SVEUČILIŠTE U ZAGREBU

FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 1673

Dodatak za Android Studio za podršku sigurnosne analize APK datoteka

Karlo Mravunac

Zagreb, lipanj 2018.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA
ODBOR ZA DIPLOMSKI RAD PROFILA

Zagreb, 9. ožujka 2018.

Predmet: **Diplomski rad**

DIPLOMSKI ZADATAK br. 1673

Pristupnik: **Karlo Mravunac (0036481662)**
Studij: Računarstvo
Profil: Programsko inženjerstvo i informacijski sustavi

Zadatak: **Dodatak za Android studio za podršku sigurnosne analize APK datoteka**

Opis zadatka:

Radi poboljšanja sigurnosti raznih aplikacija redovito se provodi njihova sigurnosna analiza, koja se ponekad naziva i penetracijsko ispitivanje. Postupak se svodi na to da stručna osoba analizira aplikaciju u potrazi za potencijalnim ranjivostima. Međutim, najčešće za aplikaciju nije na raspolaganju izvorni kod već samo prevedeni. To, zajedno s nekim metodama sprečavanja reverznog inženjerstva, značajno otežava sigurnosnu analizu. Iz tog razloga razvijen je cijeli niz alata koji pomažu prilikom analize ciljane aplikacije. Međutim, ti alati razvijani su nezavisno jedni od drugih i dosta su nepovezani, a dodatno, nemaju sposobnosti integriranih razvojnih okruženja koje imaju vrlo napredne metode analize koda na temelju čega omogućavaju korisniku vrlo jednostavno kretanje po kodu. U sklopu ovog diplomskog rada potrebno je razviti dodatak za integrirano okruženje Android Studio koje bi omogućavalo jednostavniju sigurnosnu analizu aplikacija za koje nije dostupan izvorni kod. Taj dodatak treba integrirati razne dekompajlere te omogućiti korisniku jednostavno manipuliranje njihovim izlazima i kretanje po dekompajliranom kodu aplikacije. Citirati korištenu literaturu i navesti dobivenu pomoć.

Zadatak uručen pristupniku: 16. ožujka 2018.
Rok za predaju rada: 29. lipnja 2018.

Mentor:

Doc. dr. sc. Stjepan Groš

Djelovođa:

Doc. dr. sc. Boris Milašinović

Predsjednik odbora za diplomski rad profila:

Izv. prof. dr. sc. Igor Mekterović

Zahvaljujem se mentoru, doc.dr.sc. Stjepanu Grošu na stručnim savjetima i iskazanom povjerenju prilikom izrade ovog diplomskog rada.

Sadržaj

1. Uvod

Velik broj korisnika danas koristi pametne telefone i različite aplikacije na njima. Većina tih pametnih telefona koristi Android operacijski sustav [1] na kojem se izvršavaju Android aplikacije. Android je operacijski sustav otvorenog koda temeljen na Linux operacijskom sustavu koji je razvila tvrtka Google. Primarno je namijenjen uređajima s ekranom osjetljivim na dodir, ali koriste ga i uređaji poput pametnih televizija i pametnih satova. Android aplikacije se distribuiraju i instaliraju u obliku APK datoteka (*engl. Application Package Kit*) [2] koje se najčešće preuzimaju s Google-ove PlayStore trgovine. Kako bi se poboljšala sigurnosti Android aplikacija provodi se njihova sigurnosna analiza, koja se ponekad zove i penetracijsko ispitivanje. Prilikom sigurnosne analize stručnjak analizira aplikaciju kako bi pronašao potencijalne ranjivosti u njoj. S obzirom da najčešće nije dostupan izvoran kod Android aplikacije već samo APK datoteka, odnosno njen prevedeni kod, sigurnosna analiza je otežana. Kako bi se iz APK datoteke dobio izvorni kod potrebno je koristiti alate koji obavljaju obrnuti proces od prevođenja aplikacija. Ti alati se zovu dekompajleri [3] i oni iz prevedenog koda aplikacije pokušavaju producirati čim bolji izvorni kod. S obzirom da su Android aplikacije pisane u Java programskom jeziku, ovaj rad će se baviti Java dekompajlerima. Postoji više nezavisno razvijenih Java dekompajlera koji nemaju mogućnosti napredne navigacije i analize izvornog koda poput integriranih razvojnih okruženja. Integrirano razvojno okruženje je alat koji olakšava razvoj aplikacija u jednom ili više programskih jezika. Neke od mogućnosti takvih okruženja su: uređivanje izvornog koda, automatizirano prevođenje koda i izgradnja aplikacija, otkrivanje grešaka u kodu i profiliranje aplikacije. Za razvoj Android aplikacija najčešće se koristi Android Studio integrirano razvojno okruženje. Svrha ovog rada je razviti dodatak za Android Studio integrirano okruženje koji integrira neke od dostupnih Java dekompajlera i omogućava korisniku jednostavnu manipulaciju i kretanje po produciranom izvornom kodu.

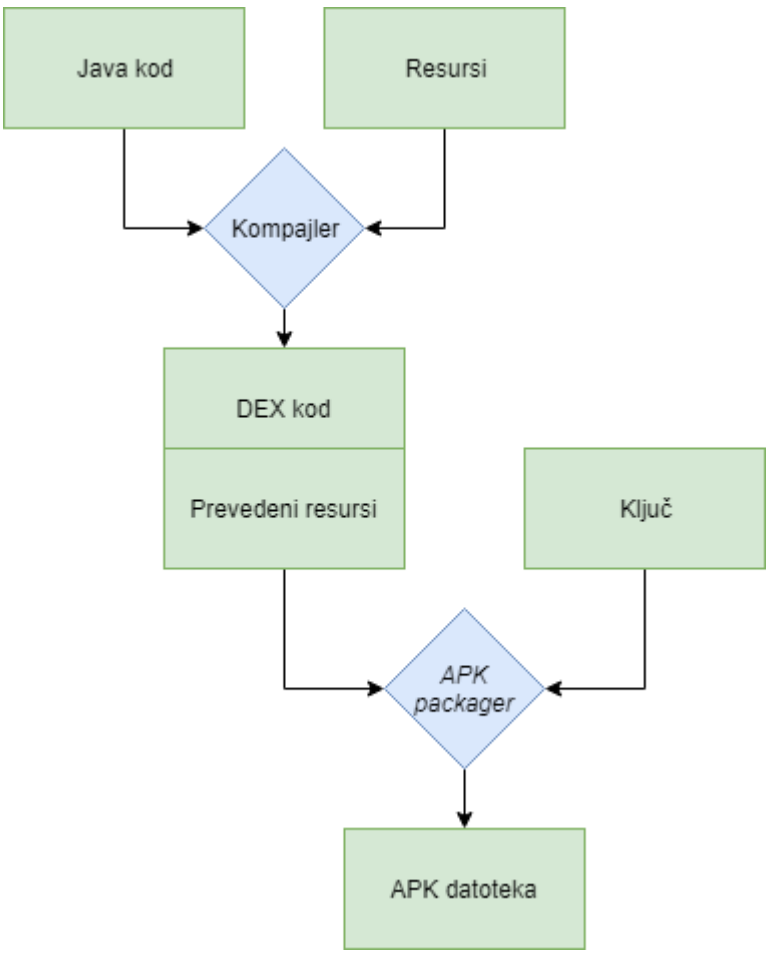
Rad je strukturiran na slijedeći način. U drugom poglavlju su objašnjeni neki bitni koncepti o Android aplikacijama poput izgradnje aplikacije, komponenti aplikacija i sigurnosnog modela aplikacija. U trećem poglavlju je detaljno objašnjen format APK datoteke i smali kod. Četvrto poglavlje se bavi Java dekompajlerima, dan je pregled Java dekompajlera koji se će se koristiti u ovom radu. Također prikazano je kako je moguće te Java dekompajlera programski koristiti. Peto poglavlje obrađuje razvoj dodataka za Android Studio u IntelliJ IDEA razvojnom okruženju. U njemu je opisano kako podesiti IntelliJ IDEA okruženje za razvoj dodatka, koja je struktura dodatka i kako razviti dodatak. U šestom poglavlju je opisan razvijeni dodatak za potporu sigurnosne analize Android aplikacija, njegove mogućnosti i kako ga koristiti. Na kraju rada dan je kratak zaključak i pregled korištene literature.

