeme konkrétne metódy zahmlievania kódu a ich delenie podľa toho, či transformujú usporiadanie kódu, dátové štruktúty alebo poradie vykonávania príkazov. Popíšeme ich účinnosť, odolnosť, cenu, nenápadnosť. Cez jednotlivé práce, ktoré tieto techniky dávali dokopy sa budeme venovať aj histórii obfuskácie. Spomenieme tiež obfuskáciu z druhej strany - deobfuskáciu. Znalosť nástrojov, ktoré má deobfuskácia k dispozícii dáva možnosť vytvárať voči deobfuskáciám odolnejšie obfuskácie.

2.1 Metriky kvality obfuskácie

Celková kvalita jednotlivých zahmlievacích metód sa posudzuje ako kombinácia štyroch rôznych metrík - účinnosti, odolnosti, ceny a nenápadnosti. Toto rozdelenie bolo spomenuté v spoločných prácach od C. Collberga, C. Thomborsona a D. Lowa [6], [7], [15].

2.1.1 Účinnosť

Po anglicky *potency*[1] - obfuskovacia transformácia je tým účinnejšia, čím viac dokáže zmiasť a znechutiť osobu, ktorá sa snaží zistiť funkciu časti kódu pred obfuskovaním. Toto znechutenie možno popísať ako komplexnosť programu, ktorá sa zvyšuje počtom podmienok, zvýšovaním hĺbky vnárania cyklov a rekurzívnych funkcií, zvyšovaním zložitosti dátových štruktúr, medziblokovými závislosťami premenných a pod. Príkladom menej účinnej obfuskácie je poprehadzovanie a vzájomné prepletenie toku riadenia programu alebo pridanie zbytočných príkazov a informácií do zdrojového kódu. Takéto transformácie sú relatívne poraziteľné automatickým nástrojom, ktorý posplietané vetvy programu

[1] definícia sa nachádza v [6] na strane 7.

usporiada do orientovaného grafu, z ktorého je jasne vidieť, ktoré príkazy po sebe nasledujú. Taktiež dokážu odstrániť zbytočne pridané úseky kódu tak, že odhalia dosiahnuteľné príkazy. Avšak človek bez pomoci automatického nástroja sa ľahko zapletie, ak je vetvenie príliš zložité, pretože by si naraz musel pamätať príliš veľa informácií o jednotlivých vetvách, najmä v miestach programu, kde sa stretáva viacero vetiev.

2.1.2 Odolnosť

Resilience[2], alebo odolnosť sa nazýva schopnosť obfuskovacej transformácie odolať deobfuskátoru - automatickému nástroju, ktorého úlohou je nájsť niektoré typy obfuskácie a odstrániť ich. Iné techniky môžu byť voči deobfuskátoru úplne odolné, sú to najmä jednosmerné transformácie, ako je zmena názvov premenných - automat nikdy nedokáže zistiť, ako človek pomenoval svojím vlastným uvážením premenné a metódy, okrem jednoduchých typu *getVlastnost()*, *setVlasnost()* a pod. Ich význam je však odhaliteľný človekom, ktorý skúma, čo v skutočnosti robia. Ďalšou automaticky neodstrániteľnou transformáciou zdrojového kódu môže byť napríklad posplietanie, rozdeľovanie, spájanie a vnáranie tried, ich metód, premenných. Automatický nástroj nie je schopný zistiť, či sú triedy rozložené tak ako sú kvôli logickému usporiadaniu a zákonom objektovo orientovaného programovania, alebo z dôvodu pridania zložitosti a neprehľadnosti. Často sa používa nedosiahnuteľný kód, ktorý sa nikdy nevykoná, ale to nie je hneď jasné, pretože sa skrýva sa zbytočné, klamlivé podmienky, ktoré zdanlivo rozvetvujú program, ale pritom je vždy splnená podmienka len pre jednu vetvu. Ich kvalita určuje odolnosť obfuskovacej transformácie.

2.1.3 Cena

Transformovaný program P' môže obsahovať predikáty, ktorých výpočet je zložitejší, môže produkovať zbytočné súbory, viac krát opakovať cykly, ako v pôvodnom programe P. *Cost*[3] - čiže cena, je metrika, ktorá opisuje, koľko zdrojov naviac program P' potrebuje a o koľko sa znižuje jeho výkon - o koľko dlhšie trvá vykonávanie. V niektorých prípadoch sa môže zložitosť aj mierne znížiť, takýto jav nastáva pri method inliningu, čo je metóda, ktorá sa používa aj mimo kontextu obfuskácie. Ide o nahradenie volaní metód ich telami, čo je rýchlejšie. Cena obfuskácií závisí aj od kontextu, v ktorom sú použité, napríklad tá istá obfuskácia vo vnútri cyklu spôsobí väčšie zdržanie ako tá istá mimo cyklu.

[2] definícia sa nachádza v [6] na strane 9.
[3] definícia sa nachádza v [6] na strane 9.

2.1.4 Nenápadnosť

Úspešnosť obfuskácie môže často závisieť aj od toho, či sa vôbec niekto pokúsi zistiť, ako vyzeral program pred sériou obfuskovacích transformácií. Ako jednoduchý príklad uveďme základnú transformáciu, akou je zmena názvov premenných. Pokiaľ sa premenným pridajú nezmyselné, nič nehovoriace názvy, je zrejmé, že ich význam niekto úmyselne skryl a útočník sa pokúsi vlastným rozumom a s použitím automatických nástrojov zistiť význam metód a premenných so zmenenými názvami. Ako ďalší príklad uvedieme, že skúsenejší spätný inžinier spozná, že kód je zahmlený, keď si všimne jeho neštandardný tok riadenia s použitím podmienok, zložito indexovaných polí, smerníkov a pod. V prvom prípade však metódy môžu mať zmyslupné názvy a argumenty, človek si bude myslieť, že chápe ich význam a bude sa snažiť pochopiť celý program, nie význam jednotlivých metód - pravdepodobne neúspešne. V druhom prípade bol príklad, ktorý väčšinou produkuje automatický obfuskátor a je teda veľmi otázne, nakoľko vôbec môže byť upravený do nenápadného ekvivalentu. Čiže *stealth* je metrika, ktorou sa meria vlastnosť transformovaného programu, ako sa podobá na program, ktorý nie je transformovaný a teda pravdepodobnosť, s akou bude tento program podrobený deobfuskácii. Rovnako ako cena je aj nenápadnosť metrika, ktorá je závislá na kontexte. Viac o nenápadnosti obfuskácie sa možno dočítať v [24].

2.2 Prehľad známych techník

Za účelom zneprehľadnenia kódu bolo vymyslených mnoho typov transformácií. Tieto sa dajú zaradiť do troch hlavných skupín, techniky v každej z nich sa vyznačujú rôznymi vlastnosťami a hodnotami metrík.

2.2.1 Zahmlievanie výzoru

Layout obfuskácie patria k tým základným, najstarším. Nie sú príliš sofistikované, boli to len prvé nápady, ako spraviť kód horšie zrozumiteľným. Podstatou je zmeniť zdrojový kód tak, aby bol na prvý pohľad nečitateľný, nepochopiteľný, aby sa nedali pochopiť ani jednoduché konštrukty (obr. 2.1). Avšak po podrobnejšom skúmaní sa dá veľa pôvodnej informácie vrátiť naspäť, veľakrát použitím automatických nástrojov, ale aj ručnou prácou spätného inžiniera. V nasledujúcich odsekoch si vymenujeme bežne používané layout obfuskácie.

```
#include <stdio.h>
main(t,_,a)char *a;{return!0<t?t<3?main(-79,-13,a+
main(-87,1-_,main(-86,0,a+1)+a)):1,t<_?main(t+1,_,
a):3,main(-94,-27+t,a)&&t==2?_<13?main(2,_+1,"%s
%d %d
\n"):9:16:t<0?t<-72?main(_,t,"@n'+,#'/*{}w+/w#cdnr/+,
{}r/*de+,/*{*+,/w{%+,/w#q#n+,/#{l,+,/n{n+,/+#n+,/#\
;#q#n+,/+k#;*+,/'r :'d*'3,}{w+K w'K:'+}e#';dq',l \
q#,+d'K#!/+k#;q#,r;eKK#}w'r}eKK{nl]'/#;#q#n,){}#}w')
{){nl]'/+#n;}d}rw' i;# \){nl]!/n{n#'; r{#w'r nc{nl]
'/#{l,+'K {rw' iK{;[{nl]'/w#q#n'wk nw' \iwk{KK{nl]!
/w%'l##w#' i; :{nl]'/*{q#'ld;r'}{nlwb!/*de}'c
\;;{nl'-{}rw]'/+,}##'*}#nc,',#nw]'/+kd'+e}+;#'rdq#w!
nr'/ ') }+}{rl#'{n' ')#
\}'+}##(!!/"):t<-50?_==*a?putchar(31[a]):main(-65,
_,a+1):main((*a=='/'+t,_,a+1)
:0<t?main(2,2,"%s"):*a=='/'||main(0,main(-61,*a,
"!ek;dc i@bK'(q)-[w]*%n+r3#l,{}:\nuwloca-0;m
.vpbks, fxntdCeghiry"),a+1);}
```

Obrázok 2.1: **Obfuskácia výzoru.** Odstránenie formátovania, komentárov a zmena názvov premenných.

Odstránenie formátovania

Slušne napísaný kód musí byť naformátovaný podľa zvyklostí, aby bol ľahko čitateľný, aby bolo jasné, kde sa ktorý blok začína aj končí, aby bola zrejmá viditeľnosť premenných, rozdelenie kódu do funkcií. Odstránením formátovania sa zo slušne napísaného kódu spraví zmes rôznych znakov, ako na obrázku 2.1. Avšak prechádzaním kódu postupne sa dá konštruovať slušne sformátovaný text, či už automaticky, alebo aj ručne. Odstránenie formátovania sa používa výlučne so zmenou názvov premenných, metód a tried, keď je cieľom spraviť nečitateľný kód, o ktorom je však zrejmé, že je obfuskovaný - čiže táto technika je najmenej nenápadná a preto sa zrejme použijú proti nej deobfuskačné techniky.

Zmena názvov premenných, metód, tried

Premenné zvyknú byť nazývané tak, aby ich názov vystihoval ich funkciu, práve kvôli pochopeniu funkčnosti celého programu. Zmena názvov premenných, metód a tried je tiež základná a jedna z prvých obfuskácií, používaná často s vyššie popísaným odstránením formátovania. Je to síce jednosmerná obfuskácia, takže automatický nástroj nezistí, aké

názvy mali premenné, metódy a triedy pôvodne, ale skúmaním, čo dané premenné obsahujú a metódy robia, sa dá približne zistiť, ako sa tieto zahmlené elementy pôvodne nazývali. Napríklad transformácia metódy *writeToFile(file, data)* na *__b8(x, y)* sa dá odstrániť tak, že sa zistí, čo daná funkcia robí - vždy sa dá postupným ponáraním do volania funkcií zistiť, že daná funkcia skutočne fyzicky zapisuje do súboru - potom sa funkcii priradí meno, ktoré bude podobne pôvodnému a bude vystihovať, že funkcia zapisuje do súboru. Ďalšou nevýhodou je rovnako ako pri odstraňovaní formátovania to, že pri pohľade na zmes symbolov miesto slušného zdrojového kódu sa dá okamžite vedieť, že bola použitá obfuskácia a príde rad na proces deobfuskácie, čo zvyšuje šancu na odhalenie funkcionality kódu. Alternatívou je použitie *stealthy layout* obfuskácie, keď sa názvy zamenia za iné, ktoré dávajú zmysel, avšak sú zavádzajúce. Pre náš príklad s funkciou *writeToFile(file, data)* by sme použili napríklad zmenu na *readFromFile(file, data)*.

Odstránenie komentárov

Každý správne napísaný zdrojový kód obsahuje komentáre, ktoré majú uľahčiť pochopenie zložitejších konštruktov. Keďže obfuskácia má opačný cieľ je samozrejmé, že v zahmlenom kóde komentáre buď nebudú, alebo budú zavádzať. V príklade s metódami *readFromFile()*, *writeToFile()* by sa napríklad zmenil opis funkcie z *funkcia číta zo súboru file údaje data* na *funkcia zapíše do súboru file údaje data*.

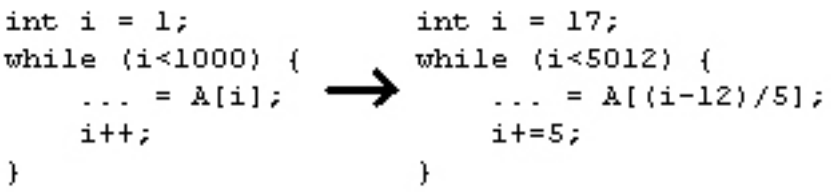
Zmena štandardných knižníc

Je to rozšírenie zmeny názvov premenných, metód a tried aj na knižnice, ktorých správanie je všeobecne známe a zdokumentované. Postupuje sa tak, že sa vytvoria kópie týchto knižníc a názvy tried a funkcií sa pomiešajú tak, aby sedeli počty a typ argumentov každej funkcie, počty a typ funkcií každej triedy.

Metriky zahmlievania výzoru

Patria medzi najstaršie a zároveň najmenej účinné zahmlievania zdrojového kódu. Celou ich podstatou je len zmena výzoru, avšak nijak sa nemení funkcionalita. Preto môžme definovať nasledovné vzťahy k metrikám na meranie kvality:

- Cena - techniky patriace pod *layout* obfuskáciu nepridávajú žiaden čas a priestor navyše, priestor dokonca môžu mierne znížiť.
- Odolnosť a účinnosť - odstránenie komentárov a zmena premenných sa nedajú automatickými nástrojmi vrátiť späť, avšak človek dokáže podrobným skúmaním aj



Obrázok 2.2: **Obfuskácia dát.** Zakódovanie indexov poľa.

napriek obfuskovaniu zistiť (s väčšou či menšou námahou) funkcionalitu. Naopak, odstránenie formátovania dokážu napraviť aj automatické nástroje.

- Nenápadnosť - okrem odstránenia formátovania, ktoré nikdy nie je nenápadné môžu a nemusia byť ostatné techniky nenápadné.

2.2.2 Zahmlievanie dátových štruktúr

Obfuskácia dát obsahuje také techniky, ktoré robia zložitejšími použité dátové štruktúry, typy a spôsoby prístupu k nim. Tejto triede zahmlievacích techník sa tiež nevenuje príliš veľa pozornosti, pretože sa o nich ťažšie ukazujú nejaké konkrétne výsledky a ani nepredstavujú také sofistikované a ťažko odstrániteľné obfuskácie, ako je v prípade zahmlievania toku riadenia, ako uvidíme v sekcii 2.2.3. V doterajších prácach na túto tému boli spomenuté spôsoby popísané v nasledujúcich statiach.

Spôsob uloženia dátových štruktúr v pamäti

Príkladom je zmena viditeľnosti premenných - globálne alebo lokálne, pre triedu, funkciu, úsek kódu. Zaujímavou možnosťou ako zahmliť funkcionalitu programu je použiť metódu, ktorá bude na prvý pohľad pracovať s lokálnymi premennými a vracať výsledok, ktorý bol na základe týchto lokálnych premenných vypočítaný, avšak jediný zmysel funkcie bude ovplyvňovať globálne premenné.

Zmena dimenzií polí

Napríklad vytvorenie jednorozmerného poľa z viacrozmerného, pristupovať k elementom poľa sa bude pomocou nejakej funkcie, ktorá mapuje toto viacrozmerné pole na jednorozmerné, prípadne opačne, zmena z menejrozmerného na viacrozmerné pole.

Zakódovanie dát

Podľa funkcie f, napríklad v poli A bude zmenené poradie indexov z $0, 1, 2, 3, ...$ na $f(0), f(1), f(2), f(3), ...$ a k prvkom tohto poľa sa bude pristupovať pomocou $A[f(x)]$, kde x je pôvodný index prvku. Ďalšou alternatívou môže byť nahradenie hodnôt funkčnými hodnotami takými, že existuje aj jednoznačná inverzná funkcia, napríklad nahradenie celého čísla i hodnotou $5 * i + 12$ (obr. 2.2). Môžme tiež použiť zakryptovanie reťazcov. Tie sa potom odkryptujú až počas behu programu nejakým algoritmom. V [6] bolo spomenuté vloženie interpreterov do kódu, ktoré sa uložia do pamäti až počas behu programu a interpretujú niektoré časti kódu, jedným z možných použití je práve pri tomto kryptovaní reťazcov. Takže reťazec je nahradený iným reťazcom, ktorý je vlastne interpreterom, ktorý produkuje tento pôvodný reťazec. Keďže spúšťanie vlastných interpeterov je veľmi drahá operácia, tak sa využíva len na menej často spúšťané časti programu, alebo na tie, ktoré vyžadujú mimoriadnu bezpečnosť.

