SVEUČILIŠTE U ZAGREBU

FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA I RAČUNARSTVA

Ocjena kvalitete dekompajliranog Java koda

Tarik Karamehmedović

Mentor: doc. dr. sc. Stjepan Groš

Zagreb, svibanj 2020.

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

Sadržaj

1. Uvod

U svijetu kibernetičke sigurnosti velika je potreba za konstantnom provjerom sigurnosti samih aplikacija u procesu koji se naziva penetracijsko ispitivanje. Ali izvorni kod aplikacije nam često niti nije dostupan, a analizu dalje otežavaju određene metode kojima se pokušava spriječiti taj postupak. U tu svrhu razvijeni su alati koji bi pri tome mogli pomoći, a od kojih su najbitniji dekompajleri.

Dekompajleri su programi koji analiziraju dobiveni prevedeni kod te iz njega pokušaju rekonstruirati originalni programski kod. S obzirom na popularnost jezika kao što je Java, za očekivati je da će se za njega također i pokušati razviti dobri dekompajleri. Ako planiramo koristiti dekompajlere za testiranje, potrebno je odrediti koji od tih dekompajlera će dati najbolju reprezentaciju originalnog koda. A nekada je potrebno odrediti kvalitetu dekompajliranog koda i u slučaju kada originalni kod nije na raspolaganju. U tu svrhu mogu se koristiti određene metrike kako bi se odredila općenita kvaliteta koda i njegova razumljivost.

Proces testiranja, izvlačenja rezultata te subjektivnog pregledavanja rezultata bi mogao biti spor i neefikasan proces. Zbog toga u sklopu ovog rada razvijena je skripta koje će automatizirano pregledati kod originalnog i dekompajliranog koda te koristeći određene metrike usporediti sličnost. Na kraju skripta će dati ukupnu ocjenu koja reprezentira ukupnu kvalitetu dekompajliranog koda. U slučaju kada originalni kod nije dostupan međusobno će se uspoređivati dvije dekompajlirane datoteke uz pomoć metrika kako bi se automatski moglo odrediti koji kod je jednostavniji, efikasniji ili lakši za razumjeti.

Ovaj rad je fokusirati na ponašanje određenih dekompajlera, algoritama i metrika kako bi u konačnici mogla biti dana ocjena kvalitete dekompajliranog koda. Kao glavna metrika za usporedbu koristi se winnowing algoritam za usporedbu tekstova, a ocjena se dalje modificira pomoću Halsteadovih metrika i fizičkih svojstava koda kako bi se odredila ukupna sličnost programa. Za metrike općenite kvalitete koda koristit će se mjere složenosti Halsteadove metrike te kompleksnost kontrole toka koda.

Rad je strukturiran na sljedeći način. U drugom poglavlju je objašnjeno kompajliranje, dekompajliranje i Javina ranjivost na dekompajlere. U trećem poglavlju će se definirati korištene metrike za ocjenu kvalitete i sličnosti koda te je definiran proces ocjene. U četvrtom poglavlju će biti pokazano na koji način su metrike i algoritmi implementirani, nakon toga će biti prikazani rezultati dekompajliranja. Potom će biti razmotreni slučajevi u kojima dekompajleri nisu uspjeli dekompajlirati cijeli kod programa te rezultati dekompajliranja obfuciranog koda.

2. Kompajliranje i dekompajliranje u jeziku Java

2.1. Kompajliranje

Općenito, kompajliranje je proces u kojem se programski kod nekog izvornog jezika prevodi u ciljni jezik, a kompajler je program koji izvodi taj proces. Proces kompajliranja razdijeljen je u fazu analize i fazu sinteze. Faza analize sadrži leksičku, sintaksnu i semantičku analizu. U leksičkoj analizi se prolazi kroz kod te se stvara niz leksičkih jedinki izvornog jezika i usput se provjerava jesu li one ispravne. Generirane leksičke jedinke se dalje šalju na sintaksnu analizu gdje se provjerava je li njihova struktura ispravna. Na kraju u semantičkoj analizi se provjerava jesu li dobivene operacije valjane, tj. jesu li one semantički točne. Ako su sva tri koraka prošla bez greške kompajler nastavlja s prevođenjem koda u ciljni jezik, tj. s fazom sinteze.

Datoteke kompajlirane u Javi se kompajliraju u Java bajtkod kojeg onda Javin virtualni stroj (engl. Java Virtual Machine, JVM) izvršava. Bajtkod je strojno nezavisni međukod koji će se pomoću JVM-a prevoditi u strojno zavisni kod ovisno o tome na kojoj platformi se kod izvršava. Java svoj prevedeni kod sprema u *Class* datoteke. *Class* datoteka se sastoji od više JVM instrukcija a jedna JVM instrukcija se sastoji od jednog okteta koji reprezentira operacijski kod, tj. dio koda koji opisuje ponašanje jedne operacije zajedno s nula ili više okteta za operande.

Proces kompajliranja kreće tako da Javin jezični procesor čita izvorni kod iz .java datoteka te pretvara dobiveni niz leksičkih jedinki u čvorove apstraktnog sintaksnog stabla (engl. Abstract Syntax Tree, AST). Svi vidljivi simboli za definicije se stavljaju u kompajlerovu tablicu simbola. Događa se razlučivanje imena i provjera tipova. Potom se analizira tok podataka na sintaksnim stablima kako bi se provjerile dodjele i doseg imena. Sintaksno stablo se prerađuje te se prevode određene sintaksne pokrate. Nakon toga se stvaraju *Class* datoteke u koje se sprema generirani bajtkod [1].

Dio informacija koji se nalazio u izvornom kodu se tijekom procesa kompajliranja izgubi. Na primjer, prije same kompilacije pretprocesor izbacuje nepotrebne dijelove koji se nalaze unutar koda kao što su komentari dok će postprocesor reorganizirati kod, maknuti redundancije te napraviti ostale moguće optimizacije kako bi povećao efikasnost koda. Ovo se može pokazati na primjeru generiranog operacijskog koda. Na primjeru Ispis 2-1 nije sigurno hoće li uvjet biti ispunjen te je zbog toga kompajler odlučio zadržati sve informacije. To se može vidjeti na operacijskom kodu u primjeru Ispis 2-2 gdje je cijeli kod napisan u Javi preveden u operacijski kod. Konkretno linije 8 i 9 predstavljaju operaciju koja sprema vrijednost u varijablu `i` koja je unutar `if` dijela uvjetne naredbe dok se na linijama 13 i 14 obavlja operacija spremanja vrijednosti u varijablu `i` koja se nalazi unutar `else` dijela uvjetne naredbe.

```
public static void main(String[] args) {
      int i = args.length;
      if (i > 2){       // komentar
            i = 1;
      } else {
            i = 2;
      }
}
```

Ispis 2-1 - Primjer programa s uvjetom koji se ne može razriješiti tijekom procesa kompajliranja

```
public static void main(java.lang.String[]);
    Code:
       0: aload_0
       1: arraylength
       2: istore_1
       3: iload_1
       4: iconst_2
       5: if_icmple     13
       8: iconst_1
       9: istore_1
      10: goto          15
      13: iconst_2
      14: istore_1
      15: return
}
```

Ispis 2-2 – Prevedeni kod iz ispisa 2-1

Sad ako se uvjet promijeni tako da je on uvijek istinit, kao što je prikazano na primjeru Ispis 2-3, kompajler će izbaciti dio koda za koji zna da se nikada neće izvršiti. To se može vidjeti na primjeru Ispis 2-4 gdje nije preveden dio koda za koji je poznato da se neće izvršiti, tj. sada se unutar prevedenog koda nalazi samo jedna operacija spremanja vrijednosti u varijablu što je prikazano na linijama 3 i 4.

```
public static void main(String[] args) {
      int i = args.length;
      if (true){        // komentar
            i = 1;
      } else {
            i = 2;
      }
}
```

Ispis 2-3 - Program s uvjetom koji je uvijek istinit

```
public static void main(java.lang.String[]);
Code:
   0: aload_0
   1: arraylength
   2: istore_1
   3: iconst_1
   4: istore_1
   5: return
```

Ispis 2-4 – Prevedeni kod iz ispisa 2-3

2.2. Dekompajliranje

U procesu dekompajliranja cilj je generirati kod koji je što sličniji originalnom kodu koristeći informacije koje se nalaze unutar kompajliranih datoteka. Ali, zbog informacija koje se gube u procesu kompajliranja, to skoro nikada nije moguće. Nužno je spomenuti kako će razumijevanje dekompajliranog koda uvijek biti teže nego razumijevanje originalnog koda.

U tablici Tablica 2-1 se može vidjeti koje su sve informacije sadržane unutar Javine *Class* datoteke, tj. koje su sve informacije dostupne dekompajlerima.

Magic number	heksadecimalna konstanta 0xCAFEBABE koji govori JVM-u da prima .class datoteku
Minor and Major version	brojevi koji govore JVM-u u kojoj verziji Jave je program bio kompajliran
Constant Pool Count	broj elemenata koji se nalazi u *Constant pool*
Constant Pool	niz elemenata promjenljive duljine u kojem su sadržane konstante i varijable koje se koriste unutar programa uključujući vrijednosti poput nizova tekstova, brojeva, imena metoda i sličnih podataka.
Access Flags	daje informaciju radi li se o razredu ili o sučelju i informacije o njihovim modifikatorima
This class	ukazuje na indeks elementa *constant pool* koji opisuje trenutačni razred
Super class	ukazuje na indeks elementa *constant pool* koji opisuje roditeljski razred
Interface count	broj sučelja koja se koriste za trenutačni razred
Interfaces	niz indeksa koji ukazuju na *constant pool* gdje su ta sučelja opisana
Fields	niz elemenata promjenljive duljine koja opisuju polja korištena u trenutačnom razredu, primjerice članske varijable
Methods	niz elemenata koji opisuje metode sadržane unutar razreda
Attributes	niz elemenata koji opisuje atribute trenutačnog razreda

Tablica 2-1 - Informacije sadržane unutar .class datoteke

Bitno je spomenuti kako je *Constant pool* jedno od značajnijih polja jer ga druge strukture koriste kako bi se referencirale na podatke koji se nalaze u njemu. Polje *Methods* je

vjerojatno i najbitnije jer je u njemu sadržan izvorni kod u obliku bajtkoda, okteta koji opisuju operacije.

Ukratko, dekompajler može koristeći ove dostupne informacije pronaći metode, proći kroz sve njene atribute koje će biti zapisane unutar *Methods* polja kako bi formirao definiciju metode, nakon toga može pronaći njenu implementaciju zapisanu u obliku okteta. Ti okteti predstavljaju operacije koje se trebaju izvršiti i na kraju dekompajler može koristeći svoju analizu dobivenih operacija rekonstruirati kod u Javi [2].