2. Android aplikacije

U ovom poglavlju je opisan proces izgradnje Android aplikacija iz izvornog koda. Također opisane su osnovne komponente Android aplikacija i kako one međusobno komuniciraju i kako komuniciraju s operacijskim sustavom. Na kraju je opisan sigurnosni model aplikacija.

2.1. Izgradnja Android aplikacije

Prije opisa sadržaja APK datoteke, opisan je proces izgradnje Android aplikacije u APK datoteku. Aplikacije se najčešće razvijaju koristeći neko od integriranih razvojnih okruženja, poput Android Studio-a koja podržavaju automatiziranu izgradnju aplikacija. Proces izgradnje [4] se sastoji od više koraka i koristi nekoliko različitih alata za izgradnju. Tipični proces izgradnje aplikacije u alatu Android Studio prikazan je na slici 2.1.



Slika 2.1. Izgradnja Android aplikacije u alatu Android Studio

U prvome koraku se kod aplikacije prevodi u DEX datoteke, a resursi koje aplikacija koristi u prevedene resurse. DEX (*engl. Dalvik executable*) je format prenosivog koda koji se izvršava na Dalvik virtualnom stroju ili u ART (*engl. Android Runtime*) virtualnom stroju

[5]. Slično kao i kod Java okruženja kod se prvo prevodi u Java međukod (*engl. bytecode*) i pohranjuje u *.class* datoteke. Zatim se *.class* datoteke pretvaraju pomoću alata *dx* u DEX format međukoda. Resursi [6] su naziv za dodatne statične podatke koji se koriste unutar aplikacije poput slika, rasporeda komponenti u korisničkom sučelju i predefiniranih znakovnih nizova. Neki od resursa se u prvome koraku prevode u jednu datoteku koju zovemo prevedeni resursi.

U drugome koraku process nazvan *APK Packager* kombinira prevedeni kod i resurse u jednu APK datoteku. Kako bi se datoteka mogla instalirati na uređaju potrebno ju je potpisati. U trećem koraku *APK Packager* potpisuje APK datoteku testnom ili produkcijskom inačicom ključa. Aplikaciju koja je potpisana produkcijskom inačicom ključa moguće je postaviti na Google-ovu trgovinu. U zadnjem koraku *APK Packager* koristi *zipalign* alat kako bi dodatno optimizirao aplikaciju kako bi koristila manje memorije prilikom izvršavanja. U prethodnom tekstu izostavljeno je objašnjenje nekih od navedenih pojmova radi preglednosti, a smatra se kako ih je bitno objasniti, pa su navedeni u slijedećem dijelu teksta.

Dalvik je virtualni stroj koja se koristio u verzijama Androida do 4.4. nakon koje je zamijenjena s ART okruženjem. Dalvik koristi *JIT* (*engl. Just-in-time*) prevođenje, tj. međukod se prevodi u strojni kod prilikom izvršavanja aplikacije i to se obavlja prilikom svakog pokretanja i uporabe aplikacije. Kod ART-a obavlja se *AOT* (*engl. Ahead-of-time*) prevođenje, gdje se prilikom instalacije aplikacije međukod prevodi u strojni kod i pohranjuje na uređaju. Ovaj proces se obavlja jednom prilikom instalacije aplikacije. Prednost ART-a u odnosu na Dalvik je ta što se manje koristi procesor prilikom pokretanja aplikacije jer se ne obavlja prevođenje koda i to rezultira manjom potrošnjom baterije. Nedostatak je taj što aplikacije koje se izvršavaju na ART-u zauzimaju više prostora zbog pohrane prevedenog koda.

2.2. Komponente Android aplikacije

Android aplikacije se sastoje od četiri tipa komponenti [10]:

- Aktivnosti (*engl. Activities*)
- Servisi (*engl. Services*)
- Višeodredišni primatelji (*engl. Broadcast receivers*)
- Pružatelji sadržaja (*engl. Content providers*)

Aktivnost je komponenta zadužena za interakciju s korisnikom. Jedna aktivnost predstavlja jedan ekran korisničkog sučelja. Aplikacije se sastoje od više aktivnosti koje su međusobno nezavisne, stoga druge aplikacije mogu pokrenuti neku od aktivnosti te aplikacije, ako to ona dozvoljava. Postoji nekoliko načina na koji su aktivnosti i operacijski sustav u interakciji. Operacijski sustav prati što se nalazi na korisničkom sučelju kako proces u kojem se izvršava aktivnost ne bi bio prekinut. Također za aktivnosti koje su prekinute, a korisnik će im se možda vratiti ne prekida procese. Interakciju dviju aktivnosti omogućava operacijski sustav. Aktivnosti se implementiraju nasljeđivanjem razreda `Activity`.

Servis je komponenta koja se koristi za obavljanje generičkog posla aplikacije u pozadini. Servisi ne pružaju korisničko sučelje i mogu se izvršavati u pozadini dok korisnik koristi neku drugu aplikaciju. Neka druga komponenta može pokrenuti servis ili biti u interakciji sa servisom. Obično se razlikuju dva načina na koji se servisi izvršavaju i kako operacijski sustav barata s njima. Servis se može izvršavati dok ne odradi neki posao ili se može

kontinuirano izvršavati u pozadini pri čemu je korisnik toga svjestan i ne želi da se taj servis prekine stoga se sustav trudi održati taj proces aktivnim. Servisi mogu biti i vezani za neku aplikaciju ili sustav na način da ih oni koriste, tj. taj servis njima pruža aplikacijsko sučelje. Iz toga razloga operacijski sustav zna da aplikacija ovisi o tome servisu i ne prekida proces koji izvršava taj servis. Servisi se implementiraju nasljeđivanjem razreda `Service`.