Spájanie, rozdeľovanie dát

Zahŕňa zmenu zoskupovania dát dokopy, napríklad spájanie viacero polí do jedného, rozdeľovanie polí, uloženie dvoch premenných typu *integer* do jednej typu *long*, zakódovanie boolovskej premennej pomocou dvoch malých integerov, potom keď sa ich hodnoty rovnajú, tak nahradia *true* hodnotu tejto boolovskej premennej, keď sa hodnoty rovnať nebudú, budú interpretovať *false* hodnotu. Výhodou je, že tabuľka takéhoto kódovania boolovských premenných a operátorov sa dá konštruovať aj počas behu programu, čo znemožní útoky automatického deobfuskátora využívajúceho len statické metódy, viď stať 2.3.2 o deobfuskáciách.

Zmena definícií tried známych knižníc

Toto sa dá spraviť na viacerých úrovniach, buď sa len modifikuje obsah známych použitých knižníc, alebo sa časť vlastného kódu zabalí do knižnice, ktorá bude niesť názov, triedy a funkcie niektorej z bežne dostupných knižníc. V oboch prípadoch spätný inžinier predpokladá správanie týchto knižníc a nevenuje im pozornosť pri deobfuskovaní.

Objektovo-orientované zmeny

Zahŕňajú zmenu štruktúry tried, popísané boli v [25]. Tam sa spomínajú tieto metódy ako *class coalescing*, čo je spojenie viac tried do jednej, *class splitting* - rozdeľovanie jednej triedy do viacerých tak, že každá nová bude mať na starosti určitú samostatnú časť

funkcionality pôvodnej triedy. Tiež sa spomína *type hiding*, čo znamená skrývanie tried za *interface*-y, abstraktné triedy a zložitú dedičnosť. Výhodou tried, ktoré len volajú metódy ďalších tried alebo ich premenné je to, že pri vhodnom použití môže byť jedna metóda alebo premenná volaná z jedného miesta viacerými spôsobmi, takže nie je zrejmé, že sa jedná o jedno a to isté volanie. Všetky tieto tri metódy je najlepšie spojiť a vznikne veľmi neprehľadná zmes tried, ktoré sa navzájom na seba odkazujú, dedia.

Metriky data obfuskácií

- Neviditeľnosť. Dátové obfuskačné techniky sa vo veľkej miere spoliehajú na ich neviditeľnosť, pretože v opačnom prípade je veľká časť z nich ľahko odstrániteľná.
- Cena je rôzna, napríklad pri uložení dvoch premenných typu *integer* do jednej typu *long* nie je pridaná skoro žiadna časová zložitosť a žiadna pamäťová náročnosť, avšak pri hre s triedami je pamäťová náročnosť pomerne vysoká, mierne sa zvýši aj čas behu programu. Pri spomenutom zakódovaní reťazcov, ktoré sa potom odkodúvajú použitím interpreterov sa pridá veľa ako časovej tak aj priestorovej zložitosti, pretože interpreter treba najskôr uložiť do pamäti, potom spustiť, získať výsledok a znova z pamäte odstrániť. Metódy, ktoré menia uloženie dát v pamäti majú veľkú pamäťovú náročnosť, pretože v pamäti sa uchovávajú globálne premenné, ktoré majú len lokálny význam počas celého behu programu. Vážne zmeny v chovaní programu spôsobujú zložité dátové štruktúry - *Garbage Collector* má s nimi oveľa viac práce. Takisto omnoho rýchlejšie zaplnia *cache* a *heap* pamäť, takže sa môže stať, že na jednej konfigurácii pôvodný program bežal bez problémov a obfuskovaný sa nespustí.
- Odolnosť je veľmi vysoká, automatický nástroj nemá dosť inteligencie a úsudku, aby zistil, že niektorá dátová štruktúra je zostrojená nelogicky a už vôbec nemá potenciál na návrat k pôvodnej dátovej štruktúre.
- Účinnosť je veľmi diskutabilná, môže byť úplne vysoká ako aj nízka, závisí od kontextu a od použitej transformácie.

2.2.3 Zahmlievanie toku riadenia

Control-flow obfuskácie menia tok vykonávania programu, aby nebolo zrejmé, v akom poradí sa vykonávajú príkazy a bloky príkazov, aby sa nedalo určiť, aké sú momentálne hodnoty priradené v premenných, takže je pri analýze neistý aj celkový výsledok. V tejto sekcii aj v kapitole 3 budeme často používať výraz graf toku riadenia, alebo v skratke *CFG*.

Definícia 2.1 (Graf toku riadenia)**.** *CFG je graf $G = (V, E)$, kde V je množina všetkých príkazov v programe a orientovaná hrana $e = (V_1 \rightarrow V_2), e \in E$ existuje práve vtedy, keď existuje výpočet v programe, v ktorom príkaz vo vrchole V_2 priamo nasleduje po príkaze vo vrchole V_1. Blokový graf toku riadenia je modifikácia* CFG *taká, že vrchol netvorí len jeden príkaz ale vždy postupnosť príkazov (blok), ktoré sa pri každom ohodnotení vstupu vykonajú vždy v poradí bezprostredne za sebou.*

Ďalej budeme pod pojmom graf toku riadenia rozumieť blokový graf toku riadenia.

Poprehadzovanie poradia vykonávania príkazov

Príkazy, ktoré sú navzájom nezávislé možno v ľubovoľnom poradí poprehadzovať tak, aby tie, ktoré spolu súvisia, boli od seba vzdialené. Toto je sama o sebe veľmi jemná obfuskácia, pretože sa ľahko dá nájsť závislosť a poradie priraďovania hodnôt do premenných automatickým nástrojom alebo aj ručne, ak kód nie je príliš zdĺhavý. Preto môžme umelo vytvoriť závislosti medzi pôvodne nezávislými úsekmi kódu, ktoré zabránia odstráneniu prehodenia poradia vykonávania príkazov programu.

```
int method1(int x, int y, int z) {
    return x + y + z;
}

int method2(int x, int y, int z) {
    int t = y;
    y = x + y;
    ...
    y = t;
    return t + y + z;
}
...
a = method1(j, k, l);
taskResult = method2(a, m, n);
```

Tu sú metódy *method1* a *method2* nezávislé, premenná a, ktorú si odovzdávajú je zbytočná, neovplyvňuje totiž výsledok funkcie *method2*. Ďalším vylepšením techniky prehadzovania príkazov je, keď sa sekvenčný program rozdelí na viacero vlákien alebo procesov a tie sa vykonávajú samostatne. V nich budú potom na niektorých miestach synchronizačné úseky, keď bude treba odovzdať vypočítané výsledky.

Rozbalenie funkcií

Rozbalenie funkcií - *inlining* znamená, že sa volania funkcií v tele programu nahradia telom týchto funkcií. Takto vznikne len veľmi dlhý sekvenčný kód bez nejakej štruktúry. Toto by podobne ako v predchádzajúcom prípade nebolo veľmi účinné, preto je vhodné inlining spojiť s poprehadzovaním poradia vykonávania príkazov. Síce Collberg v [6] tvrdí, že toto je silne odolné, pretože sa jedná o úplne jednosmernú transformáciu, dal by sa však zostrojiť automat, ktorý hľadá v kóde rovnaké úseky a tie potom označí ako funkciu a požiada o pridanie vhodného mena podľa toho, čo vykonáva.

Zbalenie funkcií

Outlining je opačný proces k *inliningu*, teda umelo sa vytvoria funkcie na miestach, kde predtým neboli. Treba pritom dodržať kritérium, že každá takto vytvorená funkcia sa spustí nie len jediný krát. Ešte lepšie ako vybrať vždy rovnaké úseky kódu by bolo zrejme to, že sa vytvorí jedna funkcia a ostatné podľa nej pridávaním zbytočného kódu do tejto funkcie, ktorý nebude výsledok nijak ovplyvňovať, bude len vypĺňať telo funkcie. Ďalšou dôležitou vecou je si uvedomiť, že keď vytvoríme metódu *Class.doSomething()* tak jej vykonávanie bude závisieť aj od typu *Class*, keďže metóda doSomething môže byť a veľmi rada bude preťažená.

Dynamický výpočet hraníc cyklov

Pri konečných iteráciách vypočítame hranicu až za behu programu, čiže táto nebude známa statickým analýzam programu. Toto môže mimoriadne zvýšiť veľkosť programu, aj dĺžku jeho vykonávania, výpočet tejto hranice totiž môže byť sám o sebe zahmlený.

Úpravy Cyklov

Veľa úprav cyklov bolo vynájdených za účelom zefektívnenia ich výpočtov, majú však aj vlastnosti zahmlievania. Jednoduchou úpravou je otočenie cyklov, niekedy môže byť užitočné, inokedy nemusí. Rozbalenie cyklov - niektoré iterácie sa nepočítajú v cykle ale osobitne, pri zmene hraníc, ide väčšinou o krajné iterácie, ale tak isto sa napríklad cyklus môže rozdeliť na párne a nepárne cykly. Príklad:

```
for(i=2; i<=(n-1); i++)
  a[i] += a[i-1]*a[i+1];
```

sa upraví na

```
for(i=2; i<(n-2); i+=2) {
  a[i] += a[i-1]*a[i+1];
  a[i+1] += a[i]*a[i+2];
}
if(((n-2)%2) == 1)
  a[n-1] += a[n-2]*a[n];
```

Blokovanie cyklov predstavuje takú úpravu cyklov, že vnútorné cykly sú menšie a potrebujú menšiu pamät:

```
for(i=1; i<=n; i++)
  for(j=1; j<=n; j++)
    a[i,j] = b[i,j];
```

sa transformuje na

```
for(I=1; I<=n; I+=64)
  for(J=1; J<=n; J+=64)
    for(i=I; i<=min(I+63,n); i++)
      for(j=J; j<=min(J+63,n); j++)
        a[i,j] = b[i,j];
```

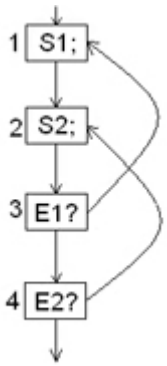
Loop fission - trhanie cyklov znamená, že ak sa v jednej iterácii cyklu spraví viac príkazov, tak tento cyklus roztrhneme na viacero tak, že v každom cykle sa bude vykonávať práve jeden príkaz, viď nasledujúci príklad:

```
for(i=1; i<n; i++) {
  a[i] += c;
  x[i+1] = d+x[i+1]*a[i];
}
```

Príkazy *a[i] += c;* a *x[i+1] = d+x[i+1]*a[i];* sa vykonajú v rôznych cykloch:

```
for(i=1; i<n; i++)
  a[i] += c;

for(i=1; i<n; i++)
  x[i+1] = d+x[i+1]*a[i]-i*c;
```



Obrázok 2.3: **Príklad neredukovateľného grafu.**

Každá z týchto techník je separátne nie príliš odolná, napríklad automatický deobfuskátor dokáže znova do cyklu zahrnúť vyňaté krajné iterácie. Ak však tieto transformácie skombinujeme, odolnosť sa dramaticky zvýši. Napríklad ak postupne aplikujeme rozbalenie, roztrhnutie, blokovanie.

Vytvorenie neredukovateľného diagramu vykonávania programu

V štruktúrovaných jazykoch ako je napríklad Java sa nedá napísať priamo zdrojový kód, ktorý by predstavoval neredukovateľný graf, čo znamená, že neexistujú dva cykly také, sa ani jeden nedá vložiť do druhého z nich. Príklad neredukovateľného diagramu behu programu je na obr. 2.3, cykly 1,2,3 a 2,3,4 sa môžu vykonávať v rôznom poradí a ani pre jeden z nich neplatí, že je vnútri toho druhého. Toto sa však dá spraviť v *bytecode*, ktorý pozná skoky a návestia. Preto silné zahmlenie je také, keď sa neobfuskuje priamo zdrojový kód ale tento *bytecode*, pretože k takémuto grafu neexistuje žiadny ekvivalent napísaný v jazyku napr. *Java*. Niektoré deobfuskátory sa toto snažia riešiť tak, že keď objavia v bytecode neredukovateľný graf, tak odstránia predikát, ktorý ich spôsobuje a nahradia neredukovateľný graf redukovateľným. Týmto predikátom hovoríme *opaque* - klamlivé a budeme sa im venovať v nasledujúcom odseku. Všeobecne platí, že každý algoritmus, ktorý mení neredukovateľné diagramy na redukovateľné pridá exponenciálnu zložitosť analýzy (viď. [4]).

Zavádzajúce podmienky a pridávanie zbytočného kódu

Definícia 2.2 (Nedosiahnuteľný kód)**.** *Nedosiahnuteľný (mŕtvy) kód je taký blok príkazov, že neexistuje platná cesta z počiatočného príkazu programu do tohto bloku, čiže mŕtvy kód sa nikdy nevykoná.*

Definícia 2.3 (Zbytočný kód)**.** *Zbytočný kód je blok príkazov zdrojového programu, ktorý je síce dosiahnuteľný, ale nemá žiadny vplyv na pozorovateľný výsledok programu.*

Najdôležitejším konštruktom nielen pri obfuskáciách toku riadenia, ale pri obfuskácii vôbec, sú takzvané zavádzajúce podmienky - *opaque predicates*. Ich cieľom je skryť skutočný výpočet programu za množstvo zbytočného kódu, ktorý sa nikdy nespustí, prípadne neovplyvňuje výsledok.

Definícia 2.4 (Zavádzajúci predikát)**.** *Zavádzajúci predikát je podmienka, ktorej splnenie je známe deobfuskátoru, ale nie je známe deobfuskátoru. Jej hodnota môže byť trojaká - buď vždy pravdivá, vždy nepravdivá, alebo rôzna podľa vstupu, závislosť na vstupe však deobfuskátor nepozná.*

Pridaním tohto konštruktu sa vytvorí falošná vetva a to tak, že obfuskátor vždy vie, aká je hodnota tohto predikátu, lenže pre automatický deobfuskátor to nemusí byť priamočiare zistiť, že niektorý predikát je vždy pravdivý alebo vždy klamlivý. Príkladom vždy pravdivého predikátu je z literatúry známy, jednoduchý predikát

```
(i*i*(i+1)*(i+1)) % 4 == 0
```

alebo

```
3 | (x*x*x - x) == true
```

Ich výsledky sú vždy pravdivé, čo nie je na prvý pohľad zrejmé.

Čím viac zavádzajúcich podmienok pridáme, tým viac bude rásť komplexnosť programu. Kvalita týchto podmienok určuje výslednú kvalitu celej obfuskácie. Aj pre automatické nástroje je prelomenie zavádzajúcich podmienok hlavnou úlohou. Generovanie zavádzajúcich podmienok musí byť automatická operácia, pretože ich je treba veľký počet na kvalitnú obfuskáciu. Rádovo sa používajú tisíce takýchto podmienok v jednoduchých programoch.

V predchádzajúcom odseku sme spomenuli, že pri snahe odstrániť neredukovateľný graf niektoré automatické nástroje odstraňujú tieto zavádzajúce podmienky. Lenže tie môžu byť navrhnuté buď tak, že ich hodnota bude niekedy *true* a niekedy *false* a teda ich odstránením sa znefunkční program. Ďalšou možnosťou, ako znefunkčniť program je vytvoriť zavádzajúce podmienky tak, že sa budú dať odstrániť len v skupinách, nikdy nie po jednej. Aj týmto zložitejším konštrukciám zavádzajúcich podmienok sa budeme venovať pri návrhu automatického generátora predikátov. Jedno z dôležitých pravidiel pri konštrukcii zavádzajúcich podmienok hovorí, že premenné, ktoré sa použijú na výpočet predikátu nebudú lokálne, ale globálne, ovplyvniteľné z viacerých miest.

Zatiaľ najlepšie výsledky pri obfuskácii sa dosiahli práve degenerovaním grafu toku riadenia programu pomocou zavádzajúcich podmienok, naša práca taktiež navrhuje ďalšie vylepšenia tohto prístupu.

Metriky obfuskácií toku riadenia

Zahmlievanie toku riadenia zahŕňa najsofistikovanejšie obfuskáčné transformácie, pri dômyselnom použití majú vysokú odolnosť aj účinnosť, závisí hlavne na tom, aký čas naviac si môžme dovoliť použiť, pretože výpočet zbytočných podmienok môže beh programu príliš spomaliť. Tu je treba nájsť vhodný pomer medzi úrovňou bezpečnosti a výkonom.

- Cena - pri prehadzovaní príkazov a blokov sa nepridá žiaden čas a priestor navyše. Pri zavedení zavádzajúcich podmienok je potrebný čas na ich výpočet a pamäť na udržiavanie premenných, podľa ktorých sa potom podmienky konštruujú. Práca s cyklami môže tiež priniesť spomalenie programu, preto sa robí len rozumne do určitej úrovne.

- Účinnosť - ako sme si povedali, ľudský útočník, ktorý nemá vedomosť o zbytočnom alebo nedosiahnuteľnom kóde je postavený pred kód, z ktorého veľká väčšina nič užitočné nerobí. Prípadné prehadzovanie príkazov a úpravu cyklov však dokáže človek logickými úvahami vrátiť späť.

- Odolnosť - je témou mnohých prác, obfuskačných a deobfuskačných a takisto sa rieši aj v našej práci, preto tu zatiaľ záver neprinesieme, nakoľko je ťažko dokázateľný.

- Nenápadnosť - v prípade zavádzajúcich podmienok nehrá veľkú rolu, pretože ich počet je veľmi vysoký, takže je treba automatický deobfuskátor na ich odstránenie.

