SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 6810

Ocjena kvalitete dekompajliranog C# koda

Petar Kovač

Zagreb, lipanj 2020.

Zahvaljujem mentoru doc. dr. sc. Stjepanu Grošu na pomoći i prenesenom znanju tijekom pisanja ovog rada.

SADRŽAJ

1. Uvod

Pri razvoju aplikacije je često potrebno provesti sigurnosnu analizu aplikacije, koja se ponekad naziva i penetracijsko ispitivanje. Pošto je pri sigurnosnoj analizi najčešće dostupan samo prevedeni kod, količina alata koji se mogu koristiti je ograničena. Najvažniji među njima su dekompajleri koji iz prevedene datoteke pokušavaju generirati izvorni kod. Taj postupak, kojeg zovemo dekompajliranje, nije savršen i generirani kod može biti vrlo različit od izvornog koda. Radi evaluacije generiranog koda, ali i poboljšanja postojećih dekompajlera je potrebno razviti metrike za ocjenu dekompajliranog koda. U ovom radu će biti proučeni dekompajleri i metrike za ocjenu efikasnosti dekompajlera u programskom jeziku C#.

Rad je strukturiran na sljedeći način. U drugom poglavlju se proučava kompajliranje, dekompajliranje i obfuskacija u jeziku C#. Proces dekompajliranja je objašnjen na dekompajleru ILSpy. U trećem poglavlju se razlažu metrike koje se koriste za ocjenu efikasnosti dekompajlera. Dodatno se proučavaju i algoritmi za računanje digitalnog otiska dokumenta koji su korišteni u metrikama. U četvrtom poglavlju se prikazuje implementacija metrika, testni slučajevi za razvijene razrede i prikazuju se rezultati primjene metrika. U zaključku će biti sažeto opisano što je napravljeno u radu, koji rezultati su dobiveni i procijenjena uspješnost rada.

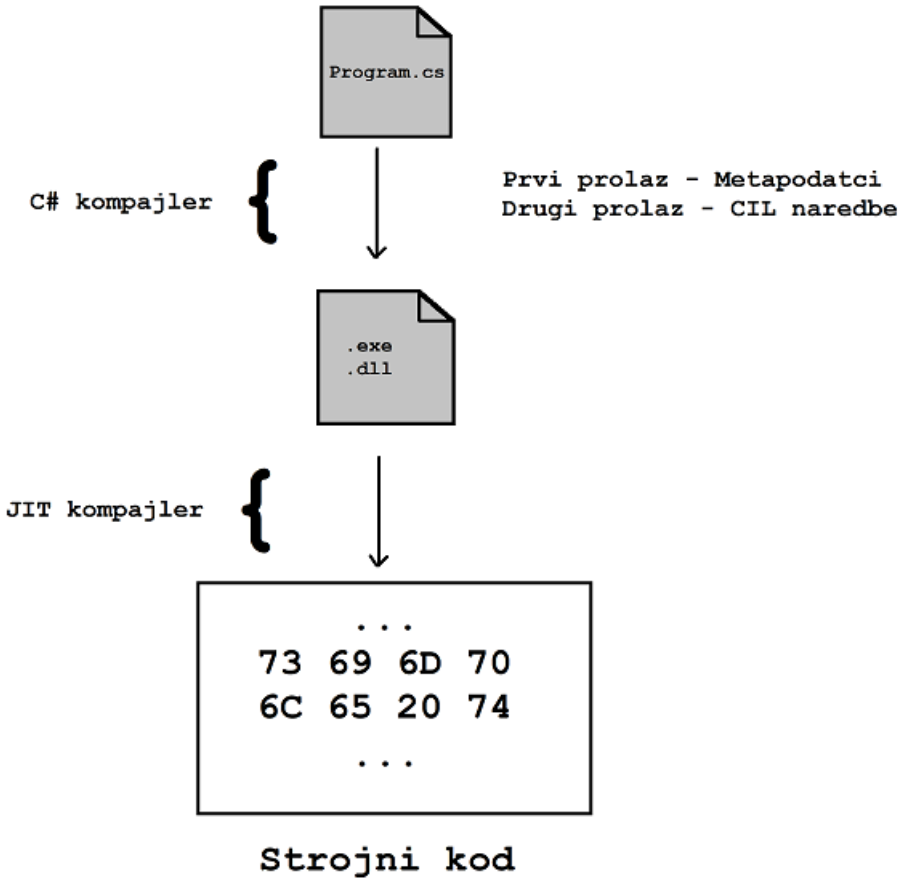
2. Kompajliranje i dekompajliranje

2.1. Kompajliranje

Kompajliranje je proces prevođenja jednog programskog jezika u drugi jezik. Program koji kompajlira zovemo kompajlerom. Kompajliranje je nužno da se kod napisan u jeziku visoke razine apstrakcije prevede u strojni kod razumljiv računalu. Kompajler može biti višeprolazan, što znači da više puta obrađuje izvorni program u procesu prevođenja. U procesu kompajliranja postoje dvije općenite faze: faza analize izvornog programa u kojoj se izvorni kod obrađuje u potrazi za greškama i faza sinteze ciljnog programa u kojoj se generira ciljni program. Tijekom ove dvije faze nepovratno se gube informacije o izvornom programu kao što su imena varijabli ili originalna struktura programa [18]. Tako se primjerice u fazi leksičke analize, početnoj fazi analize izvornog program, miču svi komentari i formatiranje iz originalnog programa. U nastavku će biti razmotreno kompajliranje u jeziku C# prikazano na slici 2.1.

Programski jezik C# spada u *Common Language Infrastructure* (CLI) skupinu jezika [2]. Jezici koji pripadaju toj skupini (C++, JScript, IronPython) se kompajliraju u oblik međukoda kojeg zovemo *Common Intermediate Language* (CIL). Kompajliranjem C# programa kao rezultat dobivamo datoteku s datotečnim nastavkom .exe ili .dll u kojoj se nalazi CIL. Pokretanjem .exe datoteke se poziva *Common Language Runtime* (CLR), program koji CIL pretvara u strojni kod kojeg računalo razumije. CLR koristi *Just-in-time* (JIT) kompajler koji kompajlira CIL svaki put kada se pokrene datoteka u kojoj se on nalazi. Ovako se postiže strojna nezavisnost u CLI skupini jezika.

Programski jezik C# koristi dvoprolazni kompajler [11]. U prvom prolazu se generiraju metapodaci, primjerice deklaracije metoda, atributa i imena klasa. Isto tako se provjerava da je program valjan C# program, ali se ne analiziraju definicije metoda. U drugom prolazu se generira CIL međukod, definicija svake metode se analizira i pokušava se odrediti tip svake varijable i izraza u metodi. Ako je ovaj postupak uspješan, provode se dodatni prolazi u kojima se gradi sintaksno stablo i optimizira kod. Drugi prolaz završava generiranjem CIL međukoda od izgrađenog sintaksnog stabla.



Slika 2.1: Kompajliranje u jeziku C#

2.2. Dekompajliranje

Dekompajliranje je proces generiranja programa više razine koristeći izvršnu datoteku koja je generirana iz izvornog jezika. Program koji dekompajlira zovemo dekompajlerom. Dekompajler u jeziku C# radi tako da prepoznaje uzorke u izvršnoj datoteci, gradi apstraktno sintaksno stablo (engl. Abstract syntax tree, AST), prepoznaje uzorke u AST-u koje zna prevesti u kod više razine i na kraju generira kod više razine [8]. Zbog gubitka informacija iz izvornog koda tijekom kompajliranja, savršeno dekompajliranje je u slučaju programskog jezika C# nemoguće. Ovo može biti prikazano na nekoliko primjera.

Na ispisu 2.1 je prikazan mrtvi kod. Mrtvi kod je dio programa koji je nedostupan te se neće ni pod kojim uvjetom izvršiti.

```
if (true) {
    int a;
} else {
    int b;
}
```

Ispis 2.1: Originalan kod

Dio programa u `else` dijelu izraza će tijekom kompajliranja (u procesu optimizacije međukoda) biti maknut jer nikada ne bi mogao biti izvršen. Zato će dekompajlirani kod na ispisu 2.2 najvjerojatnije sadržavati samo `if` izraz.

```
if (true) {
    int a;
}
```

Ispis 2.2: Dekompajlirani kod

Isto tako, dekompajler neće moći odrediti originalna imena varijabli, osim u slučaju kad mu je dostupna pripadna *Program Database* (PDB) datoteka [14]. PDB datoteka u sebi sadrži neke od metapodataka generiranih tijekom kompajliranja: imena varijabli i broj linije na kojoj se varijabla originalno nalazila. Izgled dekompajliranog koda s obzirom na postojanje PDB datoteke je prikazan na primjerima 2.3, 2.4 i 2.5.

```
byte right = 0;
byte left = 1;
byte down = 2;
byte up = 3;
```

Ispis 2.3: Izvorni kod

```
byte b = 0;
byte b2 = 1;
byte b3 = 2;
byte b4 = 3;
```

Ispis 2.4: ILSpy kod dekompajliran bez PDB datoteke

```
byte right = 0;
byte left = 1;
byte down = 2;
byte up = 3;
```

Ispis 2.5: JustDecompile Engine kod dekompajliran s PDB datotekom

Na prvom isječku koda se nalaze originalni nazivi varijabli programa. Drugi i treći isječak su uzeti iz dekompajliranog koda, ali dekompajleru ILSpy pri dekompajliranju nije bila dostupna PDB datoteka, dok je bila dostupna dekompajleru JustDecompile Engine.

Imena varijabli su važna za shvaćanje semantike programa. Zato je poželjno, iako ne uvijek i moguće da dekompajler generira originalna imena varijabli. Alati koji se koriste za otežavanje dekompajliranja, obfuskatori, često mijenjaju upravo imena varijabli.

U ovom radu će biti razmatrana dva aktualna dekompajlera, ILSpy i *JustDecompile Engine* (JDE). Postoji još nekoliko aktualnih dekompajlera kao što su: CodeReflect, JetBrains dotPeek, Reflector, ali su u ovom radu obrađeni samo oni koji su besplatni i koji se mogu pokrenuti iz naredbenog retka. Rad jednog dekompajlera će biti prikazan na primjeru dekompajlera ILSpy i JDE.

2.3. Dekompajler ILSpy

ILSpy [10] je dekompajler otvorenog koda za jezik C# kojeg su razvili Daniel Grunwald, Siegfried Pammer i David Srbecký na temelju Srbeckýove disertacije i koda napisanog 2008. godine [17]. Razvoj alata ILSpy je počeo 2011. godine te je glavna motivacija iza razvoja alata bila objava poduzeća *Redgate* da alat *.NET Reflector* više neće biti besplatan [3][7]. Danas su Daniel Grunwald i Siegfried Pammer najveći doprinosioci i održavatelji dekompajlera ILSpy.

ILSpy započinje svoj rad tako da svaku naredbu CIL-a prevede u viši oblik međukoda kojeg su autori sami razvili i zove se *Intermediate Language Abstract Syntax Tree* (ILAst) [8]. Svaka CIL naredba se pretvara u točno jednu ILAst funkciju (*ILExpression*). Glavni zadatci ILAst-a su napraviti strukturiranu reprezentaciju CIL koda i zaključiti koji se tipovi podataka koriste u naredbama CIL-a, pošto se taj dio semantike gubi tijekom prevođenja programa. Tako primjerice tip podatka „I4“ u CIL-u može označavati `int`, `bool`, `uint`, `short`, `ushort`, `byte` ili `sbyte` tip podatka. ILSpy pomoću

načina na koji se koristi ta varijabla (primjerice pri pozivu funkcije koja prima short kao argument) određuje pravi tip podatka. Za strukturiranu reprezentaciju CIL koda ILSpy koristi sintaksno stablo čiji se čvorovi zovu *ILExpression*. Jedan ILExpression u sebi sadrži kod pripadne CIL naredbe, ali može sadržavati i specifične ILAst kodove koje označavaju konstrukte u kodu kao što su strukture za kontrolu toka. Sintaksno stablo je struktura koja se koristi tijekom kompajliranja C# izvornog koda u ciljni kod i moguće je, poznavajući sintaksno stablo, odrediti u koju se naredbu neka struktura u sintaksnom stablu može preslikati. Na ovaj način ILSpy prevodi ILAst u C# kod.

