Katedra informatiky
Přírodovědecká fakulta
Univerzita Palackého v Olomouci

BAKALÁŘSKÁ PRÁCE

Obfuskace zdrojového kódu

2022

Jiří Jinda

Vedoucí práce:
Mgr. Markéta Trnečková, Ph.D.

Studijní program: Aplikovaná informatika,
prezenční forma

Bibliografické údaje

Autor:	Jiří Jinda
Název práce:	Obfuskace zdrojového kódu
Typ práce:	bakalářská práce
Pracoviště:	Katedra informatiky, Přírodovědecká fakulta, Univerzita Palackého v Olomouci
Rok obhajoby:	2022
Studijní program:	Aplikovaná informatika, prezenční forma
Vedoucí práce:	Mgr. Markéta Trnečková, Ph.D.
Počet stran:	44
Přílohy:	1 flash disk
Jazyk práce:	český

Bibliographic info

Author:	Jiří Jinda
Title:	Source Code Obfuscation
Thesis type:	bachelor thesis
Department:	Department of Computer Science, Faculty of Science, Palacký University Olomouc
Year of defense:	2022
Study program:	Applied Computer Science, full-time form
Supervisor:	Mgr. Markéta Trnečková, Ph.D.
Page count:	44
Supplements:	1 flash drive
Thesis language:	Czech

Anotace

Obfuskace zdrojového kódu je způsob šifrování, při kterém kód zůstává spustitelný počítačem, ale je nečitelný pro člověka. Tato práce uvede základní taxonomii a následně i konkrétní způsoby obfuskace. Některé z nich budou implemenovány v jazyce C++.

Synopsis

Source code obfuscation is a specific way of ciphering, where the code remains executable but is unreadable by humans. This thesis introduces basic taxonomy and concrete methods of obfuscation. Some of them will be implemented in C++ language.

Klíčová slova: obfuskace, transformace, šifrování, taxonomie

Keywords: obfuscation, transformations, ciphering, taxonomy

Děkuji Mgr. Markétě Trnečkové, Ph.D. za ochotu vést tuto práci a za pomoc při jejím zpracování.

Místopřísežně prohlašuji, že jsem celou práci včetně příloh vypracoval samostatně a za použití pouze zdrojů citovaných v textu práce a uvedených v seznamu literatury.

datum odevzdání práce podpis autora

Obsah

Seznam obrázků

Seznam tabulek

Seznam zdrojových kódů

1 Úvod

V roce 1982 formalizovala D. Denning pojem data security [1]. Když potom v roce 1988 student Cornellovy univerzity vypustil na internet svého "Morissova červa", veřejnost a odborníci začali brát problematiku zabezpečení počítačů a sítí skutečně vážně. O několik desetiletí později je zabezpečení dat nutností a naprostou samozřejmostí.

Postupem času a s masivním rozšířením osobních počítačů začala být zjevná potřeba distribuovat software koncovým uživatelům, což je dnes již běžný obchodní model. Uživatel má tak plný přístup k softwaru a používá svoji výpočetní sílu, ale tento model s sebou také nese značná bezpečnostní rizika. Uživatel se může pokusit mj. měnit program tak, aby dělal něco jiného, než k čemu byl vytvořen: dělat nelegální kopie celého softwaru a ty sám distribuovat nebo ukrást duševní vlastnictví, které software obsahuje. K redukci takových rizik se používá mnoho softwarových, hardwarových a jiných technik. Jednou z nich je právě obfuskace zdrojového kódu.

Obfuskace je logickým vyústěním problému, jak ochránit data, nad kterými ztratíme kontrolu, ale zároveň zajistit, aby byla stále použitelná. Cílem bude aplikovat transformace na data tak, aby jejich podstata zůstala nezměněna, ale byla pro koncového uživatele nečitelná. K tomu budeme používat kombinaci mnoha základních úprav, kde každá úprava sama o sobě cílí na určitou část struktury kódu, ale samotná není příliš účinná. Pro nejlepší efekt budeme transformace skládat, přidávat nové za běhu programu i po přeložení kompilátorem do strojového kódu.

Ačkoli je obfuskace primárně vyvíjena k ochraně softwaru, může být také použita k ukrytí malwaru. Protože antivirové ochrany odhalují malware primárně statickou analýzou dat, je přirozeně obfuskace svojí funkcionalitou ideální ke skrytí škodlivého kódu.

Na internetu jsou dnes lehce k dohledání obfuskátory libovolného jazyka, které lze s různou úspěšností použít. Pracují pouze se statickým kódem, který znečitelní. Jeden z takových obfuskátorů je například Tigress pro jazyk C.

2 Kontext obfuskace

V této kapitole definujeme pojmy a kontext, které budeme používat ve zbytku práce. Vymezíme zaměření této práce v rámci dalších kategorií, které se v kryptografii používají. Zmíníme se také o dalších možnostech, které lze použít k dosažení stejných cílů, a krátce zhodnotí možný konečný stav obfuskace.

2.1 Základní pojmy

Vymezení pojmů obfuskace, transformace a obfuskátoru a jejich vztah mezi nimi.

Transformace
Transformace je specifická úzká sada pravidel, která změní data, na která se aplikuje. Každá transformace, kterou budeme na programu P provádět, bude mít tu vlastnost, že sníží čitelnost programu. V jaké míře se tak stane, závisí na použité transformaci. Taktéž s provedenou transformací očekáváme, že se zvýší režie programu. Zvýšení může být často zanedbatelné a jednorázové, jindy může být kontinuální a znatelné. Čím lépe transformace program znečitelní a čím menší je její cena, tím je pro nás výhodnější ji použít. Metriky kvality transformací popíšeme v kapitole 4.

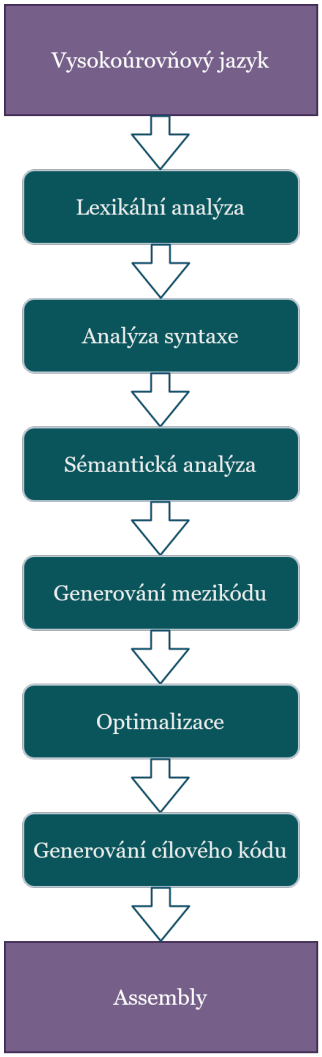
Obfuskace
Obfuskace je proces aplikování více transformací na danou část dat. Ze vstupního programu P vytvoříme nový program P' tak, že bude zachována veškerá funkcionalita P, ale P' bude mít nové vlastnosti. Pokud tedy program P vypočítá výsledek pro vstup, musí P' skončit se stejným výsledkem. Pokud P neskončí, není výsledek P' definován. Nové vlastnosti budou odpovídat součtu vlastností použitých transformací.

Obfuskátor
Obfuskátor je program, který realizuje proces obfuskace. Oproti teoretickému procesu aplikování transformací má ale navíc také praktickou funkcionalitu. Může implementovat heuristiku pro určení, jaké transformace budou pro daný kód ideální. Obfuskátor musí kód zpracovat a rozdělit na jeho jednotlivé syntaktické části, aby mohl aplikovat transformace určené např. pouze pro proměnné, podobně jako front-end kompilátoru.

2.2 Kompilátor

Mnoho transformací se nějakým způsobem odkazuje na změny, které se projeví v grafu kontroly toku dat, nebo používá techniky optimalizace, stejně jako kompilátor. Proto zde popíšeme jeho základní funkci a stavbu. Obecně se kompilace skládá ze tří částí [2, 3]:

Obrázek 1: Kompilace



1. Nahrazení (preprocessing) zástupných symbolů a maker jejich hodnotou. Následuje analýza kódu po lexikální, sémantické a syntaktické stránce. Kontroluje se, zda sekvence znaků udává rozpoznatelné instrukce, zda lze sestrojit parsovací strom, zda jsou instrukce platné, typování a podobně.

2. Sestrojení grafu kontroly toku (control-flow graph = CFG) dat a překlad do mezikódu. Graf reprezentuje možné cesty toku programu během spuštění. To umožňuje optimalizace kódu, jako třeba vypuštění uzlů, do kterých nevedou žádné hrany, nebo technika *constant folding*, která opakovaně prochází kód a nahrazuje proměnné hodnotami nebo kombinací již použitých proměnných. Vzniklý mezikód by měl být lehce přeložitelný do strojového kódu a již optimalizovaný.

3. Na základě specifické architektury procesoru se generuje konečný strojový kód, který lze spustit.

Častou transformací obfuskace bude například přidání kódu, který nemá žádnou funkci. Nicméně takový kód bude kompilátorem odstraněn během překladu. Proto bychom měli zajistit, aby ve vzniklém grafu existovala hrana mezi naším kódem a zbytkem programu.

2.3 Bezpečnostní modely

Kerckhoffsův princip říká, že systém by měl být bezpečný, i když je o něm známo vše, s výjimkou klíče [4]. Je to protiklad principu bezpečnosti skrze utajení. Mnohokrát se v historii ukázalo, že bezpečnost skrze utajení je většinou nevhodný postup, protože tajemství fungování takových bezpečnostních prvků se nikdy neudrží příliš dlouho. Dnes máme pro tyto principy dobře definované termíny, které popíšeme v kontextu kryptografie. [5, 6, 7]

Black-box
Model předpokládá, že útočník pozoruje pouze externí chování programu, tedy pro jaký vstup vydá jaký výstup. Nijak se nezaobírá vnitřním designem, strukturou, fungováním programu, ani nebere v potaz fyzické chování hardwaru. Předpokládá se, že program běží v důvěryhodném prostředí.