2.3 Deobfuskácia

Spomenieme si metódy deobfuskácie, aby sme vedeli, aké nástroje má v rukách útočník. Potom pri návrhu našich zahmlievacích transformácií budeme dbať na to, aby tieto nástroje boli neúčinné alebo aspoň ťažšie realizovateľné. Metódy automatickej deobfuskácie sa delia na statické a dynamické podľa toho, či sa analyzuje zdrojový kód alebo či sa sleduje správanie programu.

2.3.1 Statické metódy

Pri statických deobfuskáciách sa program nespúšťa, len sa analyzuje zdrojový kód. Uvažujú sa pritom všetky možné vetvenia, ktorými sa mohlo prejsť. Jasnou nevýhodou je veľká pamäťová náročnosť, keďže zahmlený program môže obsahovať mnohonásobne viac vetiev, ktorých všetky premenné a poradie vykonávania treba poznať. Uvažujú sa veľakrát aj vetvy, ktorými sa nikdy neprejde, pretože statická analýza nepracuje podľa vstupu.

Globálna analýza toku dát

Základný postup pri statickej analýze. Sleduje priradenia do premenných od začiatku vykonávania programu. Potom v bodoch vetvenia dosadí svoje informácie do podmienok a podľa výsledkov si vyberie vetvu.

Použitie teorém

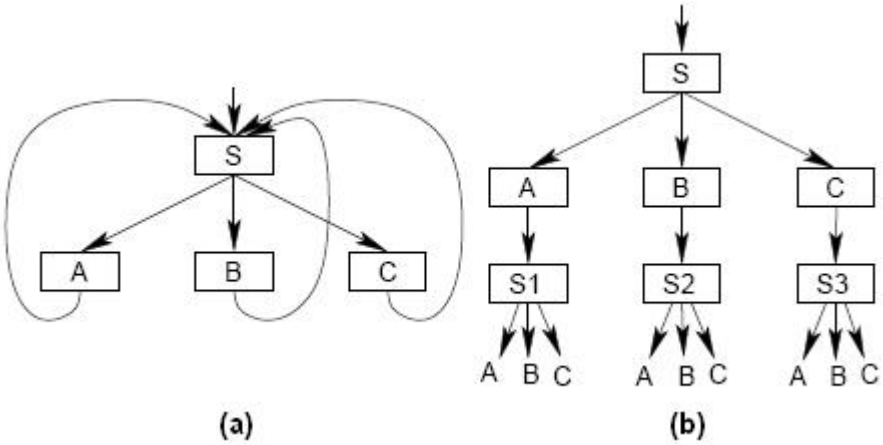
Teorémy sa používajú na odstránenie tých predikátov, ktoré používajú matematické vety na konštrukciu podmienok. Príkladom je aj náš obľúbený $3|(x * x * x - x)$. Keby analyzátor nepoužil vetu, ktorá vraví, že tento výrok vždy platí, musel by postupne skúšať všetky hodnoty x, ktorých je toľko, koľko prirodzených čísiel v programovacom jazyku.

Program slicing

Program sa rozdelí na malé úseky - *slices*, z ktorých každý reprezentuje nejakú podúlohu, ktorá ovplyvní celkový výsledok. Tieto malé úseky sú ľahšie analyzovateľné. Ako vybrať len relevantné príkazy pre jednu podúlohu je znázornené na obrázku 1 v [27]. Viac o *slicing*-u sa možno dočítať aj v [32], [1], [21].

Čiastočné vypočítanie

Čiastočné vypočítanie je často používaná metóda optimalizácie programov, viac sa možno dočítať aj v [11]. Aj tu sa rozdelí obfuskovaný program na dve časti: statickú, ktorá môže byť predvypočítaná a dynamickú, ktorá je spustená až počas behu programu. Dynamická časť korešponduje s originálnym programom, statická predstavuje náš novopridaný zbytočný kód. Ak sa podarí túto statickú časť identifikovať, je možné ju odstrániť.



Obrázok 2.4: **Klonovanie.** Zrušenie cyklov v grafe toku riadenia. **a.)** Graf pred klonovaním, **b.)** graf po klonovaní.

Klonovanie

Cieľom je zostrojiť upravený graf toku riadenia tak, aby tento neobsahoval cykly, ktoré sú zdrojom alebo výsledkom mnohých transformácií, ktoré zahmlia program. Namiesto vrátenia sa do určitého bodu programu sa tento bod skopíruje a miesto vrátenia sa späť sa vstúpi do tohto nového bodu (obr. 2.4). Výhody tohto postupu sú spomenuté v sekcii 2.4.3, ktorá opisuje prácu, ktorá použije túto techniku v praxi. My poznamenajme, že na druhej strane sa produkuje zložitejší graf, čo je zrejmá nevýhoda. Ako bolo poznamenané, táto metóda nie je dostatočná ako samostatná, ale v kombinácii s inou. Napríklad uľahčuje globálnu analýzu dát, ktorá následne môže veľa touto technikou pridaných hrán opäť odstrániť.

Dosiahnuteľnosť

Vychádza z globálnej analýzy toku dát, ktorá sleduje hodnoty všetkých premenných vo všetkých vetvách. Pri dosiahnuteľnosti chceme zistiť doprednou analýzou, ktoré premenné majú v ktorých vetvách určené hodnoty a spätnou analýzou, ktoré premenné síce majú hodnotu, tá sa však už v ďalšom nepoužije a preto ich môžme od bodu posledného použitia z programu vymazať.

2.3.2 Dynamické metódy

Dynamická analýza skúma správanie sa programu pri spustení, sleduje, ktorými vetvami riadenie prešlo. Nevýhodou je to, že kým nepustíme program so všetkými vstupmi tak nevieme presne určiť, ktorými vetvami sa mohlo prejsť. Vstupov je väčšinou toľko, že sa všetky spustenia nedajú uskutočniť, preto sa vyberá len nejaká podmnožina tak, aby zo všetkých tried ekvivalencií vstupov bola vybratá každá aspoň raz. Určiť triedy ekvivalencie je ďalší problém.

Sledovanie správania sa predikátov

Patrí pod tzv. *black-box testing*. Vytvorí sa zoznam predikátov a doň sa dopisujú hodnoty, ktoré vracajú pri konkrétnych vstupoch. Keď niektorý predikát vracia počas veľkého počtu spustení ten istý výsledok, označí sa za zavádzajúci a z programu sa odstráni. Zavádzajúce predikáty môžu byť konštruované aj tak, že niekedy vracajú *true* a inokedy *false*, závisí od vstupu. Vtedy sa deobfuskátor snaží zistiť závislosť výstupu od vstupu a keď sa táto nájde, tak je znova predikát prelomený. Výsledkom je odstránenie umelo pridaných zbytočných vetiev a tým pádom rekonštrukcia pôvodného grafu toku riadenia.

Použitie profiling tools

Je prvou tzv. *white-box testing* metódou. *Profiling tools* sledujú volania metód programu a na základe toho robia štatistiky, z ktorých sa dá zistiť, ktoré časti programu sa spúšťajú najčastejšie, ktoré menej a ktoré vôbec. Keď sa teda zistí, že niektoré časti kódu nie sú vôbec volané, deobfuskátor sa pozrie, za ktorých predikátom sa nachádza daný úsek kódu a predikát sa označí ako zavádzajúci. Toto vie veľmi znížiť priestorovú zložitosť obfuskovaného programu, pretože cieľ väčšiny control-flow obfuskácií je práve priniesť veľké množstvo zbytočného kódu skrytého za falošné podmienky.

Tracovanie programu

Tracovanie je druhou *white-box testing* metódou a znamená, že počas vykonávania programu sledujeme, aké hodnoty nadobúdajú ktoré premenné. Potom vieme identifikovať napríklad zbytočné priradenia.

2.4 Prehľad prác

V nasledujúcich odsekoch si najskôr predstavíme práce, ktoré priniesli do obfuskácie nové postupy, potom prejdeme k prácam, na ktoré najviac nadväzuje táto práca.

2.4.1 Súhrn

Obfuskácia programov je pomerne staré odvetvie ochrany softvéru, dlho však nebola formálne definovaná a používali sa len pomerne jednoduché a málo účinné metódy, zamerané prevažne na zahmlenie výzoru programu, ďalej na jednoduché transformácie kódu, ako sú preusporiadanie poradia príkazov, rozbalenie metód, zmena inštrukcií za behu.

Obfuskačné práce

V roku 1997 definovali Collberg, Thomborson a Low [6] pojem obfuskácie, popísali metriky, ako ich máme uvedené v časti 2.1, spravili súhrn dovtedy známych metód a v 1998 implementovali vlastný obfuskátor [7], [15] meniaci graf toku riadenia zavedením zavádzajúcich podmienok a vkladaním zbytočného kódu. Zavádzajúce podmienky vytvárali pomocou grafových operácií. Graf sa priebežne medziprocedurálne menil, otázky sa týkali vlastností grafu, ako napríklad jeho konektivity, umiestnenia uzlov v rovnakom komponente a pod. Takisto zaviedli aliasing dát. Ten ako najdôležitejší nástroj pri tvorbe zavádzajúcich podmienok použil vo svojich prácach [30], [31] aj Wang a kolektív, ktorí použili tzv. vyrovnávanie toku riadenia, globálne premenné a smerníky na dátové štruktúry aj funkcie, nepriame volania funkcií. Viacero týchto transformácií sme prevzali aj my, preto túto prácu dôkladnejšie popíšeme v samostatnej časti 2.4.2. Z inej strany vzal obfuskáciu Sosonkin [25], ktorý sa venoval obfuskácii zdrojového kódu v objektovo-orientovaných programovacích jazykoch, kde menil štruktúru tried spájaním, rozdeľovaním, vnáraním a pod. Generátor zavádzajúcich podmienok na základe len lokálnych premenných pomocou teórie programovania a matematických teorém na nich aplikovaných zostrojili autori v [22]. Algoritmus generovania distribuovaných zavádzajúcich podmienok v [19] používa známy 0/1 knapsack problém, o ktorom bolo dokázané, že je NP-úplný. Proces, v ktorom sa generuje podmienka využíva hodnoty od pomocných procesov. V prípade, že súčet hodnôt sa rovná vopred určenej sume, výsledok predikátu je pravdivý.

Inou kategóriou obfuskácií sú tie, ktoré priamo nezahmlievajú zdrojový kód, ale skompilovaný, buď vo forme medzikódu, alebo priamo do strojových inštrukcií. V [20] je kombinácia takejto obfuskácie spolu s kryptovaním inštrukcií. Jeden z najnovších obfuskátorov *Java bytecode* - JBCO - je detailne popísaný v [3]. Implementuje viacero účinných

transformácií od obfuskácie výzoru, cez obfuskáciu dát, obfuskáciu toku riadenia až po zmeny v štruktúre tried. Zaujímavé metódy prináša práca [14], ktorá zahmlieva medzikód takým spôsobom, aby dekompilátor nebol schopný ho vôbec transformovať na zdrojový kód, napríklad použitím neredukovateľných grafov tokov riadenia, ktoré sa nedajú zapísať v niektorých programovacích jazykoch. Možným riešením je zmena takýchto grafov na redukovateľné v medzikóde a následne preklad do zdrojového kódu. Tu ale platí, že redukovateľný graf môže byť exponenciálne väčší [4].

Deobfuskačné práce

Popri prácach, ktorých cieľom bolo priniesť nové a bezpečnejšie techniky obfuskácie, vznikali aj práce, ktorých cieľom bol pravý opak, prelomiť a odstrániť obfuskačné transformácie z kódu. Identifikácii a odstraňovaniu zavádzajúcich predikátov sa venuje práca [23], v ktorej sa matematickým modelom odstraňujú len lokálne predikáty, ktoré vznikli na základe matematických viet, čiže ako neskôr uvidíme, táto metóda nijak neodstraňuje predikáty nami pridané. Takýto typ predikátov odstraňuje [29]. Popisom tejto práce a metód bude venovaná celá sekcia 2.4.3. Tradičné statické metódy analýzy programu si nevedia poradiť so zavádzajúcimi podmienkami, pretože musia sledovať všetky novozavedené vetvy a dáta v nich, ich množstvo môže byť ľubovoľné. Preto sa deobfuskácia stále viac spolieha buď čisto na dynamické metódy popísané napríklad v [2], [26], alebo na kombináciu statických a dynamických metód [29], [17]. Práce, ktoré poskytujú konkrétne postupy, algoritmy, prípadne naprogramované deobfuskátory sú jednak [29] a tiež [5], ktorá poskytuje ochranu proti zbytočnému kódu a proti dynamicky menenému kódu. Má však významné obmedzenia, ktoré robia túto prácu neúčinnú voči nášmu postupu. Pri zbytočnom kóde sa nevie postarať o zavádzajúce podmienky a pri dynamicky menenom kóde zvládne len jednu úroveň, čiže neodstráni dynamický kód generovaný dynamickým kódom.

Práce využívajúce samomeniaci sa kód

Samomeniaci sa kód programov za účelom obfuskácie bol spomenutý najskôr v práci [13], [12]. Postup bol jednoduchý, boli náhodne vybrané inštrukcie, ktoré boli nahradené nesprávnymi. Vždy pred spustením nahradenej inštrukcie sa v kóde pred (nie bezprostredne) ňou pustila inštrukcia, ktorá vracala pôvodnú inštrukciu a v kóde po tejto správnej inštrukcii ju ďalšia znova nahradila opäť za nesprávnu. V jednom momente sa nedal zobrať kompletný obraz programu, avšak útočník mohol sledovať, ktoré inštrukcie sa naozaj vykonávajú, ďalej odstrániť inštrukcie, ktoré menia iné inštrukcie. Zložitejší postup bol

zvolený v práci [16], kde autori použili jedno miesto v pamäti pre viac funkcií naraz a tieto miesta menili v rôznych funkciách, takže zistenie, aké sú inštrukcie v pamäti si už vyžadovalo globálnu analýzu. Pomôcky na vizualizáciu grafov tokov riadenia pri dynamicky menenom kóde poskytuje práca [8], ďalšia práca, ktorá vizualizuje kód je [18] s implementovaným obfuskátorom a niekoľko pomôckami na deobfuskáciu. V [10] bol pôvodný program rozdelený na statický proces a dynamický. Dynamický proces bola implementácia tabuľky skokov, ktorá sa dynamicky modifikovala. Statický proces si pýtal od dynamického návestia skokov.

2.4.2 Implementácia obfuskácie pomocou zavádzajúcich podmienok

V roku 2000 priniesol Xenghi Wang vo svojej dizertačnej práci viacero výborných nápadov na obfuskáciu a svoju stratégiu aj naimplementoval. Pozornosť však sústredil len na odolnosť voči statickej analýze, dynamickú spomenul len stručne v kapitole 9 so slovami, že tá môže byť témou ďalšieho výskumu. Jeho postup degenerovania grafu toku riadenia programu bol nasledovný:

- Zmenil vysokoúrovňové konštrukty na sekvenciu *if-then-goto* príkazov.
- Potom zmenil *goto* príkazy tak, že ich ciele boli počítané dynamicky. Konkrétna implementácia nahradila *goto* príkazy vstupom do *switch* príkazu. V každom bloku sa táto *switch* premenná počítala, aby sa určil nasledujúci blok. Týmto sa vytvoril nový graf toku riadenia, ktorý obsahoval všetky bloky s rovnakým predchodcom a rovnakým nasledovníkom. Išlo o takzvaný vyrovnaný diagram (flattened). Toto celé zmenilo potrebnú statickú analýzu na analýzu závislú na dátach. Veľa klasických problémov analýzy dát je dokázaných, že sú NP-úplné.
- Priradenia konštánt nahradil počítaním pomocou globálneho poľa a zložitejších výrazov. Takže pred každým takýmto priradením musí statický analyzátor poznať aktuálne hodnoty tohto globálneho poľa.
- V každej funkcii boli vytvorené smerníky na niektoré premenné ako aj na prvky globálneho poľa. Naviac boli do programu popridávané príkazy, ktoré s týmito smerníkmi manipulujú.
- Referencie premenných boli nahradené smerníkmi.
- Boli pridané zbytočné príkazy ako aj nikdy nespustiteľné, ktoré boli skryté za zavádzajúcimi podmienkami, ktoré sa opierali práve o zavedený aliasing.

V ďalšom sa venuje podrobnej analýze, ktorá dokazuje účinnosť uvedenej obfuskácie proti statickej analýze. Ako budeme vidieť neskôr, viacero myšlienok bolo do nášho algoritmu prebratých, ich menšia odolnosť voči dynamickej analýze však prinútila vznik vylepšení.

2.4.3 Implementácia deobfuskácie

Cieľom tejto práce z roku 2005 bolo skonštruovať automatický deobfuskátor a tým ukázať, že viacero obfuskačných techník, ktoré boli považované za dostatočné, takými v skutočnosti nie sú. Autori sa rozhodli prelomiť prácu od Wanga (2.4.2), ktorá bola považovaná za dobrú a bezpečnú obfuskáciu. Využitá bola kombinácia statických a dynamických metód, pretože statická analýza uvažujúca všetky vetvy je príliš zložitá a naopak čisto dynamická analýza nemusí brať do úvahy všetky možné cesty v programe a produkuje preto nepresné výsledky. O statických aj dynamických metódach všeobecne sa možno dočítať v [17].