U ovom radu će biti razmatrani dekompajleri Procyon, CFR te Fernflower. U vrijeme pisanja rada sva tri dekompajlera se aktivno održavaju te konstantno izdaju nove verzije kako bi se podržavala najnovija verzija Jave.

2.3. Obfuskatori

Popularnost Jave te količina informacija o originalnom programu koje su dostupne unutar njenih kompajliranih datoteka čine ju vrlo primamljivom metom za dekompajliranje. Čak i bez komentara, dokumentacije i originalnih imena varijabli, uz dovoljan trud i uloženo vrijeme netko će iz dekompajliranog koda izvući ono što mu treba. Pravna i etička pitanja dekompajliranja su uvijek aktualna, dok je konkretni kod možda zaštićen kao intelektualno vlasništvo [3], odnosno autorsko pravo, ideje koje stoje iza tog koda ne mogu biti zaštićene [4]. Tako da ako netko želi proučiti konkretnu implementaciju nekog algoritma ili svojstva određenog programa, na njegovom putu ga ništa ne može spriječiti da to i učini ali se može otežati korištenjem obfuskatora. Obfuskatori su programi koji zamršuju originalni bajtkod sadržan unutar Javinih *Class* datoteka. Iako će uspješno dekompajliranje potpunog programa biti otežano, oni u većini slučajeva neće obraniti kod od samog procesa dekompajliranja. Glavni cilj obfuskatora je otežati cijeli proces reverznog inženjerstva jer će dekompajlirani kod biti znatno teži za razumijevanje.

Taktike koje obfuskator može primijeniti kako bi otežao proces reverznog inženjerstva su primjerice uklanjanje informacija za pronalaženje grešaka u kodu i promjena imena identifikatora. Napredniji obfuskatori provode i promjenu kontrole toka koda te ubacivanje nepotrebnog koda koji se nikada niti neće izvršiti [5].

Osim zaštite koda obfuskatori imaju i drugih funkcionalnosti, a to je optimizacija bajtkoda u smislu brzine izvođenja i veličine samog programa. Dok je optimizacija brzine upitna, optimizacija veličine će se itekako moći vidjeti.

U ovom radu će se razmatrati obfuskator ProGuard. On provodi operacije uklanjanja informacija za pronalaženje grešaka u kodu, promjene imena te eliminacije neiskorištenog koda. Također radi optimizaciju veličine izvršne datoteke.

3. Ocjenjivanje efikasnosti dekompajlera

Potrebno je pronaći neke metrike kojima se može reprezentirati koliko je neki kod sličan nekom drugom kodu. Tu se već javljaju algoritmi koji direktno uspoređuju dva niza znakova. Način rada takvih algoritama je da jednostavno traže najdulji niz jednakih znakova te na taj način računaju postotak sličnosti. Problem kod takvog načina usporedbe je da transformacije koje neki dekompajler može provesti nad kodom mogu značajno smanjiti sličnost kodova. Na primjer, jednostavna promjena poretka nekih metoda u dekompajliranom kodu bi dala znatno manju sličnost iako semantika programa nije promijenjena.

Neke od promatranih promjena u dekompajliranim kodovima koje ne mijenjaju semantiku programa:

- Dodavanje ključne riječi `final` na varijable kojima se vrijednost neće kasnije mijenjati,
- Dodavanje reference `this` kako bi se osiguralo referenciranje na trenutačni razred,
- Umjesto korištenja punog naziva neke biblioteke, promatrani dekompajleri su sve korištene biblioteke uglavnom navodili na razini razreda te bi onda skraćenim nazivom referencirali na biblioteku,
- Spajanje uzastopnih uvjetnih naredbi u jednu.

```
jPanel1 = new javax.swing.JPanel();
...
setDefaultCloseOperation(javax.swing.WindowConstants.EXIT_ON_CLOSE);
...
javax.swing.GroupLayout jPanel2Layout = new
      javax.swing.GroupLayout(jPanel2);
```

Ispis 3-1 - Originalni kod

```
this.jPanel1 = new JPanel();
...
this.setDefaultCloseOperation(3);
...
final GroupLayout jPanel2Layout = new GroupLayout(this.jPanel2);
```

Ispis 3-2 - Dekompajlirani kod

3.1. Korištene metrike za sličnost i kvalitetu

Za metodu ocjenjivanja sličnosti dekompajliranog koda odabrani su winnowing algoritam te uspoređivanje prema fizičkim svojstvima koda i Halsteadovim metrikama koda. Za ocjenu kvalitete koda koristite se kompleksnost kontrole toka i Halsteadove metrike.

Ocjena kompleksnosti kontrole toka pokazuje na koji način su uvjetne naredbe generirane te koliko ih je teško razumjeti [6].

U fizičkim svojstvima koda se broje linije, riječi i znakovi u programu [7]. To daje generalnu reprezentaciju veličine koda. Kako bi se smanjila nepotrebna razlika između originalnog i dekompajliranog koda metrika ne broji sadržaj unutar komentara.

Halsteadove metrike su metrike koje mjere svojstva nekog koda i njihove relacije [8]. One se uobičajeno koriste za procjenu kompleksnosti koda tako da proizvode neku metriku koja daje generalnu ideju o tome koliko je teško razumjeti taj kod, no pokazano je da se one mogu koristiti i za međusobnu usporedbu kodova [7]. Halsteadove metrike kao početna svojstva koriste operatore i operande sadržane unutar koda. Konkretne definicije tih svojstava su detaljnije definirane u 4. poglavlju.

Početna svojstva koda koja se koriste za izračun Halsteadovih metrika su sljedeća:

- η_1 – Broj različitih operatora,
- η_2 – Broj različitih operanada,
- N_1 – Ukupan broj operatora,
- N_2 – Ukupan broj operanada.

Iz tih elementarnih svojstava mogu se izračunati druge metrike koda koje je Halsteadova metrika definirala. U ovom radu se koriste Halsteadove metrike vokabulara, duljine, volumena i težine.

Halsteadova metrika vokabulara se dobiva zbrajanjem svih različitih operatora i operanda:

$$\eta = \eta_1 + \eta_2$$

Metrika duljine predstavlja ukupan broj operatora i operanada:

$$N = N_1 + N_2$$

Metrika volumena se računa sljedećom formulom:

$$V = N \cdot \log_2 \eta$$

Volumen se može gledati kao količina koda, tj. koliko informacija čitatelj mora apsorbirati kako bi se shvatila semantika koda. Njome se izražava kako ukupan broj operatora i operanada više utječe na kompleksnost koda koju ova metrika definira nego broj jedinstvenih operatora i operanada.

Halsteadova težina je definirana na sljedeći način:

$$D = \frac{\eta_1}{2} \cdot \frac{N_2}{\eta_2}$$

Metrikom se opisuje koliko je truda potrebno uložiti kako bi se kod mogao razumjeti.

Winnowing algoritam je algoritam za računanje digitalnog otiska [9]. Njegova primarna svrha je detekcija plagijata te se zahvaljujući nekim njegovim prednostima može iskoristiti i za provjeru sličnosti kodova. Pošto algoritam implementira listu otisaka koje je stvorio otporan je na promjenu poretka nizova znakova. Također, male promjene u nizovima koji se uspoređuju neće imati utjecaj na sličnost.

Algoritam originalni tekst pretvara u takozvane k-grame, gdje je *k* duljina podnizova. Nakon obrade k-grama koristi se klizeći prozor duljine *w* koji na određeni način bira k-grame koji na kraju tvore digitalni otisak.

Na sljedećem primjer je opisan postupak stvaranja digitalnog otiska:

```
The brown fox jumped
```

Prvo se obrađuje početni tekst:

```
thebrownfoxjumped
```

Neka je k = 7, time će se dobiti sljedeći k-grami:

```
thebrow hebrown ebrownf brownfo rownfox ownfoxj wnfoxju
nfoxjum foxjump oxjumpe xjumped
```

Nakon toga, nad k-gramima se provodi funkcija sažimanja čime se dobiva niz sažetaka:

```
316 524 381 618 547 186 561 538 434 269 536
```

U zadnjem koraku se u obzir uzimaju samo određeni sažeci koje je stvorila funkcija sažimanja. Prozor se slijedno kreće po vrijednostima koje je stvorila funkcija sažimanja te gleda minimalne vrijednosti unutar prozora, a ako postoji više vrijednosti s minimalnom vrijednošću, odabire se najdesnija vrijednost.

Neka veličina prozora bude w = 3, dobiti će se sljedeći prozori:

```
(316 524 381) (524 381 618)
(381 618 547) (618 547 186)
(547 186 561) (186 561 538)
(561 538 434) (538 434 269)
(434 269 536)
```

Te nakon odabira otisaka ostaju sljedeći sažeci koji tvore digitalni otisak:

```
316, 381, 186, 434, 269
```

3.2. Proces ocjenjivanja

Ocjena dekompajliranih kodova podijeljena je na dvije cjeline, ocjene kvalitete kodova te ocjena sličnosti između originalnog i dekompajliranog koda.

3.2.1. Ocjena kvalitete

Prva metrika koja služi za ocjenu kvalitete koda je ocjena kompleksnosti kontrole toka. Metrika u programu traži naredbe kontrole toka. Kako se naredbe nalaze tako se i računa težina njihovih uvjeta po uzoru na rad *Metrics for Measuring the Effectiveness of Decompilers and Obfuscators* [6]. Ako se radi o naredbama prisilne kontrole toka (`break` i `continue`), njima će biti dodijeljena težinska vrijednost od 2. Ako se radi o uvjetnim naredbama kontrole toka (`if`, `else if`, `switch case`), težina pojedine naredbe se određuje s obzirom na to koliko varijabla i uvjetnih operatora ta naredba sadrži. Težine pojedinih elemenata uvjetne naredbe kontrola toka su određene na sljedeći način:

- Varijable imaju težinsku vrijednost od 1,
- Pozivi metoda imaju težinsku vrijednost od 1,
- Logički operatori i i ili (`&&` i `||`) imaju težinsku vrijednost od 1,
- Uvjetni operatori poput <, >, <=, >=, ==, !=, ! imaju težinsku vrijednost od 0.5.

Prva datoteka dobiva fiksnu ocjenu 1, a druga datoteka se onda uspoređuje naspram nje. Ako dobije ocjenu veću od 1, onda druga datoteka ima jednostavniju kontrolu toka.