White-box
Model předpokládá přístup útočníka ke všem datům a algoritmům. Stejně jako v black-box modelu zná vstupy a výstupy, kromě toho ale i mezivýpočty a stav paměti v průběhu. Očekává se použití dalšího softwaru pro debugování nebo emulaci prostředí. Tento model předpokládá běh programu v prostředí plně ovládaném útočníkem.

Gray-box
Tento model je mezikrokem mezi black-box a white-box modelem. K vlastnostem black-box modelu přidává také techniky útoků postranním kanálem, tedy např. spotřebu energie, délku výpočtu, detekci elektromagnetického vlnění, akustiku nebo visuální informace.

2.4 Prostředí útoku

Alternativním způsobem k bezpečnostním modelům je klasifikace útoků popisem prostředí. V podstatě sdružuje výše popsané specifikace do tří kategorií: míra oprávnění, lokace a prostředí. [8, 9]

Míra oprávnění
Odpovídá systémovému oprávnění útočníka pro čtení, úpravu a spouštění souborů.

Lokace
Popisuje přístup útočníka k softwaru. Rozlišuje, kde je software fakticky spouštěn a zda s ním útočník pracuje přímo, nebo přes další rozhraní.

Důvěryhodný host
Narozdíl od lokace popisuje, zda útočník kontroluje přímo zařízení a prostředí, ve kterém software běží.

Obě dělení jsou navzájem porovnatelná, vztahy mezi nimi jsou popsány v tabulce 1.

Tabulka 1: Srovnání bezpečnostních modelů a prostředí útoku

Prostředí útoku	Síť	Zevnitř	Host
odpovídá modelu:	*black-box*	*gray-box*	*white-box*
Míra oprávnění	žádná	částečná	úplná
Lokace	vzdálená	lokální	host
Důvěryhodný host	ano	ano	ne

2.5 Možnosti obfuskace

Ačkoli je ideální cíl obfuskace definitivně znemožnit útočníkům rekonstrukci programu, bylo ukázáno, že to nikdy nebude zcela možné. Skutečným efektem bude vždy pouze oddálení rekonstrukce a přečtení programu. Uvažujme následující předpoklady:

1. Algoritmus obfuskace je znám (odpovídá i white-box modelu)
2. Výsledná obfuskace je generována za použití klíče K
3. Délka K vzhledem k délce programu je polynomiální
4. Časová složitost algoritmu obfuskace je polynomiální

Vzhledem k faktu, že známe jak výsledky programu P, tak i funkci obfuskace F, jedinou neznámou je klíč K. Lehce ukážeme, že když jsou veškeré dílčí problémy řešitelné v polynomiálním čase, tak potom problém deobfuskace leží ve třídě NP-jednoduchých problémů. Ačkoli je prakticky stále obtížně řešitelný, ke kvalitám jednosměrného šifrování má daleko. [11]

Barak et al. [10] ve svém výzkumu došel k zajímavým závěrům, například, že žádný obfuskátor neprovede tzv. one-way (jednosměrnou) obfuskaci. To znamená, že každý takový proces je reversibilní. Zde ale musíme doplnit, že existují i jednosměrné transformace. Takové transformace nicméně nemají efekt na správnou funkcionalitu programu, stejně jako všechny ostatní. Pokud je transformace nevratná, většinou se jedná o smazání nějaké části programu nebo o nahrazení částí jinou, funkčně ekvivalentní. Například smazání komentářů v kódu nelze vrátit zpátky, a ačkoli to útočníkovi ztíží pochopení, zabránit mu v tom nemůže. Proto, když tvrdíme, že neexistuje jednosměrná obfuskace, myslíme tím sadu transformací, nikoli jednu samotnou.

Existuje dokonce skupina programů označovaná jako *learneable*, které svým během prozrazují své vnitřní fungování (opakem jsou potom *non-learnable* programy). Na nich potom obfuskace selže úplně (představme si program, který má vytisknout svůj vlastní kód). Naopak existují transformace, které budou úspěšně fungovat na jakýkoli kód (opět například smazání komentářů). V možnostech obfuskace tedy není zašifrovat daný kód absolutně, pouze se snažíme zvýšit cenu deobfuskace na dostatečně vysokou úroveň, aby se útočníkům nevyplatilo se o ni snažit, nebo aby snaha překročila přínos výsledku. [10]

2.6 Další způsoby zabezpečení

Dalším způsobem zabezpečení je spuštění programu za asistence serveru. Na serveru potom běží ta část programu, kterou chceme chránit před potenciálním útočníkem. Znatelně tak eliminujeme rizika spojená s poskytnutím softwaru uživateli. To ale samozřejmě znamená, že bez přístupu k serveru nebo bez dostatku výpočetní síly je program nefunkční. I když je spojení se serverem navázáno bez problému, program bude nutností komunikace zpomalen. Dalším způsobem je celý program zašifrovat. Program je potom sám o sobě nefunkční a bezcenný. Server poskytne klíč nebo verifikaci programu digitálním podpisem. [12]

Pro úplnost zmíníme ještě existenci hardwarové obfuskace. Mezi dva hlavní způsoby patří pasivní a funkční obfuskace. Pasivní obfuskace má za cíl změnit to, jak hardwarová architektura vypadá navenek, a skrýt tedy účel obvodu. Jiným typem je funkční obfuskace, která vyžaduje ke svému běhu klíč, často užívající AES (Advanced Encryption Standard).

3 Analýza

Když víme kontext obfuskace a její důvod, musíme také znát způsob, jakým se budou ze softwaru informace získávat. To se nazývá *reverzní inženýrství*. To je proces, při kterém se útočník snaží zjistit interní fungování programu analýzou jeho chování. Konkrétní techniky rozlišují, zda program pozorujeme za jeho běhu nebo ne. Obfuskací se budeme snažit tyto procesy narušit nebo znemožnit. Chceme tedy alespoň zhruba porozumět tomu, jaké zdroje využívají, a na ty potom cílit během šifrování kódu.

Často je původní motivace analýzy následná úprava programu nebo zneužití programu ve svůj prospěch. Pokud při analýze zjistíme slabá místa programu, která se dají zneužít, nebo pochopíme, jak algoritmus funguje, můžeme ho poté upravit pro vlastní potřeby.

3.1 Statická analýza

Když analyzujeme program staticky, nedochází k jeho spuštění. S kódem se nejprve provedou automatizované procesy jako překlad ze spustitelného strojového kódu do assembly kódu. Ten se následně dekompiluje a vytváří se graf toku dat. Kód se rozděluje na menší logické celky a vytváří se kompaktní rutiny (funkce). Takto zpracovaná data už mají určitou výpovědní hodnotu. Některé programy také dokáží namapovat tyto nízkoúrovňové instrukce na funkce, procedury a datové struktury, které známe z vyšších jazyků. Pak už záleží na lidském faktoru, zda dokáže takové nízko nebo vysokoúrovňové instrukce interpretovat a odvodit fungování programu. [13, 14, 15]

3.2 Dynamická analýza

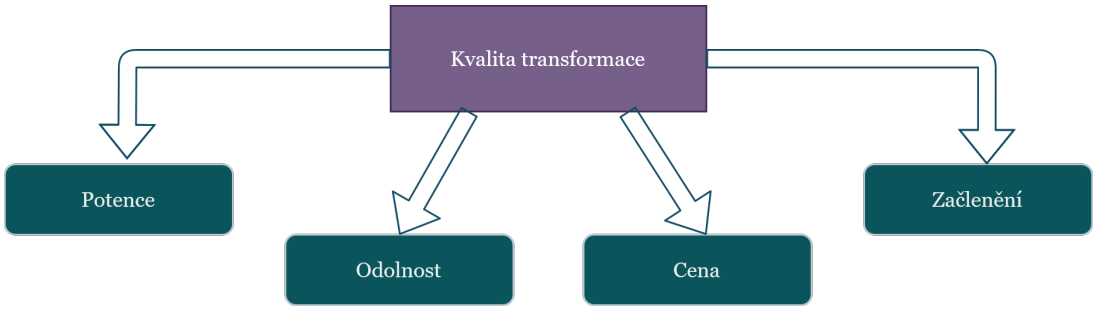
Když zkoumáme program dynamicky, díváme se na jeho průběh během spuštění. Výsledek dynamické analýzy je méně komplexní, protože jsme při ní ovlivnění prostředím, ve kterém se program spouští, vstupem, který programu předáme, nebo dostupnými zdroji. Obvykle se tedy provádí na dobře zvoleném vzorku vstupů. Při analýze se zaměřujeme na pořadí instrukcí, ve kterém se provádějí, na hodnoty v paměti v různých fázích běhu, nebo na změny v samotném kódu. [16, 37]

3.3 Hybridní analýza

Tento přístup je kombinací předchozích dvou a snaží se minimalizovat nedostatky. Úspěch statické analýzy je závislý na míře šifrování nebo virtualizaci, dynamická analýza je naopak často neúplná nebo časově náročná. Na data získaná z dynamické analýzy se používají statické metody dekompilace nebo se z kódu získaného ze statické analýzy vytváří graf toku dat. [16]

4 Kvalita transformací

V této kapitole popíšeme metriky, kterými budeme hodnotit kvalitu transformací. Podle Collberg at al. [17] definujeme čtyři základní faktory, viz obrázek 2. Pro výpočet kvality použijeme jejich kombinaci. Vzhledem k velké závislosti na lidském faktoru je obtížné definovat robustní a rigorózní systém. Ačkoli níže popsaný sýstém vznikl již v roce 1997, stále je považován za výchozí.



Obrázek 2: Přehled metrik kvality transformace

4.1 Potence

Metrikou potence budeme měřit, jak moc daná transformace sníží čitelnost programu. Takové kritérium je ovšem nutně závislé na schopnostech a znalostech čtenáře, proto bude jakákoli exaktní definice nepřesná. Budeme vycházet z předpokladu, že pokud jsou programy P a P' stejné s rozdílem, že P' má nějakou další

vlastnost navíc, je P' komplexnější program. V softwarovém inženýrství jsou definována kritéria, která by tvůrci programů měli minimalizovat, aby byl kód čitelnější. Obfuskace se bude snažit tato kritéria maximalizovat. [17, 18, 19, 20, 21, 15]

Mezi taková kritéria řadíme:

1. **Délka programu** - počet operátorů, funkcí a operandů programu.
2. **Cyklomatická složitost** - počet predikátů v programu.
3. **Úroveň zanořování** - zanořování predikátů a cyklů do sebe.
4. **Složitost toku dat** - počet referencí na data v programu (velikost grafu toku dat).
5. **Množství parametrů** - počet referenčních proměnných ve funkcích a v globálním prostředí, parametrů a hodnot, které je nutné číst nebo aktualizovat.
6. **Složitost datové struktury** - složitost implementace datové abstrakce, množství polí, referencí, dimenzí apod.
7. **Složitost tříd a objektů** - velikost stromu dědičnosti, počet potomků rodiče, metod objektů; logická spojitost tříd.