Použité statické metódy:

1. **Klonovanie.** Pri Wangovom vyrovnávaní a zavedení zbytočných blokov sa stane, že v určitom bode programu sa zíde skutočná vetva s falošnou, čo je základný problém pre analýzu. Aby sme tomuto predošli, použijeme klonovanie, viď 2.3.1. Klonovanie však musí byť použité rozumne, čiže nie vždy, ale len keď je podozrenie na falošný blok, inak by výsledný graf bol obrovský a nečitateľný.

2. **Statická analýza dosiahnuteľnosti cesty.** V skratke možno povedať, že sa jedná o sledovanie hodnôt všetkých premenných v blokoch. Tým získame určité obmedzenia na premenné a keď tieto nie sú na vstupe do ďalšieho bloku splnené, blok sa určite nevykoná.

Statické metódy sú samé o sebe konzervatívne, čo znamená, že množina získaných hrán je len supermnožina všetkých hrán. Naopak dynamické metódy, ako je napríklad tracovanie programu, alebo profilovanie hrán nemôžu nikdy zobrať na vedomie všetky možné vstupy, preto zisťujú len podmnožinu všetkých možných ciest. Duálnosť týchto prístupov povzbudzuje na spoločné použitie oboch techník. Toto môže byť spravené dvomi spôsobmi. Môžme začať dynamicky, necháme len prejdené hrany a potom spätne pridáme statickou analýzou tie, ktoré boli identifikované ako spustiteľné. Druhá možnosť je opačná. V oboch prípadoch výsledkom môže byť viac alebo menej vetiev, ako v originálnom programe. Takže nemusíme dostať kompletný alebo správny výsledok. Napriek tomu pre spätné inžinierstvo môže byť toto aj tak veľmi použiteľné. V tejto práci použili prvú metódu - najskôr hrany dynamicky osekali, potom staticky pridali.

Výsledkom experimentov na desiatich programoch, ktoré obfuskovali pomocou DIABLO a PLTO bolo, že v priemernom prípade odstránili automatickým deobfuskátorom 78,6% falošných hrán a odstránili 0,06% hrán, ktoré odstrániť nemali. V najlepšom prípade to bolo 98,4% a 1,69%. Vzhľadom k tomu, že pridali v priemere 2154253 hrán, je jasné, že veľa hrán ostalo neodstránených. Toto pozorovanie sa bude snažiť využiť náš postup.

Kapitola 3

Návrh algoritmu

Na základe prác [30] a [29] popísaných v kapitolách 2.4.2 a 2.4.3 vznikla myšlienka vylepšenia obfuskácie. V [30] boli vytvorené koncepty na zahmlievanie - snaha o zvýšenie viditeľnosti premenných z lokálnych na globálne, používanie smerníkov na premenné, aliasing týchto smerníkov, vyrovnávanie grafu toku riadenia pomocou takzvanej *dispatcher* premennej. Náš algoritmus bude implementovať tieto koncepty a bude vylepšený o niekoľko ďalších myšlienok, ktoré budú plátať jednu základnú dieru. Autori [30] totiž v krátkosti odbili možnosť použitia dynamickej analýzy pre jej predpokladanú prílišnú zložitosť, komplexnosť, ktorá sa nedá v skutočnosti implementovať. Obfuskácia týmito metódami získaná bola napadnutá v roku 2005 v článku [29]. Bola umne použitá kombinácia statických a dynamických metód. Autori vytvorili ako automatický obfuskátor tak aj deobfuskátor, ktorý odstránil pomerne veľké množstvo pridaných zbytočných vetiev, čím čiastočne rekonštruoval zdrojový kód na pôvodný. Poznamenajme, že pri obfuskácii sa pridali rádovo až milióny zbytočných vetiev, takže použitie automatického nástroja je nevyhnutné.

Našou prvou myšlienkou bolo využiť to, že automatický deobfuskátor z [29] nedokázal nikdy odstrániť všetky pridané vetvy. Využitie spočíva v tom, že v každej vetve, ktorá nasleduje po nejakom predikáte (a teda aj po tých neodstránených) sa priradí do novej premennej (my ju budeme nazývať propagujúca sa premenná) určitá hodnota, na ktorú sa bude pýtať nasledujúci falošný predikát.

Definícia 3.1. *Propagujúca premenná je novozavedená premenná, ktorá slúži na vytváranie závislostí medzi jednotlivými blokmi. Nazýva sa propagujúca preto, lebo sa odovzdáva (propaguje) z bloku do bloku. Táto premenná vzniká vo viacerých blokoch a v každom má inú definíciu. V bloku, kde sa použije potom nie je deobfuskátoru zrejmé, ktorá definícia platí. Hodnoty sa priraďujú náhodne.*

Keďže niektoré predikáty ostanú neodstránené, tak niektoré propagujúce sa premenné budú mať neznáme hodnoty a teda aj predikáty, ktoré ich použijú vo svojom výpočte ostanú tiež neprelomené. Takto sa neprelomiteľnosť bude šíriť ďalej a výsledkom bude, že po prvom neprelomenom predikáte ostanú neprelomené už všetky (alebo aspoň všetky tie, ktoré sú od jeho výpočtu závislé, lenže naša metóda počíta práve s vytváraním týchto závislostí).

Slabinou tohto prístupu môže byť predpoklad, že určite ostane nejaký neprelomený predikát. V práci [29] obfuskovali veľké programy, pri ktorých sa vždy stalo, že neprelomených predikátov ostalo dostatočne veľa. Môžme ale chciet zahmliť len pomerne malý program alebo úsek programu, kde môžu byť všetky predikáty odstránené. Potom sa nedajú použiť propagujúce premenné. Naše riešenie spočíva v čiastočnom použití metódy ochrany softvéru, ktorá je spomínaná v stati 1.2.2. Jedná sa o použitie spúšťania kódu na bezpečnom mieste, ktoré je čiernou skrinkou, takže vieme akurát zistiť, aké sú odpovede na vstupné dáta, ktoré tejto čiernej skrinke pošleme. Túto čiernu skrinku, alebo ináč povedané externý server, použijeme na získanie prvých niekoľko (ich počet bude ovplyvňovať výkon aplikácie na jednej strane a bezpečnosť na druhej, preto by mal byť konfigurovateľný, o konfigurácii algoritmu pohovoríme v sekcii 3.11) propagujúcich sa premenných a tým zaručíme, že také existovať budú a aj pomocou nich vytvoríme zavádzajúce predikáty.

Ďalším vylepšením bude dynamická zmena kódu predikátov, ktorá predstavuje ďalšie zamedzenie dynamickej analýzy predikátov, ktorá sa snaží zistiť, aké sú výsledky predikátov na určitých množinách vstupov, snaží sa zistiť tieto množiny. My však budeme telá týchto predikátov za behu meniť, takže každý z nich môže na tom istom vstupe dávať rôzne výsledky. Dynamická analýza, ktorá by chcela riešiť toto by musela pre každý predikát sledovať aj všetky jeho možné tvary (tvar predikátu - pre jedno meno predikátu všetky jeho inštancie vytvorené dynamickou zmenou tela predikátu) a k nim všetky možné množiny vstupov.

Na záver tejto kapitoly spomenieme možnosti konfigurácie algoritmu za účelom získania želaného pomeru medzi výkonom a bezpečnosťou (stať 3.11). Tiež sa budeme venovať analýze jednotlivých krokov algoritmu (v 3.12) a spomenieme možnosti ďalšieho vylepšenia algoritmu (3.10).

3.1 Prehľad činností algoritmu

V tejto stati si popíšeme detailne kroky, ktoré bude náš algoritmus zahmlievania postupne vykonávať. V skratke bude robiť nasledujúce:

1. Zoberie pôvodný zdrojový kód a spraví z neho jednoduchší, nižšieúrovňový, skladajúci sa len z niektorých príkazov, ktorými sa dajú nahradiť všetky konštrukty. Tento krok je dôležitý kvôli implementácii obfuskátora, ktorá po tomto kroku začne byť reálnejšia.

2. Vytvorí sa graf toku riadenia programu a z neho spojením príkazov do blokov sa vytvorí blokový graf toku riadenia. Z doterajšieho vieme, že cieľom našej obfuskácie je práve tento graf zdeformovať a mnohonásobne zväčšiť jeho komplexnosť pridaním zbytočného kódu.

3. Pre každý identifikovaný blok z predošlého kroku sa určia množiny premenných, ktoré do bloku vstupujú, ktoré z neho vystupujú, ktoré v ňom vznikajú a zanikajú.

4. Naviac sa pre každý blok určia funkcie vstupných premenných, ktoré tento blok vykonáva.

   Tieto kroky boli analytické, ktoré len zozbierali informácie z pôvodného programu. Po tejto prípravnej fáze už príde samotná deformácia pôvodného programu:

5. Algoritmus vykonáva deformácie grafu toku riadenia zmenou poradia vykonávania blokov. Popritom konštruuje funkcie na serveri, ktoré budú na základe vstupu posielať nazad zahmlenému programu dáta, ktoré zabezpečia rovnakú funkcionalitu zahmleného programu ako bola funkcionalita pôvodného. Funkcie zo servera budú na základe vstupu vracať dáta, ktoré budú rozhodovať o tom, ako bude graf vyzerať. Zároveň v tomto kroku sa budú získavať dáta, ktoré sa použijú na vytváranie falošných predikátov.

6. Keď máme dáta na vytváranie falošných predikátov, tak na niektorých miestach programu môžeme vložiť tieto predikáty a skrývať za ne množstvo nepotrebného kódu. Zároveň algoritmus bude vytvárať nové dáta, ktoré sa budú propagovať k ďalším predikátom.

7. Na záver pridá algoritmus ešte zmenu kódu jednotlivých predikátov na ďalšie zamedzenie dynamickej analýzy kódu.

Poďme si teraz podrobne jednotlivé kroky algoritmu rozobrať v nasledujúcich statiach.

3.2 Analýza zdrojového kódu

Túto časť zavádzame najmä kvôli implementačnej otázke, aj keď k tejto práci nepatrí samotná implementácia. Ide však o návod na implementáciu, preto do návodu patrí aj táto sekcia.

Chceme získať nízkoúrovňový kód, s ktorým sa bude ľahšie manipulovať a implementácia algoritmu bude jednoduchšia. Odstránime objektovo-orientovaný prístup, lokálne premenné, z viacerých typov cyklov spravíme len jediný - pomocou skoku po podmienke. Odstránime volanie funkcií rozbalením, čiže na konci získame len jediný graf toku riadenia, ktorý sa budeme v ďalšom pokúšať transformovať. Je nám jasné, že tu nejde o tzv. *stealthy* zahmlievanie, ale to nie je cieľom, pretože našim cieľom je spraviť účinné zahmlenie proti automatickej deobfuskácii, na spusenie ktorej je podmienkou, aby spätný inžinier vedel, že kód je zahmlený. Poďme si teraz podrobne popísať jednotlivé výstupy analyzátora zdrojového kódu v poradí, v akom sa budu vykonávať.

3.2.1 Zjednodušujúce transformácie kódu

Štandardná programová schéma obsahuje premenné, funkcie, predikáty, príkazy a výrazy. Konštanta je funkcia bez argumentov. Pozrime sa teraz na súhrn týchto elementov, ktoré budú postačovať na vyjadrenie každého jazyka. Nebudeme uvádzať, ako sa ktorý vyššieúrovňový konštrukt zmení na postupnosť nižšieúrovňových, len si vymenujeme, s čím bude náš algoritmus ďalej narábať.

1. **Premenné:** všetky sa zmenia na globálne, čiže ich deklarácia bude na začiatku programu. Budeme používať primitívne typy jazyka a polia. Pomocou polí a operáciami nad poľami budeme pracovať aj s reťazcami.

2. **Výrazy:** môžu byť rozdielne podľa programovacieho jazyka, ktorý sa používa, my sme si znova zobrali na mušku syntax javy, ktorá konštruuje výrazy z nasledovných operátorov a znakov: $+$, $-$, $*$, $/$, $\%$, unárne $-$, $<$, $>$, $=$, *and*, *or*, *not*, *null*, $<<$, $>>$, $\&$, $|$, $\wedge$, (,).

3. **Príkazy:**

   - priraďovací: $x = a$, kde sa do premennej x priradí hodnota a, kde a je premenná, konštanta, alebo výraz, ktorý sa najskôr vypočíta, až potom nastane priradenie.
   - príkaz skoku: *goto* i, kde i je návestie skoku.

- podmienkový príkaz: *if p(x) then st*, kde *st* je priradenie alebo príkaz skoku a x je výraz.
- príkaz s návestím: $i : st$, kde *st* je priradenie, skok alebo podmienka.

4. **Funkcie:** tie chceme rozbaliť, takže ich odstránime a ich telo nahradíme príkazmi tela, volanie funkcie spravíme skokom na počiatočný príkaz funkcie, návrat z funkcie spravíme skokom z posledného príkazu funkcie na príkaz, ktorý bol nasledujúci za volaním funkcie.

5. **Predikáty:** tie ostanú bez zmeny.

Okrem odstránenia funkcií musíme odstrániť aj triedy. Toto je priamočiare, všetky premenné sa stanú tiež globálnymi pre celý program, funkcie sa inlinujú. Ani tu nepovažujeme za podstatné podrobne opisovať túto tranformáciu, museli by sme brať do úvahy všetky netriviálne konštrukty, ktorých je v rôznych jazykoch viacero, ako napríklad synchronizované metódy v Jave. Pre ne musíme zaviesť semafóry, ktoré budú riadiť prístup threadov k týmto metódam. My namiesto toho len uvedieme tento jednoduchý príklad, ostatné transformácie sú nad rámec tejto práce, do úvahy by prišli až pri konkrétnej implementácii algoritmu.

Príklad 3.1.

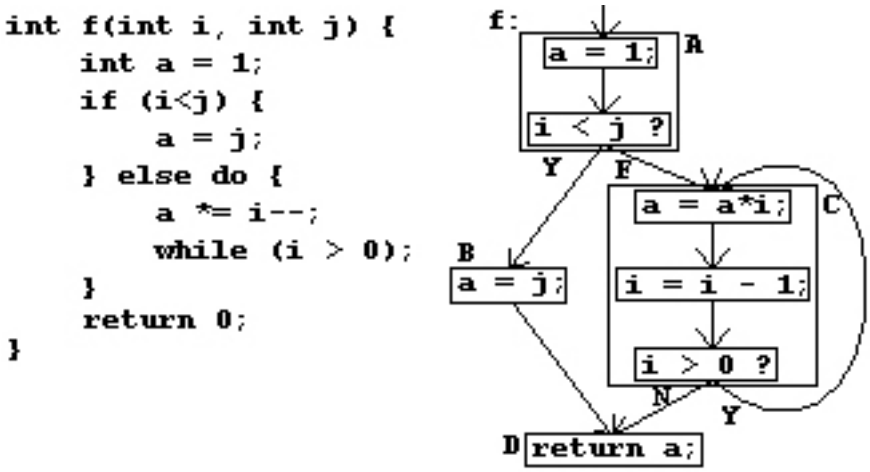
```
public class MyClass {
    private int i;

    public MyClass() {
        i = 3;
    }

    public int doSomething(j) {
        return this.i + j;
    }

    public static void main(String[] args]) {
        MyClass mc = new MyClass();
        mc.doSomething(1);
    }
}
```

sa transformuje na



Obrázok 3.1: **Graf toku riadenia. a.)** Zdrojový kód a k nemu ekvivalentný **b.)** blokový graf toku riadenia.

```
int i = 1;
int mci;
mci = 3;
int j = mci + i;
```

V príklade vidíme, že musíme jednoznačne pomenovať všetky premenné, ktoré mali v pôvodnom programe rovnaké meno. Preto každú premennú nazveme aj podľa triedy, v ktorej bola definovaná. Aj ak bola jedna inštancia triedy vytvorená viackrát s tým istým menom, stačí nám jedna globálna premenná pre každú premennú tejto triedy.

3.2.2 Graf toku riadenia

Keď už máme nízkoúrovňový program, tak k nemu zostrojíme graf toku riadenia (obr. 3.1), s ktorým budeme v ďalšom pracovať a snažiť sa ho čo najviac transformovať, aby sa nepodobal na pôvodný. Na obrázku je znázornené spojenie príkazov do blokov A, B, C, D. V bloku A sa nikdy nemôže stať, že za príkazom *a = 1;* nebude nasledovať $i < j$ *?*. Podobné fakty platia v bloku C.

3.2.3 Analýza premenných v bloku

Pre každý blok, ktorý sme získali, zostrojíme množiny premenných, ktoré je relevantné uvažovať pri analýze výpočtu bloku. Množiny pre blok B_i nazveme $KILL[B_i]$, $IN[B_i]$, $OUT[B_i]$, $GEN[B_i]$.