Ocjena druge datoteke se računa sljedećom formulom:

$$ocjena\_drugog = \frac{kompleksnost\_prvog}{kompleksnost\_drugog}$$

Drugi dio ocjene kvalitete koda određuju Halsteadove metrike. Za ocjenu kvalitete kodova, koriste se Halsteadove metrike volumena i težine. Ocjena se računa tako da se nakon izračuna potrebnih metrika gleda koja od datoteka je dobila bolji rezultat. Bolji rezultat se uzima kao baza te se lošijoj datoteci daje ocjena naspram boljoj. Ocjena lošije datoteke se računa preko omjera određenog svojstva. U oba slučaja, veća vrijednost metrike predstavlja lošiji rezultat:

$$ocjena\_volumena\_losijeg = \frac{volumen\_boljeg}{volumen\_losijeg}$$

$$ocjena\_tezine\_losijeg = \frac{tezina\_boljeg}{tezina\_losijeg}$$

3.2.2. Sličnost kodova

Prvi dio ocjene sličnosti obavlja se pomoću winnowing algoritma. Prvo se prema winnowing algoritmu određuju digitalni otisci. Nakon toga sličnost se gleda po broju otisaka koji se nalaze u oba skupa stvorenih otisaka. Ocjena sličnosti je određena sljedećom formulom:

$$winnowing\_slicnost = \frac{2 \cdot velicina\_presjeka}{velicina\_unije}$$

Drugi dio ocjene se računa pomoću Halsteadvih metrika. Kako bi bila određena sličnost koristeći Halsteadove metrike uzimaju se metrike vokabulara i duljine. Nakon što se izračunaju vrijednosti tih metrika, sličnost se gleda kao relativna udaljenost svake pojedinačne metrike te se računa omjerom vrijednosti metrika:

$$udaljenost\_vokabulara = \frac{manja\_vrijednost\_vokabulara}{veca\_vrijednost\_vokabulara}$$

$$udaljenost\_duljine = \frac{manja\_vrijednost\_duljine}{veca\_vrijednost\_duljine}$$

Računa se prosjek relativnih udaljenosti te se tom ocjenom definira sličnost Halsteadovog profila:

$$halsteadova\_slicnost = \frac{udaljenost\_vokabulara + udaljenost\_duljine}{2}$$

Na kraju, u ocjenu sličnosti se uzimaju fizička svojstva programa. U oba programa se broji broj linija, riječi te znakova. Nakon toga sličnost se gleda kao relativna udaljenost svake pojedinačne vrijednosti u oba programa. Relativna udaljenost se računa omjerom njihovih vrijednosti:

$$relativna\_udaljenost = \frac{manja\_vrijednost}{veca\_vrijednost}$$

Nakon izračuna svih relativnih udaljenosti uzima se njihov prosjek te se tim tvori sličnost fizičkog profila:

$$fizicka\_slicnost = \frac{udaljenost\_linija + udaljenost\_rijeci + udaljenost\_znakova}{3}$$

Ukupna sličnost se računa težinskim prosjekom koristeći izračunate sličnosti winnowing algoritma, Halsteadovog i fizičkog profila. Kako je winnowing algoritam jedini način usporedbe koji konkretno gleda po kodu, on ima težinsku vrijednost od 60%. Njegov jedini nedostatak je to što u nekim slučajevima može biti prestrog te pretjerano sniziti ocjenu. Kako bi nadoknadili za taj nedostatak, ocjena se modificira Halsteadovim i fizičkom profilom. Pošto se Halsteadov profil pokazao vrlo konzistentnim te ima manju vjerojatnost krivih pozitivnih rezultata od fizičkog profila, on ima težinsku vrijednost od 30%. Te na kraju kako

je fizički profil najpodložniji krivim pozitivnim rezultatima, on ima težinsku vrijednost od 10%.

Kako su sve vrijednosti sličnosti već normalizirane na intervalu [0, 1] potrebno je samo vrijednostima dodijeliti njihovu težinsku vrijednost te ih zbrojiti kako bi dobili ukupnu ocjenu sličnosti:

$$slicnost = 0.6 \cdot winnowing\_slicnost + 0.3 \cdot halsteadova\_slicnost + 0.1 \cdot fizicka\_slicnost$$

Vrijednost ocjene će se na kraju nalaziti unutar intervala [0, 1]. Što je vrijednost ocjene bliža 1, to je dekompajler dobio bolju ocjenu.

4. Implementacija i rezultati

Provođenje procesa ocjenjivanja kvalitete i sličnosti originalnog i dekompajliranog koda implementirano je u obliku skripte napisane u jeziku Python. Skripta može automatizirano dekompajlirati datoteke koristeći odabrani dekompajler te može i uspoređivati dvije datoteke koje su predane skripti.

U načinu rada dekompilacije, skripta korisnika prvo pita koji dekompajler želi koristiti. Nakon toga, korisnik mora unijeti ime direktorija u koji želi spremiti dekompajliranu datoteku. Nakon dekompajliranja datoteka se sprema u zadani direktoriji odgovarajućeg dekompajlera u kojem se dodatno stvara direktorij s imenom kojeg je korisnik zadao u prijašnjem koraku.

Ako se koristi dekompajler CFR, skripti se moraju predati putanja do neke *Class* ili Javine izvršne JAR datoteke jer dekompajler ne podržava rekurzivno pretraživanje direktorija u potrazi za datotekama. Nakon predaje putanje te odabira imena izlaznog direktorija izvršava se naredba naredbenog retka koja pokreće proces dekompajliranja za CFR dekompajler:

```
java -jar cfr-0.148 putanja_ulazne_datoteke --outputdir
putanja_izlaznog_direktorija
```

U slučaju korištenja dekompajlera Procyon, također se smiju predati samo *Class* ili JAR datoteke. Nakon odabira imena direktorija pokreće se naredba naredbenog retka kako bi se pokrenuo proces dekompajliranja:

```
java -jar procyon-decompiler-0.5.36.jar
putanja_ulazne_datoteke -o putanja_izlaznog_direktorija
```

Za razliku od prva dva dekompajlera, Fernflower ima sposobnost primanja putanje direktorija. On će potom pretraživati sve *Class* datoteke unutar direktorija te njegovih poddirektorija i redom ih dekompajlirati:

```
java -jar fernflower.jar putanja_ulazne_datoteke
putanja_izlaznog_direktorija
```

Ako mu je predana JAR datoteka, Fernflower će na kraju procesa dekompajliranja stvoriti JAR datoteku unutar koje se nalaze datoteke koje sadrže dekompajlirani kod. Zbog toga će biti potrebno još napraviti raspakiravanje proizvedene .jar datoteke kako bi došli do datoteka izvornih kodova:

4.1. Načini rada usporedbe datoteka

Usporedba ima dva načina rada. Prvi način rada je usporedba izvornog koda originalnog programa i dekompajliranog programa dok je drugi način rada usporedba dviju dekompajliranih datoteka.

U slučaju kada se uspoređuju originalni i dekompajlirani program radi se ocjena kvalitete dekompajliranog programa koristeći metriku veličine datoteke, kompleksnost kontrole toka te usporedba po Halsteadovim metrikama. Nakon toga provjerava se sličnost dekompajliranog koda s originalnim kodom. Za to se koristi sličnost po winnowing algoritmu te sličnost po Halsteadovom i fizičkom profilu kodova.

Kada se uspoređuju dvije dekompajlirane datoteke sličnost kodova nema svrhu. Zato se provodi samo ocjena kvalitete kodova koristeći metriku veličine datoteka, kompleksnost kontrole toka te se datoteke uspoređuju po Halsteadovim metrikama.

4.2. Proces usporedbe

4.2.1. Leksička analiza i procesiranje

Dekompajleri mogu uvesti male sintaksne promjene u kod koje neće nužno utjecati na semantiku programa, no hoće narušiti ocjenu sličnosti kodova. Primjerice ako dekompajler ne uspije naći originalna imena varijabli, algoritam koji direktno gleda po kodu će dati slabiju ocjenu sličnosti jer ta dva niza znakova neće biti ista. Taj problem se može riješiti leksičkom analizom koda.

Za leksičku analizu koristit će se Pythonova biblioteka *pygments*. Leksički analizator iz izvornog koda napisanog u Javi stvara niz leksičkih jedinki koje predstavljaju cijelu semantiku programa. Izlaz leksičkog analizatora koda prikazanog na primjeru Ispis 4-1 možemo vidjeti na primjeru Ispis 4-2.

```
// Komentar
class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

Ispis 4-1 - Ulazni kod za leksički analizator

```
(Token.Comment.Single, '// Komentar\n')
(Token.Keyword.Declaration, 'class')
(Token.Text, ' ')
(Token.Name.Class, 'HelloWorld')
(Token.Text, ' ')
(Token.Punctuation, '{')
(Token.Text, '\n')
(Token.Text, '\t')
(Token.Keyword.Declaration, 'public')
(Token.Text, ' ')
(Token.Keyword.Declaration, 'static')
(Token.Text, ' ')
(Token.Keyword.Type, 'void')
(Token.Text, ' ')
(Token.Name.Function, 'main')
(Token.Punctuation, '(')
(Token.Name, 'String')
(Token.Operator, '[')
(Token.Operator, ']')
(Token.Text, ' ')
(Token.Name, 'args')
(Token.Punctuation, ')')
(Token.Text, ' ')
(Token.Punctuation, '{')
(Token.Text, '\t\t')
(Token.Name, 'System')
(Token.Punctuation, '.')
(Token.Name.Attribute, 'out')
(Token.Punctuation, '.')
(Token.Name.Attribute, 'println')
(Token.Punctuation, '(')
(Token.Literal.String, '"Hello, World!"')
(Token.Punctuation, ')')
(Token.Punctuation, ';')
(Token.Text, '\n')
(Token.Text, '\t')
(Token.Punctuation, '}')
(Token.Text, '\n')
(Token.Punctuation, '}')
(Token.Text, '\n')
```

Ispis 4-2 - Izlazne leksičke jedinke leksičkog analizatora

Iz ovoga se uzimaju samo leksičke jedinke, a nizovi znakova iz kojih su stvorene leksičke jedinke se ignoriraju. Prije nego što se krene s usporedbom, dobiveni niz leksičkih jedinki se prvo preuređuje kako bi se uklonili nepotrebni znakovi i leksičke jedinke. Pri dohvaćanju

leksičkih jedinki ignoriraju se one koje predstavljaju komentare, imena uvezenih biblioteka i razreda, imena paketa u kojem se razred nalazi te leksičke jedinke koje predstavljaju samo prazne znakove poput znaka za novi red, znak za razmak i tabulator. Nadalje, svaka leksička jedinka se preuređuje tako da se iz nje maknu dijelovi naziva leksičke jedinke koje bi smetale pri usporedbi. Prema tome se iz naziva miče znakovni niz *Token* te sva pojavljivanja znaka točke.

```
Token.Keyword.Declaration -> KeywordDeclaration
```

Na kraju obrade svih leksičkih jedinki, one se sprema u jedan znakovni niz koji se onda koristi za usporedbu kodova.