Višeodredišni primatelj je komponenta koja omogućava operacijskom sustavu da razašilje događaje aplikacijama na koje one mogu reagirati. Višeodredišni primatelj je samo sučelje između drugih komponenti. Događaji mogu biti razašiljani i aplikacijama koje se trenutno ne izvršavaju, nego bivaju pokrenute kada se desi specifičan događaj. Aplikacije također mogu inicirati razašiljanje nekog događaja kako bi obavijestile druge aplikacije o nečemu. Ova komponenta tipično ne prikazuje dijelove korisničko sučelja, ali može prikazati notifikaciju kako bi obavijestila korisnika o nekom događaju. Višeodredišni primatelj se implementira nasljeđivanjem razreda `BroadcastReceiver` pri čemu je svako razašiljanje predstavljeno objetkom tipa `Intent`.

Pružatelj sadržaja je komponenta koja upravlja zajedničkim skupom podataka koje je moguće pohraniti u SQLite bazu podataka, na webu, datotečnom sustavu ili na bilo kojem mjestu kojemu aplikacija može pristupiti. Kroz pružatelje sadržaja aplikacije mogu tražiti neke podatke ili ih modificirati ako imaju odgovarajuće dozvole. Aplikacije mapiraju podatke na URI imena, pri čemu onda mogu te URI predati drugim komponentama u sustavu da ih koriste za pristup tim podatcima. URI imena podataka postoje čak i kada aplikacija koja je vlasnik tih podataka nije aktivna. Samo onda kada je potrebno pristupiti tim podatcima aplikacija se mora izvršavati. Također URI imena predstavljaju dodatan sigurnosni sloj, aplikacija može onemogućiti pristup pružatelju usluge tih podataka tako da druge aplikacije nisu u mogućnosti pristupati tim podatcima. Ukoliko neka druga aplikacija zatraži pristup tim podatcima, sustav joj može dati privremenu dozvolu da pristupim samo podatcima predstavljenim tim URI-em. Pružatelj sadržaja se implementira nasljeđivanjem razreda `ContentProvider`.

Jedna od glavnih značajki Android operacijskog sustava je ta da bilo koja aplikacija može pokrenuti komponentu druge aplikacije, odnosno ne obavlja to sama aplikacija, već aplikacija šalje poruku operacijskom sustavu da želi pokrenuti određenu komponentu i sustav će to napraviti. Kada sustav pokreće komponentu neke aplikacije, pokrenuti će proces te aplikacije i instancirati odgovarajuće razrede komponente. Zbog toga Android aplikacije nemaju tradicionalnu glavnu metodu koja je početna točka aplikacije.

Aktivnosti, servisi i višeodredišni primatelji se aktiviraju korištenjem asinkronih poziva nazvanih *Intent*. Objekti tipa *Intent* povezuju pojedine komponente prilikom njihovog izvršavanja. To su ustvari poruke koje šalje komponenta kada zahtijeva akciju od drugih komponenta pri čemu ta komponenta može pripadati istoj aplikaciji ili nekoj drugoj. Za aktivnosti i servise *Intent* definira akciju koju je potrebno izvršiti i može specificirati URI podataka nad kojima je potrebno tu akciju izvršiti. Za višeodredišne primatelje *Intent* definira obavijest koja se razašilje unutar sustava. Pružatelji usluga se aktiviraju kada *ContentResolver* pokrene zahtjev za podatcima ili transakcijom nad njima. *ContentResolver* pruža dodatan sloj apstrakcije između komponente i pružatelja usluga, na način da obavlja transakcije nad podatcima u ime komponente.

2.3. Sigurnosni model Android aplikacija

Svaka Android aplikacija se izvršava unutar svoje zaštićene okoline (*engl. security sanbdox*) [10]. Aplikacija unutar Android operacijskog sustava koristi jedan korisnički identifikator. Operacijski sustav dodjeljuje svakoj aplikaciji jedinstveni Linux identifikator koji je poznat samo sustavu. Dozvole pristupa datotekama su podešene tako da samo ona aplikacija kojoj te datoteke pripadaju ima pravo pristupa njima. Procesi se izvršava unutar svojeg virtualnog stroja tako da se kod aplikacija izvršava međusobno izolirano. Svaka aplikacija se izvršava unutar svog procesa pri čemu taj proces pokreće operacijski sustav kada je potrebno izvršiti neku od komponenti te aplikacije. Sustav zaustavlja proces aplikacije kada više nije potrebno izvršavati neku od komponenti aplikacije.

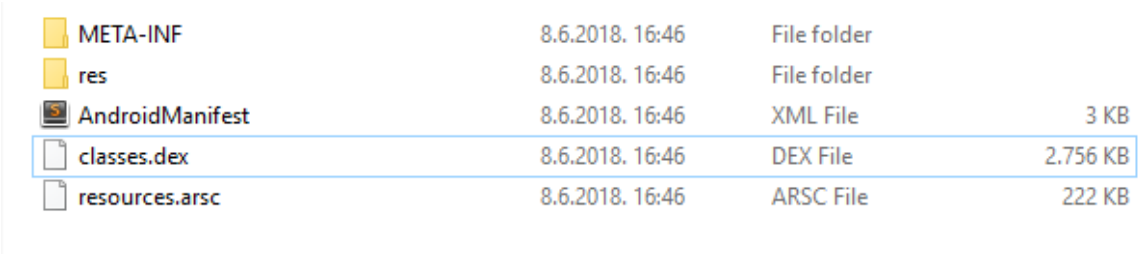
Na ovaj način Android operacijski sustav implementira princip najmanjih privilegija, svaka aplikacija ima pristup samo komponentama koje su joj potrebne za njezino izvršavanje. Aplikacija ne može pristupati dijelovima sustava za koje nema dozvole. Aplikacija može zatražiti dozvolu pristupa sistemskim servisima poput slanja poruka, pristupa Internetu ili Bluetooth-u pri čemu korisnik mora eksplicitno odobriti prava. Dvjema aplikacijama može biti dodijeljen isti Linux korisnik, pri čemu one onda mogu međusobno pristupati datotekama. Osim toga kako bi se uštedjeli resursi aplikacije s istim korisničkim identifikatorom se mogu izvršavati unutar istog virtualnog stroja i istog Linux procesa. U slučaju kada su dvije aplikacije predstavljene istim korisnikom one moraju biti potpisane istim certifikatom.