Definícia 3.2. *Množina $KILL[B_i]$ obsahuje pre každý blok všetky tie premenné, ktorých platnosť v danom bloku zaniká a ďalej sa nedajú použiť, alebo sa nepoužijú. Takže do množiny $KILL[B_i]$ vložíme premenné, ktoré:*

1. *boli v pôvodnom programe deklarované v bloku zdrojového kódu na nejakej úrovni a po vynorení sa z bloku do vyššej úrovne sa už k premennej nedalo dostať.*

2. *sa viac v pôvodnom programe nepoužijú, aj keď sú stále definované.*

Pre náš algoritmus je táto množina dôležitá, lebo poskytuje premenné, ktoré chceme použiť výlučne pre potreby obfuskácie a potrebujeme ich dostatok. Koľko, to závisí od konkrétnej implementácie algoritmu, je však možné, že globálna množina $KILL[B_1, ..., B_k] = KILL[B_1] \cup KILL[B_2] \cup ... \cup KILL[B_k]$, ktorá obsahuje po prejdení bloku B_k všetky premenné, ktoré boli označené ako mŕtve v už vykonaných blokoch, nemusí stačiť a my budeme nútení vytvárať aj vlastné premenné. Poznamenajme, že ak v pôvodnom programe bola premenná x definovaná v konkrétnom bloku, po vyjdení z neho stratila platnosť, avšak pri prípadnom ďalšom spustení tohto bloku sa znova obnovila jej platnosť. My toto riešime tak, že druhý krát už premenná bude mať iné meno a bude definovaná ako nová premenná. Preto nemusíme pri konštruovaní globálnej množiny $KILL$ uvažovať nad vytvorenými premennými v jednotlivých množinách $GEN[B_i]$.

Definícia 3.3. *Množina $GEN[B_i]$ je protikladom množiny $KILL[B_i]$ a obsahuje tie premenné, ktoré sa v bloku definovali a pri vstupe do bloku definované neboli. Globálne pole $GEN[B_1, ..., B_k]$ v bloku B_k je definované nasledovne $GEN[B_1, ..., B_k] = GEN[B_1] \cup GEN[B_2] \cup ... \cup GEN[B_k] \setminus KILL[B_1, ..., B_k]$ a obsahuje premenné, ktoré sa v ďalšom ešte použijú.*

Definícia 3.4. *Množina $IN[B_i]$ je množina všetkých premenných (s priradenými hodnotami), ktoré vstupujú do bloku B_i.*

Definícia 3.5. *Množina $OUT[B_i]$ je množina všetkých tých premenných aj s ich hodnotami, ktoré platia pri výstupe z bloku. Platí $OUT[B_i] = IN[B_i] \cup GEN[B_i] \setminus KILL[B_i]$.*

Množiny $IN[B_i]$ a $OUT[B_i]$ budeme využívať v ďalšom pri definícii blokových funkcií, tj. pri popise funkcionality blokov.

Poznámka 3.1. V sekciách 3.2.2 a 3.2.3 sme využívali postupy, ktoré sa dajú nájsť v prednáškach [28].

3.2.4 Funkcie blokov

Pre tú časť algoritmu, ktorá bude meniť poradie blokov, je dôležité, aby sme pre každý blok B_i mali definovanú jeho funkciu $f_{B_i} : IN[B_i] \cup GEN[B_i] \mapsto OUT[B_i]$ danú predpisom $f_{B_i}(v_i) = w_i, w_i \in OUT[B_i]$ ak $v_i \notin KILL[B_i]$ alebo $f_{B_i}(v_i) = null$, ak $v_i \in KILL[B_i]$. My budeme túto funkciu zapisovať v ďalšom ako $f_{B_i}(v_1, v_2, ..., v_n) = (w_1, w_2, ..., w_m)$, kde $v_i, 1 \leq i \leq n$ sú všetky vstupné premenné a $w_i, 1 \leq i \leq m$ sú všetky výstupné premenné bloku.

3.3 Použitie servera na získanie počiatočných dát

Jedným našim cieľom v tomto kroku je poprehadzovať poradie vykonávania blokov tak, aby toto poradie záviselo od konkrétnych hodnôt vstupných premenných. Z toho vyplynie, že graf toku riadenia programu sa bude meniť podľa vstupu. Aby bolo znemožnené sledovať bez spustenia programu hodnoty premenných, budú sa tieto získavať z bezpečného miesta. Dynamická analýza prijatých dát bude tiež nepoužiteľná, keďže jej cieľom je na základe vypočítaných hodnôt vyprodukovať skutočný graf toku riadenia, ktorý bude zbavený zbytočných vetiev kódu. Postup je nasledovný (obr. 3.2):

1. Algoritmus vyberie náhodne malý počet blokov, ktoré bude spracúvať. Kritériá výberu:
   (a) Vybrané bloky musia tvoriť súvislý podgraf.
   (b) Tento podgraf musí obsahovať vetvenie.
2. Zistí, aké sú podmienky, pri ktorých sa rozhoduje, do ktorého bloku riadenie programu vojde a aké sú hodnoty vstupov pre výstupy týchto podmienok. Na obr. 3.2a je to podmienka P.
3. Vytvorí ku každému spracovávanému bloku funkciu jeho vstupných premenných.
4. Náhodne určí, ako pri konkrétnych výsledkoch podmienok bude meniť poradie vykonávania blokov.
5. Ďalej algoritmus mení zdrojový kód nasledovne:
   (a) Na strane servera vytvorí funkciu, ktorá bude pre vstupné premenné, ktoré platia pred prvým vykonávaným blokom vracať tzv. *dispatcher* premennú, ktorá určí poradie vykonávania blokov a pole opravných dát, ktoré sa použijú pri

otočení poradia blokov na nápravu efektov tejto zámeny, ak boli premenné v blokoch na sebe závislé. Pre každý bod programu, ktorý si pýta od servera dáta bude osobitná funkcia. Konštruovanie nápravných dát si vyžaduje osobitnú funkciu v našom algoritme, je to riešiteľný matematický problém, ktorý nie je súčasťou tejto práce.

(b) Vytvorí v programe pred prvým blokom z upravovaného podgrafu funkciu *get(Bi, IN[Bi])*, ktorá si bude pýtať dáta od servera, parametrami budú názov bloku a pole vstupných premenných (obr. 3.2b). Na základe toho server presne vie, aké hodnoty má poslať naspäť. Pošle pole, kde nultý prvok bude hodnota *dispatcher*-a a ostatné prvky budú prvky poľa *repair*.

(c) Vytvorí v programe príkazy, ktoré spracujú prijaté dáta. Jedná sa o priradenie do *dispatcher*-a a o príkazy, ktoré naprávajú poľom *repair* zmenené hodnoty.

(d) Do programu pridá podmienku na základe *dispatcher*-a, ktorá rozhodne o poradí vykonávania blokov. Na obr. 3.2b je to podmienka *(A.contains(dispatcher)) ?*.

(e) Zmaže pôvodnú podmienku (podmienky), pretože jej výsledok bol vzatý do úvahy pri konštruovaní funkcie na serveri.

(f) Vytvorí nové cesty v *CFG*. Na konci obrátenej cesty pridá príkazy, ktoré pre každú výstupnú premennú nášho podgrafu upraví jej hodnotu na správnu. Aby podľa umiestnenia tejto opravnej rutiny nebolo zrejmé, ktorá vetva bola obrátená, alternatívou je zmeniť poradie vo všetkých.

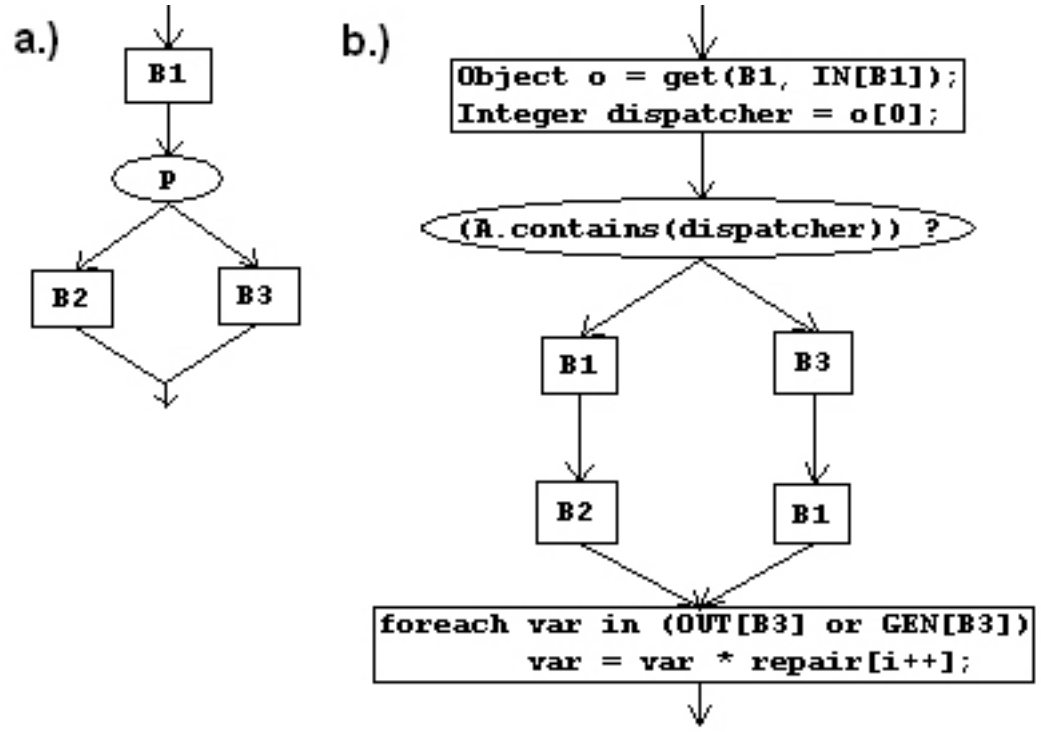
(g) Do všetkých blokov povkladá nové propagujúce premenné, ktorých hodnota na výstupe z upravovaného podgrafu bude závisieť od poradia vykonávania blokov, čiže tak, že v každom uvažovanom bloku sa budú do propagujúcich premenných priraďovať iné hodnoty, platná bude posledná.

6. Podľa konfigurácie algoritmu buď tento algoritmus skončí (ak bol dosiahnutý počet miest, kde sa mení graf toku riadenia), alebo pokračuje bodom 1.

Poznámka 3.2. Keďže sa jedná o relatívne malú ale dôležitú časť algoritmu, programátor by mal mať možnosť aj ručne pridať bloky alebo odobrať bloky, ktoré automatický obfuskátor vybral nevhodne, napríklad kvôli zbytočnej náročnosti výpočtu nápravných dát.

Rozoberme si tento postup dôkladne na príklade.

Príklad 3.2. Majme postupnosť blokov $B_1, B_2, ..., B_k$ od začiatku vykonávania programu. Každý blok B_i sa skladá z príkazov $B_{i_1}, B_{i_2}, ..., B_{i_m}$.



Obrázok 3.2: Degenerácia CFG

Krok 1 Pre prvý krok algoritmu bude vyhovovať $k = 3$ (obr. 3.2a), čiže sme vybrali tri bloky B_1, B_2, B_3, ktorým chceme meniť poradie.

Krok 2 Nech bol v pôvodnom programe najskor vykonaný blok B_1, po ňom nasledovala podmienka P

```
if (c < 5) then goto B21 else goto B31
```

a podľa jej výsledku nasledoval blok B_2, alebo blok B_3. B21 v kóde je prvý príkaz bloku B_2, čiže B_{2_1}. Pre druhý krok algoritmu teda máme podmienku $c < 5?$, keďže ale c, ktoré vstupuje do tejto podmienky je $c + a$ vstupných podmienok, budeme mať podmienku $(c + a) < 5?$.

Krok 3 Pre blok B_1 nech platí $IN[B_1] = \{a, b, c\}, OUT[B_1] = \{a, b, c, d\}$, čiže $GEN[B_1] = \{d\}, KILL[B_1] = \{\}$. Pre bloky B_2 a B_3 máme zhodne $IN[B_2] = OUT[B_2] = IN[B_3] = OUT[B_3] = \{a, b, c, d\}, GEN[B_2] = KILL[B_2] = GEN[B_3] = KILL[B_3] = \{\}$. Funkcie

pre bloky B_1, B_2 resp. B_3 sú f_{B1}, f_{B2} resp. f_{B3} definované:

$$f_{B1}(a,b,c) = (a+3, b-4, c+a, a*b+c)$$

$$f_{B2}(a,b,c,d) = (a+3, b+4, c+1, d-c)$$

$$f_{B3}(a,b,c,d) = (a-3, b+4, c+1, d+c)$$

Krok 4 Chceme otočiť poradie vykonávania blokov nasledovne: ak platí $(c+a) < 5$, tak sa program bude vykonávať v poradí $B1; B2;$, ak neplatí, tak v poradí $B3; B1;$ Otočenie teda nastáva keď platí (pre vstupné premenné) $c + a \geq 5$. Celková funkcia, ak sa najskôr vykonáva blok B_1 je:

```
if ((c + a) < 5) {
    fB12(a,b,c) = (a+6, b, c+a+1, (a*b+c)-(c+a))
} else {
    fB13(a,b,c) = (a, b, c+a+1, (a*b+c)+(c+a))
}
```

Nás zaujíma pri otočení len ten druhý prípad za *else*. Poďme teraz vykonávať program v opačnom poradí. Najskôr máme f_{B3}, potom f_{B1}, celková funkcia blokov B_3 a B_1 je

$$f_{B31}(a,b,c,d) = (a, b, c+a-2, (a-3)*(b+4)+c+1)$$

. Pri porovnaní s funkciou f_{B13} vidíme, že prvé dva členy výsledku sú rovnaké, treba upraviť len druhé dva. Pre tretí platí $f_{B13} - f_{B31} = 3$, pre štvrtý $f_{B13} - f_{B31} = -3a + c + 3b + 11$.

Krok 5 Na strane servera pridáme funkciu, ktorá keď prijme vstupné premenné a a c v bode programu, ktorý sme uvažovali, tak pošle nazad tzv. *dispatcher* premennú, ktorá určí poradie vykonávania blokov. Ak premenné c a a obsahujú také hodnoty, že $c + a \geq 5$, tak funkcia zo servera pošle naviac premenné, ktoré napravia zdeformovanie výsledku, ktoré vzniklo pri opačnom poradí spúšťania blokov. Tieto premenné budú v poli a boli vypočítané v kroku 4. Pole *repair* obsahuje nasledovné hodnoty: $repair[0] = 0, repair[1] = 0, repair[2] = 0, repair[3] = -3a + c + 3b + 11$. Po vykonaní blokov v poradí B_3, B_1 sa pridá na koniec kód, ktorý k výstupným premenným a, b, c, d pripočíta (resp. vykoná inú operáciu) hodnoty poľa *repair* - operácia $*$ na obr. 3.2.

Uvedomme si, že naším cieľom nie je si pýtať tieto dáta počas celého vykonávania programu, pretože v sekcii 1.2.2 sme spomenuli nevýhody techniky spúšťania dôležitých

algoritmov na serveri a tento prípad by bol pre komunikáciu so serverom ešte náročnejší (nejde len o jedno poslanie zadania a jedno prijatie výsledkov) a výkon by bol ešte viac znížený. My sme iba chceli zabezpečiť pre ďalšie kroky našej obfuskácie, aby existovali premenné, ktorým nebude možné statickou ani dynamickou analýzou určiť hodnotu. Vďaka týmto premenným potom budú nerozlúsknuteľné všetky predikáty, ktoré ich používajú na svoj výpočet.

Teoreticky by nám na neskoršiu konštrukciu zavádzajúcich podmienok stačili dáta, ktoré sme zatiaľ získali. My si však predstavíme ešte jedno alternatívne vylepšenie. Prijaté propagujúce premenné môžu byť aj adresy dát na serveri. V čase konštrukcie zavádzajúcej podmienky tento typ propagujúcich premenných použijeme nasledovne:

- v predikáte bude podmienka tvaru $get(p* == q*)$. Funkcia $get()$ pýta odpoveď od servera. Ten pozrie, či smerník p ukazuje na rovnaký objekt ako smerník q.

- Smerníky zo zahmleného programu na server neobsahujú priamo adresu, pretože by program vedel hneď, či sa ciele smerníkov rovnajú. Preto sú to len smerníky na ďalšie smerníky, ktoré sú uložené na serveri a tie môžu ukazovať na rôzne, alebo tie isté objekty.

- Na serveri sa vytvorí k objektov $O_1, O_2, ..., O_k$. Keď server pošle propagujúce dáta ako adresy, vytvoria sa nové smerníky na serveri, ktoré budú ukazovať na niektoré z týchto objektov. Obfuskátor bude v čase obfuskovania poznať ciele týchto smerníkov, preto ich môže použiť na vytvorenie podmienky.