2.4. Dekompajler JDE

JDE [19] je dekompajler otvorenog koda kojeg razvija poduzeće *Telerik* od 2015. godine. Alat je u vrijeme pisanja zadnji put ažuriran 18. siječnja 2019. JDE u procesu dekompajliranja isto kao i ILSpy stvara AST, ali za stvaranje AST-a koristi *Cecil.Decompiler* [5], alat koji iz definicije metode generira AST. Autori alata JDE nisu opisali na koji način JDE generira izvorni kod iz AST-a. Razumno je pretpostaviti da se to događa na sličan način kao i kod alata ILSpy, tj. da se nalaze dijelovi AST-a za koje je moguće odrediti preslikavanje u izvorni kod te se za svaki takav dio AST-a generira izvorni kod.

2.5. Obfuskatori

Postoje legitimni razlozi zašto bi se htjelo dekompajlirati izvršnu datoteku. Recimo, u slučaju da programer izgubi izvorni kod, dekompajliranje bi moglo pomoći s vraćanjem izvornog koda. Dekompajliranje isto može biti i zlonamjerno, kao u slučaju reverznog inženjerstva tuđeg koda. Postoji više načina na koje se aplikacija može zaštititi od reverznog inženjerstva. Moguće je primjerice staviti aplikaciju na aplikacijski poslužitelj i tražiti od korisnika da aplikaciju koristi isključivo putem poslužitelja. Korisnik tada šalje podatke koje želi obraditi, a aplikacija nakon obrade vraća rezultat. Program se neće izvršiti na korisničkom računalu i korisnik neće biti u mogućnosti dekompajlirati kod.

Jedan od poznatih načina zaštite izvornog programa su obfuskatori. Obfuskatori su programi koji provode obfuskaciju. Obfuskacija je postupak kojim se otežava razumijevanje izvornog programa ili dekompajliranje izvršnog programa. Obfuskatori nad programima vrše postupke kao što su: promjena imena varijabli, promjena struk-

ture programa i zamjena jednostavnih izraza puno složenijim, ali istoznačnim izrazima [1]. Obfuskatori ne miču korisne informacije iz programa te će dekompajliranjem iste ciljne datoteke prije i nakon obfuskacije biti dobiveni funkcionalno jednaki programi, ali će za razumijevanje obfusciranog programa biti potrebno značajno više vremena. U ovom radu će biti obrađeni sljedeći obfuskatori: Eazfuscator.NET, Orange Heap i neo-ConfuserEx.

3. Ocjenjivanje efikasnosti dekompajlera

Važan problem u ocjenjivanju efikasnosti dekompajlera je određivanje metrika po kojima će biti dana ocjena. Od dekompajliranog koda se traži da semantički bude što bliži originalnom programu. Postoje radovi koji ocjenjuju dekompajlirani kod s obzirom na to koliko je točna sintaksa i semantika dekompajliranog koda. Primjerice u jednom radu [9] se programu daje ocjena 0-10 s obzirom na točnost sintakse ili semantike i mogućnost njihovog ispravljanja u slučaju greške. Iako rad daje dobar pregled trenutnog stanja dekompajlera, problem s ovim pristupom je što nije automatiziran. Moguće je da želimo vidjeti koji od dva obfuskatora bolje obfuscira kod i to pomoću koda kojeg dekompajler generira iz obfusciranih datoteka. U tom slučaju je potrebno koristiti metrike koje se mogu računati na temelju dekompajliranog koda. U ovom radu će biti objašnjene i implementirane neke od metrika koje su predstavljene u radu autora Naeem, Hendren i Batchfelder [12]. Predlaže se i korištenje dodatne metrike koja se bazira na algoritmima za računanje digitalnog otiska.

3.1. Algoritmi za računanje digitalnog otiska

Algoritmi za računanje digitalnog otiska su algoritmi koji sažimaju veliku količinu podataka u relativno mali sažetak. Algoritmi za računanje digitalnog otiska se koriste za jedinstvenu identifikaciju neke datoteke, identifikaciju audio uzorka, ali se isto tako mogu koristiti za detekciju plagijata [20].

3.1.1. RKR-GST

Running Karp-Rabin Greedy String Tiling (RKR-GST) je algoritam za računanje digitalnog otiska koji se koristi za detekciju plagijata, ali i tamo gdje je potrebno saznati sadrži li veća količina originalnog teksta u sebi neki manji uzorak [21][20]. Cilj algo-

ritma je naći što veće podudaranje između originalnog teksta i uzorka. Za shvaćanje algoritma je prvo potrebno definirati nekoliko pojmova.

Maksimalno podudaranje je najdulje moguće podudaranje između originalnog teksta i uzorka počevši od neke pozicije u uzorku. U jednoj iteraciji algoritma može postojati više maksimalnih podudaranja od kojih jedan može sadržavati dio drugoga. Ovakva veza između dva podniza u tekstovima nije trajna niti jedinstvena, jedan podniz u originalnom tekstu može imati više maksimalnih podudaranja s podnizovima uzorka. Maksimalno podudaranje postaje trajno kada se oba podniza u kasnijoj fazi algoritma označe.

Označavanje je postupak u kojem se odabire podskup svih maksimalnih podudaranja tako da se obuhvate što veći dijelovi tekstova. Maksimalno podudaranje koje je jednom bilo označeno se više ne može koristiti u budućim maksimalnim podudaranjima te je ovakva asocijacija trajna.

Pri određivanju maksimalnog podudaranja važno je uočiti da jako kratka podudaranja ne moraju implicirati sličnost između originalnog teksta i uzorka i da takva podudaranja trebamo zanemariti. Što je veća duljina podudaranja, to je vjerojatnije da je takvo podudaranje značajno, te će u algoritmu postojati granica duljine podudaranja ispod koje zanemarujemo podudaranje.

GST algoritam se sastoji od dva manja algoritma – *scanpattern* i *markstrings*. U algoritmu *scanpattern* se traže maksimalna podudaranja iznad trenutne duljine pretraživanja. U algoritmu *markstrings* se nađena maksimalna podudaranja označuju.

Za nalaženje maksimalnih podudaranja se koristi RKR algoritam. Definira se duljina pretraživanja *D* i svaki podniz duljine *D* u originalnom tekstu se sažima korištenjem kotrljajuće funkcije sažimanja. *Kotrljajuća funkcija sažimanja* je funkcija sažimanja kojom se može izračunati sljedeći sažetak koristeći samo novi element i prijašnju vrijednost sažetka. Ovo omogućuje efikasno računanje sažetaka pošto je vremenska složenost linearna. Na isti način se sažima i tekst uzorka. Na kraju se i sažetci sažimaju što omogućuje brzo uspoređivanje sa sažetcima iz uzorka.

RKR-GST ocjena se računa kao omjer broja označenih elemenata i ukupnog broja elemenata u oba programa. Parametri *a* i *b* su programi koji se uspoređuju.

$$\text{RKR\_GST}(a, b) = \frac{\text{broj\_označenih}(a, b)}{\text{broj\_elemenata(a)} + \text{broj\_elemenata(b)}}$$

3.1.2. Winnowing

Winnowing algoritam je algoritam za računanje digitalnog otiska koji se, slično kao i algoritam RKR-GST, koristi za detekciju plagijata [16]. U winnowing algoritmu se isto tako sažimaju uzastopni podnizovi originalnog teksta. Podnizove nazivamo *k-gramima* gdje je *k* parametar duljine. Slično kao i kod RKR-GST algoritma, može se pretpostaviti da su vrlo male vrijednosti parametra *k* zanemarive. Na primjeru 3.1 je prikazan tekst, obrađeni tekst i pripadni 6-grami. Iz teksta se prvo miču svi razmaci i interpunkcijski znakovi. Potom se iz tako obrađenog teksta uzimaju uzastopni nizovi od 6 slova koje zovemo 6-gramima.

```
Na vrhu brda vrba mrda.

Navrhubrdavrbamrda

Navrhu avrhub vrhubr rhubrd hubrda ubrdav brdavr
rdavrb davrba avrbam vrbamr rbamrd bamrda
```

Primjer 3.1: Generiranje k-grama

Ovi k-grami se sažimaju, te se odabiru samo neki od sažetaka koji se nazivaju otisci teksta. Za odabir sažetaka prvo se definira prozor veličine *w*, gdje je *w* parametar veličine prozora. Prozor je niz koji sadrži *w* slijednih sažetaka k-grama. Otisak se odabire tako da se uzme najmanji sažetak u prozoru ili, ako postoje više jednakih sažetaka se uzima najdesniji takav sažetak. Nakon ovoga se prozor pomiče za jedno mjesto i postupak se ponavlja. Na primjeru 3.2 su prikazani sažetci, pripadni prozori veličine 3 i odabrani otisci. U svakom prozoru je označena trenutna najdesnija minimalna vrijednost. U prvom prozoru je za vrijednost prvog otiska odabrana vrijednost 2 jer je ona najmanja vrijednost u prozoru. U trećem prozoru se odabire vrijednost drugog otiska koja je jednaka vrijednosti prvog otiska, ali se nalazi desnije od nje u prozoru. U šestom prozoru se odabire treći otisak jer se vrijednost prijašnjeg odabranog otisak više ne nalazi u prozoru. U sedmom prozoru se odabire otisak kojem je vrijednost 1 jer je njegova vrijednost manja od vrijednosti prijašnjeg odabranog otiska. Prozori u kojima nisu odabrani novi otisci su oni u kojima se nalazi zadnji odabrani otisak te je on ujedno i najdesnija minimalna vrijednost u prozoru.

```
3 7 2 4 2 5 7 3 1

(3 7 |2|) (7 |2| 4) (2 4 |2|)
(4 |2| 5) (|2| 5 7) (5 7 |3|)
          (7 3 |1|)

2 2 3 1
```

Primjer 3.2: Generiranje otisaka uz w = 3

Ocjena sličnosti dva teksta se računa kao omjer dvostruke veličine presjeka skupova otisaka i veličine unije skupova otisaka. Parametri *a* i *b* su programi koji se uspoređuju.

$$\text{winnowing}(a, b) = \frac{2 \cdot \text{velicina presjeka}(a, b)}{\text{velicina unije}(a, b)}$$

3.2. Metrike

U ovom poglavlju će biti razmatrane metrike koje će kasnije biti implementirane i korištene u ocjenjivanju efikasnosti dekompajliranja.

3.2.1. Metrika sličnosti

Ukupna ocjena RKR-GST i winnowing algoritama se računa kao aritmetička sredina njihovih vrijednosti. Parametri *a* i *b* su programi koji se uspoređuju.

$$\text{ukupno}(a, b) = \frac{\text{winnowing}(a, b) + \text{RKR-GST}(a, b)}{2}$$

Kada se ocjena sličnosti koristi kao metrika za računanje ukupne ocjene efikasnosti dekompajliranja, ona je izražena na način prikazan na formuli.

$$\text{metrika\_sličnosti}(a, b) = \frac{1}{\text{ukupno}(a, b)^2}$$

Odabir ovog izraza temelji se na pretpostavci da je teže razumjeti dekompajlirani kod jer je on različit od izvornog koda, tj. što je dekompajlirani kod bliži izvornom, to

ga je lakše razumjeti. Pošto će ukupna ocjena sličnosti uvijek biti manja ili jednaka 1, metrika sličnosti će uvijek biti veća ili jednaka 1. Ova vrijednost se potom i kvadrira jer je pretpostavka da čak i mali pad u sličnosti između originalnog i dekompajliranog koda može jako otežati razumijevanje dekompajliranog programa

3.2.2. Broj C# konstrukata

Za ovu metriku je važno odrediti koji konstrukti impliciraju kompleksnost u programu. Pretpostavka je da je veći broj konstrukata značajka kompleksnijeg koda. Određeni su sljedeći konstrukti po kojima se ova metrika mjeri:

1. `If-else` konstrukti,
2. `Break`, `continue` i `goto`,
3. Labele,
4. Lokalne varijable.