```
KeywordDeclarationNameClassPunctuationKeywordDeclarationKeywordDeclarationK
eywordTypeNameFunctionPunctuationNameOperatorOperatorNamePunctuationPunctua
tionNamePunctuationNameAttributePunctuationNameAttributePunctuationLiteralS
tringPunctuationPunctuationPunctuationPunctuation
```

Ispis 4-3 - Krajnji rezultat leksičke analize i obrade leksičkih jedinki

4.2.2. Winnowing sličnost

Prije nego što se započne s usporedbom, nad ulaznim kodovima se provodi leksička analiza te se definiranom metodom niz dobivenih leksičkih jedinki obrađuje kako bi se povećala točnost i konzistentnost rezultata. Za veličinu k-grama je odabrana vrijednost od 70 dok je veličina prozora 20. Testiranjem su se ove vrijednosti pokazale kao dobra ravnoteža između krivih pozitivnih vrijednosti i pretjerane osjetljivosti na pogreške.

Nakon obrade ulaza, `createFingerprints` metoda stvara k-grame na kojima se onda primjenjuje funkcija sažimanja.

```
def createFingerprints(text, k):
   fingerprintArr = []
   for i in range(0, len(text) - k):
      fingerprintArr.append(h11(text[i: i + k]))
   return fingerprintArr
```

Ispis 4-4 - Funkcija biranja digitalnih otisaka

```
def h11(word):
   return int(hashlib.md5(word.encode('utf8')).hexdigest()[:9], 16)
```

Ispis 4-5 - Funkcija sažimanja

Nakon generiranja otisaka pokreće se algoritam koji bira otiske po winnowing algoritmu.

```
def winnowingAlgorithm(fingerprints, w):
   winnowingArr = []
   minRelativeIndex = -1
   minRelativeValue = 0
   for i in range(0, len(fingerprints) - w):
      if minRelativeIndex == -1:
         minRelativeIndex = findMinIndex(fingerprints[i:i + w])
         minRelativeValue = fingerprints[i + minRelativeIndex]
         winnowingArr.append(minRelativeValue)
         minRelativeIndex -= 1
      else:
         minRelativeIndex -= 1
         if int(minRelativeValue) > int(fingerprints[i + w - 1]):
            minRelativeIndex = w - 1
            minRelativeValue = fingerprints[i + w - 1]
            winnowingArr.append(minRelativeValue)
   return winnowingArr
```

Ispis 4-6 - Winnowing algoritam

```
def findMinIndex(fingerprints):
   minimum = fingerprints[0]
   minIndex = 0
   for i in range(0, len(fingerprints)):
      if int(fingerprints[i]) < int(minimum):
         minimum = fingerprints[i]
         minIndex = i
   return minIndex
```

Ispis 4-7 - Funkcija traženja minimalnih vrijednosti otisaka

4.2.3. Razred MetricProfiler

Kako bi se implementiralo računanje metrika koje se koristi za usporedbu kvalitete te za ocjenu sličnosti, napravljen je razred `MetricProfiler`. Razred sadrži sve podatke koji su potrebni kako bi se napravila usporedba kodova. Pri stvaranju razreda, odmah se stvara Halsteadov i fizički profil datoteke. Za stvaranje Halsteadovog profila je potrebno dobiti početna svojstva koda, tj. potrebno je prebrojati broj ukupnih i različitih operatora i operanada.

Operatori i operandi su definirani na sljedeći način:

Operatori:

- Rezervirane riječi (ne uključujući tipove varijabla)
- Aritmetički operatori
- Unarni operatori
- Operatori jednakosti i relacija
- Uvjetni operatori

- Operatori za operacija nad bitovima

Operandi:

- Svi identifikatori koji nisu rezervirane riječi
- Rezervirane riječi koje definiraju tip varijable
- Numeričke i znakovne konstante

Kako bi se uspješno razvrstali i prebrojali različiti operatori opet se koristi *pygments* biblioteka.

Leksičke jedinke iz biblioteke *pygments* koje predstavljaju Halsteadove operatore su prikazane na primjeru Ispis 4-8 dok su leksičke jedinke koje predstavljaju Halsteadove operande prikazane na primjeru Ispis 4-9.

```
Token.Keyword.Declaration
Token.Keyword
Token.Operator
Token.Punctuation
Token.Keyword.Constant
```

Ispis 4-8 - Leksičke jedinke biblioteke pygments za Halsteadove operatore

```
Token.Name.Attribute
Token.Keyword.Type
Token.Literal.*
Token.Name.*
```

Ispis 4-9 - Leksičke jedinke biblioteke pygments za Halsteadove operande

Nakon prebrajanja provodi se izračun svih potrebnih metrika te se nastavlja s izračunom fizičkog profila. Za stvaranje fizičkog profila se broje sve linije koje nisu isključivo komentari, prazne linije ili linije koje su nazivi uvezenih biblioteka i vanjskih razreda. Nakon toga se broje riječi sadržanih u kodu. Brojanje se radi uz pomoć biblioteke *pygments* te se kao riječ broji svaka leksička jedinka koja nije prazan znak, komentar ili uvezena biblioteka. Na kraju, za fizički profil se broji znakove. Broje se svi znakovi osim onih unutar komentara te znakova unutar linija koje predstavljaju uvoz biblioteka i vanjskih razreda.

4.2.4. Usporedba kompleksnosti kontrole toka

Za početak potrebno je naći linije koje sadržavaju neki oblik kontrole toka. To je riješeno pomoću regularnih izraza. Kako se pronalaze takvi izrazi tako se i računaju njihovi pojedinačni doprinos ukupnoj kompleksnosti kontrole toka. Za izračun kompleksnosti prvo

se broje svi logički operatori relacija i jednakosti (<, >, <=, >=, ==, !). Njihov broj se nakon toga množi sa 0.5, s obzirom na to da je njihova definirana težina za kompleksnost kontrole toka jednaka 0.5 za pojedinačni operator te se pribraja ukupnoj kompleksnosti. Nakon toga prebrajaju se uvjetni operatori (`&&` i `||`) te se njihov broj pribraja ukupnoj kompleksnosti.

4.3. Testiranje i rezultati

Za testiranje dekompajlera, ocjena kvalitete kodova te ocjene sličnosti korišteni su programi *reflexTest* i *PacManTopDownShooter* [10] [11].

PacManTopDownShooter je igra otvorenog koda te sadrži nekoliko datoteka koje se uzimaju kao primjer dekompajliranja.

reflexTest je program koji je napravljen u grafičkom sučelju *JavaFX* te je njegov kod gotovo u potpunosti automatski generiran te se zbog toga u metrikama pojavljuje posebni slučaj.

Zanimljivo je pokazati kako se metrike ponašaju ako se izvorni program ciljano promijeni. Za primjer se uzima izvorni kod Game.java iz programa *PacManTopDownShooter* te reflex.java iz programa *reflexTest*. Potom je iz oba programa stvorena kopija u kojoj je svaka druga linija koda izbrisana čime nastaju dvije nove datoteke koje su nazvane GameHalf.java te reflexHalf.java. One su međusobno uspoređene kako bi se pokazalo ponašanje metrika u kontroliranim uvjetima. Na primjerima Ispis 4-10 Ispis 4-11 su prikazani ispisi koje je skripta generirala nakon usporedbe navedenih programa.

```
Game.java je teža od GameHalf.java te je dobila ocjenu: 56.97%
Game.java je veća po volumenu od GameHalf.java ocjenu naspram drugoj:
49.976%

Winnowing slicnost: 76.568%
                  (lines, words, chars, h_len, h_vol, h_vocabulary)
Game.java         :(258, 1956, 9638, 1956, 16086.147, 299)
GameHalf.java     :(129, 1028, 4936, 1028, 8039.144,  226)
        Delta     :(129, 928,  4702, 928,  8047.003,  73)
halstead slicnost : 64.071%
fizicka slicnost : 51.257%
Ukupna ocjena slicnosti : 70.288%
```

Ispis 4-10 - Usporedba Game.java i GameHalf.java

```
reflex.java je teža od reflexHalf.java te je dobila ocjenu: 35.903%
reflex.java je veća po volumenu od reflexHalf.java ocjenu naspram drugoj:
44.167%

Winnowing slicnost: 86.478%
                  (lines, words, chars, h_len, h_vol, h_vocabulary)
reflex.java     :(306, 3079, 14876, 3079, 23999.006, 222)
reflexHalf.java :(157, 1392, 7514, 1392, 10599.676, 196)
          delta :(149, 1687, 7362, 1687, 13399.33, 26)
halstead slicnost : 66.749%
fizicka slicnost : 49.009%
Ukupna ocjena slicnosti : 76.812%
```

Ispis 4-11 - Usporedba reflex.java i reflexHalf.java

Iz ovih rezultata može se odmah primijetiti kako su Halsteadove metrike za ocjenu kodova više-manje prepolovljene što je i za očekivati s obzirom na to da metrike koje se koriste za usporedbu uglavnom ovise o veličini koda. No to je i jedan od nedostataka korištenja Halsteadovih metrika u ocjeni dekompajliranja. Halsteadova metrika ovdje pokazuje da manje koda znači i da je taj kod lakši za razumjeti. To je slučaj ako se pomoću te metrike želi gledati kompleksnost nekog originalno napisanog programa, ali to ne vrijedi ako dekompajler ne uspije dekompajlirati originalni program u potpunosti. Nadalje, vidi se kako su metrike fizičkog profila gotovo točno prepolovljene što je također očekivano ponašanje jer one ovise isključivo o količini koda. Gledajući winnowing algoritam, ovdje se jasno može vidjeti kako njegov način na koji bira digitalne otiske utječe na sličnost. Naime, pošto algoritam vrlo specifično bira otiske, ti otisci ne opadaju linearno kako se miču dijelovi programa već je to opadanje bliže opisano logaritamskom funkcijom. Halsteadova metrika sličnosti je sastavljena od Halsteadove metrike duljine i vokabulara. Gledajući kako se te metrike ponašaju kada bi se za primjer uzeo proces pisanja koda može se vidjeti kako je u ovom slučaju njihovo ponašanje sasvim očekivano. Naime očito je da duljina programa raste linearno kako se taj program piše. U slučaju vokabulara, kako proces pisanja napreduje tako se brzina njegovog porasta smanjuje s obzirom na to da se sve češće koriste isti operatori i operandi, tj. sve rjeđe se uvode novi. Time Halsteadova metrika vokabulara oponaša porast koji je približan logaritamskoj funkciji. Uzevši to u obzir može se primijetiti kako se te metrike ponašaju na isti način i u slučaju kada nedostaju neki dijelovi programa. Dok je relativna udaljenost Halsteadove metrike duljine u oba slučaja približno prepolovljena, metrika vokabulara je zadržala veću sličnost. Zanimljivo je primijetiti korelaciju koja postoji između winnowing algoritma i Halsteadove metrike vokabulara. Gledajući primjer Ispis 4-10 winnowing je ostvario sličnost od 76.57% dok vokabular ima relativnu udaljenost od originala jednaku 75.58%, a u primjeru Ispis 4-11 winnowing je dobio sličnost od 86.478%, dok je relativna udaljenost vokabulara 88.29%.

4.3.1. Ocjena kompleksnosti kontrole toka

Nakon obavljenog testiranja primijećeno je kako dekompajlirani kodovi uglavnom imaju lošiju ocjenu kompleksnosti koda te u slučajevima kada su ocjene dekompajlera bolje te razlike su minimalne. To je vjerojatno rezultat činjenice da su sva tri dekompajlera koristila spajanje više ulančanih uvjetnih naredbi kontrole toka u jednu čime se dodaje više operatora kako bi se proizveo istovjetni uvjet. Zbog dodavanja operatora uvjetu je dodijeljena veća ocjena kompleksnosti.

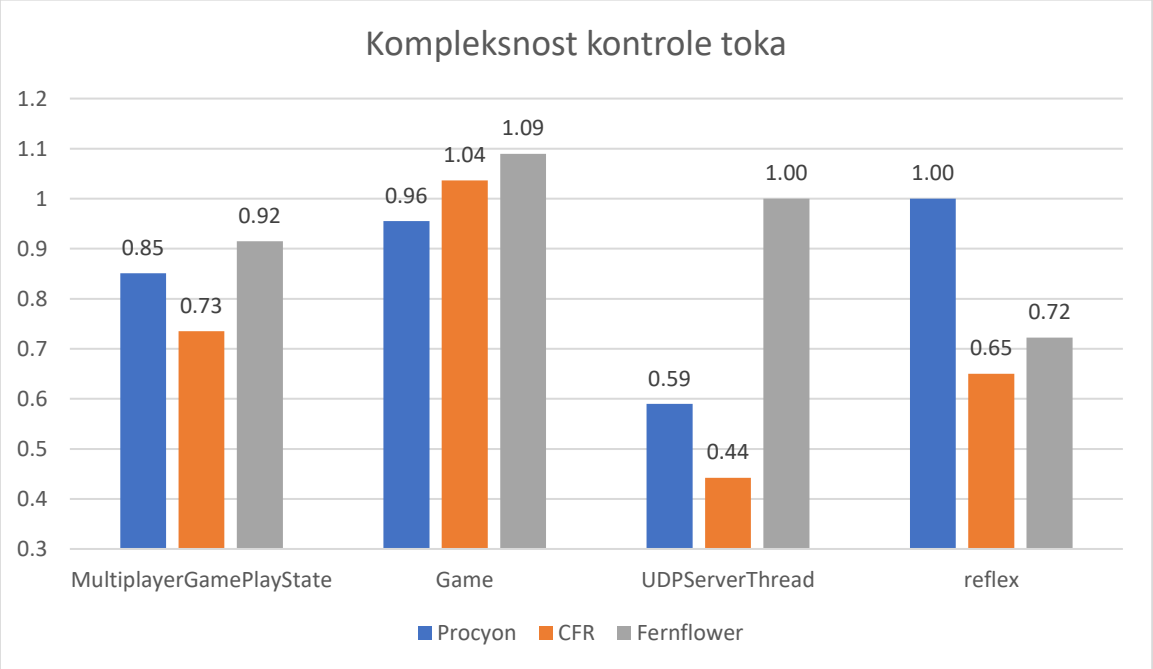
```
if (player.isAlive()) {
  if (input.isMouseButtonDown(Input.MOUSE_LEFT_BUTTON) && canFire <= 0) {
     ...
   }
}
```

Ispis 4-12 - Uvjetna naredba originalnog koda