Príklad 3.3. Povedzme, že program P zabezpečuje určitý výpočet pri kliknutí myšou na určité miesto. Na server sa budú posielať súradnice x, y myši pri kliknutí a server odpovie nejakými dátami, ktoré potom budú znamenať vždy rovnaké vykonanie predikátu, po ktorom bude nasledovať blok, ktorý spracúva kliknutie myšou. Chceme, aby odpoveď od servera bola na každom vstupe iná, aby deobfuskátor a spätný inžinier nemohli sledovať závislosť výstupu od vstupu. Zároveň tieto dáta budú smerníky na dátové štruktúry vytvorené na serveri. Keďže sa jedná o smerníky, nie je problém posielať vždy rozdielne hodnoty. Konkrétne, nech pri prvom poslaní súradníc na server sa tam vytvoria dva ľubovoľné objekty O_1 a O_2 a server pošle smerník na O_1. Potom v ďalšom bode programu, ktorý bude nasledovať neskôr sa tiež pošlú údaje na server, ten vie, že program zatiaľ drží len jeden smerník na dvojicu objektov a preto pošle ďalší, ktorý bude ukazovať buď na ten istý objekt, alebo na O_2. Pri vytváraní zahmleného programu P' sme zároveň vytvárali funkcie na serveri. Takže keď sme v zahmlenom programe uviedli predikát p,

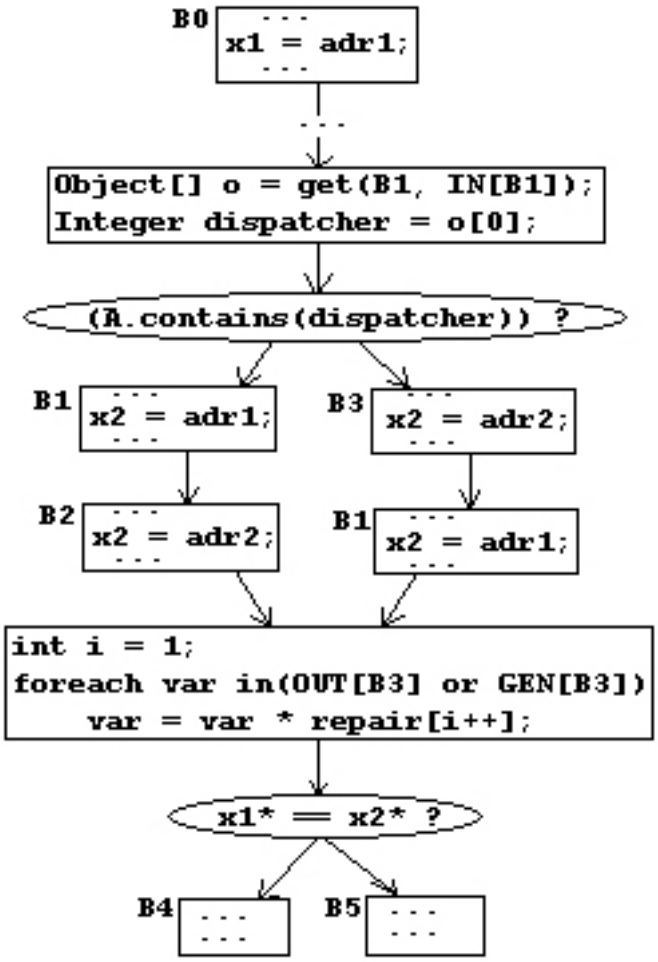
ktorý chceme, aby vždy vrátil *true* resp. vždy $false$ na určitých vstupoch, tak funkcia na serveri podľa toho vráti druhý smerník buď na objekt O_1 alebo O_2.

Príklad 3.4. Skúsme tento postup aplikovať pre program z príkladu 3.2. Tam mám bloky B_1, B_2, B_3, ako sú znázornené na obrázku 3.2. Teraz nech naviac po nich nasleduje falošný predikát Q. V pôvodnom programe by sa vykonával blok B_4 pri kliknutí myšou na určitý objekt a blok B_5 pri kliknutí mimo (obr. 3.3). Ako skonštruujeme podmienku Q? Nech niekedy pred blokmi B_1, B_2, B_3 bola do premennej x_1 priradená adresa smerníka na serveri. V bloku B_1 nech sa priradí do ďalšej premennej x_2 adresa iného smerníka, ktorý však ukazuje na ten istý objekt a v blokoch B_2 a B_3 zhodne nech sa priradí adresa smerníka na iný objekt. Podmienka P potom bude $x_1* == x_2*$?. Server vie podľa toho, akú *dispatcher* premennú poslal, ktoré priradenie pre x_2 platí. Máme teda príklad, keď sa kliklo myšou na tlačítko a chceme vykonávať blok B_4, ak sa kliklo inde, tak blok B_5. Čiže ak platí $c + a < 5$, tak je v x_2 priradená adresa na smerník na rovnaký objekt, ako v premennej x_1 a ak $c + a \geq 5$, tak je v x_2 priradená adresa smerníka na iný objekt, ako v x_1.

3.4 Propagácia dát použitých na zavádzajúce podmienky

Poznamenajme, že v kóde, ktorý vznikol v kroku 1 sa nachádzajú všetky premenné ako globálne. Keď premenná v pôvodnom kóde stratila platnosť, napríklad z dôvodu, že sa beh programu vynoril z vnorenejšej úrovne, v ktorej bola premenná definovaná, do vyššej, vtedy ju náš algoritmus v danom mieste vloží do množiny $KILL[B_i]$ a táto premenná sa ďalej môže použiť na propagáciu dát ovplyvňujúcich vykonávanie nasledujúcich predikátov. V konkrétnom bode programu sa môžu na propagáciu použiť premenné zo všetkých množín $KILL[B_i]$ blokov B_i, ktoré sa už vykonávať nebudú, alebo sa budú vykonávať znova až po použití propagujúcej premennej.

Pripomeňme si prvú myšlienku našej práce, ktorou bolo vytvoriť mechanizmus, ktorý využije nesprávny predpoklad z práce [29], kde autori popísali vytvorený deobfuskačný nástroj, ktorý dokázal viac či menej úspešne odstraňovať umelo pridané bloky kódu, nikdy však neodstránil všetky, autori to však považovali za výborný deobfuskačný výsledok. Odstránenie zbytočného bloku znamená odhalenie falošného predikátu a jeho odstránenie. My teraz popíšeme postup, ako vytvoriť závislosti medzi jednotlivými predikátmi tak, aby nevyriešenie jedného z nich znamenalo automaticky nevyriešenie ďalších, ktoré nasledujú. Formálnejšie, majme predikát p, definujeme preň množinu predikátov $Succ(p)$ ako množinu všetkých predikátov, ktoré sa vykonajú vždy až po prvom volaní predikátu p. Po



Obrázok 3.3: **Vytvorenie novej zavádzajúcej podmienky.** Nová podmienka Q.

každom predikáte nech nasledujú dve vetvy, z ktorých jedna môže byť falošná. V obidvoch vždy definujeme ľubovoľnú (ale rovnakú) premennú z množiny $KILL[B_1, ..., B_k]$, ak taká existuje a následne ju z tejto množiny odstránime. Ak je množina $KILL[B_1, ..., B_k]$ po vykonaní bloku B_k prázdna, tak vytvoríme novú premennú a vložíme ju do množiny $GEN[B_k]$. Nech je táto premenná *pv* (propagating variable), v obidvoch vetvách priradíme do premennej *pv* inú hodnotu.

V práci [29] bola existencia neprelomených predikátov len ako výsledok experimentu, nebola dokázaná a nemusí sa zvlášť pri menších programoch v programe ani nachádzať. Preto sme v sekcii 3.3 použili techniku pýtania si počiatočných dát z bezpečného servera, ktorý sa pre používateľa javí ako čierna skrinka, ktorá produjuje na základe nejakých vstupov výstupy, pričom je veľmi obtiažne zistiť závislosti medzi vstupmi a výstupmi.

Tieto dáta ďalej použijeme pri úprave predikátov. Do vybraných predikátov $\{r|r \in Succ(p)\}$, pridáme otázku týkajúcu sa nejakej propagujúcej premennej, prípadne viacerých premenných. Nech máme ľubovoľný predikát p a podmienku q, nový predikát $r = p \wedge q$ alebo $r = p \vee q$. Z toho, že nepoznáme dáta určujúce podmienku q, nepoznáme ani výslednú r. Podmienka q môže byť

- test na rovnosť (nerovnosť) s konštantou,
- test na rovnosť (nerovnosť) s nejakou používanou premennou programu,
- test na rovnosť (nerovnosť) s ďalšou propagujúcou premennou,
- takisto to znova môže byť porovnanie s niečím na serveri, aby sme znova získali istotu, že analyzátor toku riadenia a dát nebude vedieť výsledok podmienky q.

Podľa ladenia algoritmu, spomenutého v sekcii 3.11 vyberieme podmnožinu všetkých (aj novovytvorených predikátov podľa sekcie 3.8) predikátov, do ktorej pridáme takéto podmienky. Do každej vetvy, ktorá nasleduje potom pridáme definíciu ďalšej propagujúcej premennej viď príklad 3.5.

Príklad 3.5. Do predikátu $r1$ vložíme definíciu premennej a nasledovne:

```
boolean r1() {
    if ( q ) {
        int a = x;
        return 1;
    else {
        int a = y;
        return 0;
    }
}
```

pričom *return 1;* bude znamenať pokračovanie jednou vetvou a *return 0;* pokračovanie druhou, jedna z nich bude falošná, pretože predikát q podľa predpokladu má pevne stanovenú návratovú hodnotu *true* alebo $false$. Premenné x a y predstavujú v našom prípade celočíselné hodnoty generované algoritmom. Ich hodnota nie je dôležitá, dôležité je jedine to, aby si algoritmus počas zahmlievania pamätal, ktorá definícia bude platiť a aby ju v ďalšom mohol použiť. Vytvorili sme si teda premennú a, ktorej hodnotu pozná obfuskátor, ale nepozná deobfuskátor. Môžme ju teda použiť na tvorbu ďalších predikátov ($r2$) alebo rozšírenie iných ($r3$).

```
boolean r2() {
    if ( a <= 5 ) {
        int b = xx;
        return 1;
    } else {
        int b = yy;
        return 0;
    }
}
```

```
boolean r3() {
    if ( q1 && a <= 5) {
        int b = xy;
        return 1;
    } else {
        int b = yx;
        return 0;
    }
}
```

No a nakoniec môžme použiť aj spojené výsledky, ako ich používa predikát $r4$:

```
boolean r4() {
    if (a <= b) {
        int c = a;
        return 1;
    } else {
        int c = a + b;
        return 0;
    }
```

3.5 Zmena viditeľnosti premenných

V sekciách 3.2 až 3.9 opisujeme procedúry algoritmu, ktoré sú nevyhnutné pre jeho chod a až na záver v sekcii 3.10 spomenieme naviac technické vylepšenia, ktoré môžu podporiť buď účinnosť, alebo výkon algoritmu. Tu si však dovolíme menšiu výnimku. Ide o to, že algoritmus by dokázal pracovať aj s lokálnymi premennými, lenže to by výrazne znižovalo jeho odolnosť voči statickej analýze, ktorá by dokázala sledovať hodnoty premenných v rámci jedného bloku programu. Statická analýza nedokáže sledovať globálne dáta, pretože nedokáže určiť, ktorá definícia premenných platí v danom bode programu. Takže na prelomenie tohto bude treba dynamickú analýzu a náš algoritmus sa snaží popasovať aj s tou, ako uvidíme v ďalších častiach.

Čo znamená zmena všetkých premenných na globálne pre zavedené množiny GEN, $KILL$, IN, OUT? Zopakujme si, že tieto množiny nie sú štyri, ale pre každý blok jedna a navyše existujú takzvané globálne množiny GEN a $KILL$, v ktorých sa počas zahmlievania udržujú zjednotenia množín GEN a $KILL$, ktoré platia v konkrétnych bodoch programu. My sme sa rozhodli, že premenné, ktoré boli zmenené z lokálnych na globálne sa iniciálne

budú nachádzať v množine $KILL$, aby boli použiteľné pri produkovaní zavádzajúcich predikátov. Musíme však dávať pozor na miesto v programe, kde nastane reálne priradenie podľa pôvodného programu, čiže priradenie, ktoré sa podieľa na výsledku výpočtu programu. Po tomto priradení si bude algoritmus udržovať o premennej informáciu, že je v stave $useful$ a až kým sa nedostane do množiny $KILL$ tak s ňou máme zakázané pracovať s výnimkou použitia na porovnávanie v nových podmienkach. Ak algoritmus zistí, že takúto premennú definoval a od tej definície nepoužil na žiadnu podmienku, tak túto definíciu spätne zruší z dôvodu neznižovania výkonu programu.

3.6 Použitie dynamických dát

Výrazné zvýšenie zložitosti analýzy programu prináša zavedenie dynamických výpočtov miesto statických. Toto vylepšenie je o to účinnejšie, že už všetky premenné sú globálne a ku každej musíme nájsť jej definíciu nielen vo svojom výhľade, ale v celom programe, čo je netriviálna záležitosť (viď. sekcia 3.12 o analýze), keďže treba určiť, ktorý blok s definíciou uvažovanej premennej sa vykonal ako posledný. To pôjde len dynamickým sledovaním vykonávania programu a aj obmedzeniam dynamickej analýzy sa venuje náš algoritmus.

Konkrétne ide o zavedenie dynamických priradení v týchto prípadoch:

- Miesto priradenia konštanty v priraďovacom príkaze použijeme na pravej strane výraz pozostávajúci z premenných a operátorov, ktorého hodnota sa rovná nahrádzanej konštante.
- Skoky budú tiež dynamické, čo pri našom algoritme vlastne znamená, že *dispatcher* premenná, ktorá pre série blokov určuje poradie ich vykonávania nebude konštanta, prípadne určená premennou tesne pred použitím.
- Nakoniec aj všetky porovnania budú dynamické, čiže premenné nebudeme porovnávať s konštantami.

Príklad 3.6. Jednoduché priradenie

```
x = a;
```

nahradíme

```
x = y1 + y2;
```

kde *y1* a *y2* sú ľubovoľné premenné také, že platí $y1+y2 = a$, ak takéto premenné existujú, alebo

```
x = y1 + y2 + b;
```

ak takéto premenné $y1$, $y2$ neexistujú. b je vhodná konštanta. Návestia skokov nahradíme nasledovne:

```
if (E) then goto Bi else goto Bj
```

sa stane napríklad

```
if (E) then goto B(y1 + y2 + y3) else goto B(z1 + z2);
```

Nakoniec

```
if (x == a) ?
```

kde a je konštanta, dynamicky počítanou pravou stranou porovnania, napríklad

```
if (x == x1 + x2);
```

3.7 Použitie smerníkov a vytváranie aliasov

Aby sme sa dostali až k vytváraniu aliasov, potrebujeme najskôr zaviesť pre každú premennú smerník, ktorý na ňu bude ukazovať, respektíve viacero smerníkov. Majme globálne premenné $x_1, x_2, ..., x_n$, zavedieme smerníky $p_1, p_2, ..., p_m$, kde $m > n$ tak, že na každú premennú bude existovať aspoň jeden smerník (prípadne smerník nemusí existovať, ak by sa tento prístup ukázal ako príliš náročný na použitú pamäť).

V [30] pri vytváraní aliasov dát používali funkcie, v ktorých boli definované lokálne premenné. Vo vnútri týchto funkcií sa aliasmi stávali tieto premenné a globálne premenné, ktoré do funkcie vstupovali ako formálne parametre. Túto techniku nebudeme používať, keďže sme si funkcie rozbalili. Ukážeme si iný postup pri vytváraní aliasov, ktorý sa opäť spolieha aj na dáta z bezpečného servera. Pri implementácii algoritmu za použitia postupov z [30] by došlo buď k nerozbaleniu všetkých funkcií a k deklarácii niektorých premenných ako lokálnych. Alternatívou by bolo vytvorenie funkcií len za účelom vytvorenia aliasov na dáta.

My pre vytváranie aliasov vytvoríme nasledujúce metódy, ktoré sa budú odlišovať svojou bezpečnosťou a výkonnosťou.

1. Menej bezpečný postup je ten, keď jednoducho v kóde postupne viacerým smerníkom priradíme adresu tých istých dát. Deobfuskátor by proti tomuto postupu mohol sledovať ciele všetkých smerníkov ako ich adresy. My však už máme zavedené premenné, ktoré sme získali z bezpečného miesta. Môžme ich použiť dvojako:

   (a) Niektoré premenné môžu obsahovať priamo relatívnu adresu pamäte. Smerník potom nasmerujeme, aby ukazoval na túto adresu. Keď dve premenné x_1 a x_2 budú obsahovať rovnakú adresu - adresu premennej y, tak dva smerníky p_1 a p_2, pre ktoré platí $p_1 = x_1$ a $p_2 = x_2$ vytvoria alias na y. Ďalší konkrétny postup, ako vytvoriť alias je definovanie smerníka p pomocou cieľa iného smerníka q:

   ```
   p = q + offset;
     ...
   p = p - offset;
   ```

   takto vznikne alias, smerníky p aj q ukazujú na to isté miesto.

   (b) Použijeme ich ako podmienku, ktorá vytvorí dve vetvy. V prvej je do smerníka priradená jedna adresa, v druhej iná. V jednej z nich je vytvorený alias, v druhej nie. Výsledok podmienky nie je známy deobfuskátoru, takže ani existencia aliasu od tohto bodu.