`If-else` konstrukti su jednostavni elementi programa, ali povećana količina ovih konstrukata naspram originalnog programa implicira nepotrebnu kompleksnost.

Pretpostavka je da je od dva programa koji imaju istu funkcionalnost teže shvatiti onaj koji koristi `break`, `continue` ili `goto` naredbe od onog koji bi koristio samo `If-else` konstrukte. Zato su ove naredbe veći pokazatelj kompleksnosti nego `If-else` konstrukti.

Labele su primjer kompliciranе kontrole toka i pretpostavka je da su one najveći pokazatelj kompleksnosti u ovoj metrici.

Pojedinačno, jedna lokalna varijabla ne unosi veliku kompleksnost u program. Odabir ovog konstrukta za mjerenje kompleksnosti programa se temelji na pretpostavci da je za program koji ima više lokalnih varijabli od originalnog programa potrebno i više vremena za shvaćanje uloge svake od dodatnih varijabli. Zbog ovih razloga je pretpostavka da su lokalne varijable slab pokazatelj kompleksnosti u programu.

3.2.3. Uvjetna kompleksnost

Pod *uvjetnom kompleksnosti* se podrazumijeva ukupna kompleksnost koju uvjetni izraz unosi u program. Uvjetni izraz može biti dio uvjetnog konstrukta (primjerice u `if`, `while` i `for` konstruktima), definicija logičke varijable ili impliciran, kao u slučaju `switch-case` konstrukta.

Uvjetna kompleksnost se računa tako da se svakom jednostavnom izrazu kojeg uvjetni izraz sadrži pridružuje vrijednost uvjetne kompleksnosti jednostavnog izraza te se tako dobivene vrijednosti zbrajaju [12]. Jednostavnim `true`, `false` i unarnim logičkim izrazima se daje vrijednost 1. Agregacijskim izrazima kao što su `&&` i `||` se daje vrijednost 1 jer je za određivanje logičke vrijednosti izraza potrebno odrediti logičku vrijednost njihova dva podizraza. Relacijskim operatorima $(<=, >=, ==, != , <, >)$ i operatoru negacije se daje vrijednost $0,5$. U slučaju relacijskih operatora argument je da su oni kompleksniji od jednostavne logičke varijable, ali da su lakši za razumjeti. Operator negacije ima vrijednost $0,5$ jer, za razliku od agregacijskih izraza, za određivanje logičke vrijednosti ukupnog izraza je potrebno odrediti logičku vrijednost samo jednog podizraza. Uvjetna kompleksnost takvog izraza je tada zbroj svih vrijednosti u izrazu. Na primjeru 3.3 će biti prikazan način izračuna ove vrijednosti.

```
!weather.isRaining() || wetStreets
```

Ispis 3.3: Uvjetni izraz s kompleksnošću 3,5

S obzirom na pridružene vrijednosti, uvjetna kompleksnost je zbroj operatora negacije $(0,5)$, logičkog izraza (1), operatora agregacije (1) i još jednog logičkog izraza (1). Uvjetna kompleksnost izraza je dakle $3,5$.

3.2.4. Kompleksnost identifikatora

Imena identifikatora mogu pružati uvid u semantiku programa i dati informacije o njegovoj primjeni. Zato obfuskatori jako često mijenjaju imena identifikatora u besmislene nizove znakova što dodatno i otežava čitljivost. Ova metrika se računa kao zbroj dvije manje metrike, kompleksnost znakova i kompleksnost leksičkih jedinki [12].

Kompleksnost znakova se računa kao omjer svih znakova i jednostavnih znakova. Kompleksni znakovi su znakovi koji su dio niza koji se sastoji od znakova koji nisu alfanumerici i ima duljinu veću od 1. Jednostavni znakovi su svi znakovi koji nisu kompleksni. Na primjeru 3.4 će biti prikazan način izračuna ove vrijednosti.

```
idn%#%name$
```

Primjer 3.4: Kompleksan identifikator

Identifikator ima 3 kompleksna znaka i 8 nekompleksnih znakova te je kompleksnost znakova varijable $1,375$. Znak '$' na kraju identifikatora nije kompleksan znak jer, iako nije alfanumerik, duljina tog podniza je jednaka 1.

Kompleksnost leksičke jedinke se računa kao omjer ukupnog broja riječi nađenih u varijabli i riječi koje su nađene u rječniku. U ovom radu ova metrika nije implementirana.

Svakom identifikatoru se pridodaje faktor važnosti u ovisnosti o tome kojeg je tipa identifikator. U konačnom rezultatu, faktor važnosti se množi s kompleksnosću znakova jednog identifikatora da se dobije ukupna kompleksnost identifikatora za taj identifikator. Faktor važnosti je najviši za imena klasa i imena metoda jer oni daju najviše informacije o funkcionalnostima programa. Metodama je dodijeljen faktor važnosti 4, a klasama 3. Članske varijable mogu dati važan uvid u stanja i način rada klase i njima je dodijeljen faktor važnosti 2. Formalnim argumentima metoda se pridodaje faktor važnosti $1,5$ jer su važniji za shvaćanje rada metode od lokalnih varijabli metode, kojima se pridodaje faktor važnosti 1.

4. Implementacija i rezultati

Za implementaciju je korišten programski jezik Python 3. Prvi algoritmi koji su bili implementirani su winnowing i RKR-GST algoritam pošto je početna ideja bila ocijeniti efikasnost rada dekompajlera tako da ga se usporedi s originalnim kodom, ponašajući se da je dekompajlirani program zapravo plagijat originalnog programa.

4.1. Winnowing algoritam

Winnowing algoritam je implementiran na način prikazan na ispisu 4.1

```
def process_compare(original, decompiled):
    win_size = determine_window_size(original,
        decompiled)
    fprint_size = win_size // 5
    if fprint_size < 3:
        return 0
    decompiled_fprints =
        create_fingerprints(decompiled, fprint_size)
    original_fprints = create_fingerprints(original,
        fprint_size)
    winnowing_original =
        winnowing_algorithm(original_fprints, win_size)
    winnowing_decompiled =
        winnowing_algorithm(decompiled_fprints,win_size)
    return (len(set(winnowing_original).intersection
            (winnowing_decompiled)) * 2) /
                (len(set(winnowing_original)) +
                len(set(winnowing_decompiled)))
```

Ispis 4.1: Implementacija winnowing algoritm

Metoda `determine_window_size` dinamički određuje veličinu prozora. Početne postavke su 40 za veličinu prozora, a 8 za parametar k, ali postoje rubni slučajevi kada je jedan od programa premalen da se napravi prozor zadane veličine. Naime, ako neki program ima n elemenata, tada je najveći broj prozora $w = (n - k + 1)$. Za programe koji imaju manje od 47 elemenata početni uvjet nije zadovoljen i potrebno je iznova izračunati veličinu prozora. Ako jedan od programa ima manje od 15 elemenata, usporedba se uopće ne provodi jer je tada duljina k-grama premala da se donose značajni zaključci. Nakon odabira veličine prozora se stvaraju sažetci koristeći MD5 algoritam za sažimanje poruke.

Sažetci se koriste u glavnom winnowing algoritmu prikazanom na ispisu 4.2 gdje se odabiru otisci koji reprezentiraju program.

```
def winnowing_algorithm(fingerprints, w):
    winnowing_arr = []
    min_relative_index = -1
    min_relative_value = 0
    for i in range(0, len(fingerprints) - w):
        if min_relative_index == -1:
            min_relative_index =
                find_min_index(fingerprints[i:i + w])
            min_relative_value = fingerprints[i +
                min_relative_index]
            winnowing_arr.append(min_relative_value)
            min_relative_index -= 1
        else:
            min_relative_index -= 1
            if int(min_relative_value) >
                int(fingerprints[i + w - 1]):
                min_relative_index = w - 1
                min_relative_value = fingerprints[i + w
                    - 1]
                winnowing_arr.append(min_relative_value)
    return winnowing_arr
```

Ispis 4.2: Winnowing algoritam

U algoritmu se u prvoj iteraciji nalazi i pamti indeks najmanjeg elementa u prozoru. Najmanji element se dodaje u polje otisaka. Novi najmanji element se ne traži u svakoj

iteraciji već samo kada stari najmanji element više nije u prozoru (`min_relative_index == -1`), a u slučaju da je novi element dodan na kraj prozora manji od trenutnog najmanjeg elementa se mijenjaju pripadne `min_relative_index` i `min_relative_value` varijable te se novi element dodaje u polje otisaka. Nakon ove funkcije se računa ocjena sličnosti.

4.2. RKR-GST

Algoritam RKR-GST je složeniji od winnowing algoritma i zbog količine programskog koda će biti objašnjeni samo važni dijelovi i općeniti algoritam prikazan na ispisu 4.3.

```
def algorithm(initial_size, min_len, tokens_t,
    tokens_p):
    search_len = initial_size
    coverage = 0
    while True:
        match_dict = dict()
        hash_dict = dict()
        max_len = scanpattern(search_len, tokens_t,
            tokens_p, match_dict, hash_dict)
        if max_len > 2 * search_len:
            search_len = max_len
        else:
            coverage += markstrings(max_len,
                search_len, match_dict, tokens_t,
                tokens_p)
            if search_len > 2 * min_len:
                search_len = search_len // 2
            elif search_len > min_len:
                search_len = min_len
            else:
                break
    return (coverage * 2) / (len(tokens_t) +
        len(tokens_p))
```

Ispis 4.3: RKR-GST algoritam

Općeniti GST algoritam se sastoji od dva dijela, `scanpattern` i `markstrings`. Kao što je prije opisano, u funkciji `scanpattern` se traže i bilježe sva maksimalna podudaranja. Ako se naiđe na maksimalno podudaranje čija je duljina dvostruko veća od trenutne duljine pretraživanja, onda se opet ulazi u funkciju, ali s dvostruko većim parametrom duljine pretraživanja. Za stvaranje sažetaka u funkciji `scanpattern` se koristi kotrljajuća funkcija sažimanja. Vrijednost sažetka svakog elementa je implementirana kao zbroj ASCII vrijednosti znakova u elementu. Ukupna vrijednost sažetka je jednaka zbroju vrijednosti sažetka svakog elementa pomnožena s konstantom kojoj je eksponent relativni indeks elementa.

Ako nije poznat prethodni sažetak, onda se koristi funkcija `create_hash` prikazana na ispisu 4.4. Ovaj slučaj je moguć u početnom slučaju algoritma `scanpattern` ili u slučaju da su upravo bili preskočeni elementi teksta koji su već označeni.

```
def create_hash(index, length, tokens):
    base = 7 if length < 20 else 3
    base = base if length < 100 else 1
    hash_val = 0
    for i in range(index, index + length):
        hash_val += compute_token_ord(tokens[i][1]) *
            (base ** (i - index))
    return hash_val
```

Ispis 4.4: Metoda za stvaranje sažetka

Ako je poznata vrijednost prethodnog kotrljajućeg sažetka, koristi se funkcija `update_hash` prikazana na ispisu 4.5. Ova funkcija računa novu vrijednost kotrljajućeg sažetka na način da se od prethodne vrijednosti oduzme vrijednost prvog člana, podijeli se s konstantom i doda se vrijednost novog člana pomnoženog s potenciranom bazom.

```
def update_hash(index, length, tokens,
    previous_hash):
    base = 7 if length < 20 else 3
    base = base if length < 100 else 1
    return (((previous_hash -
        compute_token_ord(tokens[index - 1][1])) //
        base) + (compute_token_ord(tokens[index +
        length - 1][1]) * (base ** (length - 1))))
```

Ispis 4.5: Metoda za računanje nove vrijednosti kotrljajućeg sažetka

U metodi `markstrings` se provjerava da je par maksimalnih podudaranja jednak te da im se sažetci ne podudaraju slučajno. Osim toga, provjerava se da elementi maksimalnih podudaranja nisu već označeni. Ako nisu, onda se označuju. Završna ocjena se računa nakon što trenutna duljina postane jednaka minimalnoj duljini.