```
if (Game.player.isAlive() && input.isMouseButtonDown(0) &&
      Game.canFire <= 0) {
     ...
}
```

Ispis 4-13 - Uvjetna naredba dekompajliranog koda

Može se vidjeti kako se metrikom na primjeru Ispis 4-12 izračunava ukupnu kompleksnost od 5,5 dok će na primjeru Ispis 4-13 dekompajliranog koda vrijednost biti 6,5.



Graf 4-1 - Kompleksnost kontrole toka

4.3.2. Halsteadova ocjena kvalitete

Halsteadove metrike za ocjenu kvalitete koda uvijek ukazuju na to da je originalni kod lakši za razumjeti. Jedina iznimka ovoga je program reflexTest. Tu se opet javlja problem kojeg Halsteadove metrike imaju, a to je da manji kod implicira i jednostavniji kod. To generalno i je slučaj jer, uzevši primjere Ispis 4-14 i Ispis 4-15, u obzir možemo vidjeti da je leksički analizator generirao više leksičkih jedinki kod originalnog koda što direktno utječe na Halsteadovu metriku duljine. Na tu ocjenu uglavnom utječe način na koji dekompajleri deklariraju vanjske biblioteke. Primjerice dok je u primjeru Ispis 4-14 originalni kod koristio puni naziv pozivane biblioteke, dekompajleri su koristili skraćeni naziv kao što je pokazano na primjeru Ispis 4-15.

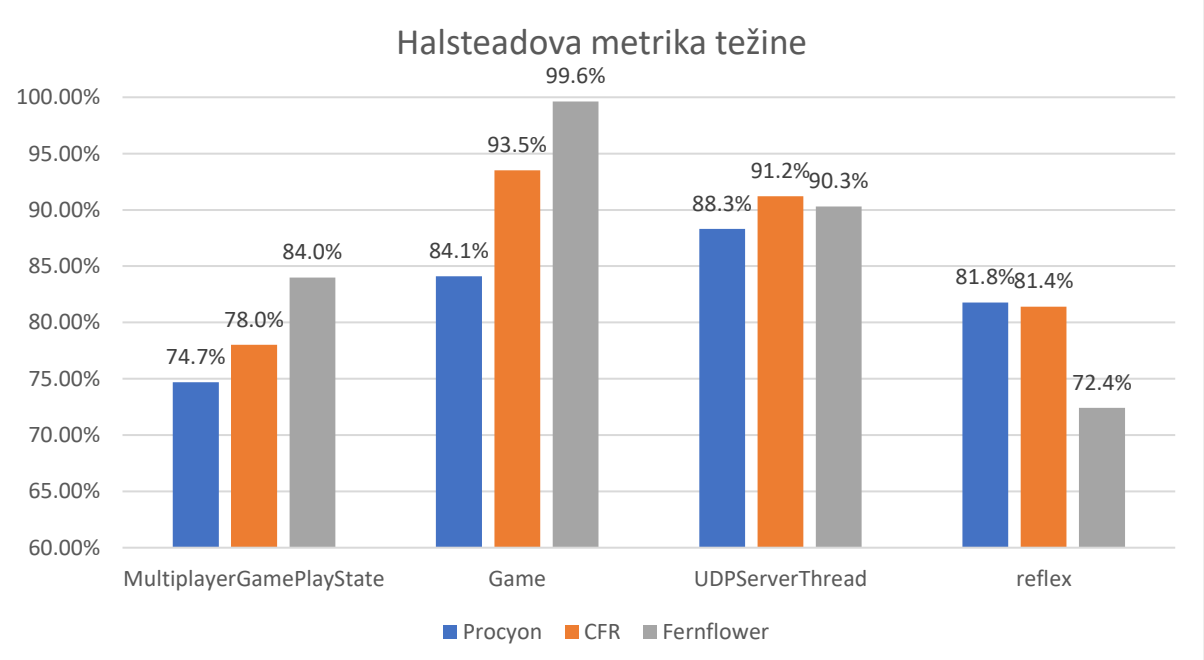
```
jPanel1.setBorder(javax.swing.BorderFactory.createTitledBorder(null,
"Upute", javax.swing.border.TitledBorder.DEFAULT_JUSTIFICATION,
javax.swing.border.TitledBorder.DEFAULT_POSITION, new
java.awt.Font("Arial", 0, 18)));
```

Ispis 4-14 - Utjecaj na Halsteadovu metriku težine, originalni kod

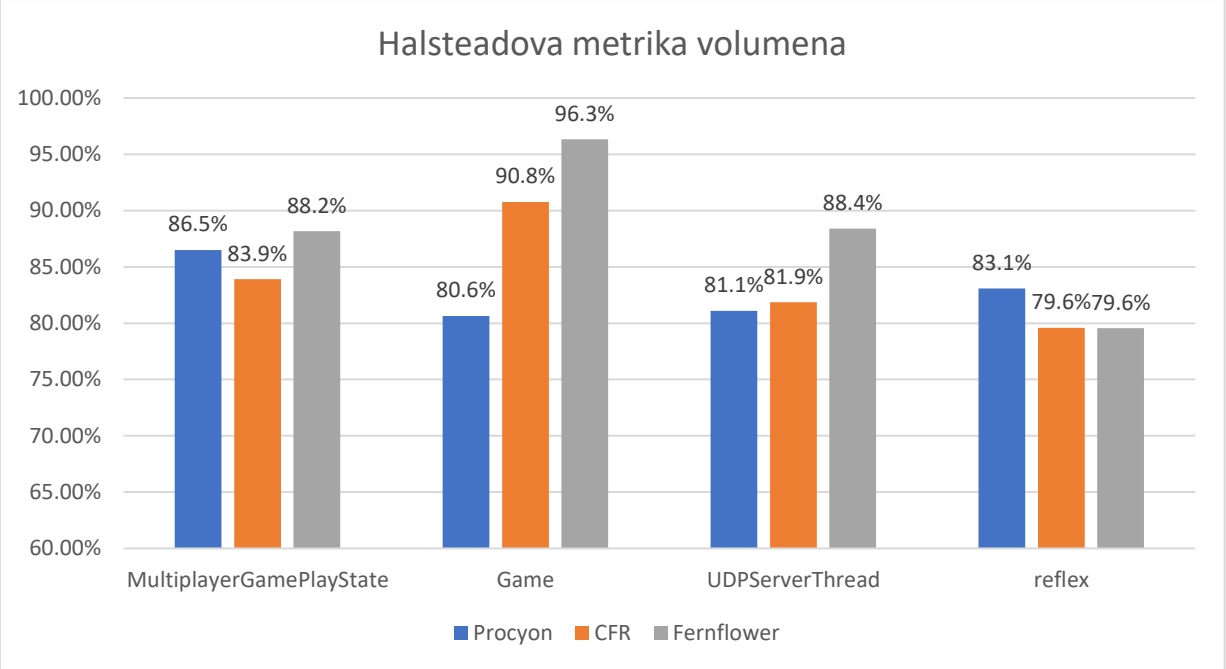
```
this.jPanel1.setBorder(BorderFactory.createTitledBorder(null, "Upute", 0,
0, new Font("Arial", 0, 18)));
```

Ispis 4-15 - Utjecaj na Halsteadovu metriku težine, dekompajlrani kod

No, s druge strane može se argumentirati da deklariranje biblioteka na razini razreda te korištenje skraćenih imena tih biblioteka olakšava čitanje koda što ova Halsteadova metrika i pokazuje. Općenito Fernflower stvara najmanje nepotrebnih leksičkih jedinki te je kod kojeg on generira generalno i najsličniji originalom po tom pitanju.



Graf 4-2 - Halstead metrika težine



Graf 4-3 - Halstead metrika volumena

4.3.3. Winnowing sličnost

Najveću razliku koja se stvara kod ocjene sličnosti prema winnowing algoritmu je razlika u načinu referenciranja varijabla i vanjskih razreda. Kao što je već viđeno, dekompajleri često deklariraju biblioteke na razini razreda te koriste skraćena imena tih biblioteka. Time se kroz program nakuplja veći broj manjih razlika te se i time smanjuje sličnost. Dekompajleri također imaju problema s korištenjem globalno definiranih konstanti te umjesto korištenja njihovih naziva samo ubacuju vrijednost konstante. Winnowing algoritam je relativno otporan na takve manje greške no ako su te greške relativno blizu, one mogu napraviti veliku razliku. Na primjeru koji je uzet od rezultata programa Game.java koristeći dekompajler Procyon (Ispis 4-16) može se vidjeti kako jedno nizanje takvih razlika može imati utjecaj na sličnost. Dekompajler Procyon je koristio vrijednost konstante umjesto njenog naziva i puno ime varijable kao što je vidljivo na primjeru Ispis 4-18. Druga dva dekompajlera su u ovom slučaju umjesto punog naziva ipak koristili skraćeni naziv varijable `input`, što je i vjerojatno slučaj u ostatku programa te i razlog zašto su oba dobila bolju ocjenu po tom pitanju.