3. APK datoteke

U ovom poglavlju je opisana struktura APK datoteke, tj. od čega se sve ona sastoji i koja je uloga tih komponenti. Jedan od bitnijih komponenti je tzv. manifest datoteka koje je detaljnije opisana. Također navedeni su neki alati koji nam olakšavaju razumijevanje APK datoteke. Na kraju je dan pregled Smali koda koji predstavlja čitljivu reprezentaciju Dalvik međukoda i ponekad je bitno razumjeti ga.

3.1. Struktura APK datoteke

Kao što je već prije navedeno, APK je format u kojem se Android aplikacije prenose i instaliraju na uređaje. APK datoteka je ustvari ZIP arhiva koja sadrži datoteke koje sačinjavaju Android aplikaciju. Promjenom ekstenzije APK datoteke iz .apk u .zip moguće ju je raspakirati ili otvoriti koristeći neki od postojećih alata za manipulaciju ZIP arhivama. Sadržaj APK datoteke [7] nakon što je raspakirana prikazana je na slici 3.1.



Slika 3.1. Sadržaj APK datoteke

META-INF je direktorij koji sadrži slijedeće datoteke: MANIFEST.MF, CERT.RSA i CERT.SF. MANIFEST.MF je manifest datoteka koja sadrži listu svih datoteka i njihovih SHA-1 sažetaka. Datoteka CERT.RSA sadrži lanac certifikata javnog ključa koji se koristio za potpisivanje sadržaja APK datoteke. CERT.SF slično kao i manifest datoteka sadrži listu datoteka sa sažetcima te sadrži dodatno sažetak manifest datoteke.

Res direktorij sadrži više poddirektorija u kojima se nalaze resursi koji nisu prevedeni u *resources.arsc* datoteku. XML datoteke unutar tih direktorija su u nečitljivom binarnom formatu.

Datoteka *classes.dex* sadrži prevedeni kod u DEX formatu koji se izvršava na virtualnom stroju.

Datoteka resources.arsc sadrži prevedene resurse. Prilikom izgradnje aplikacije, sve XML datoteke unutar *res/values/* projektnog direktorija se prevode u binarni oblik i pakiraju u resources.arsc. Prilikom razvoja Android aplikacije u *res/values/* direktoriju se pohranjuju podatci u obliku znakovnih nizova kojima se onda može pristupati.

Posljednja komponenta APK datoteke je AndroidManifest datoteka. S obzirom da ona sadrži informacije koje su iz perspektive ovoga rada zanimljive i potrebno ju je detaljnije objasniti.

AndroidManifest [8] je XML datoteka koja se sastoji od različitih informacija koje su bitne za alate koji izgrađuju aplikaciju, operacijski sustav i Google-ovu trgovinu. AndroidManifest je također u binarnom XML format i kao takav nije čitljiv. Binarne XML datoteke iz APK datoteke je moguće pretvoriti u čitljiv XML format koristeći alat *apktool* [9]. Navedeni alat potpomaže reverzanje Android aplikacija i nudi nekoliko mogućnosti: dekodiranje resursa u originalni oblik, dobivanje smali koda iz DEX datoteke te ponovnu izgradnju APK datoteke iz smali datoteka. Naredbom `apktool decode <apk_datoteka>` apktool će dekodirati binarne XML datoteke u čitljivi oblik i producirati smali kod iz DEX datoteke.

Primjer dekodiranog sadržaja AndroidManifest datoteke prikazan je u ispisu 3.1.

```
1  <?xml version="1.0" encoding="utf-8" standalone="no"?><manifest
       xmlns:android="http://schemas.android.com/apk/res/android"
       package="com.example.comp.myapplication"
       platformBuildVersionCode="26"
       platformBuildVersionName="8.0.0">
2      <meta-data android:name="android.support.VERSION"
       android:value="26.0.0-alpha1"/>
3      <application android:allowBackup="true" android:debuggable="true"
       android:icon="@mipmap/ic_launcher" android:label="@string/app_name"
       android:roundIcon="@mipmap/ic_launcher_round"
       android:supportsRtl="true"
       android:theme="@style/AppTheme">
4          <activity
             android:name="com.example.comp.myapplication.MainActivity">
5              <intent-filter>
6                  <action android:name="android.intent.action.MAIN"/>
7                  <category
                   android:name="android.intent.category.LAUNCHER"/>
8              </intent-filter>
9          </activity>
10     </application>
11 </manifest>
```

Ispis 3.1. Sadržaj jedne AndroidManifest datoteke

U korijenskom elementu (`manifest`) je obavezan atribut koji predstavlja korijenski paket aplikacije, u navedenom primjeru to je `com.example.comp.myapplication` (linija 1). Ovaj atribut se koristi prilikom izgradnje aplikacije kako bi se pronašao kod odgovarajućih komponenti.

U `meta-data` elementu su definirane dodatne proizvoljne informacije u obliku parova *ime-vrijednost* (linija 2).

Element `application` definira cijelu aplikaciju (linija 3). Unutar njega se nalaze elementi koji definiraju komponente aplikacije i sadrži atribute koji se odnose na te komponente. Atributi poput `icon` i `label` definiraju pretpostavljene vrijednosti za pripadajuće atribute u komponentama, a atributi poput `debuggable` i `allowBackup` su definirani za cijelu aplikaciju i njihove vrijednosti nije moguće mijenjati.

Element `activity` deklarira aktivnost koja definira jedan ekran korisničkog sučelja aplikacije (linija 4). Atribut `name` definira razred koji implementira taj ekran. Unutarnji element `intent-filter` definira s kojim drugim komponentama aktivnost može biti u interakciji (linija 5). Osim `activity` komponente postoje još `service`, `receiver` i `provider` komponente, gdje se za svaku implementiranu komponentu aplikacije definira jedan element. Detaljno objašnjenje komponenti aplikacija dano je u poglavlju 2.2.

Bitan element koji nije vidljiv u ovom primjeru manifest datoteke je `uses-permission`. Kako bi aplikacije mogle pristupiti osjetljivi podatcima na uređaju (poput SMS poruka i kontakata) te koristiti mogućnosti operacijskog sustava (pristup Internetu i kameri) one moraju tražiti dozvolu za to. Dozvole se definiraju unutar `uses-permission` elementa. Ukoliko aplikacija zahtijeva pristup Internetu dozvolu je potrebno definirati ovako:

```
<uses-permission
    android:name="android.permission.INTERNET"/>.
```

Ovdje nisu navedeni svi elementi koji se mogu pojaviti u manifest datoteci, već samo najbitniji i najzanimljiviji za razumijevanje manifest datoteke.