4.3. Leksička analiza

Kada se uspoređuju dva programa onda formatiranje programa, interpunkcijski znakovi i komentari ne bi smjeli imati utjecaj na ocjenu sličnosti [20]. Za bolju implementaciju RKR-GST i winnowing algoritama je potrebna leksička analiza i obrada leksičkih jedinki. Leksička analiza je implementirana pomoću *pygments* biblioteke. Leksičkom analizom se postiže mogućnost micanja dijelova programa koji su nepotrebni za usporedbu, primjerice komentara i praznina, ali se i omogućava obrada leksičkih jedinki. Obradom se imena varijabli, brojčanih tipova podataka i znakovnih nizova mijenjaju u jedinstvene leksičke jedinke. Ovo je pokazano na ispisu 4.6 gdje se u prva četiri retka ispisa nalazi originalni kod, a u zadnja tri retka isti kod nakon obrade.

```
int i1 = 1   ;
string s = "";
Random randomGenerator   =   new Random();; /*
   trailing comment */


<NUMERIC_TYPE><IDENTIFIER>=<NUMERIC_VALUE>
<STRING_TYPE><IDENTIFIER>=<STRING_VALUE>
<IDENTIFIER><IDENTIFIER>=new<IDENTIFIER>()
```

Ispis 4.6: Obrada leksičkih jedinki

Prednosti koje dobivamo ovakvom obradom su da formatiranje programa, komentari, interpunkcijski znakovi, imena varijabli, promjene brojevnog tipa pri dekompajliranju i promjena vrijednosti konstanti više nemaju utjecaja na ocjenu sličnosti dva programa. Koristeći ovakvu obradu, ocjene sličnosti su bliže očekivanju, što će i biti pokazano na testnim slučajevima. Jedna negativna posljedica ovog postupka je što winnowing algoritam za dva različita velika programa (iznad 2000 linije koda) često daje neočekivano visoku ocjenu sličnosti. Pretpostavka je da je to posljedica toga što

se leksičkom analizom i obradom povećava broj čestih otisaka, tj. otisaka koje će winnowing algoritam vrlo vjerojatno generirati i da su veliki programi slični upravo prema tim čestim otiscima.

4.4. Metrike

Potpora za računanje metrika je implementirana pomoću 4 razreda: *QualityMeasure, Scope, IdentifierFinder* i *ConditionalMeasure*. Ovi razredi koriste leksičku analizu. U poglavlju će detaljno biti objašnjen način rada ovih razreda.

4.4.1. Razred *Scope*

Tijekom brojanja lokalnih varijabli moguće je da se dvije varijable istoga imena nalaze unutar dvije različite metode. Za precizno brojanje potrebno je razlikovati takve dvije varijable. Jedan od načina rješavanja ovog problema je da se odredi opseg u kojem vrijedi deklaracija varijable. Tada se može jednoznačno odrediti da je deklaracija varijable jednakog imena koja se nalazi u drugom djelokrugu drukčija od prve deklaracije. S tom idejom je razvijen razred *Scope* koji definira novi doseg svaki put kada se naiđe na tijelo metode ili na for petlju. Takav primjerak razreda *Scope* zovemo *običnim dosegom*. Jedan primjerak razreda *Scope* može referencirati više primjeraka razreda *Scope* koji definiraju dosege sadržane u roditeljskom dosegu. Tako se obradom programa dobiva struktura podataka stabla.

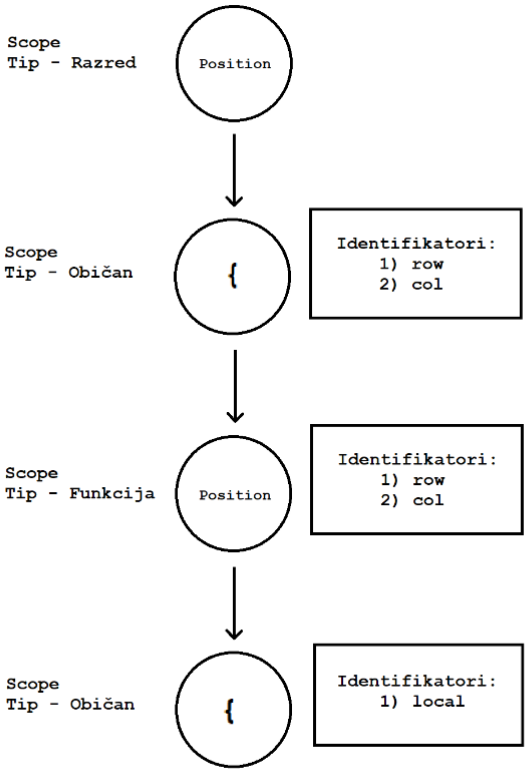
Razred *Scope* se dodatno koristi za nalaženje članskih atributa i formalnih argumenata tako da se stvori novi primjerak razreda *Scope* kada se prvi put naiđe na ime razreda ili na ime funkcije. Primjerak stvoren kada se obradi ime razreda se naziva *razrednim dosegom*, a primjerak stvoren obradom funkcije *funkcijskim dosegom*.

Pomoću tih definicija se članska varijabla može definirati kao identifikator koji se nalazi u primjerku razreda *Scope* čiji roditeljski *Scope* pripada razrednom dosegu. Formalni argumenti se na sličan način definiraju kao identifikatori koji se nalaze u primjerku razreda *Scope* koji pripada funkciji.

Način rada razreda *Scope* može biti prikazan na primjeru 4.7. Na slici 4.1 je prikazano pripadno stablo. Identifikatori *row* i *col* pripadaju običnom dosegu, ali pošto je roditeljski doseg razredni, ovi identifikatori su članski atributi. Drugi put kada se nađu identifikatori *row* i *col*, trenutni doseg je funkcijski te su to formalni argumenti. Identifikator *local* se nalazi u običnom dosegu, a roditeljski doseg nije razredni te je taj identifikator lokalna varijabla.

```
struct Position
    {
        public int row;
        public int col;
        public Position(int row, int col)
        {
            this.row = row;
            this.col = col;
            int local;
        }
    }
```

Ispis 4.7: Isječak programa Snake



Slika 4.1: Stablo primjeraka razreda *Scope*

4.4.2. Razred *QualityMeasure*

QualityMeasure je razred u kojem se na kraju obrade programa trebaju nalaziti informacije o broju različitih C# konstrukata koji se nalaze u programu i imena svih identifikatora potrebnih za računanje kompleksnosti identifikatora. U razredu se isto tako nalazi jednostavna metoda `process_token` za obradu jedne leksičke jedinke. Biblioteka *pygments* daje informaciju kojeg je tipa obrađena leksička jedinka tako da je jednostavno identificirati sve konstrukte. Metoda `process_token` prikazana na ispisu 4.8 nalazi upravo te elemente u programu. Pronalazak identifikatora se delegira razredu *IdentifierFinder*.

```
def process_token(self, token):
    if str(token[0]) == "Token.Keyword":
            if token[1] == "if" or token[1] ==
               "else" or token[1] == "case":
                    self.ifelse_cnt += 1
            elif token[1] == "break" or token[1] ==
               "continue" or token[1] == "goto" or
               (token[1] == "return" and
               self.previous_value[1] == "yield"):
                    self.brk_cont_cnt += 1
    elif str(token[0]) == "Token.Punctuation":
            if token[1] == ";":
                    self.label_flag = True
            elif str(self.previous_value[0]) ==
               "Token.Name" and self.label_flag and
               token[1] == ":":
                    self.labeled_blocks += 1

    if (not (token[1].isspace() or token[1] == ";"))
       and str(token[0]) != "Token.Name":
            self.label_flag = False
```

Ispis 4.8: Metoda za nalaženje i brojanje konstrukata u kodu

Razred implementira i potporu za računanje metrike kompleksnosti identifikatora.

4.4.3. Razred *IdentifierFinder*

Alat *pygments* koji se koristi za leksičku analizu ne može točno prepoznati identifikatore u programu jer ne razlikuje ime varijable od imena razreda varijable. Osim toga ne razlikuje članske atribute, formalne argumente i lokalne varijable, što je potrebno za implementaciju metrike kompleksnosti identifikatora. Zato je razvijen razred *IdentifierFinder* koji nalazi identifikatore u programu i određuje tip nađenog identifikatora. Razred *IdentifierFinder* je automat stanja s 3 stanja i definira metodu `process_token` koja obrađuje leksičke jedinke. Stanja automata su sljedeća:

1. stanje pretrage,
2. stanje deklaracije,
3. stanje definicije.

U *stanju pretrage* se leksičke jedinke preskaču dok se ne nađe leksička jedinka koja može biti tip varijable. Tip varijable mogu biti imena razreda ili primitivni tipovi podataka. Primjer leksičke jedinke koja se traži u ovom stanju je prikazan na primjeru 4.9 gdje je tražena leksička jedinka označena okomitim crtama.

```
|tip_varijable| ime_varijable;
|int| a;
```

Primjer 4.9: Stanje pretrage

U *stanju deklaracije* se provjerava je li sljedeća leksička jedinka identifikator. Na primjeru 4.10 je prikazana leksička jedinka koja se traži u ovom stanju te je tražena leksička jedinka označena okomitim crtama.

```
tip_varijable |ime_varijable|;
int |a|;
```

Primjer 4.10: Stanje deklaracije

Ako leksička jedinka nije identifikator, automat se vraća u stanje pretrage. Ako se nakon identifikatora naiđe na zarez, automat ostaje u stanju deklaracije, a ako je sljedeća leksička jedinka znak jednakosti, automat prelazi u stanje definicije.

U *stanju definicije* se definira vrijednost varijable te se preskaču leksičke jedinke dok se ne dođe do sljedećeg zareza, kada se vraća u stanje deklaracije. Na primjeru

4.11 je prikazan primjer definicije koja se preskače. Definicija je okružena okomitim crtama.

```
tip_varijable ime_varijable = |definicija_varijable|;
int a = |5 + b + c.func(3, 6, 7)|;
```

Primjer 4.11: Stanje definicije

Kada automat obradi leksičku jedinku točke sa zarezom, uvijek prelazi u stanje pretrage.

Ovo je pojednostavljen opis i postoje provjere posebnih slučajeva (primjerice kada se u definiciji varijable nalazi funkcija koja prima više od jednog argumenta) koje omogućuju jednoznačnu pretragu identifikatora. Kada se nađe novi identifikator, određuje se je li on lokalna varijabla, formalni argument ili članski atribut s obzirom na trenutni primjerak razreda *Scope*.

4.4.4. Razred *ConditionalMeasure*

Za računanje metrike uvjetne kompleksnosti programa je razvijen razred *ConditionalMeasure*. Razred u sebi ima metodu `process_token` i ponaša se kao automat stanja. Slično kao i kod razreda *IdentifierFinder* traže se deklaracije varijable, ali samo onih koje su tipa `bool`.

Ako se pronađe deklaracija `bool` varijable, pripadna varijabla se dodaje u rječnik poznatih `bool` varijabli. Pamćenje poznatih `bool` varijabli je važno jer je moguće da se definicija varijable ne nalazi na istom mjestu kao deklaracija varijable kao na primjeru 4.12.

```
bool a;
.
.
.
a = b && c;
```

Primjer 4.12: Definicija logičke varijable nakon njezine deklaracije

Leksičke jedinke se obrađuju dok se ne nađe uvjetni izraz. Uvjetni izraz se očekuje i prepoznaje u sljedećim oblicima prikazanima na primjeru 4.13.

```
bool b = uvjetni_izraz, c = uvjetni_izraz
b = uvjetni_izraz
if(uvjetni_izraz)
for(...;uvjetni_izraz;...)
while(uvjetni_izraz)
```

Primjer 4.13: Uvjetni izrazi

Kada se uvjetni izraz pronađe, obrađuje se na način opisanom u poglavlju o uvjetnoj kompleksnosti.