Z uvedených bodů můžeme vyvodit, že transformace budou často zvětšovat velikost původního programu, přidávat nové třídy, struktury, funkce nebo predikáty. Transformace s vysokou potencí by měla splňovat více z výše uvedených bodů.

Definice 1 (Potence [17])

Nechť P je program a P' je transformovaný program s funkcionalitou P. Dále E(P) je komplexita programu P. Výslednou potenci potom dostaneme následujícím výpočtem:

$$T_{pot}(P) \stackrel{\text{def}}{=} E(P')/E(P) - 1$$

4.2 Odolnost

Abychom za kvalitní transformaci nepovažovali třeba přidání triviálních predikátů, definujeme i odolnost transformace, která udává, jak těžké je ji odčinit. Např. podmínka: *if (sizeof(char) != 1)* bude při kompilaci nahrazena hodnotou false a daná větev bude z kódu úplně vypuštěna. [15, 17, 19]

Míru odolnosti budeme brát jako kombinaci dvou kategorií:

- náročnost sestrojení deobfuskátoru,
- náročnost deobfuskace.

První z uvedených udává, jak dlouho lidskému útočníkovi potrvá sestrojit deobfuskátor, který bude schopen transformace částečně nebo úplně zvrátit. Zde vstupuje do úvahy lidský faktor: znalosti, vědomosti a zkušenosti daného útočníka. Nicméně můžeme vycházet z předpokladu, že je lehčí sestrojit algoritmus, který má za úkol zvrátit transformaci ovlivňující malou část kódu, než na transformaci aplikovanou globálně. Z tohoto předpokladu odvodíme následující stupně odolnosti:

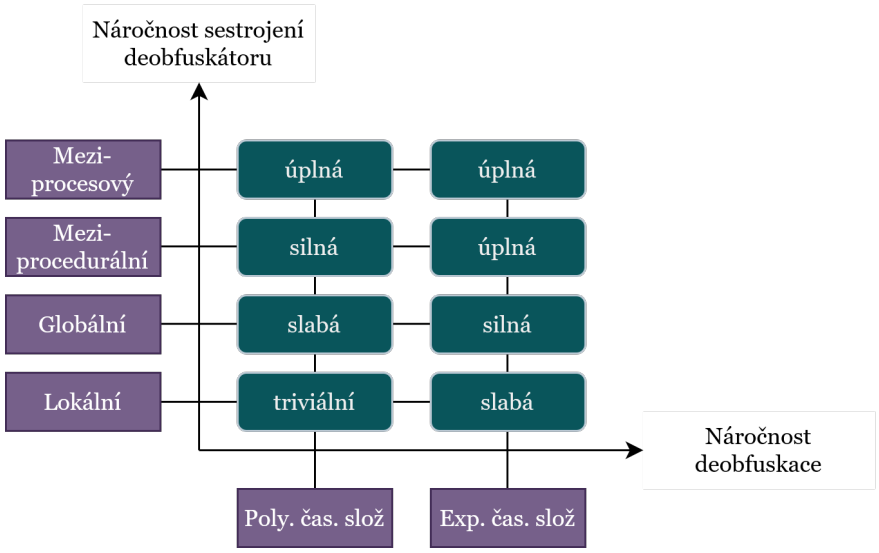
1. **Lokální** - jeden blok (uzel) grafu toku dat,
2. **Globální** - graf toku dat celé procedury,
3. **Meziprocedurální** - tok dat mezi jednotlivými procedurami,
4. **Meziprocesový** - interakce mezi samotnými vlákny procesů.

Druhá z uvedených udává, jak dlouho takto sestavenému deobfuskátoru bude trvat, než zpětné transformace provede. Zde už se můžeme držet teorie časové složitosti algoritmů a Landauovy notace. Barak et al. ve své taxonomii vyděluje dvě základní třídy a to:

- polynomiální časovou složitost řešení,
- exponenciální časovou složitost řešení.

Nyní již můžeme definovat matici, která bude hodnotit odolnost transformací na škále pěti hodnot:

Obrázek 3: Odolnost transformace



Obrázek 4: Graf vztahu mezi druhy náročností deobfuskátoru a programátora a stupni odolnosti

Na každou stranu spektra v obrázku 4 bychom mohli přidat další časové složitosti, ale pokud definujeme, že transformace s lokálním rozsahem a polynomiální časovou složitostí řešení je triviální, potom by nemělo smysl uvažovat jinou než meziprocesní transformaci s lineární složitostí. Stejně tak větší složitost než exponenciální bude určitě velmi účinná. [19]

Definice 2 (Odolnost [17])
Nechť je T transformace, která zachová funkcionalitu P a změní P na P'. $T_{od}(P)$ je odolnost T vzhledem k P. $T_{od}(P)$ je jednosměrná, pokud je informace z P smazána a nelze zpětně získat z P'. Jinak platí:

$$T_{od} \stackrel{\text{def}}{=} Odolnost(T_{sestrojení}, T_{deobfuskace})$$

4.3 Cena

Každá transformace bude mít na program i určitý negativní dopad. Ten se projeví buď v prostorové složitosti, časové složitosti nebo velikosti paměti, kterou následně program zabírá. Velikost použité paměti na pevných discích zanedbáme, protože ta většinou není omezující faktor ani při větším nárůstu, a budeme se soustředit na časovou a prostorovou složitost. Cena je také jediný faktor, který se ze všech definovaných snažíme maximálně snížit. Ačkoli si v definici obfuskace neklademe požadavky na její cenu, je zjevné, že pokud nám naroste exponenciálně časová složitost algoritmu, je daná transformace, byť sebelepší, nepoužitelná, kromě velmi specifických případů.

Budeme rozeznávat čtyři základní kategorie, u nichž opět využijeme Landauovy notace, jsou to vzestupně [15, 17, 19]:

1. **Bezplatná** - pokud P' nemá o více než O(1) horší časovou/prostorovou složitost než P,
2. **Levná** - pokud P' nemá o více než O(n) horší časovou/prostorovou složitost než P,
3. **Drahá** - pokud P' nemá o více než $O(n^k)$, k>1 horší časovou/prostorovou složitost než P,
4. **Velmi drahá** - pokud P' nemá více než exponenciálně horší časovou/prostorovou složitost než P.

Přirozeně také záleží na prostředí, ve kterém transformaci aplikujeme. Uvnitř cyklu s n iteracemi se bezplatná transformace změní na levnou. Chceme tedy aplikovat dražší transformaci v méně frekventovaných místech kódu a naopak.

4.4 Začlenění

Některé transformace, které přidávají nové části kódu, sice mohou být velmi obtížně automaticky deobfuskovatelné, ale snadno rozpoznatelné lidským útočníkem. Často se bude jednat o části, které v kontextu zbytku kódu evidentně nedávají žádný smysl. Takový kód je potom snadno rozpoznatelný a snižuje časovou náročnost sestrojení deobfuskátoru. Měli bychom se proto vždy snažit, aby přidaná část co nejlépe splynula se zbytkem. V tom nám pomůže analýza původního programu z hlediska užití syntaxe (operátory, proměnné, funkce apod.). Ne všechny syntaktické prvky se nutně musí v programu vyskytovat ještě před transformací, ale takové prvky potom budou míru začlenění pravděpodobně snižovat. [21, 22]

Pro přesné určení kvality budeme muset analyzovat prvky jazyka v kódu použité. Kvalitu začlenění budeme měřit na stupnici:

Obrázek 5: Míra začlenění

Definice 3 (Začlenění [17])

Nechť T je transformace zachovávající funkcionalitu a Q je program. $P_s(Q)$ je pak množina prvků jazyku použitá v Q a $P_s(T)$ je množina prvků jazyka, kterou přidává T. Začlenění potom dostaneme jako:

$$T_{začlenění}(Q) \overset{\text{def}}{=} \begin{cases} 1.0, pokud |P_s(T) = 0| \\ 1.0 - \frac{|P_s(T) \ P_s(Q)|}{|P_s(T)|}, jinak \end{cases}$$

4.5 Kvalita

Ideální transformace bude bezplatná, snadno začlenitelná a s vysokou odolností a potencí. V praxi ale takovou transformaci budeme hledat těžko. Mezi jednotlivými metrikami existují vztahy. Například transformace s vysokou odolností bude mít horší začlenění, protože útočník si snadněji všimne složitých výpočtů a velkých částí kódu. Některé transformace se budou soustředit právě na odolnost, jiné budou jednodušší, ale s vysokou mírou začlenění. Výpočet kvality definujeme následujícím vzorcem [17]:

$$T_{kvalita}(P) \stackrel{\text{def}}{=} \frac{(T_{potence}(P), T_{odolnost}(P), T_{začlenění}(P))}{T_{cena}(P)}$$