3.2. Smali kod

Kao što je navedeno APK datoteka sadrži prevedeni Dalvik međukod u datoteci *classes.dex*. Taj kod je u binarnom obliku i nije prikladan za čitanje i modifikaciju. Smali kod predstavlja čitljivu reprezentaciju Dalvik međukoda koju je lakše modificirati. Smali je važan zato što dekompajleri često ne mogu dati potpun Java kod iz prevedenog koda u dex formatu, dok je uvijek moguće iz Dalvik koda dobiti potpun smali kod. Iz toga razloga kod reverzanja i analize Android aplikacija je ponekad potreban smali kako bi se u potpunosti mogao rekonstruirati i razumjeti dio aplikacije. Smali kod se može dobiti iz Dalvik međukoda koristeći baksmali [11] alat. U poglavlju 3.1. navedeno je da alat apktool također producira smali kod iz APK datoteke, razlog tome je što je baksmali integriran u apktool, stoga je za potrebe ovog rada korišten apktool alat.

Dalvik međukod podržava dvije vrste tipova podataka: primitivni tip i reference [12]. Polja i objekti su reference, a sve ostalo su primitivni tipovi podataka. Primitivni tipovi su reprezentirani jednim slovom. Tablica 3.1. daje popis primitivnih tipova podataka i njihovih imena.

Reprezentacija tipa podataka	Puno ime tipa podataka
V	void – koristi se samo kao povratna vrijednost
Z	boolean
B	byte
S	short
C	char
I	integer
J	long (64 bita)
F	float
D	double (64 bita)

Tablica 3.1. Primitivni tipovi podataka Dalvik međukoda

Objekti se imenuju na slijedeći način `Lpaket/ime/ImeObjekta;`:

- Fiksni znak `L` označava da se radi o objektu
- `paket/ime` je paket u kojem se taj objekt nalazi
- `ImeObjekta` predstavlja ime objekta

Npr. `Ljava/lang/String;` je ekvivalent `java.lang.String` u Java programskom jeziku.

Polja se označavaju nizom „`[S`“ čiji elementi imaju slijedeće značenje:

- `[` označava da se radi o jednodimenzionalno polju, u općem slučaju može biti više znakova `[` te se tako označavaju višedimenzionalna polja
- `S` označava da polje sadrži elemente tipa *short*

Analogno moguće je polje objekata, npr. jednodimenzionalno polje znakovnih nizova se definira ovako: `[java/lang/String;`.

Metode se imenuju tako da uključuju objekt koji sadrži metodu, ime metode, tipove parametara i tip povratne vrijednosti. Forma imenovanja metode prikazana na slijedećem konkretnom primjeru: `Lpaket/ime/ImeObjekta;->ImeMetode(IC)V`:

- `Lpaket/ime/ImeObjekta;` označava objekt koji sadrži metodu
- `ImeMetode` predstavlja ime metode
- `I, C` su fiksni argumenti navedene metode tipa integer i char, argumenti se uvijek navode jedan iza drugoga bez dodatnih znakova
- `V` je povratna vrijednost metode tipa void

Atributi klasa su imenovani slično kao i metode, tako da uključuju tip koji sadrži metodu, ime atributa i tip atributa. Primjerice `Ljava/lang/Integer;->MAX_VALUE:I;` je atribut koji se nalazi u klasi `Integer`, imena `MAX_VALUE` i cjelobrojnog je tipa.

Kod Dalvik virtualnog stroja registri su 32-bitni, a za pohranu 64-bitnih vrijednosti se koriste dva registra [13]. Broj registara unutar metode može se definirati na dva načina: pozivom `.registers` instrukcije koja definira ukupan broj registara u metodi, alternativno, pozivom instrukcije `.locals` koja definira broj registara pri čemu nisu uključeni oni koji se koriste za argumente metode. Navedene instrukcije se izvršavaju na početku metode i

služe za definiranje broja registara. Ukupan broj registara uključuje i one koji se koriste za argumente metode.

Kada je neka metoda pozvana njezinih *n* argumenata se stavlja u zadnjih n registara. Prvi registar kod nestatičkih metoda je uvijek objekt nad kojim se metoda poziva odnosno `this`. Primjerice metoda je definirana na slijedeći način: `MojObjekt;->poziv(CC)V;`. Ova metoda ima dva parametra tipa *char* te implicitni parametar tipa `MojObjekt`. Ako su specificirana četiri registra koristeći instrukciju `.registers 4` ili instrukcije `.locals 2` u registrima v2 i v3 se nalaze parametri metode, u registru v1 se nalazi implicitni parametar. Za statičke metode vrijedi je isto, samo nema implicitnog parametra.

Registri su imenovani na slijedeći način: korištenjem slova `v` i korištenjem slova `p`. Ukoliko metoda koristi četiri registra oni će biti nazvani `v0-v3`, odnosno registri koji se koriste za parametre metode (`v1-v3`) su još nazvani `p0-p2`. Registri koji sadrže parametre se mogu referencirati na oba načina.

4. Java dekompajleri

U ovome poglavlju su prikazani neki od Java dekompajlera. Na početku je navedeno nešto osnovno o dekompajlerima i u što se prevodi Java kod i kako se izvršava. Za dekompajlere koji se koriste u daljnjim djelovima ovoga rada prikazano je kako ih koristiti i koje su njihove mogućnosti. Također za svaki od tih dekompajler je prikazano kako ih integrirati programski. Na kraju je objašnjen alat *dex2jar* koji transformira DEX datoteke u JAR datoteke, s obzirom da većina dekompajlera ne radi direktno s DEX kodom.

4.1. Java dekompajleri

Kao što je već navedeno dekompajleri su alati koji iz prevedenog koda pokušavaju rekonstruirati izvorni kod. Naravno, proces nije savršen, tj. ti alati nekada neće moći u potpunosti rekonstruirati izvorni kod iz prevedenog. Ova činjenica je bitna zato što je kod Android aplikacija moguće rekonstruirati potpuni smali kod, a iz njega se može zaključiti kako je izgledao izvorni kod.

Java kod se prevodi u Java međukod koji se izvršava u Java virtualnom stroju (*engl. Java Virtual Machine, JVM*) [14]. Java virtualni stroj je definiran specifikacijom, što osigurava prenosivost Java koda na različite platforme. Programe pisane u jezicima osim Jave također je moguće izvršavati u Java virtualnom stroju ukoliko su oni prevedeni u Java međukod. Java prevoditelj prevodi izvorni kod u *.class* datoteke koje sadrže Java međukod. Java međukod se zatim izvršava na ciljnoj platformi koristeći interpreter. Kod koji se interpretira će se izvršavati sporije nego kod koji je preveden u strojni kod platforme, stoga se koristi *JIT* (*engl. Just-in-time*) prevodioc koji će prilikom izvođenja programa prevoditi njegov kod u strojni kod platforme. Važno je napomenuti da neke implementacije Java virtualnog stroja koriste samo *JIT* prevodioc, a neke uključuju i interpreter [14].

U poglavlju 2.1. je navedeno da se Android aplikacije prevode u Dalvik međukod (DEX) koji se izvršava na Dalvik ili ART virtualnom stroju na sličan način kao i Java aplikacije.