```
Ocjena kompleksnosti uvjeta upravljanja tokom:
      Ocjena prve detoteke: 1
      Ocjena druge datoteke: 0.95506
Ocjena težine dekompajlirane datoteke: 84.091%
Ocjena volumena dekompajlirane datoteke: 80.641%
Winnowing slicnost: 50.885%
halstead slicnost : 89.069%
fizicka slicnost : 84.361%
Ukupna ocjena slicnosti : 65.688%
```

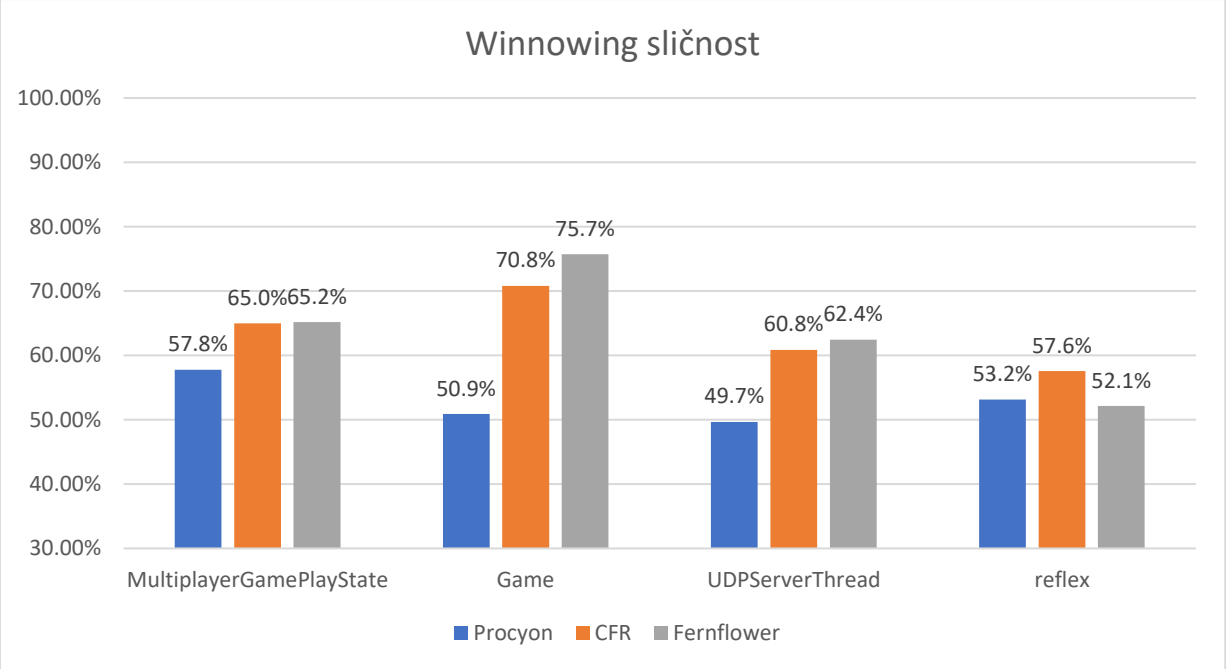
Ispis 4-16 - Rezultat dekompajlera Procyon na kodu Game.java

```
switch (state) {
 case MENUSTATE:
        MenuState.update(gc, input, delta, mouseX, mouseY);
        break;
 case GAMEPLAYSTATE:
        GamePlayState.update(gc, input, delta, mouseX, mouseY);
        break;
 case MULTIPLAYERMENUSTATE:
    MultiplayerMenuState.update(gc, input, delta, mouseX, mouseY);
    break;
    …
}
```

Ispis 4-17 - Utjecaj na winnowing algoritam, Game.java, originalni kod

```
switch (Game.state) {
  case 0: {
       MenuState.update(gc, Game.input, delta, Game.mouseX, Game.mouseY);
       break;
  }
  case 1: {
       GamePlayState.update(gc, Game.input, delta, Game.mouseX,
Game.mouseY);
        break;
  }
  case 2: {
       MultiplayerMenuState.update(gc, Game.input, delta, Game.mouseX,
Game.mouseY);
       break;
   }
}
```

Ispis 4-18 - Utjecaj na winnowing algoritam, Game.java, dekompajlirani kod, dekompajler Procyon



Graf 4-4 - Winnowing sličnost

4.3.4. Halsteadova sličnost

Metrikom se provjerava koliko su kodovi slični po broju operatora i operanada koje kod koristi. Na ovaj dio ocjene sličnosti može utjecati nepotrebno stvaranje dodatnih varijabli, odvojena deklaracija i inicijalizacija varijabli. Također dodavanje dodatnih modifikatora varijablama i način na koji se dekompajler referencira na određene varijable, metode ili biblioteke može utjecati na ocjenu. Na primjeru Ispis 4-19 se može jasno vidjeti kako Procyon uvijek koristi potpuni naziv varijabli i metoda na koje se referencira.

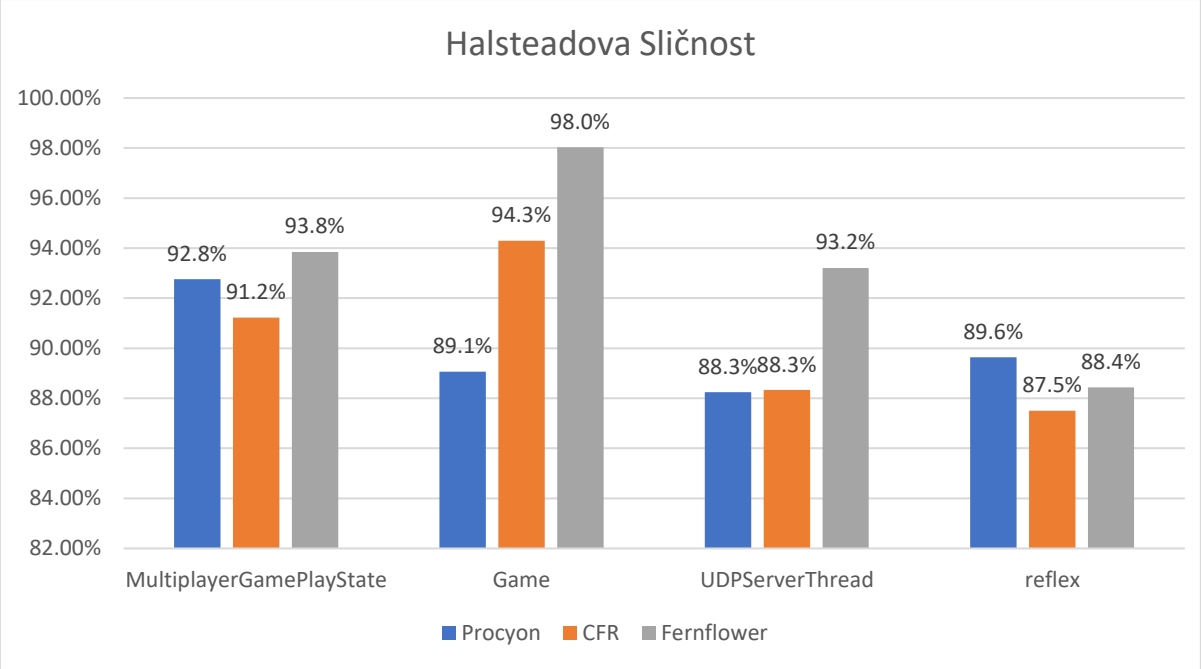
```
Game.infoString = "" + Game.player.getX() + ":" + Game.player.getY() + ":"
+ mouseX + ":" + mouseY + ":" + Game.player.getHealth() + ":" +
Game.Score.getScore();
```

Ispis 4-19 - Korištenje punog imena varijable, dekompajler Procyon

```
infoString = "" + player.getX() + ":" + player.getY() + ":" + mouseX + ":"
+ mouseY + ":" + player.getHealth() + ":" + Score.getScore();
```

Ispis 4-20 - Originalni kod koristi kraći oblik varijable

Kako Procyon takav način referenciranja koristi kroz cijeli kod tako mu se i povećava broj leksičkih jedinki te smanjuje sličnost s originalnim programom po ovoj metrici. Iako CFR i Fernflower znaju koristiti isti način referenciranja, oni to rade u manjoj mjeri. Također Procyon dodaje ključnu riječ `final` na svaku varijablu kojoj se vrijednost kasnije ne mijenja iako ta varijabla nije bila tako definirana u originalnom programu.



Graf 4-5 - Halstead sličnost

4.3.5. Fizička sličnost

Fizička sličnost može dosta ovisiti o načinu na koji dekompajleri generiraju određene dijelove koda. Ponovo se ističe primjer dekompajliranja reflex.java programa. Kroz sva tri dekompajlera sličnost je oko 70%. Gledajući rezultate dekompajliranja u primjeru Ispis 4-22 može se primijetiti velika razlika u broju generiranih linija. Gdje je originalni program imao 306 linija, sva tri dekompajlera su generirala kod koji je imao oko 190 linija. To je posljedica stila generiranja koda kojeg provode dekompajleri. Dok su u originalnom kodu dekoratori uredno napisani, dekompajleri su ih sve stavili u jednu liniju. S obzirom na to da je to jako česta pojava u kodu reflex.java to je i dosta utjecalo na rezultat.