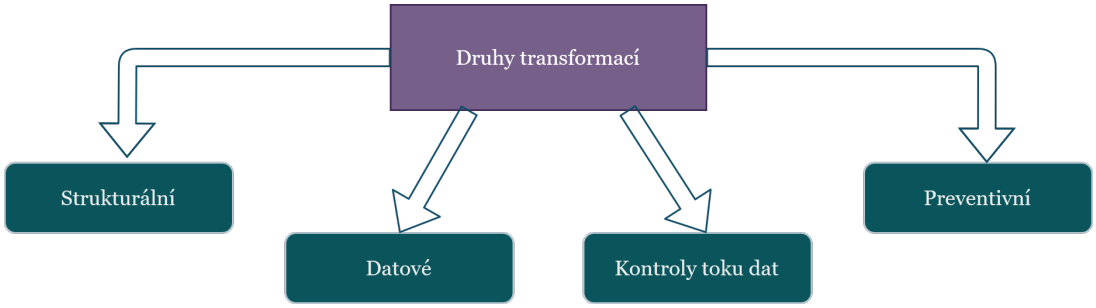
Předtím, než můžeme vzorec uvedený výše prakticky použít, měla by se zvolit určitá heuristika, která stupňům metrik definovaným výše (bezplatná, silná apod.) přiřadí číselnou hodnotu. Tu můžeme definovat podle preference určitých kategorií (například zda v dané situaci preferujeme vyšší odolnost na úkor ostatních vlastností, nebo se snažíme o vyvážené ohodnocení). Hodnoty samotné nejsou příliš informativní, jejich užitek je hlavně v porovnávání transformací mezi sebou. Jeden takový přístup by mohl vypadat následovně:

Tabulka 2: Příklad exaktního ohodnocení stupňů kvality

Metrika	Stupeň				
Potence	nízká	střední	vysoká		
	1	10	100		
Odolnost	triviální	slabá	silná	úplná	jednosměrná
	1	10	100	1000	10000
Cena	bezplatná	levná	drahá	velmi drahá	
	1	10	100	1000	
Začlenění	zjevné	průměrné	začleněné		
	1	10	100		

5 Druhy transformací

Transformace budeme klasifikovat na základě jejich cíle a toho, jak daný cíl ovlivňují. Collberg et al. [17] navrhuje čtyři kategorie: struktura, data, kontrola toku a preventivní transformace. V pozdější publikaci [22] potom pojmy rozšiřuje a zobecňuje: abstrakce, data, kontrola a dynamické transformace. V této práci budeme používat první čtyři zmíněné, protože plně obsáhnou veškeré popsané transformace.



Obrázek 6: Přehled druhů transformací podle cíle

5.1 Strukturální transformace

Transformace spadající do této kategorie ovlivňují lexikální prostředky jazyka, které se nijak netýkají funkcionality kódu (jde pouze o dodatečné informace pro programátora, které jsou v kódu obsaženy). Zahrnuje především změny názvů proměnných, funkcí, struktur nebo jiných entit jazyka, dále také komentáře či informace o ladění. Zde lze použít i sofistikovanější řešení než nahrazení smysluplných jmen náhodnými znaky. Proměnné nebo funkce můžeme přejmenovávat tak, aby jejich název smysl dával, ale byl zavádějící pro pochopení zbytku programu. Většina na internetu volně dostupných obfuskátorů se soustředí primárně na tuto kategorii transformací, protože nevyžaduje žádnou hlubší analýzu kódu a pracuje s programem pouze jako s textem. [9, 18]

5.2 Datové transformace

Datové transformace se aplikují na konstantní data programu. Rozdělují se do tří základních skupin podle způsobu, kterým data mění [16, 23]:

1. **Kódování dat** - Zaměřuje se na interní reprezentaci dat v paměti nebo její interpretaci programem. Můžeme například místo 32 bitového integeru uložit 64 bitové číslo, kde bude původní integer výsledkem součtu prvních a posledních 32 bitů.

2. **Agregace dat** - Je skupina transformací, která rozděluje logicky sdružené datové celky a přerozděluje je tak, aby mezi nimi souvislost neexistovala.

Aplikace agregace na konstantní data mění způsob jejich uložení v softwarové rovině. Můžeme tak spojový seznam nahradit například dvoudimenzionálním polem a přidat k němu další data navíc. Obecně chceme rozbíjet datovou abstrakci na její prvočinitele.

3. **Řazení dat** - V souvislosti s konstantními daty rozumíme řazením změnu systému jejich uložení. Například u pole, kde je n-tý prvek na n-té pozici v poli, můžeme použít funkci f, která bude ukládat prvky na jiné pozice určené danou funkcí.

5.3 Transformace kontroly toku dat

Transformace cílí na změnu grafu kontroly toku dat, který kompilátor vytvoří. [9, 17, 19]

1. **Agregace toku** - V kontextu toku dat můžeme měnit seskupování deklarací, sdružovat nesouvisející výpočty a rozdělovat od sebe související nebo nahradit volání funkce přímo kódem funkce a naopak.

2. **Řazení toku** - Při výpočtech můžeme například rozdělit vícekrokový výpočet a vložit jeho části mezi jiné, nesouvisející. Obecně chceme změnit pořadí, v jakém by se výpočty logicky prováděly. Narozdíl od agregace se soustředíme na pořadí provedení za běhu programu.

3. **Výpočty toku** - Reprezentují transformace, které mění tok dat v programu. Přidávají do kódu nové funkce, vytvářejí funkční duplikáty existujících funkcí a mění volání v programu nebo úplně mění vnitřní algoritmus určitého výpočtu.

5.4 Preventivní transformace

Transformace struktury, dat a toku dat mají za cíl ztížit práci lidskému útočníkovi. Narozdíl od nich je cílem této skupiny transformací ztížit automatizovanou deobfuskaci. Dělíme je do dvou skupin a to [17, 19]:

1. **Obecné** - Jsou transformace s vysokou odolností a nízkou potencí. Obvykle se využívají v kombinaci s ostatními druhy transformací. Často se do kódu vkládají byty nesmyslných dat, aby se ztížilo rozpoznání začátku instrukce nebo deklarace.

2. **Cílené** - Narozdíl od obecných preventivních transformací se zaměřují na specifické slabiny konkrétních deobfuskátorů. Pokud existuje často používaný deobfuskátor pro daný jazyk, můžeme jeho analýzou zjistit, jaké transformace nezvládne identifikovat nebo zvrátit, a ty pak použít.

6 Aplikace transformací

Transformace můžeme na program aplikovat ve třech hlavních časových fázích. Před, během nebo po kompilaci. Každá z těchto možností znesnadňuje jiný druh analýzy. Fungování kompilátoru jsme stručně popsali v kapitole 2. Pro implementaci obfuskace během a po kompilaci jsou transformace závislé na architektuře a instrukční sadě procesoru, místo na programovacím jazyku. [16]

6.1 Před kompilací

V této kategorii bude většina transformací popsaných níže. Většinou se jedná o jednodušší operace a mnoho dostupných obfuskátorů pracuje pouze na této úrovni. Čím vyšší jazyk a čím pokročilejší abstrakci (OOP > funkcionální) obfuskujeme, tím lépe můžeme čitelnost kódu degradovat z původní úrovně. Do této kategorie také patří transpilery, které překládají kód z jednoho programovacího jazyka do stejného nebo jiného. Běžně se můžeme setkat s překladem vyšších jazyků do C nebo JavaScriptu. Můžeme potom obfuskovat i přeložený kód v nižším jazyce.

6.2 Během kompilace

Kompilátory již dnes dělají mnohem více, než pouhý překlad programovacího jazyka na strojový. V průběhu překladu vytvářejí pomocné struktury, mezikód a grafy. To vše může útočníkovi pomoci porozumět programu během dynamické analýzy. Překladač také dělá mnoho optimalizací, se kterými je třeba počítat i v obfuskaci před samotnou kompilací. Musíme tak například vnášet závislosti do kódu, který přidáváme, aby při optimalizaci nebyl odstraněn. Vzhledem k faktu, že kompilátor sám o sobě znatelně mění kód programu, je to ideální způsob, jak vnést do kódu i další transformace, které program při analýze znečitelní.

6.3 Po kompilaci

Stejně jako u zdrojového kódu můžeme používat stejné nebo podobné principy obfuskace u strojového kódu. Měnit pořadí instrukcí, adresaci místa nebo zaměňovat instrukce za složitější, ale ekvivalentní konstrukce.

7 Transformace

V této kapitole uvedeme výčet základních transformací a jejich popis. Jde o informace vybrané napříč mnoha zdroji. [17, 19, 24, 25, 26, 27]

7.1 Strukturální

Strukturální transformace jsou nejčastěji používané a nejlépe zdokumentované. Jsou bezplatné a s poměrně vysokou potencí. Jejich zvláštností je, že jsou všechny (zmíněné níže) jednosměrné, takže nelze získat stav programu před transformací, nicméně nijak neovlivňují samotné fungování algoritmů ani běh programu. Aplikují se na kód před kompilací.

7.1.1 Přejmenování identifikátorů

Nejjednodušší cestou, jak program znečitelnit, je změnit názvy funkcí, struktur a proměnných. Zde se nabízí více možností přístupu v závislosti na stupni začlenění, nebo jejich libovolná kombinace [28]:

- Nahradit názvy náhodnými znaky s náhodnou délkou. Když takto nahradíme uniformně názvy v celém kódu, virtuálně tím zvýšíme i jeho začlenění.
- Nahradit názvy malou skupinou těžko rozlišitelných znaků.
- Nahradit názvy jinými, které dávají smysl, ale jsou zavádějící. V jazycích, které umožňují přetěžování názvů, operátorů apod. můžeme situaci ještě více zkomplikovat. Některé jazyky taktéž rozeznávají mezi názvy metod a datových struktur, takže se ve stejném prostředí mohou jmenovat stejně. Taková metodika ale vyžaduje určitou heuristiku a implementace bude individuální pro každý jazyk.

7.1.2 Změna formátování

Kód, který píše programátor, je nějakým způsobem stylizovaný, zarovnává se řádkováním, tabulátory apod. Změnou formátování takové rozdělení smažeme. Odebereme všechny bílé znaky a kód bude jeden velký blok textu.

7.1.3 Změna lokality

Při psaní kódu je kvůli přehlednosti tendence vytvářet logické celky. Kód, který spolu souvisí je zpravidla poblíž. Přesuneme náhodně všechny celky, které můžeme (některé jazyky nedovolují volání funkcí před jejich deklarací). V této strukturální transformaci přesouváme pouze celé deklarace a části, nepřeskupujeme pořadí samotných výpočtů.

7.1.4 Odebrání komentářů

Transformace, která smaže veškerý text, který je v kódu obsažen a nemá žádný účinek na běh. Je velmi jednoduchá na implementaci, ale v složitějším programu má znatelný dopad na čitelnost. Taktéž smaže debug inforomace, pokud se někde v kódu nachází z předchozí kompilace.