Java aplikacije i biblioteke često su pohranjene u obliku JAR (*engl. Java archive*) datoteke [15] dok s druge strane Java dekompajleri većinom kao ulaznu točku prihvaćaju skup *.class* datoteka. S obzirom da je kod Android aplikacija u dex formatu, potrebno je pretvoriti DEX datoteku u JAR. To je moguće alatom *dex2jar* koji je detaljnije opisan u poglavlju 4.3.

4.2. Korišteni Java dekompajleri

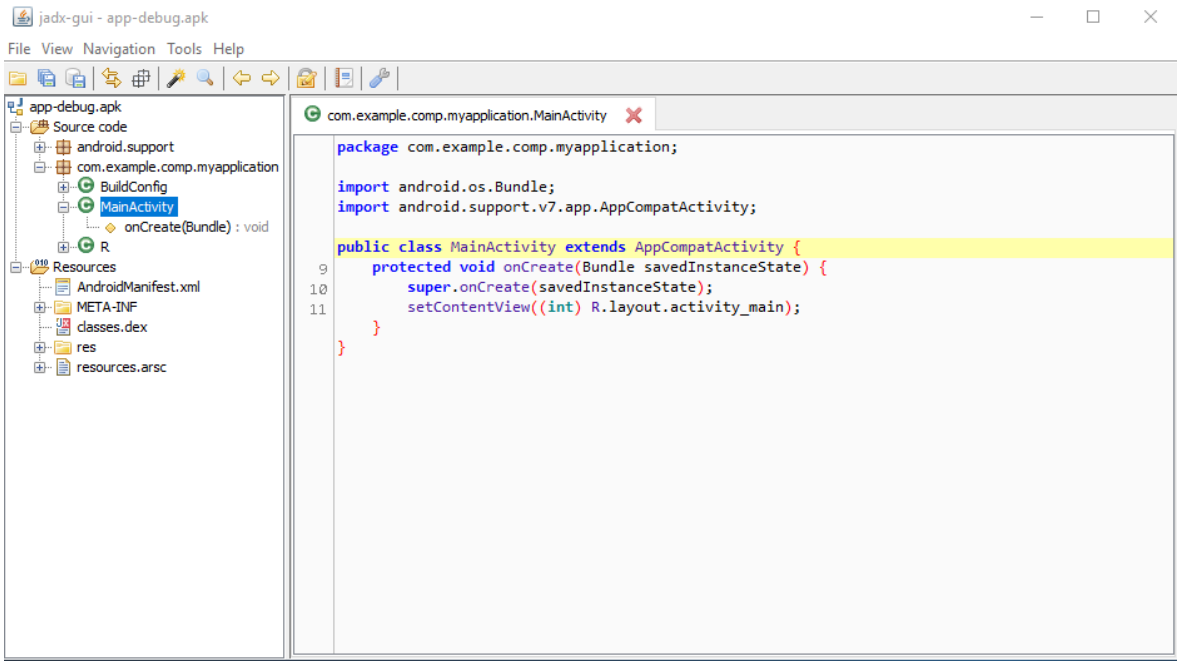
Za potrebe ovoga rada korišteni su Jadx, Procyon i Fernflower dekompajleri. Sva tri navedena dekompajlera su besplatni za korištenje i otvorenog koda. Također svi se još uvijek aktivno razvijaju čime postaju kvalitetniji s vremenom. Osim navedenih Java dekompajlera postoje i drugi, kao što su CFR [16], Krakatau [17] i JD-GUI [18].

Jadx [19] nije tipični Java dekompajler, već je specijalizirani dekompajler za Android aplikacije koji iz DEX i APK datoteka producira Java kod. Osim DEX i APK datoteka kao ulaz može primiti JAR i .class datoteke. Jadx osim što dekompajlira međukod, također dekodira binarne resurse u čitljivi oblik. Dostupan je kao alat koji se može koristiti iz komandne linije, a postoji i grafičko sučelje. Kao i većina dekompajlera podržava dodatne

mogućnosti koje se definiraju kao dodatni argumenti u komandnoj liniji. Neke od tih mogućnosti su:

- Broj dretvi koje se koriste prilikom dekompajliranja – omogućava nešto brže dekompajliranje
- Isključivo dekodiranje resursa
- Isključivo dekompajliranje koda
- Prikazivanje koda koji se pogrešno dekompajlirao
- Spremanje koda kao Android gradle projekt
- Korištenje deobfuskacije
- Onemogućavanje korištenja uvoza imena, već korištenje cjelokupnih imena paketa

Unutar grafičkog sučelja postoje mogućnosti pretraživanja koda i navigacije po njemu. Primjer korištenja Jadx dekompajlera iz njegovog grafičkog sučelja dan je na slici 4.1.



Slika 4.1. Grafičko sučelje Jadx dekompajlera

Prilikom pokretanja potrebno je odabrati APK datoteku iz koje želimo dekompajlirati DEX kod. Nakon što je odabrana APK datoteka pokreće se dekompajliranje i u statusnoj traci se prikazuje napredak dekompajliranja. U lijevom dijelu sučelja se nalazi dekompajlirani kod u dijelu *Source code* te sadržaj APK datoteke s dekodiranim resursima u dijelu *Resources*. Opcija *Navigation* nudi opcije pretraživanja i navigacije po kodu. Dodatne mogućnosti se mogu podesiti u opciji *File* > *Preferences* nakon čega je potrebno pokrenuti dekompajliranje ponovo s opcijom *File* > *Open file*. U opciji *Tools* se nalazi alat za dekompajliranje uz deobfuskaciju te preglednik log datoteke u kojoj je detaljno zabilježeno koji dijelovi nisu uspješno dekompajlirani.

Procyon [20] je skup alata za analizu i generiranje koda. Sastoji se od slijedećih komponenti, tj. biblioteka:

- *Core* radni okvir
- *Reflection* radni okvir
- *Expression* randi okvir
- Alati za kompajliranje
- Java dekompajler

U sklopu ovoga rada objašnjen je Java dekompajler. Prema navodima autora [21] dekompajler uspješno barata s naprednijim mogućnostima Java programskog jezika koje su dodane u verzijama Jave 5 i više. Neke od tih konstrukata su: deklaracije enumeracija, *switch* naredbe koje sadrže znakovne tipove podataka i enumeracije, lokalne i anonimne klase, anotacije te lambda izrazi. Procyon je dostupan kao komandno linijska aplikacija u obliku JAR datoteke. Kao ulaz prima *.class* datoteke ili JAR datoteku. Primjerice ako se želi dekompajlirati JAR datoteka poziv dekompajlera izgleda ovako:

```
# java -jar procyon.jar -jar moj_jar.jar -o izlaz [dodatni_argumenti]
```

Navedeni poziv kroz drugi argument `-jar` kao ulaznu datoteku prima datoteku *moj_jar.jar* i dekompajlirani Java kod pohranjuje u direktorij nazvan *izlaz*, `[dodatni_argumenti]` predstavljaju dodatne neobavezne argumente dekompajlera koje mogu biti uključeni. Neke od dodatnih mogućnosti Procyon dekompajlera su slijedeće:

- Spajanje varijabli – rezultira manjim brojem deklaracija varijabli i sažetijim kodom
- Korištenje eksplicitnih *import* naredbi
- Korištenje točnih tipova kod generičkih metoda
- Ostavljanje eksplicitnih *cast* konstrukcija
- Produciranje sintaksnog stabla međukoda umjesto izvornog koda
- Produciranje čistog međukoda umjesto Java koda
- Prikazivanje neoptimiziranog koda
- Isključivanje ugniježđenih tipova

Navedene mogućnosti se definiraju kao dodatni komandno linijski argumenti prilikom poziva dekompajlera. Procyon nudi i programsko sučelje koje se može koristiti za integraciju u drugim aplikacijama.