```
jPanel1Layout.setVerticalGroup(
      jPanel1Layout.createParallelGroup(javax.swing.GroupLayout.Alignment.L
EADING)
        .addGroup(jPanel1Layout.createSequentialGroup()
            .addContainerGap()
            .addComponent(jLabel3)
            .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.UNR
      ELATED)
            .addComponent(jLabel1)
            .addGap(18, 18, 18)
            .addComponent(jLabel5)
            .addContainerGap())
);
```

Ispis 4-21 - Originalno generirani kod, svaka linija je posebna linija u kodu

```
jPanel1Layout.setVerticalGroup(jPanel1Layout.createParallelGroup(GroupLayou
t.Alignment.LEADING).addGroup(jPanel1Layout.createSequentialGroup().addCont
ainerGap().addComponent(this.jLabel3).addPreferredGap(LayoutStyle.Component
Placement.UNRELATED).addComponent(this.jLabel1).addGap(18, 18,
18).addComponent(this.jLabel5).addContainerGap()));
```

Ispis 4-22 - Dekompajlirani kod, sve je jedna linija

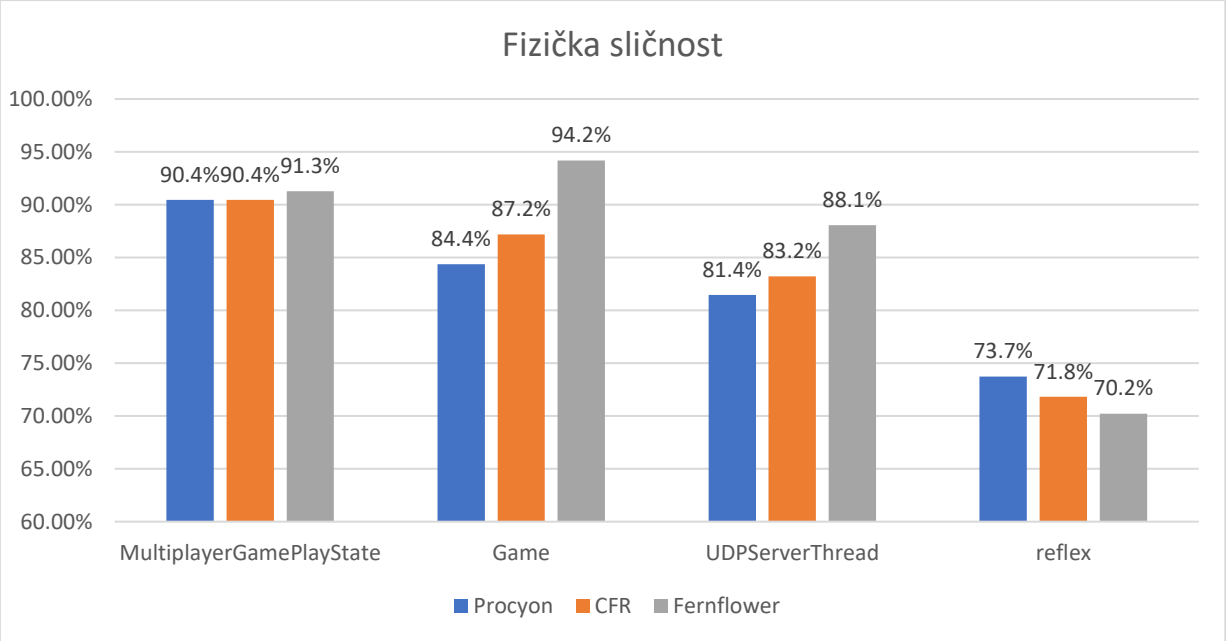
Sljedeće, u kodu Game.java se nalazi veliki broj statičkih globalnih varijabli koje su u vrijeme deklaracije i inicijalizirane na svoje početne vrijednosti. No sva tri dekompajlera su odlučila prvo deklarirati varijable, i tek onda ih unutar svog dodatnog bloka inicijalizirati kao što je prikazano na primjeru Ispis 4-24. Dekompajleri nisu ovo napravili sa svim varijablama te se Fernflower ističe kao najbolji s obzirom na to da je on ovo napravio na najmanjem broju varijabla čime je i najmanje puta ponovio istu varijablu.

```
public class Game {
	static ScoreManager Score;
	static LinkedList<Bullet> bulletList = new LinkedList<>();
	static ArrayList<Enemy> enemyList = new ArrayList<>();
	static Ammo[] ammos = new Ammo[10];
	static Player player;
	static int bulletCount = 0, enemyCount = 0, ammosCount = 0;
	static final int START_DELAY = 500, AMMOS_DELAY = 10000,
		NEW_DIFFICULTY_DELAY = 15000, BULLET_DELAY = 200;
		static int ENEMY_DELAY = 500;
	…
}
```

Ispis 4-23 - Deklaracija i inicijalizacija u originalnom kodu

```
public class Game{
	static ScoreManager Score;
	static LinkedList<Bullet> bulletList;
	static ArrayList<Enemy> enemyList;
	static Ammo[] ammos;
	static Player player;
	static int bulletCount;
	static int enemyCount;
	static int ammosCount;
	static final int START_DELAY = 500;
	static final int AMMOS_DELAY = 10000;
	static final int NEW_DIFFICULTY_DELAY = 15000;
	static final int BULLET_DELAY = 200;
	static int ENEMY_DELAY;
	...
	static {
		Game.bulletList = new LinkedList<Bullet>();
		Game.enemyList = new ArrayList<Enemy>();
		Game.ammos = new Ammo[10];
		Game.bulletCount = 0;
		Game.enemyCount = 0;
		Game.ammosCount = 0;
		Game.ENEMY_DELAY = 500;
		...
	}
}
```

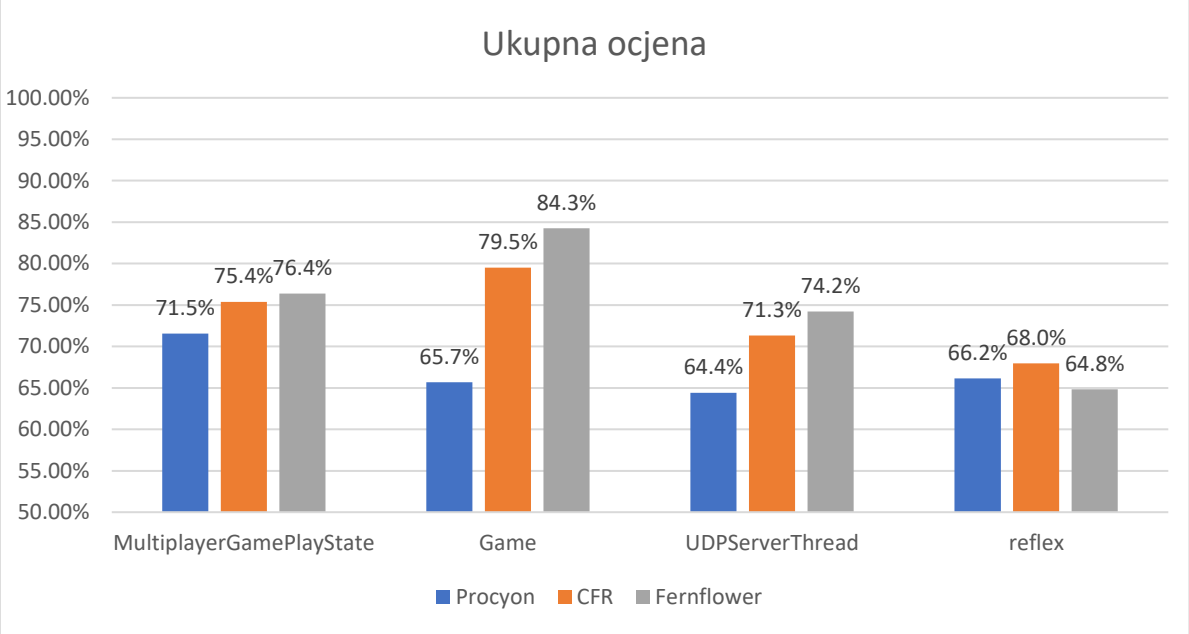
Ispis 4-24 - Deklaracija i inicijalizacija varijabli u dekompajliranom kodu



Graf 4-6 - Fizička sličnost

4.3.6. Ukupna ocjena

Gledajući metrike sličnosti a i metrike kvalitete koda, može se zaključiti kako je dekompajler Fernflower konzistentno najbolji. Dok su druga dva dekompajlera dobivala lošiju ocjenu uglavnom zbog stvaranja dodatnih leksičkih jedinki koje nisu potrebne.



Graf 4-7 - Ukupna ocjena

4.4. Rezultati neuspješne dekompilacije

Na primjeru Newton.java je pokazano kako se metrike ponašaju kada dekompajleri ne uspiju dekompajlirati program u cijelosti. Konkretno za ovaj kod niti jedan od dekompajlera nije uspio dekompajlirati tri metode koje su se nalazile u originalnom programu. Analizirajući kod, sva tri dekompajlera su uspješno prepoznala da ta metoda postoji te su njenu definiciju stavili unutar svog generiranog koda, no metoda nema implementaciju. Problem je vjerojatno nastao jer se nisu mogli referencirati na određeni vanjski razred. Svejedno rezultati ocjene dekompilacije će pokazati da se neuspješna dekompilacija može prepoznati.

```
Ocjena kompleksnosti uvjeta upravljanja tokom:
     Ocjena prve detoteke: 1
     Ocjena druge datoteke: 6.42857
Ocjena težine originalne datoteke: 65.734%
Ocjena volumena originalne datoteke: 55.828%
Winnowing slicnost: 49.851%
halstead slicnost : 63.392%
fizicka slicnost : 60.999%
Ukupna ocjena slicnosti : 55.028%
```

Ispis 4-25 - Rezultati dekompajlera Procyon na programu Newton.java

```
Ocjena kompleksnosti uvjeta upravljanja tokom:
     Ocjena prve detoteke: 1
     Ocjena druge datoteke: 6.42857
Ocjena težine originalne datoteke: 63.313%
Ocjena volumena originalne datoteke: 52.664%
Winnowing slicnost: 59.144%
halstead slicnost : 60.525%
fizicka slicnost : 58.437%
Ukupna ocjena slicnosti : 59.487%
```

Ispis 4-26 - Rezultati dekompajlera CFR na programu Newton.java

```
Ocjena kompleksnosti uvjeta upravljanja tokom:
     Ocjena prve detoteke: 1
     Ocjena druge datoteke: 6.42857
Ocjena težine originalne datoteke: 65.42%
Ocjena volumena originalne datoteke: 52.738%
Winnowing slicnost: 55.052%
halstead slicnost : 59.885%
fizicka slicnost : 58.642%
Ukupna ocjena slicnosti : 56.861%
```

Ispis 4-27 - Rezultati dekompajlera Fernflower na programu Newton.java

Očekivano je da sve metrike sličnosti ukazuju na malenu sličnost, dok Halsteadove metrike kvalitete koda ukazuju na to da je originalni program teži za razumjeti jer se u njemu nalazi veća količina koda. No vjerojatno najzanimljiviji rezultat daje metrika ukupne kompleksnosti kontrole toka. Kod nje neočekivano visoka ocjena u svakom od rezultata nam bez sumnje može pokazati da nedostaje velika količina logike u dekompajliranim kodovima.

4.5. Rezultati dekompajliranja obfusciranog koda

Zbog izuzetno kompliciranog procesa pokretanja obfuskacije obfuskatora ProGuard, samo program *reflexTest* je uspješno obfusciran. Bez obzira na to što se taj program pokazao kao posebni slučaj moguće je vidjeti na koji način će obfuskator utjecati na metrike.

Vrijedno je napomenuti kako je korišteni obfuskator ProGuard smanjio veličinu izvršne datoteke s 24 KB na samo 9 KB.

Nakon dekompajliranja CFR je generirao program kojeg nije moguće kompajlirati. Pogreške nastale u kodu su ručno uklonjene ali program nakon uspješnog pokretanja ne radi kako bi trebao. Procyon je nakon dekompilacije također generirao kod koji se ne može pokrenuti, no nakon ručnog micanja pogrešaka program se uspješno pokreće te radi kako bi trebao. Fernflower je dao najbolje rezultate. Generirani kod je imao minimalne pogreške te nakon ručnog uklanjanja pogrešaka kod se uspješno pokreće te program radi kako bi trebao.