2. Bezpečnejší postup je opäť si pýtať adresy zo servera. Postup pýtania si dát už bol popísaný v 3.3.

Poznámka 3.3. Uviedli sme zápisy, ktoré nie je možné použiť v javovskom zdrojovom kóde, pretože ten neumožňuje prácu so smerníkmi. V Jave sa miesto smerníkov používajú referencie na objekty, nie je umožnená smerníková aritmetika. Možnosti ako používať smerníky v Jave sú viaceré. Štandard je *JNI (Java Native Interface)*, ktorý umožňuje vkladanie natívneho kódu do Javy. Natívne metódy sú priamo spúšťané *JVM*, nekompilujú sa do *Java bytecode*. Týmto sa ale program stáva závislým na operačnom systéme, čím padá jedna z veľkých výhod Javy. Druhou možnosťou je použitie niektorých nástrojov, ktoré kompilujú *C* kód do *Java bytecode*. V tomto prípade skutočné smerníky zanikajú, používajú sa rôzne alternatívne spôsoby nahrádzania smerníkov a adries, napríklad použitím globálneho poľa. Príklady týchto programov: *AMPC*, *C2J*, *Jazillian*. Poslednou možnosťou je vytvorenie triedy *Pointer* s jedným členom typu *Object* a metódami *void setObject(Object obj)* na nastavenie objektu, na ktorý tento nepravý smerník ukazuje a *Object getObject()* na získanie objektu na jeho následné testovanie na ekvivalenciu s iným objektom.

3.8 Tvorba zavádzajúcich podmienok

V tomto momente už máme zavedené propagujúce premenné, všetky premenné sú globálne, k premenným boli vytvorené smerníky, ktoré na ne ukazujú a pomocou týchto smerníkov bol vytvorený aliasing premenných. Keď už máme všetky potrebné prostriedky, môžme si konečne popísať postup na samotné vkladanie zavádzajúcich podmienok do zdrojového kódu a pridávanie zbytočného kódu do falošných vetiev.

1. Vyberieme blok, ktorý chceme rozdeliť, kritériom výberu bude počet príkazov v bloku, vyberieme ten, kde ich je najviac. Naviac by sa programátor mohol rozhodnúť, kam vložiť podmienku na základe vlastného uváženia.

2. V tomto bloku sa vyberie náhodne príkaz, za ktorý vložíme túto podmienku.

3. Vložíme podmienku *if (E) then goto Sx else Sy*, kde S_x alebo S_y je prvý príkaz novej vetvy, ktorú vytvoríme ako falošnú (zbytočnú).

4. Falošnú vetvu vytvoríme použitím živých premenných, ktoré sa nachádzajú v globálnom poli GEN blokov, ktoré sa vykonali pred týmto blokom a zároveň sa nenachádzajú v poli $KILL$ tých istých blokov. Túto vetvu nekonštruuje algoritmus automaticky, pretože náhodne generovaný kód by očividne mohol nerobiť nič užitočné, napríklad pri postupnosti príkazov

   ```
   S1. x := y;
       ...
   Sk. x := x';
   ```

   Pričom medzi príkazmi S_1 a S_k nie je ani jedno priradenie do x. Takéto príkazy by vedel automatický deobsuskátor odstrániť. Preto algoritmus skopíruje postupnosť príkazov z ľubovoľného bloku z programu, kde sa vykonáva užitočný výpočet. Je potrebné, aby naviac do zbytočnej vetvy pridal aj priradenia do propagujúcich premenných (samozrejme cez smerníky), ktoré však nemajú žiadny vplyv na hodnoty týchto premenných.

5. Samotnú podmienku E vytvorí obfuskačný algoritmus tak, že vyberie dvojicu dát, na ktoré sa bude pýtať, čiže buď smerník na propagujúcu premennú a konštantu, premennú alebo ďalší smerník na propagujúcu premennú. Podobne môže vybrať aj smerník na dáta na bezpečnom serveri, ako bolo uvedené v 3.3. K týmto dátam pridá jeden z boolovských operátorov $<$, $>$, $\leq$, $\geq$, $=$. Podľa konfigurácie algoritmu

alebo podľa nedeterministického rozhodnutia algoritmu sa ďalej produkujú takéto podmienky tak, že k podmienke E sa pripojí ďalšia podmienka F pomocou logickej spojky $\vee$ alebo $\wedge$.

3.9 Dynamická zmena kódu

Možnosťou vylepšenia nášho postupu je použiť v obmedzenej miere dynamickú zmenu kódu. Nápad použiť počas behu sa meniace inštrukcie za účelom obfuskácie nie je úplne nový, prvé myšlienky sú popísané v [13]. Autori pracujú so skompilovaným kódom, v ktorom vyberú cieľové inštrukcie, ktoré sa nahradia falošnými inštrukciami. Ďalej je nutné pred tieto cieľové inštrukcie vložiť inštrukcie, ktoré nahradia falošné inštrukcie originálnymi. Za cieľ potom vložia ďalšiu inštrukciu, ktorá cieľ znova nahradí niečím falošným. Tento postup, ak by nebol vylepšený o ďalšie techniky zahmlievania by nebol dostatočný, pretože útočník by si pamätal len tie inštrukcie, ktoré sa v skutočnosti vykonali a ktoré nemenia iným inštrukciám kód. Čiže dynamickým spúšťaním programu na rôznych vstupoch by sme dostali skutočný kód, ktorý sa vykonáva. V [16] tento základný postup vylepšili, nemenia len jednotlivé náhodne vybrané inštrukcie, ale menia naraz celé úseky kódu, ktoré sú úsekmi viacerých funkcií naraz.

My v našej práci chceme zmeniť zdrojový kód falošných predikátov, aby sme zamedzili dynamickej analýze správania programu sledovaním výsledkov podmienok ako v 2.3.2. Náš prístup k samotnej zmene kódu za behu je iný, nebudeme meniť strojové inštrukcie, ale zmeny implementujeme priamo do zdrojového kódu. Z tohto dôvodu je termín *dynamická zmena kódu* trochu zavádzajúci. My budeme dynamickú zmenu len simulovať. Zavedieme nový typ premenných, ktorých hodnoty budú textové reťazce. Tieto reťazce budú obsahovať časti predikátov, napríklad operátory $<, >, \geq, \leq, =$, ďalej logické spojky, názvy použitých premenných, konštánt, výrazy - všetky konštrukty, z ktorých sa dá poskladať predikát.

Popíšeme dva konkrétne postupy, ktorými simulujeme dynamickú zmenu kódu predikátov.

1. Prvý postup je závislý na konkrétnom jazyku, my si ho ukážeme ako vždy pre Javu. Z novozavedených premenných vyskladáme triedu, ktorá bude mať jedinú metódu, ktorou bude samotný predikát. Túto triedu uložíme do súboru s príponou *.java*, skompilujeme a následne pustíme. Toto všetko sa vie udiať aj v operačnej pamäti, takže výkon je dostatočný. Zabezpečuje to trieda *javax.tools.JavaCompilerTool*, ktorá sa nachádza v JDK 6+. V predchádzajúcich verziách Javy sa používal napríklad

sun.tools.javac.Main, ktorý mal viacero obmedzení. Znova nahrávať tie isté triedy jedným *classloader*-om nie je možné. Riešenie poskytuje *JavaAssist* - nástroj, ktorý umožňuje pomocou vlastného *classloader*-a nahrávať jednu triedu viackrát a popritom ju meniť.

2. Druhý postup nepoužíva žiadnu prácu s pamäťou, takže už naozaj nemá nič spoločné s dynamickou zmenou pamäte, okrem myšlienky. Symboly do predikátu vyberieme pomocou *if-then-else* konštruktov.

Oba postupy si ukážeme na príklade.

Príklad 3.7. Majme predikát p, ktorého zdrojový kód je nasledovný

```
boolean p(a, b) {
    if (a > b) return true
        else return false;
}
```

Chceme, aby sa v tomto predikáte zmenilo porovnanie $(a > b)$ na $(a < b)$, nazvime si tento predikát pracovne p'. Podľa postupu so skompilovaním máme definované tieto premenné:
x = *'public class P {boolean p(a,b) {if(a'*
y = *'<'*
w = *'b)return true else return false;}}'* Vytvorime subor P.java, don vlozime $x + y + w$, skompilujeme vyššie popísaným postupom. V kóde potom miesto predikátu p použijeme sekvenciu

```
P p = new P();
if (p.p()) then {} else {}
```

Majme teraz ten istý predikát p, ktorý chceme zmeniť na p'. Máme k tomu zavedenú premennú p_x, ktorá drží boolovskú hodnotu, podľa ktorej sa rozhodne, ako bude predikát vyzerať.

```
boolean p(a, b) {
    if (px) {
        if (a > b) return true
        else return false;
    } else {
        if (a < b) return true
        else return false;
    }
```

V čom je to dynamická zmena a v čom je jej účinnosť? Zopakujme si, že cieľom útočníka je zostrojiť originálny graf toku riadenia, čiže odstrániť falošné predikáty. My sme vložili falošné predikáty, ktorým sa bez zmeny ich kódu dá sledovať, aké dávali pri viacerých spusteniach výsledky a na základe toho sa dá usúdiť, ktorá vetva za podmienkou je zbytočná. Keď však zmeníme zdrojový kód predikátu, tak na rovnakých vstupoch vráti rôzne výsledky. Útočník síce dokáže sledovať premenné, z ktorých sa vyskladáva predikát, ale kvôli globalizácii všetkých premenných veľmi zložito - je to vlastne doterajší postup nášho algoritmu. Naviac, ak pre každý symbol v predikáte existuje k ďalších, ktoré ho nahrádzajú a v predikáte je n takýchto symbolov, potom útočník musí zrazu sledovať k^n predikátov. Kde k aj n sú pomerne malé konštanty, takže aj k^n je konštanta a jej veľkosť závisí od zložitosti predikátu.

Poznámka 3.4. Pri dynamickej zmene predikátov musíme upraviť aj vetvy, ktoré sa nachádzajú za týmito predikátmi tak, aby pri všetkých vetvách sa vykonával správny program, čiže už nemôžme len náhodne pridať zbytočný kód. Jedna vetva musí byť platná pri nejakom tvare predikátu a druhá pri inom.

3.10 Technické vylepšenia

Algoritmus, ako sme ho doteraz popísali, robí len nutné činnosti, aby naša myšlienka fungovala a ostala relatívne jednoduchá. Obfuskácia je však na nápady neobmedzená téma, ani náš výpočet ako bol uvedený v sekcii 2.2 zďaleka nepokrýva všetky rôzne zlepšujúce myšlienky, od významne ovplyvňujúcich celé zahmlievanie až po pomerne malicherné úpravy. Takisto aj pre náš algoritmus môže existovat viacero vylepšení. Tie, ktoré nás napadli a považujeme za zaujímavé a užitočné sú uvedené v tejto sekcii.

1. Kvôli zvýšeniu výkonu programu (skôr by sa dalo povedať, že kvôli zníženiu nepriaznivého dopadu na výkon) pri pýtaní dát zo servera zavedieme výpočet všetkými vetvami bez ohľadu na vykonávanie podmienok. Toto je dôležitý krok, pretože sieťové odozvy môžu byť dlhé, čiže program by pri každom čakaní na dáta zo servera príliš dlho stál. Ak by bol algoritmus nakonfigurovaný na produkovanie veľmi bezpečného programu, ktorý si často pýta dáta zo servera, dlhé odozvy by sa mimoriadne odrazili na výkone programu. Výpočet programu sa pri tomto zlepšení nezastaví, ale pokračuje ďalej, aj keď nemá v niektorej premennej priradenú hodnotu. To znamená, že všetky podmienky, ktoré používajú túto hodnotu sa budú ďalej vykonávať všetkými vetvami. Program si musí pamätať všetky podmienky a cesty, ktorými sa uberá

samostatne. Hodnotou nedefinovanej premennej sa môže definovať hodnota inej premennej, takže pri takomto predbiehajúcom výpočte by aj jej hodnota ostala nedefinovaná a aj všetky podmienky, ktoré závisia na tejto premennej by sa museli vykonávať všetkými vetvami. Zrejme by čoskoro došlo k preplneniu pamäte. Preto algoritmus bude pokračovať len niekoľko úrovní definícií premenných od definície pomocou dát zo servera a potom ostane stáť. Koľko úrovní bude predbiehať sa stane predmetom konfigurácie algoritmu. Po prijatí potrebných dát zo servera sa priradí hodnota čakajúcej premennej a pokračuje sa vo výpočte, vetvy, do ktorých sa nevstúpilo sa neberú už do úvahy, dopĺňajú sa všetky definície, ktoré nemohli byť vykonané. Výhodou je, že premenné, ktoré nezávisia od dát na serveri sú už vypočítané. Nastáva otázka, či je tento postup rýchlejší, ako čakanie na dáta. Tu však nevieme poskytnúť odpoveď, tá závisí od konkrétnej implementácie, od konfigurácie algoritmu a predovšetkým od konkrétneho zahmleného programu. Jediná možnosť, ako zistiť odpoveď je experiment na konkrétnej implementácii, dátach a programe.

2. Ďalšie vylepšenie si vyžaduje ručnú manipuláciu s pôvodným zdrojovým kódom, ktorej môžu napomáhať *profiling tools*, ktoré označia bloky, ktoré sú najčastejšie spúšťané. Ide o označenie miest v kóde, ktoré sú náročné na výkon a preto nie je vhodné sa v nich pýtať na dáta zo servera, čo môže program spomaliť, v najhoršom prípade však aj úplne zastaviť. Druhá úroveň označovania kódu určí miesta, na ktoré dokonca nie je ani možné vložiť nové inštrukcie, špeciálne inštrukcie zavádzajúcich podmienok.

3. Veľkosť mapovacej tabuľky na bezpečnom serveri môže byť obmedzená. Preto môžme výpočet dát, ktoré sa vracajú zo servera spraviť pomocou mapovacích funkcií, ktoré budú vstup mapovať na výstup. Výrazne sa znížia pamäťové nároky na server, avšak zníži sa bezpečnosť - odstránenie náhodnosti dáva útočníkovi šancu zistiť tieto mapovacie funkcie.

3.11 Konfigurácia algoritmu

Postupy v sekciách 3.3 až 3.9 sa snažili zaviesť maximálnu možnú ochranu softvéru a nehľadeli na výkon výsledného programu. Všetky postupy pri nadmernom použití však vedú buď k spomaleniu programu, alebo môžu viesť k príliš veľkým pamäťovým nárokom. Preto je treba nájsť kompromis medzi bezpečnosťou a výkonom. V tejto kapitole preto prichádzajú obmedzenia obfuskácie.

1. Základným nastavením algoritmu je určenie hraníc pomeru počtu pôvodných blokov originálneho programu ku počtu novouvedených predikátov a tým aj falošných vetiev.

2. Veľmi dôležitým výkonnostným parametrom je množstvo dát, ktoré sa pýtajú zo servera. Patria tu počiatočné propagujúce premenné, *dispatcher* premenné, dáta používané na vytváranie aliasov.

3. V 3.6 sme sa snažili nahradiť všetky statické výrazy dynamickými, čiže počítanými na mieste. To samozrejme uvedie ďalšie výpočty a dôjde k spomaleniu. Algoritmus musí byť konfigurovateľný na percentuálny podiel nahradenia týchto výrazov.

4. Všeobecne môžme nastaviť maximálne prípustné spomalenie programu. Určíme výpočtovú zložitosť pôvodného programu a výsledného buď staticky - spočítame počet operácií, každá má svoju váhu, pretože každá operácia môže mať inú zložitosť, napríklad už aj sčítanie na krátkych celých číslach je rýchlejšie ako na reálnych číslach, to, že násobenie je zložitejšia operácia ako sčítanie je známe. Môžme určiť výkon programu aj dynamicky, spúšťaním a meraním času vykonávania. Ak je program príliš pomalý, zmeníme nastavenia algoritmu tak, že obmedzíme množstvo uvedených nových výpočtov, najmä vo falošných predikátoch.

5. K zníženiu pamäťových nárokov môže dôjsť, keď nastavíme percento premenných, na ktoré existuje smerník. Nie je to však radikálne riešenie, pretože smerníky zaberajú v pamäti málo miesta.

6. Pamäťové nároky sa dajú znížiť aj na serveri. V sekcii 3.3 sme uviedli, že na serveri sa zavádzajú premenné, na ktoré ukazujú smerníky. Môžme pri každom novom smerníku zaviesť nový objekt, alebo niektoré objekty používať viackrát, aby sme šetrili pamäťou.

7. Ďalšie dôležité šetrenie pamäte na serveri môže byť poskytnuté obmedzením veľkosti mapovacej tabuľky použitím funkcií, ktoré budú mapovať vstup na výstup. Počet takýchto funkcií ovplyvňuje hranicu medzi bezpečnosťou a šetrnosťou.

8. Mimoriadne môže spotrebúvať pamäť postup zo sekcie 3.2, kde sme pri znovuoživení premennej v pôvodnom programe vytvorili v obfuskovanom programe novú premennú, pretože tá pôvodná už mohla byť použitá na naše zahmlievacie účely. Existujú dve šetrenia pamäte pre tento prípad. V prvom z nich dáme nové meno premennej

len v prípade, že táto bola použitá na obfuskáciu. Druhá možnosť je označiť premenné, ktoré ešte môžu byť znovupoužité ako *useful* a tým znemožniť ich použitie na vedľajšie úlohy.