U slučaju da se obradi ključna riječ `case` koja pripada `switch-case` konstruktu, ukupnoj kompleksnosti se dodaje vrijednost $2,5$ jer je pretpostavka da je `case` semantički jednak jednostavnoj `if` naredbi.

4.5. Testiranje

Za razvijene programe i algoritme su napravljeni testni slučajevi kako bi se lakše našle greške u programu. Testnih slučajeva ima ukupno 17 i u vrijeme pisanja svi uspješno prolaze. Testni slučajevi $1-2$ testiraju ispravan rad leksičke analize. Testni slučajevi $3-6$ testiraju rad RKR-GST i winnowing algoritma. Testni slučajevi $7-12$ testiraju rad razreda *ConditionalMeasure*. Testni slučajevi $13-17$ testiraju rad razreda *IdentifierFinder, Scope* i *QualityMeasure*. Ovi razredi su testirani zajedno jer ovise jedno o drugom za ispravan rad.

4.5.1. Testiranje leksičke analize

U prvom testnom slučaju se testira ispravno micanje nepotrebnog formatiranja, komentara i interpunkcijskih znakova iz programa. Dva istovjetna programa a.cs i b.cs se uspoređuju s programom Program.cs. Program b.cs dodatno sadrži nepotrebno formatiranje, komentare i interpunkcijske znakove. Očekuje se da će oba programa dati istu ocjenu usporedbe kada se uspoređuju pomoću RKR-GST i winnowing algoritama.

U drugom testnom slučaju se testira ispravna promjena imena varijabli, njihovih pripadnih tipova i vrijednosti u jedinstvene tipove podataka kao na primjeru 4.14. Uspoređuju se programi a.cs i b.cs koji definiraju jednak broj varijabli jednakih tipova, ali s drukčijim imenima i vrijednostima. Očekuje se da će ocjena usporedbe ta dva programa biti 1 za winnowing i RKR-GST algoritam.

```
static int left = 0;
static string name = "snake";
static int up = 2;

static int right = 7;
static string long_name_variable = "different
   string\n";
static int down = 0;

<NUMERIC_TYPE><IDENTIFIER>=<NUMERIC_VALUE>
<STRING_TYPE><IDENTIFIER>=<STRING_VALUE>
<NUMERIC_TYPE><IDENTIFIER>=<NUMERIC_VALUE>
```

Ispis 4.14: Testni slučaj 2

4.5.2. Testiranje RKR-GST i winnowing algoritama

Za RKR-GST i winnowing algoritam su napravljeni testni slučajevi kod kojih se može pretpostaviti koju ocjenu bi algoritmi trebali vratiti.

Kada se uspoređuju dvije kopije istog programa te se iz jedne makne oko 20% programa, očekuje se da oba algoritma daju visoku ocjenu sličnosti oko $0,90$.

Kada se uspoređuju dvije kopije istog programa te se iz jedne makne polovica koda, očekuje se da RKR-GST algoritam daje ocjenu sličnosti oko, ali ispod $0,66$, a da winnowing algoritam daje ocjenu sličnost $0,8 - 1,0$. Ako se u modificiranu kopiju dodatno stavi nepovezan kod, očekuje se da će ocjena sličnosti za oba algoritma pasti razmjerno količini dodanog koda.

4.5.3. Testiranje razreda *ConditionalMeasure*

Testovi za *ConditionalMeasure* provjeravaju ispravno računanje prosječne uvjetne kompleksnosti za jednostavne i komplicirane izraze. U jednostavnijem slučaju se očekuje da će izrazu 4.15 biti dana ocjena 5.

```
bool foo = a && b && c;
```

Ispis 4.15: Testni slučaj 7

U kompliciranijem slučaju se očekuje da će izrazu 4.16 biti dana ocjena $10,5$.

```
bool foo = (a & b) >> c <= d && e || f && !(s.getF()
   + s.getG() >= 77 + 4);
```

Ispis 4.16: Testni slučaj 9

Isto tako se testirala istovjetnost uvjetnih izraza bez obzira nalaze li se u uvjetnim konstruktima ili u definiciji varijable. Očekuje se da će dijelovi koda na ispisu 4.17 imati jednaku ocjenu prosječne uvjetne složenosti.

```
bool first = b + c > 4, second = true, third =
   !raining;

if (int i = 0; b + c > 4; i++) { . . . }
for(int i = 0; true; i++) { . . . }
while(!raining) { . . .}
```

Ispis 4.17: Testni slučaj 11

Naposljetku je razred testiran na programu *Snake* čija je ukupna i prosječna uvjetna složenost poznata.

4.5.4. Testiranje razreda za nalaženje identifikatora

Testiranjem razreda *Scope, IdentifierFinder* i *QualityMeasure* se provjerava nalaženje točnog broja identifikatora i konstrukata. Na ispisu 4.18 je prikazan najjednostavniji testni slučaj. Može se istaknuti da je za pronalaženje identifikatora i ispravan rad implementiranih razreda potrebno pronaći otvorenu vitičastu zagradu i stvoriti primjerak razreda *Scope*. Bez otvorene vitičaste zagrade pronalaženje identifikatora ne bi bilo moguće.

```
{
   int a;
   Random c, d, e;
   float x = 5, y, z = 6;
   f bar = new f();
   bool g, h;
}
```

Ispis 4.18: Testni slučaj 13

Testni slučaj prikazan na ispisu 4.19 je kompleksniji jer je potrebno raspoznati različite vrste identifikatora.

```
struct SomeClass {
   public double first;
   public bool x;
   float third;
   private Member member;
   public SomeClass(double first, bool x) {
      int inner;
      this.first = first;
      this.x = x;
   }

   public Main(string []args) {
      List<Integer> numbers = new List<>();
      int []othernumbers;
   }
}
```

Ispis 4.19: Testni slučaj 15

Isječak programa sadrži dvije metode, jedan razred, četiri članske varijable, tri formalna argumenta i tri lokalne varijable.

Zadnja dva slučaja testiraju da se obradom programa *Snake* nalazi točan broj različitih tipova identifikatora.

4.6. Rezultati

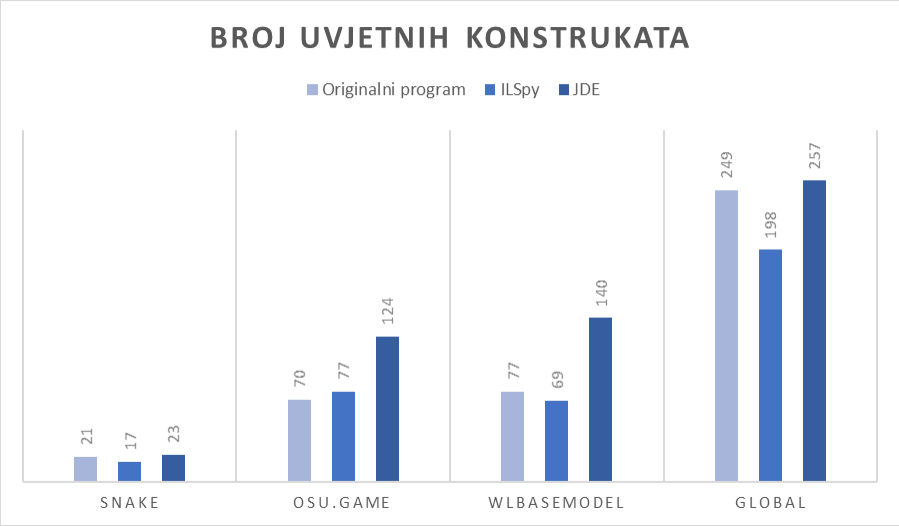
U ovom poglavlju će biti pokazane i komentirane ocjene koje su dobivene pri dekompajliranju odabranih uzoraka programa kada se koriste opisane metrike. Metrike će biti iskorištene na dekompajliranim kodovima generiranim iz neobfusciranih i obfusciranih izvršnih datoteka. Na tablici 4.1 su prikazani i kratko opisani odabrani programi.

Ime programa	Kratak opis	Broj linija	Licenca
Snake [6]	Sadrži logiku za simuliranje igre zmijica.	200	Otvoreni kod
Osu.Game [13]	Sadrži logiku potrebnu za rad popularne igre *osu!*.	1000	MIT licenca
WLBaseModel [15]	Dio koda programskog sustava za rad s bazama podataka.	1000	Otvoreni kod
Global [4]	Metode i atributi potrebni za rad sustava za upravljanje web sadržajem.	4000	MIT licenca

Tablica 4.1: Izvorni kodovi

4.6.1. Broj uvjetnih konstrukata

Na malim programima, kao što je *Snake* dekompajleri generiraju rezultate slične originalu. No, kod ostalih primjera na grafu 4.2 se mogu primijetiti razlike među dekompajlerima. JDE generira mnogo više `if-else` konstrukata nego ILSpy. To ukazuje na mogućnost da JDE složene uvjetne izraze razdvaja u više `if` konstrukata ili da dekompajlira `for` ili `switch-case` konstrukte na neočekivan način. Nakon analize dekompajliranog koda *WLBaseModel* nađeno je više primjera gdje je umjesto `for` petlje generirana istovjetna `while` petlja koja u sebi ima `if` izraze za kontrolu toka. Osim toga, primijećeno je više primjera gdje JDE umjesto originalnog ternarnog izraza generira `if-else` konstrukt.



Slika 4.2: Broj uvjetnih konstrukata

Kod dekompajlera ILSpy je uočena manja količina generiranih `if-else` konstrukata od originalnog *Global* programa. Analizom izvornog koda je uočeno da ILSpy iz jednostavnih `if` izraza koji određuju koja će biti vrijednost jedne varijable generira ternarni izraz. Ovo je prikazano na ispisu 4.20.

```
//Originalni kod
if (strMessage.StartsWith("ERROR")) {
    if (IsInstalled())
            tempStatus = UpgradeStatus.Error;
        else
            tempStatus = UpgradeStatus.Install;
    }

//Dekompajlirani kod
if (text.StartsWith("ERROR")){
    upgradeStatus = ((!IsInstalled()) ?
       UpgradeStatus.Install : UpgradeStatus.Error);
}
```

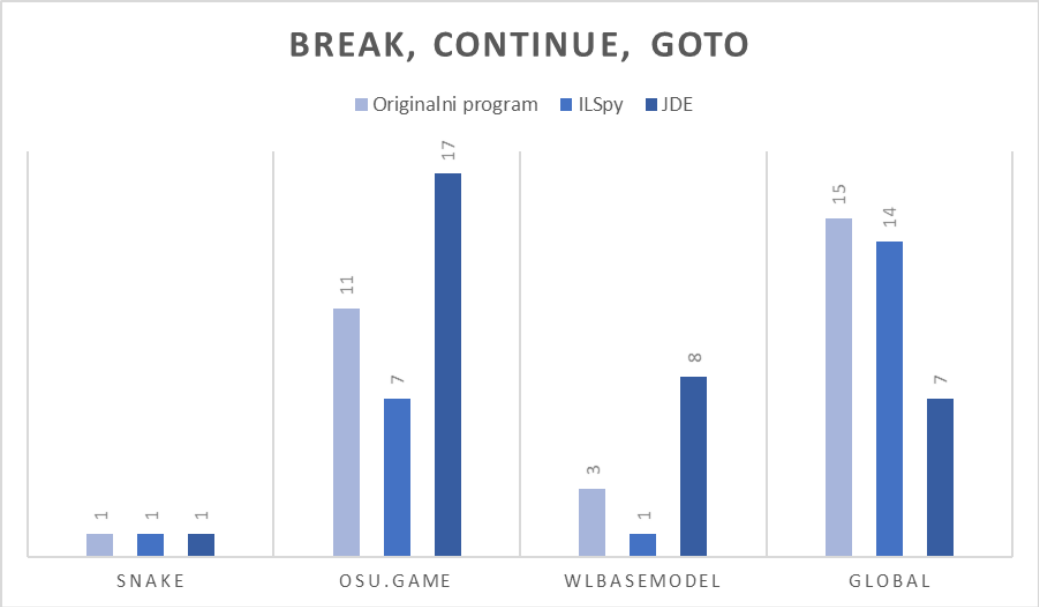
Ispis 4.20: Originalan i ILSpy dekompajliran kod

4.6.2. Break i continue

Kod ove metrike je zanimljivo da JDE nekad generira manje, a nekad više `break` i `continue` naredbi od originalnog programa, kao što je vidljivo na grafu 4.3.