Fernflower [22] je Java dekompajler otvorenog koda koji se još uvijek razvija. Dostupan je u obliku izvornog koda na navedenom Github repozitoriju, te dolazi u obliku kompajlirane biblioteke (JAR datoteka) uz IntelliJ IDEA razvojno sučelje. Biblioteka se nalazi u instalacijskom direktoriju IntelliJ IDEA alata u *plugins\java-decompiler\lib* direktoriju. Slično kao i Procyon dekompajler Fernflower dekompajler je dostupan kao komandno linijski alat. Kao ulaz prima *.class*, JAR ili ZIP datoteke ili direktorij pri čemu se direktorij rekurzivno pretražuje te kao izlaz producira dekompajlirani Java kod u obliku JAR datoteke. Primjer u kojem se dekompajlira JAR datoteka:

```
# java -jar fernflower.jar [dodatni_argumenti] moj_jar.jar izlaz
```

Navedeni poziv Fernflower dekompajlera producira JAR datoteku s izvornim kodom iz datoteka *moj_jar.jar* te ju pohranjuje u direktorij *izlaz*. `[dodatni_argumenti]` označavaju neobavezne dodatne mogućnosti dekompajlera. Neke od dodatnih mogućnosti FernFlower dekompajlera su slijedeće:

- Dekompajliranje ugniježđenih razreda
- Dekompajliranje tvrdnji (*engl. assertions*)
- Sakrivanje praznog pretpostavljenog konstruktora

- Sakrivanje praznog *super* poziva
- Dekompajliranje enumeracija
- Pretvaranje lambda izraza u anonimne razrede prilikom dekompajliranja
- Preimenovanje obfusciranih razreda i njihovih elemenata

Navedene dodatne mogućnosti se definiraju kao komandno linijski argumenti prilikom poziva dekompajlera na slijedeći način: `[opcija]=vrijednost` pri čemu je vrijednost 0 kada je opcija deaktivirana ili 1 kada je aktivirana.

4.3. Korištenje odabranih Java dekompajlera u programima

U sklopu ovog rada navedeni Java dekompajleri su korišteni programski u razvijenom dodatku. S obzirom da se dodatak razvija u IntelliJ IDEA alatu koristeći Java programski jezik, te da su dekompajleri dostupni u obliku JAR biblioteka razvijeni su Java razredi koji omataju funkcionalnosti dekompajlera. Ti razredi iskorištavaju funkcionalnosti dekompajlera na način da ih se poziva uz mogućnost navođenja opcionalnih argumenata.

Jadx dekompajler nudi sučelje [19] s kojim je jednostavno iskoristiti njegovu funkcionalnost. Sučelje se nalazi u razredu *jadx.api.JadxDecompiler.java*. Navedeni razred u konstruktoru prima opcionalne argumente koji su tipa *JadxArguments*. Objekt tipa *JadxDecompiler* je moguće instancirati bez i sa dodatnim argumentima. Metoda *loadFile* u razredu *JadxDecompiler* prima datoteku koja se dekompajlira i to može biti APK, JAR, *.class* ili DEX datoteka. Metoda *setOuputDir* služi za definiranje izlaznog direktorija u koji će se pohraniti dekompajlirani kod. Na kraju još jedna bitna metoda je *save* koja nema argumenata i ona pokreće dekompajliranje. Za ovaj rad razvijena je klasa *JadxWrapper* čiji je dio koda prikazan u ispisu 4.1.

```
1  public class JadxWrapper {
2      private File apkFile;
3      private File outDir;
4      private JadxArgs arguments;
5      private JadxDecompiler decompiler;

6      public JadxWrapper(File apkFile, File outDir, String args) {
7          this.arguments = parseArguments(args);
9          this.apkFile = apkFile;
10         this.outDir = outDir;
11         this.decompiler = new JadxDecompiler(arguments);
12     }

13     public JadxWrapper(File apkFile, File outDir) {
14         this.apkFile = apkFile;
15         this.outDir = outDir;
16         this.decompiler = new JadxDecompiler();
17     }

18     public void decompile() {
19         try {
20            decompiler.loadFile(apkFile);
21         } catch (JadxException e) {
22            System.out.println("Something went wrong,
                  jadx exception: ");
23            e.printStackTrace();
24         }
25         decompiler.setOutputDir(outDir);
26         decompiler.save();
27     }
       ...
28 }
```

Ispis 4.1. Dio koda razvijenog razreda *JadxWrapper*

Razred ima dva konstruktora, ovisno jesu li zadani opcionalni argumenti ili ne (linije 6 i 13). Argumenti se zadaju u obliku znakovnog niza, te kada su zadani pomoćna metoda *parseArguments* ih parsira i pretvara u *JadxArguments* objekt (linija 7). Metoda *decompile* obavlja dekompajliranje pozivanjem navedenih metoda *JadxDecompiler* razreda, pri čemu su ulazna datoteka i izlazni direktorij specificirani u konstruktoru (linije 19-26). Za korištenje navedenog koda potrebno je referencirati JAR biblioteku Jadx dekompajlera unutar projekta kako bi njegov kod bio dostupan. Primjer korištenja *JadxWrapper* razreda je dan u ispisu 4.2.

```
JadxWrapper wrapper = new JadxWrapper(apkFile, outDir, arguments);
Wrapper.decompiler();
```