9. Veľkosť výsledného programu (ktorá v dnešných časoch často nehrá žiadnu úlohu) sa dá ovplyvňovať nastavením maximálnej a minimálnej dĺžky nových zbytočných blokov.

10. v prípade technického vylepšenia číslo 1 zo sekcie 3.10 treba nastaviť maximálnu hĺbku, do ktorej sa má program vnoriť počas čakania na dáta zo servera.

11. Tak ako sa všeobecne nastavila maximálna konštanta spomalenia programu, tak sa nastaví aj maximálna možná miera zvýšenia pamäťových nárokov, čiže pamäť zabratá propagujúcimi premennými, smerníkov, nových aliasovaných dát a počet zglobalizovaných premenných.

3.12 Analýza algoritmu

Viaceré naše časti algoritmu sa spoliehajú na aliasing dát, preto najskôr uvedieme vetu, ktorá hovorí o teoretickej dostatočnej zložitosti tohto postupu. Skutočnú zložitosť pochopenia pri obfuskácii nie je možné určiť, pretože viacero veličín, ktoré sú potrebné nie je možné merať (domýšľavosť človeka), prípadne niektoré veličiny sú reálne obmedzené, napríklad počet nových vetiev, počet nových premenných. Idea vytvárania zavádzajúcich podmienok použitím aliasingu dát bola prevzatá z [30], kde bola aj ukázaná zložitosť statickej analýzy.

Veta 3.12.1. Pri použití globálnych smerníkov je problém presného určenia adries cieľov nepriameho vetvenia NP-ťažký[1].

Nebudeme sa venovať časovej a priestorovej zložitosti samotného algoritmu, pretože ten zbehne len jediný krát pred vydaním programu do obehu. Zaujíma nás jedine zložitosť a bezpečnosť výsledného programu. Zopakujme si, ktoré činnosti robí náš algoritmus, rozdelíme si ich podľa toho, ako ich treba osobitne analyzovať. Potom si pre každú z nich podiskutujeme o jej zložitosti.

- Analýza a úprava kódu
- Získanie dát z bezpečného miesta
- Zmena grafu toku riadenia

[1] Dôkaz je v [30], strany 11,12.

- Použitie dynamických návestí
- Vytváranie aliasingu
- Tvorba zavádzajúcich podmienok
- Propagácia dát
- Dynamická zmena predikátov

Analýza a úprava kódu

Tento krok síce nespadá pod hlavnú myšlienku algoritmu, má len pomocný charakter, no napriek tomu si treba uvedomiť, akú silu nám ponúka. Ide vlastne o efektívnu obfuskáciu, pretože z programu odstránime všetky konštrukty, ktoré nám napovedajú o správaní a vlastnostiach sa programu. Odstránime triedy, procedúry, dátové štruktúry okrem primitívnych typov a polí. Spätný inžinier je nútený hľadať v nízkoúrovňovom kóde také sekvencie (makrá), z ktorých sa dá zostrojiť naspäť vyššieúrovňový kód. Existujú techniky, ako transformovať tento nízkoúrovňový kód, aby takéto makrá neboli odhaliteľné, to je však už obfuskácia na úrovni skompilovaného kódu, čo nie je cieľom tejto diplomovej práce. Úprava kódu ovplyvňuje výkon aplikácie len minimálne, mierne zrýchlenie prinesie rozbalenie a tiež to, že sa nevytvárajú objekty, len primitívne typy.

Získanie dát z bezpečného miesta

Ak sa nám podarí po tejto fáze mať premenné, ktorých hodnotu nebude možné zistiť statickou, dynamickou a ani kombináciou týchto metód, máme vyhrané. Útočník môže jedine zisťovať závislosť vstupu od výstupu - ktorá je však náhodná. Povedzme, že v nejakom bode programu posielame na server k premenných $x_1, x_2, ..., x_k$, pričom každá premenná x_i môže mať m rôznych hodnôt. Máme teda m^k rôznych vstupov a môžme teda dostať toľko isto rôznych výstupov. Tabuľka na serveri je konštruovaná tak, že neexistuje funkcia, ktorá by dokázala mapovať vstup na výstup, výstup je v čase obfuskácie generovaný náhodne. Preto zistiť celú tabuľku pre jeden bod programu zákonite musí znamenať m^k spustení programu. Naviac máme n miest, na ktorých si pýtame dáta zo servera. Ak sa útočníkovi nepodarí prelomiť túto fázu, všetky ostatné sa budú mocť spoľahnúť na útočníkovi neznáme dáta.

Avšak čo sa týka pridaného času, ten pri pýtaní dát zo servera nie je zhora obmedzený, keďže server môže prestať reagovať úplne. Aj pri jeho funkčnosti môže byť zdržanie spôsobené sieťovou komunikáciou značné. Možné alternatívy sú spomenuté v sekciách 3.10

alebo 3.11. Algoritmus by mal vedieť fungovať aj bez pýtania dát zo servera, prvotná myšlienka fungovala bez týchto dát, len je znížená odolnosť.

Použitie dynamických návestí

Miesto lokálne počítaných alebo konštantných skokov použijeme globálne premenné, ktoré vzniknú vo vetvách za falošnými predikátmi, k nim vieme zložitosť ich odhalenia, preto použitie dynamických návestí pridá rovnakú zložitosť (NP-úplný problém, viď úvod do tejto sekcie) ako vytvorenie zavádzajúcich predikátov.

Časová zložitosť je zvýšená o výpočty cieľov skokov, priestorová je vyššia, ak sa uvedú nové premenné, algoritmus môže byť naimplementovaný aj tak, že sa použijú propagujúce premenné. Príklad: chceme nahradiť príkaz skoku *goto(5)*. Máme propagujúcu premennú x, ktorej hodnota je 4. Skok nahradíme $goto(x + 1)$.

Zmena grafu toku riadenia

Pridaná výpočtová zložitosť nemusí byť veľká, ak tento postup použijeme len niekoľko krát na deformovanie toku riadenia, kým nemáme dostatok premenných. Pri každej zmene musíme počítať s tým, že na konci musíme pridať blok, ktorý dá do poriadku zmeny zapríčinené otočením poradia blokov. Táto oprava môže byť taká zložitá ako výpočet samotný.

Tvorba zavádzajúcich podmienok

Ak mal pôvodný program n príkazov, môže vzniknúť $O(n)$ nových vetiev a v nich $O(n)$ ďalších a tak ďalej, čiže pridaná zložitosť môže byť až exponenciálna, toľko treba aj premenných aj výpočtov zavádzajúcich predikátov, preto musíme mať konštanty na obmedzenie počtu pridaných vetvení.

Vytváranie aliasov

Uviedli sme niekoľko postupov, ako získať aliasy. Ak si adresy pýtame priamo zo servera, bezpečnosť je nám známa. V opačnom prípade je potrebná globálna analýza celého programu, ktorá, ako už tiež vieme, je NP-úplná. Na aliasy sa spolieha tvorba zavádzajúcich podmienok, takže aká je pravdepodobnosť odhalenia aliasov, taká je aj pravdepodobnosť odhalenia každej podpodmienky v predikáte.

Propagácia dát

Propagácia dát je esenciálna pre celý postup, pretože tá nám zaručuje, že po jednom neprelomenom predikáte máme dáta, ktorých hodnoty nie sú známe a tieto dáta potom môžme používať na rôzne účely. O prelomení propagácie dát nemožno hovoriť, jedine o prelomení predikátu, za ktorým sa tieto dáta vytvárajú, takže to je vec sekcie o tvorbe zavádzajúcich podmienok. Propagácia dát vytvára umelo závislosti medzi blokmi kódu, čím zamedzuje deobfuskačnej technike nazvanej *program slicing*, pretože v transformovanom programe sa nedajú vyčleniť samostatné úseky kódu - tie vždy používajú dáta z iných častí a poskytujú nové dáta pre ďalšie časti.

Dynamická zmena predikátov

Dynamická zmena predikátov zabraňuje dynamickej analýze programu, čiže znemožňuje značenie predikátov podľa výsledkov, aké dávajú - zabraňuje *black-box* analýze. Jej zložitosť, ako sme si ju popísali pre Javu je znova ťažko popísateľná - závisí hlavne od konfigurácie algoritmu, podľa toho, ako často sa použije, od spôsobu spúšťania zmenených predikátov. Pri ich kompilácii a nahrávaní do pamäte dôjde k väčšiemu zdržaniu ako pri ostatných spôsoboch.

Záver

V práci sme najskôr rozobrali doteraz vyvinuté obfuskačné techniky. Ešte skôr, ako sme začali s opisom našich vylepšení sme taktiež zosumarizovali možnosti automatickej deobfuskácie. Tie nám potom pomohli pri vývine našich myšlienok, pretože tie boli navrhnuté tak, aby existujúce deobfuskačné techniky boli neúčinné. Tento postup sa v uvedenej použitej literatúre často nespomína. Aj preto majú niektoré zaužívané postupy bezpečnostné diery, ako platí aj o práci [30], ktorú sme si zobrali za vzor, ale keďže sme poznali postup, ako odstrániť obfuskačné transformácie použité v tejto práci, navrhli sme odolnejšie vylepšenia. Tie sa na rozdiel od predošlých prác spoliehajú na komunikáciu so vzdialeným serverom, sú však prakticky použiteľné, pretože algoritmus je možné konfigurovať tak, aby komunikácia prebiehala len počas fáz, keď program nerobí výpočty náročné na výkon. Vďaka tomuto postupu sme získali dáta, ktoré pozná len obfuskátor a preto je náš postup bezpečnejší, ako ostatné. Ani o našich postupoch sa nedá s určitosťou povedať, že neexistuje deobfuskačná technika, ktorá by ich odstránila, pretože zahmlenosť programov je len relatívny pojem. To je aj dôvod, prečo obfuskačné práce, túto nevynímajúc, neobsahujú presné a dokázané tvrdenia. Veľký význam získavajú práve experimenty, ktoré neboli cieľom tejto diplomovej práce. Preto by prirodzeným pokračovaním práce bola implementácia navrhovaného algoritmu pre konkrétny programovací jazyk. Práca je písaná všeobecne tak, aby bola implementovateľná pre všetky jazyky, jej praktickú použiteľnosť zdôrazňujeme tým, že sme si zvolili konkrétny programovací jazyk - Java, v ktorom sme uvádzali príklady.

Literatúra

[1] H. AGRAWAL and J. R. HORGAN. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, volume 25, pages 246–256, White Plains, NY, June 1990.

[2] THOMAS BALL. The concept of dynamic analysis. In *ESEC / SIGSOFT FSE*, pages 216–234, 1999.

[3] MICHAEL BATCHELDER and LAURIE HENDREN. Obfuscating Java: the most pain for the least gain. In *16th International Conference on Compiler Construction*, Braga, Portugual, March 2007.

[4] LARRY CARTER, JEANNE FERRANTE, and CLARK THOMBORSON. Folklore confirmed: reducible flow graphs are exponentially larger. In *POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 106–114, New York, NY, USA, 2003. ACM Press.

[5] MIHAI CHRISTODORESCU, JOHANNES KINDER, SOMESH JHA, STEFAN KATZENBEISSER, and HELMUT VEITH. Malware normalization. Technical Report 1539, University of Wisconsin, Madison, Wisconsin, USA, November 2005.

[6] CHRISTIAN COLLBERG, CLARK THOMBORSON, and DOUGLAS LOW. A taxonomy of obfuscating transformations. Technical Report 148, July 1997. http://www.cs.auckland.ac.nz/collberg/Research/Publications/CollbergThomborsonLow97a/index.html.

[7] CHRISTIAN COLLBERG, CLARK THOMBORSON, and DOUGLAS LOW. Manufacturing cheap, resilient, and stealthy opaque constructs. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 184–196, New York, NY, USA, 1998. ACM Press.

[8] BRADLEY DUX, ANAND IYER, SAUMYA DEBRAY, DAVID FORRESTER, and STEPHEN KOBOUROV. Visualizing the behavior of dynamically modifiable code. In *IWPC '05: Proceedings of the 13th International Workshop on Program Comprehension*, pages 337–340, Washington, DC, USA, 2005. IEEE Computer Society.

[9] DESCLAUX FABRICE. Skype uncovered, November 2005. www.ossir.org/ windows/supports/2005/2005-11-07/EADS-CCR_Fabrice_Skype.pdf.

[10] JUN GE, SOMA CHAUDHURI, and AKHILESH TYAGI. Control flow based obfuscation. In *DRM '05: Proceedings of the 5th ACM workshop on Digital rights management*, pages 83–92, New York, NY, USA, 2005. ACM Press.

[11] NEIL D. JONES. An introduction to partial evaluation. *ACM Comput. Surv.*, 28(3):480–503, 1996.

[12] YUICHIRO KANZAKI. *Protecting Secret Information in Software Processes and Products*. PhD thesis, Nara Institute of Science and Technology, February 2006.

[13] YUICHIRO KANZAKI, AKITO MONDEN, MASAHIDE NAKAMURA, and KENICHI MATSUMOTO. Exploiting self-modification mechanism for program protection. In *Proc. of the 27th Annual International Computer Software and Applications Conference*, 2003.

[14] CULLEN LINN and SAUMYA DEBRAY. Obfuscation of executable code to improve resistance to static disassembly. In *CCS '03: Proceedings of the 10th ACM conference on Computer and communications security*, pages 290–299, New York, NY, USA, 2003. ACM Press.

[15] DOUGLAS LOW. Java control flow obfuscation, 1998. www.cs.arizona.edu/ collberg/Research/Students/DouglasLow/thesis.ps.

[16] M. MADOU, B. ANCKAERT, P. MOSELEY, S. DEBRAY, B. DE SUTTER, and K. DE BOSSCHERE. Software protection through dynamic code mutation., 2005. www.cs.arizona.edu/solar/papers/wisa05.ps.

[17] MATIAS MADOU, BERTRAND ANCKAERT, BJORN DE SUTTER, and KOEN DE BOSSCHERE. Hybrid static-dynamic attacks against software protection mechanisms. In *DRM '05: Proceedings of the 5th ACM workshop on Digital rights management*, pages 75–82, New York, NY, USA, 2005. ACM Press.

[18] MATIAS MADOU, LUDO VAN PUT, and KOEN DE BOSSCHERE. Loco: an interactive code (de)obfuscation tool. In *PEPM*, pages 140–144, 2006.

[19] ANIRBAN MAJUMDAR and CLARK THOMBORSON. Manufacturing opaque predicates in distributed systems for code obfuscation. In Vladimir Estivill-Castro and Gillian Dobbie, editors, *Twenty-Ninth Australasian Computer Science Conference (ACSC 2006)*, volume 48 of *CRPIT*, pages 187–196, Hobart, Australia, 2006. ACS.

[20] AKITO MONDEN, ANTOINE MONSIFROT, and CLARK THOMBORSON. A framework for obfuscated interpretation. In *ACSW Frontiers*, pages 7–16, 2004.

[21] MANGALA GOWRI NANDA and S. RAMESH. Slicing concurrent programs. In *International Symposium on Software Testing and Analysis*, pages 180–190, 2000.

[22] MILA DALLA PREDA and ROBERTO GIACOBAZZI. Control code obfuscation by abstract interpretation. In *SEFM*, pages 301–310, 2005.

[23] MILA DALLA PREDA, MATIAS MADOU, KOEN DE BOSSCHERE, and ROBERTO GIACOBAZZI. Opaque predicates detection by abstract interpretation. In *AMAST*, pages 81–95, 2006.

[24] ARJAN DE ROO and LEON VAN DEN OORD. Stealthy obfuscation techniques: misleading the pirates, February 2003.

[25] MIKHAIL SOSONKIN, GLEB NAUMOVICH, and NASIR MEMON. Obfuscation of design intent in object-oriented applications. In *DRM '03: Proceedings of the 3rd ACM workshop on Digital rights management*, pages 142–153, New York, NY, USA, 2003. ACM Press.

[26] ELENI STROULIA and TARJA SYSTA. Dynamic analysis for reverse engineering and program understanding. *SIGAPP Appl. Comput. Rev.*, 10(1):8–17, 2002.

[27] FRANK TIP. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995.

[28] JÁN ŠTURC. Prednáška z kompilátorov - analýza toku dát, February 2006.

[29] SHARATH K. UDUPA, SAUMYA K. DEBRAY, and MATIAS MADOU. Deobfuscation: Reverse engineering obfuscated code. In *WCRE '05: Proceedings of the 12th*

Working Conference on Reverse Engineering, pages 45–54, Washington, DC, USA, 2005. IEEE Computer Society.

[30] CHENXI WANG. *A security architecture for survivability mechanisms*. PhD thesis, 2001. Adviser-John Knight.

[31] CHENXI WANG, JONATHAN HILL, JOHN C. KNIGHT, and JACK W. DAVIDSON. Protection of software-based survivability mechanisms. In *DSN '01: Proceedings of the 2001 International Conference on Dependable Systems and Networks (formerly: FTCS)*, pages 193–202, Washington, DC, USA, 2001. IEEE Computer Society.

[32] XIANGYU ZHANG and RAJIV GUPTA. Cost effective dynamic program slicing. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 94–106, New York, NY, USA, 2004. ACM Press.