7.2 Datové

Tato skupina transformací je stále poměrně hojně využívána. Nevyžaduje hluboké znalosti jazyka nebo kompilátoru. Všechny následující procesy mají za cíl změnit to, jak se reprezentují a uchovávají data, která program využívá za běhu nebo před spuštěním.

7.2.1 Změna pořadí ve strukturách

Pro zvolený způsob uložení dat ve struktuře (např. v poli uložení podle indexů) definujeme funkce *f1* a *f2*. Funkce *f1* bude obfuskovat uložení dat za běhu programu (nebo před jeho spuštěním, pokud se jedná o statická data). Funkce *f2* bude následně mapovat data zpět na původní pořadí. Obě funkce mohou implementovat zároveň slučování a rozdělování struktur. [24]

7.2.2 Slučování a rozdělování struktur

Datovou strukturu můžeme rozdělit (a zároveň změnit pořadí jejích prvků) na více či méně stejných struktur. Vhodným přístupem je rozdělit jednu strukturu na více menších částí a některé z nich opět sloučit dohromady s jinými, nesouvisejícími daty. Pro mapování prvků definujeme funkce *f1* a *f2*, kde *f1* data obfuskuje a *f2* data rekonstruuje.

7.2.3 Reprezentace struktur

Reprezentace dat je indikátorem jejich vlastností a použití (např. pokud chce programátor reprezentovat 2D prostor může intuitivně použít matice nebo dvoudimenzionální pole). Tomu můžeme zamezit změnou reprezentace dat. U příkladu pole můžeme měnit jeho dimenze (zploštit vícedimenzionální na jedno nebo naopak), namísto fronty můžeme data namapovat do stromů apod. Zde je potřeba brát v potaz vlastnosti daných struktur. Pokud využíváme binární strom pro časté hledání, převod na zásobník bude mít za následek znatelné zvýšení časové složitosti.

7.2.4 Kódování

Místo uložení čísla 100 jako posloupnost 32-bitů 1 a 0 můžeme definovat funkci *f1*, která bude dané číslo před uložením měnit, aby v paměti nebylo čitelné. K tomu použijeme buď ustálené šifrovací algoritmy nebo jednodušší prostředky (např. numerické *f1*(i)= i * c, kde c je konstanta, bitové nebo booleovské operace). Data potom můžeme dešifrovat funkcí *f2* před spuštěním nebo za běhu programu. Na důležité a malé kolekce nebo hodnoty se nabízí použít bezpečnější algoritmy.

7.2.5 Statická data jako funkce

Tato transformace nahrazuje statická data funkcí, která dané hodnoty vytváří až za běhu programu. Samotná funkce potom může být předmětem dalších transfor-

mací kontroly toku dat. Collberg et al. [17] navrhuje využít pro obfuskaci řetězců jednu funkci, která bude na základě jejich argumentů vracet požadovanou hodnotu.

7.2.6 MBA

MBA (Mixed Boolean-Arithemtic) je typ transformace, kde se kombinuje booleova algebra a aritmetika spolu s polynomy, pro které existuje jejich inverze. Výsledkem je kódování proměnné, které je velmi odolné proti deobfuskaci a taktéž optimalizaci kompilátoru. Například pro hodnotu k=0x87654321 a polynom druhého stupně s koeficienty 2^{32}:

$$f(x) = 727318528x^2 + 3506639707x + 6132886(mod 2^{32})$$

Výsledkem f(k) je potom číslo 1704256593. Za běhu programu je zpět převedeno polynomem

$$f(x)^{-1} = 1428291584x^2 + 1257694419x + 4129091678(mod 2^{32})$$

Zhou et al. [29] popisuje, jak takové polynomy vybrat a jak k nim zjistit funkci inverzní.

7.2.7 Rozdělení proměnných

Na konstantní proměnné můžeme aplikovat kombinaci předešlých transformací. Rozdělení jedné proměnné na více, výslednou proměnnou potom jako funkci s argumenty předchozích částí. Booleovské hodnoty jako logické formule využívající více nově vytvořených deklarací. Získání výsledné proměnné se potom může dít jak před spuštěním programu (u složitějších konstruktů) nebo za jeho běhu.

7.2.8 Povýšení proměnných

V mnoha jazycích jsou datové typy v hierarchii, kde základ tvoří obecnější typy a od nich jsou potom odvozené další, specifičtější. V jazyce C++ namísto enumerace, jejíž účel je jednoznačný, můžeme použít běžnou třídu. V Javě jsou potom základní datové typy potomkem třídy Object. Kromě úpravy samotného typu můžeme také měnit její životnost a viditelnost. Změna životnosti proměnné z lokální na globální je zvláště účinná, pokud na ni později v běhu programu vytvoříme závislost v toku dat.

7.2.9 Klonování kódu

Cílem této transformace je vzít část kódu, která je v programu použitá vícekrát a takovou část nakopírovat. Nově vzniklé kopie budou mít nějaké odlišnosti od původní části (jiný vnitřní algoritmus, název apod.), ale budou s ní funkčně ekvivalentní. Při opakovaném volání původní části kódu budeme následně střídavě používat různé kopie nebo originál. Chceme tím vyvolat dojem, že se s daty

dějí různé procedury. Oproti vkládání kódu se všechny kopie aktivně volají a používají.

7.2.10 Vkládání kódu

Jednou z nejrozšířenějších transformací je vkládání nesmyslného kódu do programu. Takový kód je ale mrtvý (v programu není způsob, jak ho spustit). Aby tato transformace obstála i při dynamické analýze, musíme v grafu toku dat vytvořit propojení se zbytkem programu, nebo ho zkombinovat s neprůhlednými predikáty, aby nešlo před spuštěním zjistit, zda je skutečně mrtvý. V opačném případě ho kompilátor během optimalizace smaže. Kód nemusí dělat nic nebo může provádět zbytečné výpočty s daty, která se nepoužijí ke skutečnému výpočtu programu.

7.2.11 Volání knihoven

Volání standardních knihoven nelze obfuskovat a stejně tak veškerá funkcionalita v knihovně obsažená. Abychom mohli transformace v tomto případě použít, můžeme vytvořit vlastní kopie knihoven, které již lze upravovat. Otázka podpory v překladači a distribuce knihovny už je věcí praxe.

7.3 Kontroly toku dat

Transformace toku dat jsou složitější implementaci a komplexnější v jejich aplikaci. Zde uvádíme několik základních. [17, 30, 31, 32, 33]

7.3.1 Neprůhledné predikáty

Obecně se jedná o predikát, jehož pravdivostní hodnota je známa při obfuskaci, ale obtížně se hodnotí během statické analýzy. Různé druhy predikátů podle jejich výsledné hodnoty jsou: vždy pravda, vždy nepravda nebo v závislosti na kontextu. Jejich účelem je ztížit eliminaci vloženého kódu a proměnných. Predikáty s triviální a slabou odolností jsou takové, které lze vyhodnotit pouze lokální statickou analýzou.

Predikáty mohou být založeny na matematických operacích nebo jiných technikách např. Aliasingu, která bude podrobně vysvětlena v 7.3.2. Pokud již v kódu existuje složitý výpočet, jehož výsledek má s určitostí nějakou vlastnost, můžeme tuto vlastnost v predikátu použít. Další možností jsou numerické tautologie platné pro všechny hodnoty, např.

$$(x^2 + x) mod 2 == 0$$

.

Paralelizace vláken je vhodná ke konstrukci velmi odolných neprůhledných predikátů. Kvůli povaze paralelního výpočtu je těžké určit, jakou hodnotu bude mít výraz V prvního vlákna v čase, kdy bude V používat jiné vlákno.

7.3.2 Aliasing

Tato transformace využívá adresaci místa v paměti, která je staticky nerozhodnutelná před spuštěním programu [33]. Zpravidla se vytváří datové struktury, které se za běhu programu mění a mají za cíl mapovat ukazatele v paměti na jiné ukazatele. V různém stádiu běhu programu jsou potom jeden nebo více ukazatelů namapovány na různé proměnné nebo jiné ukazatele.

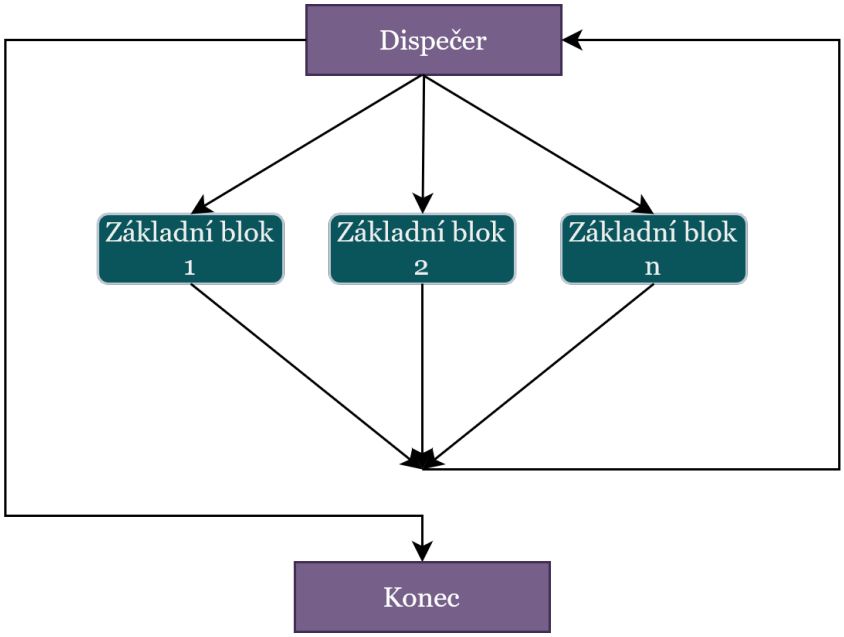
7.3.3 Paralelizace

Jak již bylo zmíněno v kapitole 7.3.1, paralelizace programu ztěžuje jeho pochopení. Je výhodné paralelizaci spojit s dalšími transformacemi a využít souběhu vláken a jejich prokládání. Kromě užití hodnot v predikátech můžeme mezi vlákna rozdělovat funkce a výpočty nebo vytvářet nové nesmyslné. Ačkoli je paralelizace oblíbenou technikou optimalizace programů, je její statická analýza obecně problematická a nespolehlivá.