Analizom dekompajliranog *Global* programa je uočeno da JDE `switch-case` konstrukte koji sadrže `break` naredbe mijenja u istovjetne `if` konstrukte bez `break` naredbi. ILSpy na isti način mijenja `switch-case` konstrukte, ali jedino ako su jedine naredbe u `case` konstruktima dodjele vrijednosti varijabli.

S druge strane u slučaju *Osu.Game* originalni program sadrži funkciju koja sadrži `switch` konstrukt s više `case` konstrukata. U svakom `case` konstruktu se vraća logička vrijednost kao povratna vrijednost funkcije. JDE s druge strane generira kod s istim `switch-case` konstruktima, samo što logičku vrijednost pamti u varijabli i generira `break` naredbu za svaki `case` konstrukt da dođe do kraja funkcije gdje vraća vrijednost varijable.



Slika 4.3: break, continue, goto

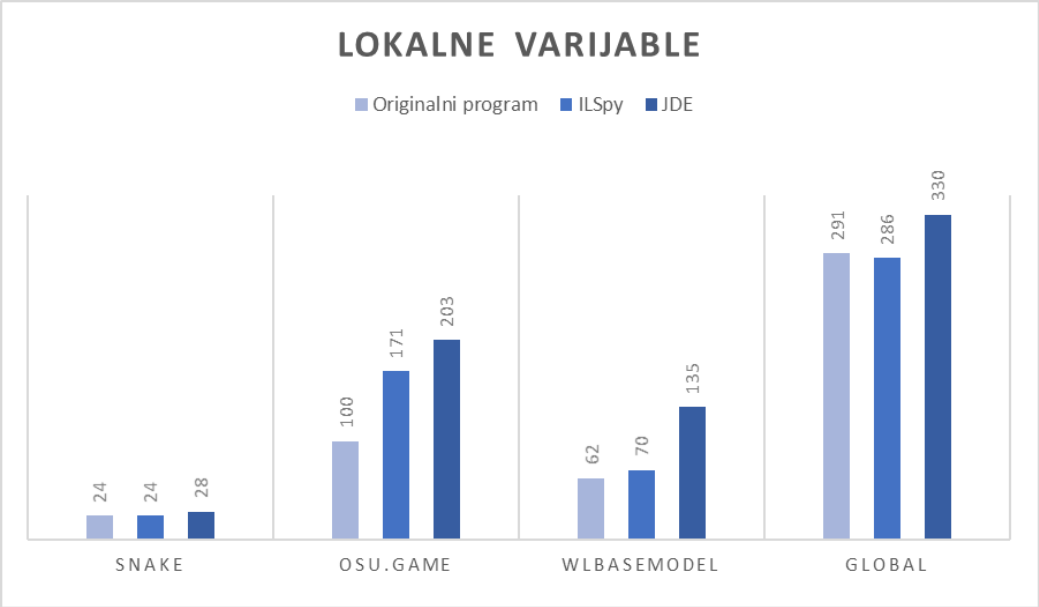
4.6.3. Labele

Nijedan dekompajler nije generirao labele što je i očekivano s obzirom na to da je pretpostavka da labele predstavljaju lošu programersku praksu i kompliciranu kontrolu toka.

4.6.4. Lokalne varijable

Pretpostavka je da su lokalne varijable slab pokazatelj kompleksnosti u programu, no velik broj redundantnih lokalnih varijabla može značajno otežati razumijevanje programa.

Na grafu 4.4 dekompajler JDE generira mnogo više lokalnih varijabli od originalnog programa, dok ILSpy generira rezultat koji je blizak originalu, osim u slučaju *Osu.Game* programa.



Slika 4.4: Lokalne varijable

Kod dekompajlera JDE je uočeno nekoliko problema. Za programe koji nemaju ternarne izraze, funkcije u obliku lambda izraza i `switch-case` konstrukte, najveći izvor redundantnih varijabli su `bool` zastavice. Tako su primjerice 4 dodatne varijable u programu *Snake* `bool` varijable. Još problematičnije, deklaracija varijable često nije blizu definicije varijable. Ovo možemo prikazati na ispisu 4.21.

```
bool flag;
. . .
do
{
    position = new Position();
    flag = (positions1.Contains(position) ? true :
          positions.Contains(position));
}
while (flag);
```

Ispis 4.21: Isječak iz JDE dekompajliranog programa

Osim toga JDE generira redundantne varijable kada se u konstruktima za kontrolu toka vraća vrijednost funkcije te povratnu vrijednost sprema u redundantnoj varijabli.

Kod oba dekompajlera je primijećeno generiranje redundantnih varijabli, ako se

u originalnom programu nalaze ternarni izrazi ili definicije polja što je i prikazano na ispisu 4.22. I u ovim slučajevima je JDE generirao više redundantnih lokalnih varijabli.

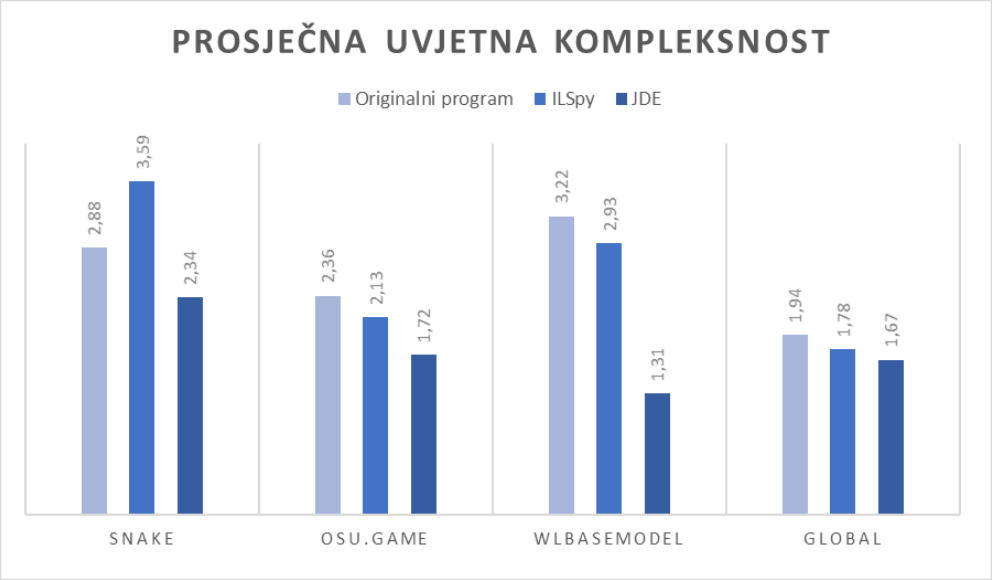
```
//Originalni kod
Item []items = new Item[]{new Item(), new Item()};

//Dekompajlirani kod
Item []items = new Item[2];
Item item = new Item();
items[0] = item;
Item item1 = new Item();
items[1] = item1;
```

Ispis 4.22: Definicija polja u originalnom i dekompajliranom programu

4.6.5. Prosječna uvjetna kompleksnost

Metrika prosječne uvjetne kompleksnosti je zanimljiva jer ukazuje na način na koji dekompajler generira uvjetne konstrukte sa složenim uvjetnim izrazima. Prosječna uvjetna kompleksnost koja je viša od originalne uvjetne složenosti primjerice može ukazivati na agregaciju uvjeta, dok niža prosječna uvjetna kompleksnost ukazuje na suprotan učinak razdvajanja kompleksnog uvjetnog izraza u više jednostavnih. Na grafu 4.5 se mogu vidjeti rezultati mjerenja ovom metrikom.



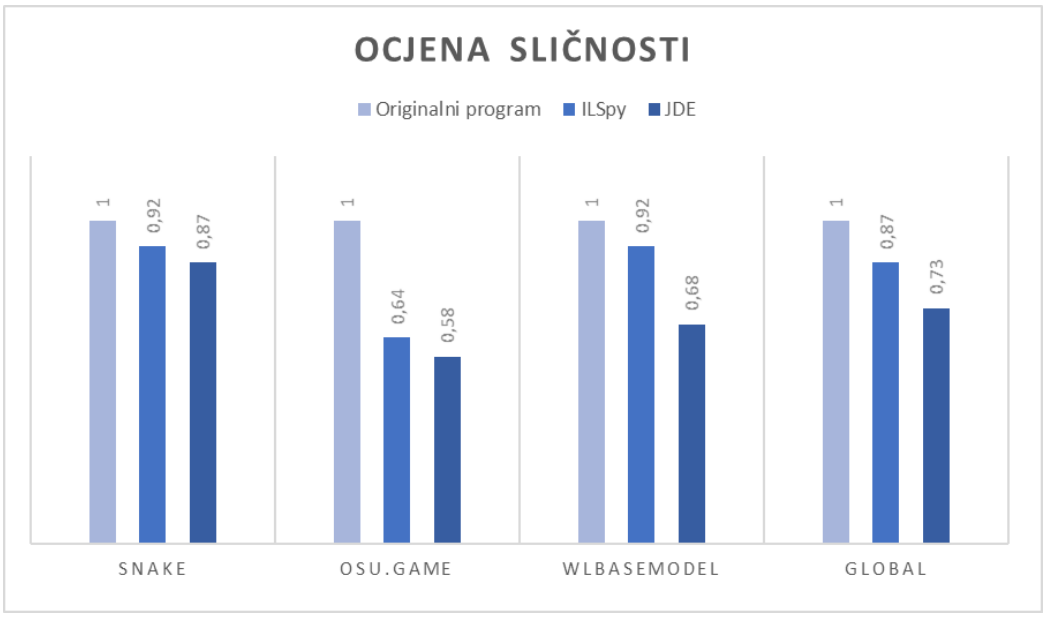
Slika 4.5: Prosječna uvjetna kompleksnost

Analizom dekompajliranog koda *Snake* kojeg generira ILSpy, može se uočiti agregacija jednostavnih izraza u složeni, dok su uzrok relativno niskoj prosječnoj kompleksnosti koda kojeg JDE generira česte redundantne `bool` varijable koje povećavaju broj uvjetnih izraza dok ne doprinose značajno ukupnoj kompleksnosti programa. Ovaj učinak je najprimjetljiviji kod programa *WLBaseModel* gdje se `for` petlje pretvaraju u istoznačne `while` petlje s dvostruko više uvjetnih konstrukata i dodatno se generiraju desetci nepotrebnih `bool` varijabli.

4.6.6. Ocjena sličnosti

Metrika ocjene sličnosti programa može biti jako važan pokazatelj efikasnosti dekompajliranja. Teško je odrediti koja ocjena sličnosti se može nazvati visokom, a koja niskom jer su uočeni slučajevi visoke ocjene sličnosti gdje su nedostajali ključni dijelovi programa i obratno. No, s obzirom na to ocjena sličnosti veća od $0,8$ u većini slučajeva implicira dobro dekompajliranje, dok ocjena ispod $0,7$ implicira da velika količina koda nedostaje ili čak da je dodan redundantni kod. Isto tako je moguće da dekompajliranje generira kod koji je semantički istovjetan s originalnim kodom, ali neprepoznatljiv kada se direktno uspoređuje s originalnim kodom.

Ocjene sličnosti su prikazane na grafu 4.6. Analizom programa s niskom ocjenom sličnosti je moguće uočiti koje konstrukte dekompajler ne može točno dekompajlirati.