```
Ocjena kompleksnosti uvjeta upravljanja tokom:
     Ocjena prve detoteke: 1
     Ocjena druge datoteke: 1.00000
Ocjena težine originalne datoteke: 73.111%
Ocjena volumena originalne datoteke: 70.651%
Winnowing slicnost: 47.669%
halstead slicnost : 78.413%
fizicka slicnost : 60.131%
Ukupna ocjena slicnosti : 58.138%
```

Ispis 4-28 - Rezultat dekompajlera Procyon na obfusciranom kodu reflex.java

```
Ocjena kompleksnosti uvjeta upravljanja tokom:
     Ocjena prve detoteke: 1
     Ocjena druge datoteke: 0.65000
Ocjena težine originalne datoteke: 83.162%
Ocjena volumena originalne datoteke: 74.696%
Winnowing slicnost: 60.238%
halstead slicnost : 83.318%
fizicka slicnost : 65.228%
Ukupna ocjena slicnosti : 67.661%
```

Ispis 4-29 - Rezultat dekompajlera CFR na obfusciranom kodu reflex.java

```
Ocjena kompleksnosti uvjeta upravljanja tokom:
      Ocjena prve detoteke: 1
      Ocjena druge datoteke: 0.72222
Ocjena težine originalne datoteke: 74.342%
Ocjena volumena originalne datoteke: 71.07%
Winnowing slicnost: 51.128%
halstead slicnost : 78.817%
fizicka slicnost : 58.657%
Ukupna ocjena slicnosti : 60.188%
```

Ispis 4-30 - Rezultat dekompajlera Fernflower na obfusciranom kodu reflex.java

Osim metrike kompleksnosti kontrole toka, može se vidjeti kako su rezultati po svim metrikama lošiji nego što su bili u dekompajliranom kodu koji nije bio obfusciran. Pri dekompajliranju obfusciranog programa svaki dekompajler je generirao dodatne datoteke, tj. razrede u kojima se nalazi dio koda koji je se prije nalazio u istom glavnom razredu. Time se izvukao dio koda iz glavne metode koje se koristi za usporedbu koda, što se i može vidjeti na rezultatima. Pored toga, imena svih varijabli su promijenjena, poredak i način na koji dekompajleri deklariraju i instanciraju varijable je promijenjen. Znatno je promijenjen način na koji se pozivaju metode što je pokazano na primjeru Ispis 4-32.

Potrebno je dodatno pojasniti Halsteadovu metriku težine i volumena. Zbog toga što se u glavnim razredima koji se koriste za usporedbu nalazi još manje koda, rezultati koje je metrika generirala impliciraju da je kod lakši za razumjeti. Kada bi se testiranje moglo obaviti na više testnih slučajeva postoji mogućnost da bi u dekompajliranom kodu često mogla biti manja količina koda čime bi ova metrika bila vrlo nekonzistentna. Ova metrika bi se u tom slučaju morala ignorirati jer često ne bi točno obavljala svoj posao.

```
public class reflex{
      ...
      public static void main(String args[]) {
            ...
            java.awt.EventQueue.invokeLater(new Runnable() {
                  public void run() {
                        try {
                              new reflex().setVisible(true);
                        } catch (InterruptedException ex) {
                        Logger.getLogger(reflex.class.getName()).log(Level.
                        SEVERE, null, ex);
             }
            }
      });
}
```

Ispis 4-31 - Pokretanje programa u originalnom kodu

```
public class refelex{
      ...
      public static void main(String[] var0) {
            ...
            EventQueue.invokeLater(new g());
      }
}

final class g implements Runnable {
   public final void run() {
      (new reflex()).setVisible(true);
   }
}
```

Ispis 4-32 - Pokretanje programa u dekompajliranom kodu, dekompajler Fernflower

5. Zaključak

U radu je prikazano ponašanje metrika u normalnim okolnostima te u nekim posebnim slučajevima i kako ti posebni slučajevi utječu na metrike. Pokazano je kako se sve metrike ponašaju konzistentno i u skladu s očekivanjima osim u posebnim slučajevima kada je pokazano da metrike mogu i ukazati na greške u procesu dekompajliranja. Moglo se vidjeti kako dekompajlirani programi gotovo uvijek imaju veću složenost kod naredbi kontrole toka. Halsteadove metrike su pokazale kako razmatrani dekompajleri uvijek stvaraju višak leksičkih jedinki te općenito opširnije pišu program što ga kao rezultat čini težim za razumjeti. Winnowing algoritam se koristio kako bi se dva koda direktno usporedila te se po tome i dala ocjena sličnosti. Također, pokazalo se kako veći broj malih razlika može smanjiti sličnost kodova. To se moglo primjerice vidjeti na rezultatima dekompajlera koji su stavljali pune nazive razreda i biblioteka te su imali lošiju ocjenu od onih koji to nisu radili.

Razmotreni su dekompajleri Procyon, CFR i Fernflower te se pomoću korištenih metrika Fernflower pokazao kao najbolji dekompajler s obzirom da je konzistentno generirani najmanje nepotrebnih leksičkih jedinki te je i njegov kod na razmatranim primjerio bio najrazumljiviji. Taj rezultat je dodatno potvrdilo dekompajliranje obfusciranog programa u kojem je Fernflower generirao kod s najmanje grešaka. Nadalje CFR je u većini slučajeva imao bolje rezultate od Procyona. Procyon je dao puno bolji rezultat pri dekompajliranju obfusciranog koda te je nakon uklanjanja sintaksnih pogrešaka program kojeg je on generirao radio korektno dok se kod CFR-a uklanjanje pokazalo kao znatno teži zadatak nakon kojeg program i dalje nije radio.

6. Literatura

[1] Oracle Corporation, »Compilation Overview,« Oracle Corporation, [Mrežno]. Available: https://openjdk.java.net/groups/compiler/doc/compilation-overview/index.html. [Pokušaj pristupa 23 5 2020].

[2] G. Nolan, Decompiling java, APress Media, LLC, 2004.

[3] »Zakon o autorskom pravu i srodnim pravima, Čl 107 - 112 računalni programi,« [Mrežno]. Available: https://www.zakon.hr/z/106/Zakon-o-autorskom-pravu-i-srodnim-pravima. [Pokušaj pristupa 20 5 2020].

[4] »Zakon o autorskom pravu i srodnim pravima, Čl 8. Nezaštičene tvorevine,« [Mrežno]. Available: https://www.zakon.hr/z/106/Zakon-o-autorskom-pravu-i-srodnim-pravima.

[5] A. Kalinovsky, Covert Java: Techniques for Decompiling, Patching, and Reverse Engineering, Sams Publishing, 2004.

[6] N. Naeem, M. Batchelder i L. Hendren, »Metrics for Measuring the Effectiveness of Decompilers and Obfuscators,« p. 16, Lipanj 2006.

[7] E. L. Jones, »Metrics based plagarism monitoring,« *Journal of Computing Sciences in Colleges,* pp. 253-261, Travanj 2001.

[8] »Halstead complexity measures,« [Mrežno]. Available: https://en.wikipedia.org/wiki/Halstead_complexity_measures. [Pokušaj pristupa 20 5 2020].

[9] S. Schleimer, D. S. Wilkerson i A. Aiken, »Winnowing: Local Algorithms for Document Fingerprinting,« Lipanj 2003.

[10] T. Karamehmedović, »reflexTest Github,« [Mrežno]. Available: https://github.com/Tarik-fer/reflexTest. [Pokušaj pristupa 11 6 2020].

[11] C. Blasi, »PacManTopDownShooter GitHub,« [Mrežno]. Available: https://github.com/carloblasi/PacManTopDownShooter. [Pokušaj pristupa 11 6 2020].

[12] M. Strobel, »Procyon, BitBucket,« [Mrežno]. Available: https://bitbucket.org/mstrobel/procyon/wiki/Home. [Pokušaj pristupa 15 06 2020].

[13] L. Benfield, »CFR - another java decompiler.,« [Mrežno]. Available: http://www.benf.org/other/cfr/. [Pokušaj pristupa 16 06 2020].

[14] S. A. W. Todd A. Proebsting, »Krakatoa: Decompilation in Java (Does Bytecode Reveal Source?),« p. 14, Lipanj 1997.

Sažetak

Ocjena kvalitete dekompajliranog Java koda

U procesu reverznog inženjerstva jedan od najbitnijih alata su dekompajleri. Oni omogućavaju rekonstrukciju originalnog koda iz izvršnih datoteka te imaju razne primjene, primjerice povratak izgubljenog koda te dekompajliranje tuđih kodova radi proučavanja korištenih algoritama. Kako je Java zbog svojeg dizajna posebno ranjiva na ovakve vrste napada s obzirom na ostale jezike poput C++. Kako bi se ocijenili korišteni dekompajleri koriste se metrike sličnosti te metrike ocjene kvalitete koda koje će pokazati koliko je dekompajlirani kod sličan i kvalitetan u usporedbi s originalnim kodom. Na kraju se testiranjem može vidjeti kako metrike mogu ukazati na bolji kod te neuspješno dekompajlirane datoteke.

Ključne riječi: dekompajliranje, kompajliranje, kvaliteta, ocjenjivanje, Java, obfuskacija, efikasnost.

Abstract

Assessing quality of decompiled Java code

In the process of reverse engineering one of the most important tools are decompilers. They are capable of retrieving the original source code from compiled files and therefore have a wide range of usages such as retrieving lost code and decompiling someone else's code in order to analyse a specific algorithm. Because of its design, Java is much more prone to being decompiled in comparison to other programming languages such as C++. To grade decompilers, we use metrics for measuring the similarity and the quality of decompiled code in comparison to the original source code. With testing it is possible to see that the metrics can show the quality of the code as well as show that the code has not been decompiled properly.

Keywords: decompiling, compiling, quality, grading, Java, obfuscation, efficiency