7.3.4 Zplošťování CFG

Graf kontroly toku dat je strom, který zobrazuje přechody a stavy za běhu programu. Jeho sestavením dostaneme o programu mnoho důležitých informací. Odhalí některé nekonečné cykly, nedosažitelný kód a obecný běh programu. Sestrojuje ho kompilátor pro lepší optimalizaci kódu.

Aby graf kontroly toku dat neposkytl útočníkovi informace o programu, budeme touto transformací graf deformovat. Cappaert [16] navrhuje místo uzlů stromu, které tvoří hlavně podmínky a instrukce přechodu (goto), vytvořit jeden uzel, který bude fungovat jako dispečer. Kořen stromu (dispečer) bude obsahovat všechny podmíněné a jiné přechody a přechod do konečného stavu. Jeho listy budou veškeré ostatní základní bloky, z nichž povede hrana zpět do dispečera. Takto jsme eliminovali informativní hodnotu grafu, ale vytvořili slabý článek transformace, dispečera. Řešením je podle Cappaerta [16] dispečera (obrázek 7) různými způsoby obfuskovat nebo zašifrovat.



Obrázek 7: Příklad zplošťování grafu pomocí dispečera

7.3.5 Úpravy cyklů

Často se cykly deklarují inkrementováním čítače a vyhodnocuje se splnění podmínky. Úpravou deklarace můžeme například čítač snižovat místo zvyšování (a změnit odpovídající ukončovací podmínku), změnit inkrement iterace apod. Při úpravě cyklů je výhodné přidat k ukončovací podmínce další, která zastře přesný čas ukončení. Vhodné jsou pro to například neprůhledné predikáty závislé na kontextu. Další možnosti je replikovat tělo cyklu vícekrát do jedné iterace, abychom ztížili rozpoznání počtu opakování nebo vkládat dovnitř cyklu další cykly, na které aplikujeme další transformace.

7.3.6 Změna instrukcí

Stejně jako v datových a strukturních transformacích můžeme měnit pořadí instrukcí ve funkcích a metodách. Zde je potřeba analyzovat, zda přesunutím nenarušíme daný výpočet. Instrukce můžeme také rozdělovat na více jednodušších nebo naopak slučovat do složitějších. Do této transformace také zahrneme změnu volání. Pokud v jazyce existuje více způsobů volání stejné funkce nebo lze použít zcela jiná, funkčně ekvivalentní, je výhodné volání diverzifikovat.

7.3.7 In-line funkce

Vkládání (in-linování) funkcí do kódu místo jejich volání je dalším způsobem snižování míry abstrakce v kódu. Pokud se daná funkce v kódu volá vícekrát, lze část volání v kódu ponechat a část vložit. Vkládání je i jednou z technik optimalizace, kterou běžně používá kompilátor.

7.3.8 Slučování/rozdělování funkcí

Tato transformace je ekvivalentní co do principu se slučování/rozdělování dat. Ucelené funkce můžeme rozdělovat na menší celky a ty potom slučovat s nesouvisejícími výpočty. Můžeme tak činit lokálně, globálně nebo i mezi vlákny programu. Kromě toho můžeme měnit počet argumentů, které funkce vyžaduje a vracet více hodnot, než je potřeba. Taková data potom můžeme využívat například ve vytváření závislostí mezi jednotlivými částmi kódu nebo k přidání funkčnosti vloženého kódu.

7.3.9 Úprava objektů

Úpravy objektů, dědičnosti apod. lze využít pouze v objektově orientovaných jazycích (např. Java). Vybrané transformace uvedeme souhrnně v tomto bodě.

- Složitost hierarchie tříd roste (a čitelnost se snižuje) spolu s hloubkou stromu dědičnosti. Můžeme namísto jedné třídy definovat více nových, které budou mít díky dědičnosti stejnou funkcionalitu jako by měla původní.
- Stejně jako s daty nebo tokem dat, tak i v objektové abstrakci můžeme vytvářet nové nesmyslné entity. Instance těchto entit potom můžeme vytvářet, mazat a používat ve zbytku kódu.
- Kombinací předchozích dvou a nahrazením referencí na původní objekt referencemi na jeho vytvořeného potomka se vyhneme automatickému procesu faktorizace, která přesouvá společnou funkcionalitu tříd do předka.

7.3.10 Interpretace

Nejkomplexnější způsob obfuskace kódu. Jedná se o vytvoření nového jazyka a jeho překladače. Taková "transformace" je jistě časově náročná na implementaci, ale bude mít i značný dopad na časovou složitost programu. Proto je vhodné takto obfuskovat pouze vybrané části kódu.

7.4 Preventivní

Preventivní transformace cílí na automatické deobfuskátory. Obsahují transformace s nižší potencí a vyšší odolností.

7.4.1 Neprůhledné predikáty s vedlejším efektem

Aby se nedali neprůhledné predikáty během analýzy nahradit jednodušším výrazem, můžeme mu přidat vedlejší efekt, například zvyšování globální proměnné, na které bude závislý další predikát. Zjednodušením nebo odstraněním predikátu potom dojde k chybné funkcionalitě programu.

7.4.2 Závislosti ve vložených datech

Pro ztížení analýzy (především slicingu a optimalizací kompilátoru), je vhodné do nově vytvořeného nebo rozděleného kódu vnést závislosti na zbytku programu. Ty můžeme přidat skrze reference, úpravy proměnných apod. Pro útočníka je obtížnější označit takový kód jako zbytečný a ignorovat jeho deobfuskaci.

8 Přehled a hodnocení

Následuje přehled a hodnocení transformací. Data v tabulce 3 se zakládají na hodnocení, které navrhl Collberg et al. [17]. Transformace, které neuvádí, jsou založeny na vlastní pokusné implementaci. Mnoho metrik je závislých na dané realizaci, na umístění v kontextu kódu nebo na druhu závislostí, které jsou do transformace vložené.

Tabulka 3: Přehled transformací

Druh	Transformace	Potence	Odolnost	Cena
Strukturální	Přejmenování indetifikátorů	střední	jednosměrná	bezplatná
	Změna formátování	nízká	jednosměrná	bezplatná
	Změna lokality	nízká	jednosměrná	bezplatná
	Odebrání komentářů	vysoká	jednosměrná	bezplatná
Datové	Změna pořadí ve strukturách	nízká	jednosměrná	bezplatná
	Slučování/rozdělování struktur	Záleží na počtu dílčích částí		
	Reprezentace struktur	nízká	slabá	bezplatná
	Kódování	Závislé na kódovací funkci		
	Statická data jako funkce	Záleží na složitosti vytvořené funkce		
	MBA	střední	silná	levná
	Rozdělení proměnných	Záleží na druhu predikátu		
	Povýšení proměnných	nízká	silná	bezplatná
	Klonování kódu	střední	Druh záv.	bezplatná
	Vkládání kódu	střední	Druh záv.	bezplatná
	Volání Knihoven	Záleží na úpravě knihoven		bezplatná
Kontrola toku	Neprůhledné predikáty	Záleží na druhu predikátu		
	Aliasing	střední	silná	bezplatná
	Paralelizace	vysoká	silná	drahá
	Zplošťování CFG	vysoká	úplná	drahá
	Úpravy cyklů	nízká	jednosměrná	bezplatná
	Změna instrukcí	nízká	jednosměrná	bezplatná
	In-line funkce	střední	jednosměrná	bezplatná
	Slučování/rozdělování funkcí	nízká	silná	bezplatná
	Úprava objektů	střední	slabá	bezplatná
	Interpretace	vysoká	silná	drahá
Preventivní	Neprůhledné predikáty s vedlejším efektem	střední	slabá	bezplatná
	Závislosti ve vložených datech	Záleží na druhu závislostí		

9 Demonstrace transformací

Součástí práce je demonstrace vybraných transformací v jazyce C++. Ty byly vybrány tak, aby ke své realizaci nevyžadovaly kontext zbytku kódu. Například k ukázce paralelizace bychom potřebovali znát celý kód nějakého programu a ten složitě analyzovat. Také uděláme předpoklady o vstupu, který nám již zpracoval imaginární obfuskátor a budeme uvažovat pouze interní fungování daných transformací. Stejně tak výstup předáme ve zvoleném formátu a předpokládáme, že obfuskátor již zvládne výsledek správně využít.

V této práci budeme uvažovat white-box model, protože ten nejvíce odpovídá reálné situaci, ve které se obfuskace používá. Jejím ekvivalentem je potom Untrusted host model. Teoretický základ popsaný v první části práce nám bude stačit pro jejich implementaci a porovnání.

9.1 Jazyk

C++ je multiparadigmatický programovací jazyk navržený jako nadstavba jazyka C. Z vlastností jazyka C přebírá efektivitu a nízkoúrovňovost a přidává k němu vyšší abstrakci a hlavně systém tříd a podporu objektového programování. Poslední stabilní verze je C++20.

9.2 Knihovny

V kódu se používají pouze standardní C++ knihovny, konkrétně [34]:

- **<string>** - implementace funkcionality řetězců
- **<time.h>** - generování náhodných čísel a inicializace seedu
- **<set>** - implementace funkcionality množin
- **<cmath>** - základní matematické operace
- **<vector>** - implementace funkcionality vektorů

9.3 Implementace

Následuje implementace vybraných transformací, popis jejich vstupů a výstupu a obecná myšlenka fungování.

9.3.1 Přejmenování indetifikátorů

Vytvoří množinu unikátních názvů dané mohutnosti. Na výběr je ze tří přístupů:

1. Lehce zaměnitelné znaky
2. Výběr náhodně zvolených znaků

3. Smysluplné názvy, které simulují často využívané konvence pojmenování

Funkce v prvních dvou případech náhodně zvolí velikost a znaky a vytvoří název. Ve třetím případě spojí obecné názvy konceptů v programování a specifičtější přívlastky. Z nich potom vybere náhodně požadované množství názvů. Tento přístup je omezen na 144 unikátních názvů kvůli počtu deklarovaných slov (12 konceptů a 12 přívlastků).