Slika 4.6: Ocjena sličnosti

Niska ocjena sličnosti dekompajlera JDE kod programa *WLBaseModel* i *Global* većinom dolazi od rastavljanja `switch-case` konstrukata i složenih `if-else` konstrukata na više manjih i jednostavnijih `if-else` konstrukata. Pošto originalni programi sadrže mnogo takvih izraza, dekompajlirani kod je značajno drukčiji.

Najnižu su ocjenu sličnosti oba dekompajlera dobila kod programa *Osu.Game*. Dekompajleri najvjerojatnije dobivaju nisku ocjenu zbog velike količine definicija polja sličnih onima prikazanim na primjeru 4.22. Tako metoda `loadComplete` koja u izvornom programu sadrži oko 200 linija koda, u dekompajliranim programima sadrži više od 300 linija. Ovo je ujedno i jedini od izvornih kodova kojeg dekompajleri nisu mogli dekompajlirati bez vraćanja greške. Greška se pojavljuje kod funkcija u obliku lambda izraza. Izvorni kod koji nije uspješno dekompajliran, komentar kojeg generira ILSpy i kod kojeg generira JDE su prikazani na ispisu 4.23.

```
ScoreManager.PresentImport = items =>
    PresentScore(items.First());
//IL_0283: Unknown result type (might be due to
    invalid IL or missing references)
//IL_028d: Expected O, but got Unknown
OsuGame.u003cu003ec__DisplayClass56_0 variable =
    null;
```

Ispis 4.23: Primjer greške pri dekompajliranju

4.6.7. Ukupna ocjena

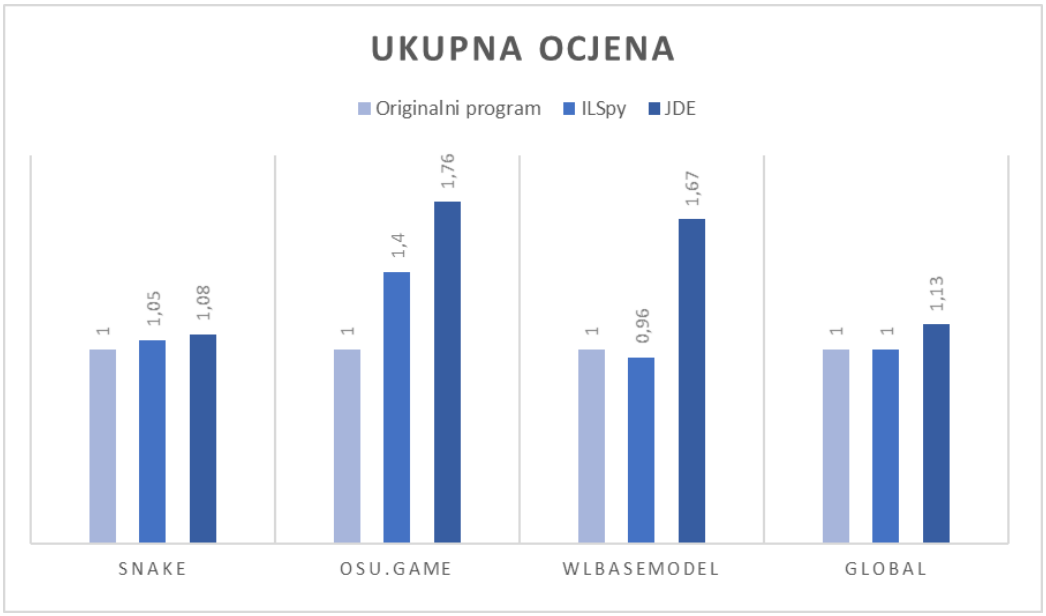
Dobivene metrike za dekompajlirane programe se normaliziraju s obzirom na originalni program i svaka od metrika se množi s konstantom. Te se vrijednosti potom zbrajaju kako bi se dobila ukupna ocjena. Zbroj konstanti s kojima se množe metrike je 1 tako da se za originalan program uvijek dobije ocjena 1. Ukupna ocjena se dobiva na način prikazan na formuli. Sve metrike su normalizirane.

$$\begin{aligned}\text{ocjena\_efikasnosti} = 0.2 \cdot \text{if\_else} + 0.1 \cdot \text{break\_continue} + 0.1 \cdot \text{labele} \\ +0.2 \cdot \text{lokalne\_varijable} + 0.2 \cdot \text{uvjetna\_složenost} + 0.2 \cdot \text{sličnost}\end{aligned}$$

Konstante su određene s obzirom koliko je očekivano da metrike karakteriziraju

program. Tako primjerice metrika broja labela koje dekompajleri rijetko generiraju, a originalni kod rijetko sadrži, se množi s niskom vrijednošću.

Na grafu 4.7 je prikazana ukupna ocjena koju su dekompajleri dobili. Može se uočiti da dekompajler ILSpy generira kod koji je po odabranim metrikama najbliži izvornom programu. Najveći uočeni problem kod ILSpy dekompajlera je generiranje greške pri dekompajliranju funkcije u obliku lambda izraza.

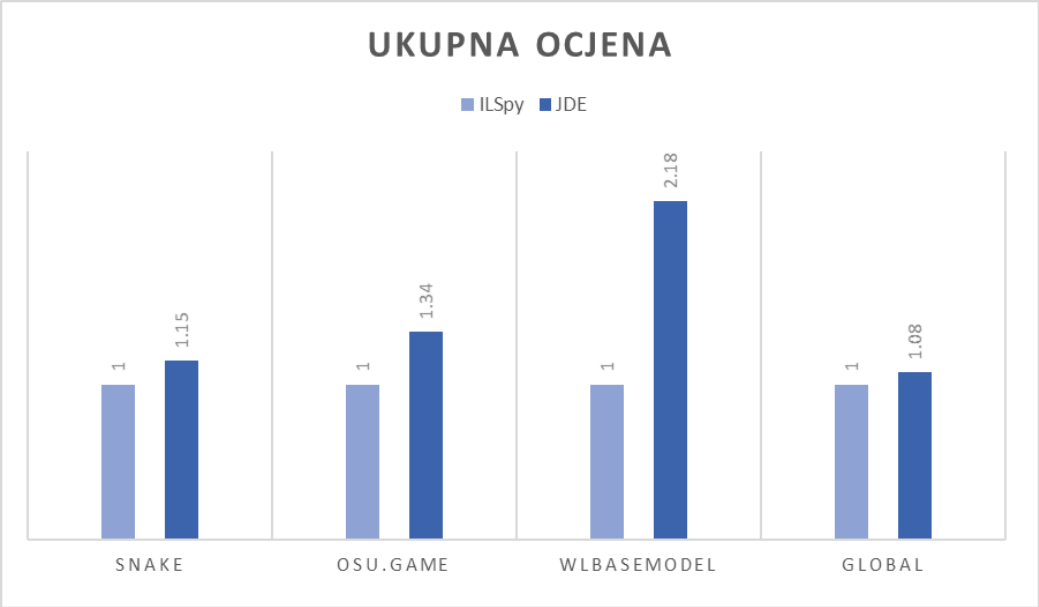


Slika 4.7: Ukupna ocjena

JDE s druge strane općenito generira kod koji je teži za razumjeti od ILSpy koda. Najveći uočeni problemi kod JDE dekompajlera su generiranje nepotrebno velikog broja `if-else` naredbi i `bool` varijabli.

U slučaju da originalan program nije dostupan, efikasnost dekompajliranja se može računati bez metrike sličnosti programa. Ispis na grafu 4.8 je dobiven s vrijednostima prikazanima na formuli.

$$\begin{aligned}\text{ocjena\_efikasnosti} = 0.3 \cdot \text{if\_else} + 0.1 \cdot \text{break\_continue} + 0.1 \cdot \text{labele} \\ +0.3 \cdot \text{lokalne\_varijable} + 0.2 \cdot \text{uvjetna\_složenost}\end{aligned}$$



Slika 4.8: Ukupna ocjena

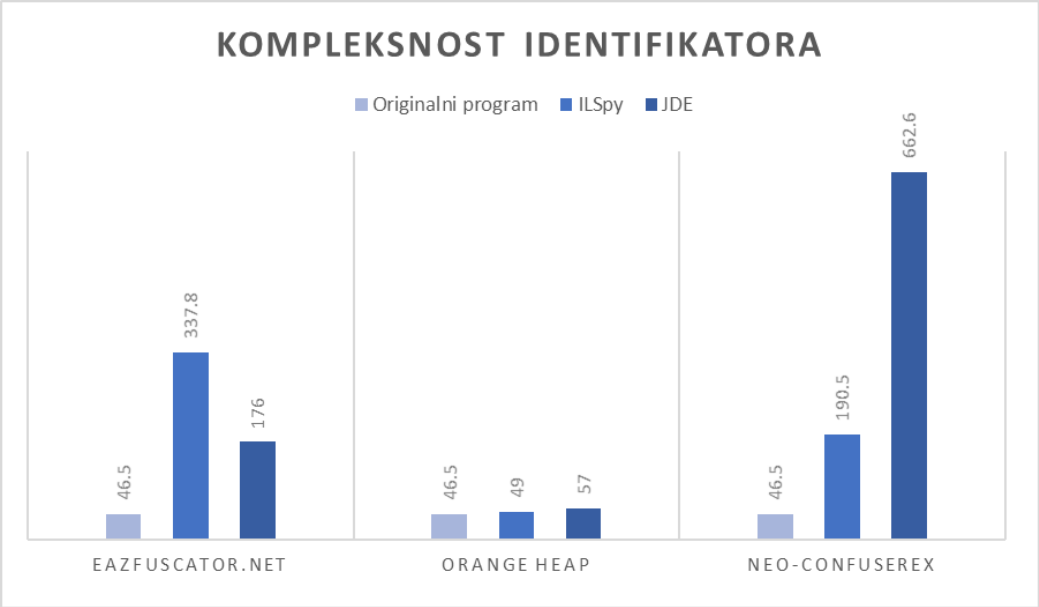
4.7. Obfuscirani programi

U ovom poglavlju će biti prikazani rezultati metrika za ocjenu efikasnosti dekompajlera nakon što je provedena obfuskacija. Obfuscira se program *Snake* koji sadrži oko 200 linija koda koristeći 3 obfuskatora: EazFuscator.NET, Orange Heap i neo-ConfuserEx.

4.7.1. Kompleksnost identifikatora

Kompleksnost identifikatora je metrika na kojoj se lagano može vidjeti učinak obfuskatora čak i na najjednostavnijim primjerima. Visoka ocjena kompleksnosti identifikatora naspram ocjene originalnog programa može implicirati da je obfusciranjem generiran višak klasa, funkcija, članskih atributa i varijabli. Osim toga može implicirati da su imena varijabli promijenjena u kompleksan niz koji sadrži mnogo znakova koji nisu alfanumerici.

Na grafu 4.9 je prikazana metrika kompleksnosti identifikatora za kodove koji su dekompajleri generirali iz obfusciranih izvršnih programa. Dodatno je i prikazana ista metrika za originalni program.