- **Vstup** - počet požadovaných názvů; volba přístupu
- **Výstup** - množina nových názvů

```
set<string> rename_identifiers(int identifiers_count, int set_option)
{
srand(time(NULL));

string set0[] = { "_", ",", "." };
string set1[] = { "#", "$", "2", "Q", "O", "/", "|", "\\", "7", "1", "4", "3",
 "0", "+", "%", "g", "o" , "p" , "*" , "_" };
string set2_generals[12] = { "Index", "Value", "Name", "Data", "Arg", "Result",
 "Param", "Size", "Length", "Position", "Test", "Item"};
string set2_conretes[12] = {"Output", "Input", "Current", "Previous", "Used",
 "Check", "Right", "Left", "Top", "Bottom", "First", "Second"};
string set2[144];

int counter = 0;
for (int k = 0; k < 12; k++)
        for (int i = 0; i < 12; i++)
                {
                        set2[counter] = set2_generals[i] + set2_conretes[k];
                        counter++;
                }

string* used_set;
int set_len;
switch (set_option)
{
        case 0: used_set = set0; set_len = 3; break;
        case 1: used_set = set1; set_len = 20; break;
        case 2: used_set = set2; break;
        default:
                return rename_identifiers(identifiers_count, rand() % 3);
}

set<string> set_result;
int index = 0;
if (set_option < 2)
{
        int ident_len;
        while (index < identifiers_count)
        {
                string new_ident;
                ident_len = rand() % 10 + 3;

                for (int len = 0; len < ident_len; len++)
                        new_ident += used_set[rand() % set_len];
                pair<set<string>::iterator, bool> ret =
                        set_result.insert(new_ident);
                if (ret.second)
                        index++;
        }
}
```

```
else
{
        while (index < identifiers_count)
        {
                pair<set<string>::iterator, bool> ret =
                        set_result.insert(used_set[rand() % 144]);
                if (ret.second)
                        index++;
        }
}
return set_result;
}
```

Zdrojový kód 1: Přejmenování indentifikátorů

9.3.2 Statická data jako funkce

V této demonstraci vygenerujeme dva různé řetězce na základě dvou různých identifikátorů. Nejdříve vytvoříme tabulku znaků, které funkce podporuje. Pomocí Ascii tabulky naplníme tabulku malými a velkými písmeny bez diakritiky a vybranými znaky (zde si kvůli demonstraci dovolíme znaky deklarovat explicitně namísto podmínek a cyklů). Následně na základě vstupu získáme unikátní 18 místné číslo, které reprezentuje souřadnice znaků v tabulce, a řetězec z nich rekonstruujeme.

- **Vstup** - identifikační číslo řetězce
- **Výstup** - výsledný řetězec

```
string strings_as_function(long long string_id)
{
char alphabet[8][8];
int interpunction[] = { 32, 44, 46, 58, 61, 33, 37, 42, 45, 62, 60, 0 };
int index = 0;

for (int i = 0; i < 8; i++)
        for (int k = 0; k < 8; k++)
        {
                if (index < 26)
                        alphabet[i][k] = index + 65;
                else if (index < 52)
                        alphabet[i][k] = index + 71;
                else
                        alphabet[i][k] = (char)interpunction[index - 52];
                index++;
        }

string temp = to_string(string_id);
string string_id_1, string_id_2;

while (temp.length() < 18)
        temp = "0" + temp;
for (int i = 0; i < 9; i++)
        string_id_1 += temp[i];
```

```
for (int i = 9; i < 18; i++)
        string_id_2 += temp[i];

int component1 = stoi(string_id_1, nullptr, 10) + 372291;
int component2 = stoi(string_id_2, nullptr, 10) - 806385;
temp = to_string(component1) + to_string(component2);

while (temp.length() < 18)
        temp = "0" + temp;

string result;
for (int i = 0; i < 18; i += 2)
        if (alphabet[(int)temp[i] - '0'][(int)temp[i + 1] - '0'] != '\0')
                result += alphabet[(int)temp[i] - '0'][(int)temp[i + 1] - '0'];
        else break;

return result;
}
```

Zdrojový kód 2: Statická data jako funkce

9.3.3 Kódování

V tomto příkladě se budeme snažit změnit interní reprezentaci v paměti. Číslo na vstupu projdeme po číslicích a pro každou číslici zvolíme náhodně odpovídající velké písmeno. To nalezneme tak, že k ascii hodnotě čísla přičteme náhodně 0,1 nebo v možných případech 2 desítky tak, aby výsledné číslo stále reprezentovalo písmeno v Ascii tabulce. To potom uložíme do výsledného řetězce. Pro izolaci číslice na daném indexu používáme pomocnou rekurzivní funkci *get_digit*.

Podle toho, zda chceme vstup zakódovat nebo dekódovat zaměníme vstup za výstup:

- **Vstup** - celé číslo
- **Výstup** - číslo zakódované do řetězce

```
string change_coding(int input_plain)
{
srand(time(NULL));
int index=0;
string res;

while (pow(10, index) < input_plain)
{
        int digit = get_digit(input_plain, index);
        if (digit < 6)
                digit += (10 * (rand() % 3));
        else
                digit += (10 * (rand() % 2));
        res += (char)digit + 65;

        index++;
}
        return res;
}
```

Zdrojový kód 3: Kódování

```
int change_coding(string input_string)
{
        int res = 0;
        for (int k = 0; k < input_string.length(); k++)
        {
                res += pow(10, k) * ((input_string[k] - 65) % 10);
        }
        return res;
}
```

Zdrojový kód 4: Kódování - inverzní

9.3.4 Změna pořadí ve strukturách a rozdělení

Rozdělení složitějších struktur na jednodušší si ukážeme na příkladu vektorů. Rozdělení vektorů je velmi snadné, stačí deklarovat nové a příslušné části původního do nich překopírovat. Zde se omezujeme na rozdělení v polovině a na dva nové, ale dělení lze zvolit i náhodně a počet výsledných vektorů může být vyšší.

Změna pořadí probíhá s pomocí funkcí *mapping* a *inverse_mapping*. Zvolený postup posune začátek vektoru do poloviny a poté začne čísla ob jedno vkládat zpět. Pokud by došel na konec, začně znovu od začátku. Pokud by narazil na plný index a přitom mu zbývaly hodnoty, posune se na další prázdný.

Podle toho, zda chceme vektory rozdělit nebo spojit zaměníme vstup za výstup a naopak:

- **Vstup** - vstupní vektor
- **Výstup** - pár výstupních vektorů

```
vector<int> divide_and_reorder(pair<vector<int>, vector<int>> input_pair)
{
        vector<int> semi_res;
        for (int i = 0; i < input_pair.first.size(); i++)
                semi_res.push_back(input_pair.first.at(i));
        for (int i = 0; i < input_pair.second.size(); i++)
                semi_res.push_back(input_pair.second.at(i));

        vector<int> res(semi_res.size(), 0);
        for (int i = 0; i < res.size(); i++)
                res.at(index_inverse_mapping(i, semi_res.size()))
                                = semi_res.at(i);

        return res;
}
```

Zdrojový kód 5: Změna pořadí ve strukturách a rozdělení

```
pair<vector<int>, vector<int>> divide_and_reorder(vector<int> input_vec)
{
        int in_len = input_vec.size();
        int res_index = in_len / 2;
        vector<int> semi_res(in_len, 0);

        for (int i = 0; i < in_len; i++)
        {
                int index = index_mapping(i, in_len);
                semi_res.at(index) = input_vec.at(i);
        }
        vector<int>res_p1;
        vector<int>res_p2;

        for (int k = 0; k < in_len / 2; k++)
                res_p1.push_back(semi_res.at(k));

        for (int l = in_len / 2; l < in_len; l++)
                res_p2.push_back(semi_res.at(l));

        pair<vector<int>, vector<int>> res = make_pair(res_p1, res_p2);
        return res;
}
```

Zdrojový kód 6: Změna pořadí ve strukturách a rozdělení - inverzní

```
int index_mapping(int in_index, int len)
{
        int res_index = (len / 2 + in_index * 2) % len;
        if (len % 2 == 0 && in_index >= len / 2)
                res_index = (res_index + 1) % len;
        return res_index;
}
```

Zdrojový kód 7: Změna pořadí ve strukturách a rozdělení - mapování

```
int index_inverse_mapping(int in_index, int len)
{
        int res_index = in_index - len / 2;
        if (res_index < 0)
                res_index += len;
        if (res_index % 2 == 0)
                res_index /= 2;
        else
        {
                if (len % 2 != 0 )
                        res_index++;
                res_index /= 2;
                res_index += len / 2;
        }

        return res_index;
}
```

Zdrojový kód 8: Změna pořadí ve strukturách a rozdělení - inverzní mapování

9.4 Použití

Funkce by měly být použity jako součást obfuskátoru. Pro tento účel je v kapitole 9.3 popsáno chování funkcí. Pro samostatné použití jsem vytvořil demonstrační příklady jako součást přiloženého kódu. Soubor ve složce `bin/` po spuštění vypíše do konzole příklady použití a počká na ukončení uživatelem. Soubor ve složce `src/` je projektový adresář generovaný Visual Studiem. Ve zdrojovém kódu lze upravit podle omezení uvedených v `readme.txt/` funkci main tak, aby generovala jiné příklady.

10 Diskuze

Popis transformací je spíše než rigorózní popis definice hlavní myšlenky, podle které by mělo být možno transformaci realizovat. Variace v následné implementaci bude vysoká vzhledem k povaze konkrétního kódu, různým nárokům na určité aspekty běhu softwaru (rychlost, míry zabezpečení, aj.) nebo programovacímu jazyku. Práce takřka nic neříká o samotném obfuskátoru. Ten potom bude muset kód zpracovávat a používat mnohé algoritmy na rozpoznání vhodných transformací pro aplikaci a jejich nedeterministický výběr.

Vlastní implementace v kapitole 9 má demonstrovat některé popsané transformace. Funkcionalita je poměrně úzká, často zaměřená na jeden datový typ. Vnitřní fungování jistě není optimalizované nebo ideální, ale názorně uvádí v praxi teoretický popis. Některé transformace budou vyžadovat velkou znalost pozadí (lexikum a syntax jazyka, fungování kompilátoru) a některé budou fungovat na základě robustních matematických základů. Často tak bude samotné navržení algoritmu jen prvním krokem k jeho aplikaci. Při programování jsem také často narážel na komplikace ve vyhodnocování velkých číselných výrazů, do hry se dostávala nepřesnost výpočtů a omezení architektury.

Nicméně se domnívám, že vlastní implementace má výhody oproti zavedeným a dostupným softwarům. Velkou roli bude hrát, že kromě obecného principu bude vlastní algoritmus zcela neznámý. To má samo o sobě velkou přidanou hodnotu pro bezpečnost.

Závěr

Výsledkem práce je úvod do problematiky obfuskace zdrojového kódu, vymezení základních pojmů a vysvětlení kontextu, taxonomie a souborný přehled transformací, při jejichž zpracování se čerpalo z mnoha vědeckých publikací. Poskytuje výchozí zdroj informací pro další studium pokročilejších způsobů obfuskace. Definice metrik, výčet a přehled transformací, jsou podstatné podklady pro sestrojení vlastního obfuskátoru. V závěru práce jsou vybrané transformace demonstrovány v jazyce C++.

Do budoucna je možné implementovat celý obfuskátor, který bude zpracovávat kód a aplikovat transformace. Dále se nabízí realizovat transformaci interpretace a vytvořit nový překladač. V neposlední řadě je také zajímavá možnost zabývat se tématikou obfuskace z opačného pohledu a to deobfuskace.

Conclusions

The result of this work is an introduction to source code obfuscation, the definition of fundamental concepts and their context, taxonomy and review of transformations, using many different scientific papers. It provides a starting point for further research. Definition of metrics, listing and overview of transformations are essential for constructing our own obfuscator. At the end of the thesis, chosen transformations are demonstrated in C++ language.

For future work, it is possible to implement whole obfuscator, which will be processing code and applying transformations. There is also an opportunity to put into practice the interpretation transformation. Last but not least, it would be intriguing to study topic of obfuscation from the other side, that is, deobfuscation.

A Obsah přiloženého datového média

bin/
: Program BP_IMPLEMENTACE_TRANSFORMACI, spustitelný přímo z média.

doc/
: Obsahuje text práce ve formátu PDF a všechny soubory potřebné pro vygenerování PDF dokumentu textu (v ZIP archivu), tj. zdrojový text textu, vložené obrázky, apod.

src/
: Kompletní zdrojové texty programu BP_IMPLEMENTACE_TRANSFORMACI se všemi potřebnými zdrojovými texty, knihovnami a dalšími soubory potřebnými pro bezproblémové vytvoření spustitelných verzí programu.

readme.txt
: Instrukce pro spuštění programu BP_IMPLEMENTACE_TRANSFORMACI, včetně všech požadavků pro jeho bezproblémový provoz. Obsahuje také množinu vstupů, prot které je implementace navržena.

Literatura

[1] ROBLING DENNING, Dorothy Elizabeth. *Cryptography and data security*. Addison-Wesley Longman Publishing Co., Inc., 1982.

[2] GOOS, Gerhard; WAITE, William M. *Compiler Construction*. New York, NY: Springer, 2013. Monographs in Computer Science.

[3] PAHADE, Prajakta; DAWALE, Mahesh.*Introduction to Compiler and its Phases*. International Research Journal of Engineering and Technology. 2019, p. 1318-1322.

[4] KERCKHOFFS, Auguste. *La cryptography militaire*. Journal des sciences militaires, IX:5–38, Janvier 1883.

[5] WYSEUR, Brecht. *White-Box Cryptography*. Boston. 2011.

[6] CHOW, Stanley; EISEN, Phil; JOHNSON, Harold. *A White-Box DES Implementation for DRM Applications*. 2003, 1-15. ISSN: 0302-9743.

[7] BHATIA, Jasleen. *Comparison of White Box, Black Box and Gray Box Cryptography*. S N Education Society, 2017.

[8] MAIN, Alec; VAN OORSCHOT, Paul C. *Software protection and application security: Understanding the battleground*. International Course on State of the Art and Evolution of Computer Security and Industrial Cryptography, Heverlee, Belgium. 2003, vol 19.

[9] BANESCU, Sebastian; PRETSCHNER, Alexader. *A Tutorial on Software Obfuscation*. 2018, 283-353. ISSN: 0065-2458.

[10] BARAK, Boaz; GOLDREICH, Oden; IMPAGLIAZZO, Russel; RUDICH, Steven; SAHAI, Amit; VADHAN, Salil; YANG, Ke. *On the (Im)possibility of Obfuscating Programs*. 2001, 1-18. ISSN: 0302-9743.

[11] APPEL, Andrew. *Deobfuscation is in NP*. Princeton University, Aug. 2002, vol 21, p. 2.

[12] ANIRBAN SENGUPTA, Dipanjan Roy. *SP Design Protection in CE through Algorithmic Transformation Based Structural Obfuscation*. IEEE Transactions on Consumer Electronics. 2017, vol 63, p. 467-476.

[13] STRIEWE, Michael;GOEDICKE, Michael. *A Review of Static Analysis Approaches for Programming Exercises*. 2014, 100-113. ISSN: 1865-0929.

[14] MØLLER, Anders; SCHWARTZBACH, Michael I. *Static program analysis*. Feb, 2012.

[15] ARVIDSSON, Oskar. *Platform Independent Code Obfuscation*. 2014.

[16] CAPPAERT, Jan; OTHERS. *Code obfuscation techniques for software protection*. Katholieke Universiteit Leuven. 2012, p. 1–112.

[17] COLLBERG, Christian; THOMBORSON, Clark; LOW, Douglas. *A taxonomy of obfuscating transformations*. 1997.

[18] ANCKAERT, Bertrand; MADOU, Matias; DE SUTTER, Bjorn; DE BUS, Bruno; DE BOSSCHERE, Koen; PRENEEL, Bart. *Program obfuscation: a quantitative approach*. 2007.

[19] LOW, Douglas. *Java Control Flow Obfuscation*. University of Auckland. 1998.

[20] FREILING, Felix C.; PROTSENKO, Mykola; ZHUANG, Yan. *An empirical evaluation of software obfuscation techniques applied to Android APKs*. 2014.p. 315–328.

[21] COLLBERG, Christian; NAGRA, Jasvir; WANG, Fei-Yue. *Surreptitious software: Models from biology and history*. 2007.p. 1–21.

[22] KANZAKI, Yuichiro; MONDEN, Akito; COLLBERG, Christian. *Code artificiality: a metric for the code stealth based on an n-gram model*. 2015.p. 31–37.

[23] VITICCHIÉ, Alessio; REGANO, Leonardo; TORCHIANO, Marco; BASILE, Cataldo; CECCATO, Mariano; TONELLA, Paolo; TIELLA, Roberto. *Assessment of source code obfuscation techniques*. 2016.p. 11–20.

[24] SHIRAZI, Matthieu Tofighi. *Analysis of obfuscation transformations on binary code*. 2019.

[25] XU, Hui; ZHOU, Yangfan; MING, Jiang; LYU, Michael. *Layered obfuscation: a taxonomy of software obfuscation techniques for layered security*. Cybersecurity. 2020, vol 3, num. 1, p. 1–18.

[26] XU, Hui; ZHOU, Yangfan; KANG, Yu; LYU, Michael R. *On secure and usable program obfuscation: A survey*. arXiv preprint arXiv:1710.01139. 2017.

[27] SCHRITTWIESER, Sebastian; KATZENBEISSER, Stefan; KINDER, Johannes; MERZDOVNIK, Georg; WEIPPL, Edgar. *Protecting software through obfuscation: Can it keep pace with progress in code analysis?*. ACM Computing Surveys (CSUR). 2016, vol 49, núm. 1, p. 1–37.

[28] CHAN, Jien-Tsai; YANG, Wuu. *Advanced obfuscation techniques for Java bytecode*. Journal of systems and software. 2004, vol 71, num. 1-2, p. 1–10.

[29] ZHOU, Yongxin; MAIN, Alec; GU, Yuan X; JOHNSON, Harold. *Information hiding in software with mixed boolean-arithmetic transforms*. 2007.p. 61–75.

[30] MING, Jiang; XU, Dongpeng; WANG, Li; WU, Dinghao. *Loop: Logic-oriented opaque predicate detection in obfuscated binary code*. 2015.p. 757–768.

[31] COLLBERG, Christian, THOMBORSON, Clark; LOW, Douglas. *Manufacturing cheap, resilient, and stealthy opaque constructs*. 1998.p. 184–196.

[32] BANESCU, Sebastian; COLLBERG, Christian; GANESH, Vijay; NEWSHAM, Zack; PRETSCHNER, Alex; ER. *Code obfuscation against symbolic execution attacks*. 2016.p. 189–200.

[33] RAMALINGAM, Ganesan. *The undecidability of aliasing*. ACM Transactions on Programming Languages and Systems (TOPLAS). 1994, vol 16, num. 5, p. 1467–1471.

[34] JOSUTTIS, Nicolai M. *The C++ standard library: a tutorial and reference*. Addison-Wesley. 2012.

[35] HOSSEINZADEH, Shohreh; RAUTI, Sampsa; LAURÉN, Samuel; MÄKELÄ, Jari-Matti; HOLVITIE, Johannes; HYRYNSALMI, Sami; LEPPÄNEN, Ville. *Diversification and obfuscation techniques for software security: A systematic literature review*. Information and Software Technology. 2018, vol 104, p. 72–93.

[36] EBAD, Shouki A.; DAREM, Abdulbasit; ABAWAJY, Jemal H. *Measuring Software Obfuscation Quality—A Systematic Literature Review*. IEEE Access. 2021.

[37] MAVROGIANNOPOULOS, Nikos; KISSERLI, Nessim; PRENEEL, Bart. A taxonomy of self-modifying code for obfuscation. 2011, vol 30, p. 679-691.

[38] HOFHEINZ, Dennis; MALONE-LEE, John; STAM, Martijn. *Obfuscation for cryptographic purposes*. Journal of Cryptology. 2010, vol 23, núm. 1, p. 121–168.