Slika 4.9: Kompleksnost identifikatora

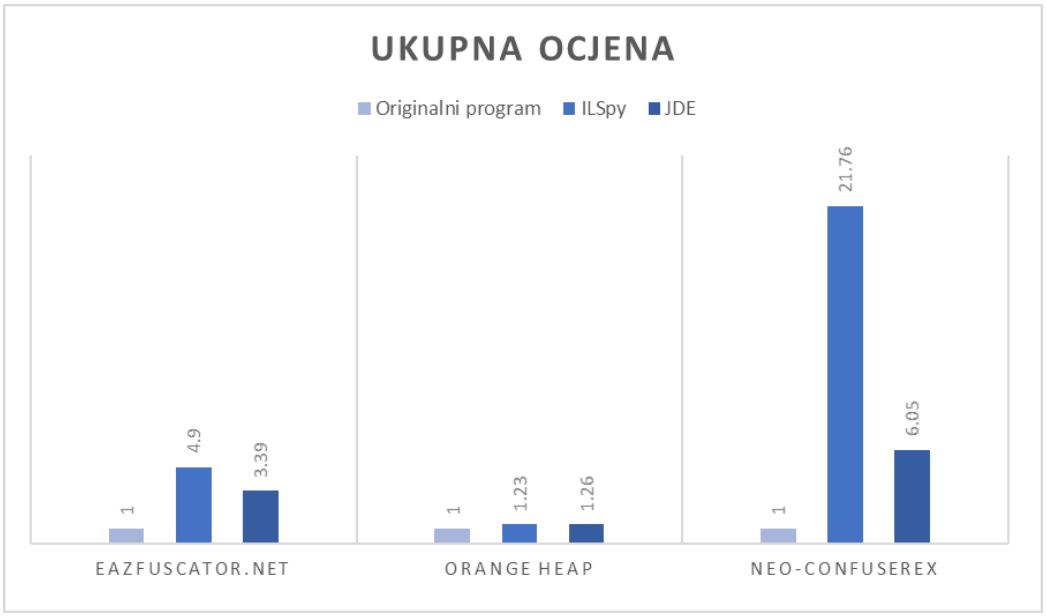
Dekompajliranjem koda obfusciranog alatom EazFuscator, je dobiven kod koji nije nimalo sličan originalnom kodu. ILSpy generira jednu dekompajliranu datoteku, dok JDE generira nekoliko manjih datoteka. EazFuscator tijekom obfuskacije generira mnogo redundantnih varijabla i kompleksnih imena identifikatora razreda.

Obfuskacija alatom Orange Heap nije vrlo kvalitetna jer su samo promijenjena imena tipova podataka, naslov datoteke i sadržaji znakovnih nizova te je ocjena kompleksnosti identifikatora vrlo slična originalnoj.

Alatom neo-ConfuserEx su dobiveni najzanimljiviji rezultati. Program koji je dobiven dekompajliranjem alatom ILSpy ima više od tisuću linija koda, dok program dobiven alatom JDE ima tek 150. Visoka ocjena kompleksnosti identifikatora dobivenog dekompajliranjem alatom JDE proizlazi iz činjenice da su obfuscirani identifikatori sadržavali velik broj znakova koji nisu alfanumerici. Na taj način relativno mali broj identifikatora u dekompajliranom programu ima vrlo visoku ocjenu kompleksnosti, u ovom slučaju njih dvadesetak. Za usporedbu, ILSpy generira stotinjak varijabli, a kompleksnost identifikatora generiranog koda je manja od kompleksnosti identifikatora programa kojeg je generirao JDE. Iz ovoga se može zaključiti da je ILSpy otporniji na obfuskaciju.

4.7.2. Ukupna ocjena

Na grafu 4.10 su prikazani rezultati izračuna ukupne ocjene efikasnosti dekompajliranja nakon obfuskacije prevedenih programa.



Slika 4.10: Ukupna ocjena

Za računanje ukupne ocjene je uključena i metrika sličnosti dva programa. Ona ima velik učinak na ukupnu ocjenu jer su dekompajlirani programi uglavnom jako različiti od originalnog programa. Kada ova metrika ne bi bila korištena, kod dobiven dekompajliranjem pomoću alata JDE bi imao ocjenu koja je puno bliža ocjeni originalnog koda, čak iako je dekompajliranje bilo većinski neuspješno. JDE većinski neuspješno generira kod kada se prevedeni kod obfuscira alatom neo-ConfuserEx.

Kod dobiven dekompajliranjem prevedenog koda obfusciranog alatom EazFuscator sadrži nekoliko puta više `if-else` konstrukata i lokalnih varijabli. Osim toga, generirani kod sadrži komplicirane i neobične izraze koji dodatno otežavaju razumijevanje koda i smanjuju ocjenu sličnosti između dekompajliranog i originalnog programa. Na ispisu 4.24 je prikazan jedan takav neobičan izraz.

```
num6 ^= ~(-(-(~(~((num ^ -1946300104) + num2)))))
```

Ispis 4.24: Izraz kojeg generira obfuskator neo-ConfuserEx

U slučaju alata Orange Heap, struktura dekompajliranih programa i broj varijabli su vrlo slični originalnom programu, a obfuscirani aritmetički izrazi nisu komplicirani.

Orange Heap je alat koji se već nekoliko godina ne održava, ali se pomoću njega može vidjeti koliko su današnji obfuskatori napredniji.

Najbolji rezultati su dobiveni koristeći obfuskator neo-ConfuserEx. Pri obfuskaciji je korisniku omogućeno oko deset mogućnosti načina obfuskacije, primjerice hoće li imena identifikatora biti obfuscirana ili hoće li se generirati nevaljan CIL međukod. Kada su sve opcije uključene, niti jedan dekompajler nije uspješno dekompajlirao obfuscirani kod. Rezultat prikazan na grafu 4.10 je dobiven kada su neke od opcija isključene. JDE nije niti u ovom slučaju generirao program bez javljanja greške tijekom dekompajliranja. Kada ne bi bila korištena metrika sličnosti, JDE bi imao ocjenu vrlo sličnu originalnom programu.

Kod kojeg je generirao alat ILSpy sadrži vrlo složenu kontrolu toka te najveći dio njegove loše ocjene efikasnosti dekompajliranja je uzrokovano velikim brojem `break`, `continue` i `goto` naredbi. U kodu se nalazi i 5 labela, što implicira loše dekompajliranje. Prema prikazanim metrikama, alat Neo-ConfuserEx najbolje obfuscira prevedeni kod.

Dobiveni rezultati ukazuju na to da su obfuskatori važan alat za povećanje sigurnosti i zaštitu aplikacije od dekompajliranja.

5. Zaključak

U ovom radu su implementirane metrike za ocjenu efikasnosti dekompajlera i pokazano je da jednostavne metrike broja konstrukata, uvjetne složenosti i sličnosti programa mogu dobro karakterizirati dekompajlirani program. Primjenom metrika na dva dekompajlera u programskom jeziku C#, JustDecompile Engine i ILSpy je pokazano da loša ocjena dekompajlera s obzirom na ocjenu izvornog koda implicira probleme u dekompajliranju.

Metrika za ocjenu efikasnosti dekompajlera je isprobana i nad obfusciranim prevedenim kodovima i zaključeno je da obfuskatori značajno pridonose povećanju sigurnosti aplikacije. Zaključak je autora da se pokazane metrike mogu koristiti za automatiziranje usporedbe različitih dekompajlera te da dekompajler ILSpy bolje generira kod iz prevedenog koda od dekompajlera JustDecompile Engine.

Mogućnosti za budući rad su poboljšanje trenutnih i dodavanje novih metrika. Osim toga, planira se olakšavanje korištenja skripte i isprobavanje metrika na većem broju dekompajlera i programa koji trenutno nisu dostupni.

6. LITERATURA

[1] Andrew Binstock. Obfuscation: Cloaking your code from prying eyes, 2003. URL https://web.archive.org/web/20080420165109/http://www.devx.com/microsoftISV/Article/11351, pristupljeno 18. 5. 2020.

[2] Compilation process. C# compilation process, 2018. URL https://codeasy.net/lesson/c_sharp_compilation_process, pristupljeno 17. 5. 2020.

[3] Bob Cramblitt. Red gate to charge $35 for .net reflector. URL https://www.businesswire.com/news/home/20110202005972/en/Red-Gate-charge-35-.NET-Reflector, pristupljeno 6.6.2020., 2011.

[4] dnnsoftware. Dnn.platform. URL https://github.com/dnnsoftware/Dnn.Platform/blob/develop/DNN%20Platform/Library/Common/Globals.cs, pristupljeno 6.6.2020.

[5] Jb Evain. Cecil.decompiler 15 dec 2008. URL https://evain.net/blog/articles/2008/12/15/cecil-decompiler/, pristupljeno 6.6.2020., 2008.

[6] Carmelo Garcia. Project: – simple games collection in c#. URL https://code-projects.org/simple-games-collection-in-c-with-source-code/, pristupljeno 6.6.2020., 2019.

[7] Daniel Grunwald. www.danielgrunwald.de. URL http://www.danielgrunwald.de/, pristupljeno 6.6.2020.

[8] Daniel Grunwald. Ilspy - decompiler architecture, 2011. URL https://web.archive.org/web/20160826024646/http://community.sharpdevelop.net/blogs/danielgrunwald/archive/2011/02/19/ilspy-decompiler-architecture.aspx, pristupljeno 20. 5. 2020.

[9] James Hamilton i Sebastian Danicic. An evaluation of current java bytecode decompilers. U *2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*, stranice 129–136. IEEE, 2009.

[10] icsharpcode. Ilspy. URL https://github.com/icsharpcode/ILSpy, pristupljeno 6.6.2020.

[11] Eric Lippert. How many passes?, 2010. URL https://docs.microsoft.com/en-us/archive/blogs/ericlippert/how-many-passes, pristupljeno 17. 5. 2020.

[12] N.A. Naeem, M. Batchelder, i Laurie Hendren. Metrics for measuring the effectiveness of decompilers and obfuscators. stranice 253 – 258, 07 2007. ISBN 0-7695-2860-0. doi: 10.1109/ICPC.2007.27.

[13] ppy. osu. URL https://github.com/ppy/osu/blob/master/osu.Game/OsuGame.cs, pristupljeno 6.6.2020.

[14] John Robbins. Pdb files: What every developer must know, 2014. URL https://www.wintellect.com/pdb-files-what-every-developer-must-know/, pristupljeno 23. 5. 2020.

[15] Ryan Samiee. Weblight 2.00. URL https://www.codeproject.com/Articles/436341/WEBLIGHT-2-00-OpenSource-Component-Library-for-MVC, pristupljeno 6.6.2020., 2012.

[16] Saul Schleimer, Daniel S Wilkerson, i Alex Aiken. Winnowing: local algorithms for document fingerprinting. U *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, stranice 76–85, 2003.

[17] David Srbecký. .net decompiler. URL https://github.com/icsharpcode/ILSpy/blob/f1021d18afc6548fd9396b5d449aacec46297311/doc/Dissertation/Dissertation.pdf, pristupljeno 6.6.2020., 2008.

[18] Siniša Srbljić. *Prevođenje programskih jezika*. Element, 2009.

[19] telerik. Justdecompileengine. URL https://github.com/telerik/JustDecompileEngine, pristupljeno 6.6.2020.

[20] Zoran Đurić i Dragan Gašević. A source code similarity system for plagiarism detection. *The Computer Journal*, 56(1):70–86, 2013.

[21] Michael J Wise. Neweyes: a system for comparing biological sequences using the running karp-rabin greedy string-tiling algorithm. U *ISMB*, stranice 393–401, 1995.

Ocjena kvalitete dekompajliranog C# koda

Sažetak

Za nalaženje sigurnosnih propusta u aplikaciji se često provodi postupak penetracijskog testiranja kod kojeg se koriste dekompajleri. Za olakšavanje ovog postupka potrebno je razviti metrike za ocjenjivanje efikasnosti dekompajlera koje se mogu automatizirati. U tu svrhu su implementirane jednostavne metrike s kojima se ocjenjuje efikasnost dekompajliranja dva C# dekompajlera i pokazuje se da metrike mogu ukazivati na dobro dekompajliranje, ali i na ozbiljne nedostatke u dekompajliranju prevedene datoteke.

Ključne riječi: kompajliranje, dekompajliranje, obfuskacija, efikasnost, C#, metrike

Assessing quality of decompiled C# code

Abstract

A technique called penetration testing is often used to find security vulnerabilities in applications. One of the most important tools for penetration testing are decompilers. To make this process easier, it's crucial to develop automated metrics for assessing decompiler quality. To achieve that goal, an implementation of metrics for assessing the quality of two C# decompilers is provided. It is shown that even simple metrics can be used to indicate good or error-prone decompilation.

Keywords: compiling, decompiling, obfuscation, quality, C#, metrics