Ispis 4.2. Isječak koda koji prikazuje korištenje *JadxWrapper* razreda

Procyon dekompajler nudi bogato programsko sučelje koje je opisano na Bitbucket stranicama dekompajlera [20]. Za ovaj rad nije bilo potrebno toliko detaljno sučelje već je iskorištena glavna (*engl. main*) metoda u razredu *com.strobel.decompiler.DecompilerDriver.java* koja predstavlja ulaznu točku dekompajlera. Glavna metoda prima polje znakovnih nizova koje predstavlja argumente te pokreće dekompajliranje. Za potrebe ovog rada razvijen je razred *ProcyonWrapper* koji omata funkcionalnost Procyon dekompajlera. Isječak koda iz navedenog razreda dan je u ispisu 4.3.

```
1  public class ProcyonWrapper {
2      private String outDir;
3      private String jarFile;

4      public ProcyonWrapper(String outDirPath, String jarFilePath) {
5         this.outDir = outDirPath;
6         this.jarFile = jarFilePath;
7      }

8      public void decompile(String... arguments) {
9         String[] args = null;

10        String[] parsedArgs = parseArguments(arguments);

11        args = new String[4 + parsedArgs.length];
12        args[0] = "-jar ";
13        args[1] = jarFile;
14        args[2] = "-o ";
15        args[3] = outDir;

16        for(int i = 0; i < parsedArgs.length; i++) {
17            args[4 + i] = parsedArgs[i];
18        }
19        com.strobel.decompiler.DecompilerDriver.main(args);
20    }
      ...
21 }
```

Ispis 4.3. Isječak koda iz *ProcoynWrapper* razreda

U konstruktoru razreda se definiraju ulazna JAR datoteka koja će biti dekompajlirana te izlazni direktorij u koji se pohranjuje kod (linije 5 i 6). Metoda *decompile* prima varijabilni broj znakovnih nizova koji predstavljaju dodatne argumente. U metodi se prvo s pomoćnom metodom *parseArguments* filtriraju argumenti tako da se izbace pogrešno definirani argumenti (linija 10). Zatim se konstruira polje znakovnih nizova koji su argumenti koji se predaju glavnoj metodi razreda *DecompilerDriver*. Na početku se dodaju argumenti koji definiraju ulaznu JAR datoteku i izlazni direktorij, te se zatim dodaju opcionalni argumenti ukoliko ih ima (linije 11-18). Kao i kod Jadx dekompajlera potrebno je referencirati JAR biblioteku Procyon dekompajlera unutar projekta za korištenje navedenog koda. Primjer korištenja je prikazan u ispisu 4.4.

```
ProcyonWrapper wrapper = new ProcyonWrapper(outDir, jarFile);
wrapper.decompile(arguments); //bez argumenata: wrapper.decompiler();
```

Ispis 4.4. Primjer korištenja *ProcyonWrapper* razreda

Fernflower dekompajler [22] slično kao i Procyon decompiler sadrži glavnu metodu u razredu *org.jetbrains.java.decompiler.main.decompiler.ConsoleDecompiler* i koja je ulazna točka dekompajlera. Glavna metoda prima argumente koji su u obliku polja znakovnih nizova i pokreće dekompajliranje. Razred *FernFlowerWrapper* iskorištava navedenu metodu i implementira funkcionalnost Fernflower dekompajlera. Ispis 4.5. prikazuje isječak koda navedenog razreda.

```
1  public class FernFlowerWrapper {
2      private String outDir;
3      private String jarFile;

4      public FernFlowerWrapper(String outDir, String jarFile) {
6          this.outDir = outDir;
7          this.jarFile = jarFile;
8      }

9      public void decompile(String... arguments) {
10         String[] args = new String[2 + arguments.length];

11         for(int i = 0; i < arguments.length; i++) {
12             args[i] = arguments[i];
13         }

14         args[arguments.length] = jarFile;
15         args[arguments.length + 1] = outDir;

16         org.jetbrains.java.decompiler.main.decompiler.
             ConsoleDecompiler.main(args);

17         File jar = new File(jarFile);
18         String jarName = jar.getName();

19         String toUnzip = new String(outDir + File.separator + jarName);
20         ZipUtility.unzip(toUnzip, outDir);

21         File toDelete = new File(toUnzip);
22         toDelete.delete();
23     }
       ...
24 }
```

Ispis 4.5. Isječak koda iz *FernFlowerWrapper* razreda

U konstruktoru se definiraju JAR datoteka koja će biti dekompajlirana i izlazni direktorij (linije 6 i 7). Zatim se konstruiraju argumenti u obliku znakovnih nizova, na početku su opcionalni argument, a na kraju se dodaju argumenti koji definiraju ulaznu datoteku i izlazni direktorij (linije 11-15). S obzirom da Fernflower dekompajler producira JAR datoteku na kraju se ta datoteka raspakirava i dobiva se izvorni kod u standardnom obliku (linije 16-20). Za korištenje navedenog koda također je potrebno referencirati Fernflower biblioteku u projektu. Ispis 4.6. prikazuje primjer korištenja.

```
FernFlowerWrapper wrapper = new FernFlowerWrapper(outDirectory, jarFile);
wrapper.decompile(argumetns); //bez argumenata: wrapper.decompile();
```

Ispis 4.6. Primjer korištenja *FernFlowerWrapper* razreda

Sav prikazani kod je dostupan na Github repozitoriju razvijenog dodatka [23].

4.4. Dex2jar alat

Dex2jar [24] je skup alata koji se koristi analizu DEX i *.class* datoteka. Sastoji se od slijedećih komponenti:

- *dex-reader/dex-writer* – alat za čitanje i pisanje DEX datoteka
- *d2j-dex2jar* – alat za transformaciju DEX datoteka u *.class*, tj. JAR datoteke
- *smali/baksmali* – alat za transformaciju dex koda u smali kod i obrnuto
- *d2j-decrypt-string* – alat za deobfuskaciju znakovnih nizova

Na Github repozitoriju je dostupan izvorni kod ovog alat, a dolazi i u obliku JAR biblioteka. U ovome radu je objašnjen *d2j-dex2jar* (dalje u tekstu *dex2jar*) alat koji je korišten za pretvaranje DEX datoteka u JAR datoteke jer neki dekompajleri ne mogu raditi direktno s APK i dex datotekama već ih je potrebno transformirati u JAR datoteku. Dex2jar je dostupan kao komandno linijski alat sa *batch* i *shell* skriptama koje pozivaju alat iz JAR datoteke. Primjer korištenja *dex2jar*-a za pretvaranje .dex datoteke u JAR datoteku je slijedeći:

```
# d2j-dex2jar.bat classes.dex -o moj_jar.jar
```

Navedeni poziv datoteku *classes.dex* transformira u datoteku *moj_jar.jar*, sa opcijom -o se definira put do izlazne datoteke. U ovom slučaju to je samo ime datoteke koja će biti kreirana u direktoriju iz kojeg je pozvana skripta. Dex2jar nudi neke dodatne mogućnosti koje se mogu navesti kroz argumente u komandnoj liniji. Neki od njih su:

- pohrana detaljnih informacija o iznimkama
- transformacija *debug* informacija
- isključivanje rukovanja pogreškama *dex2jar-a*
- optimizacija *synchronized* blokova
- ponovo iskorištavanje registara prilikom generiranja *.class* datoteka

Za dex2jar je također razvijen Java razred koji omata njegovu funkcionalnost. Slično kao i kod Fernflower i Procyon dekompajlera iskorištena je glavna metoda koja je ulazna točka dex2jar alata. Glavna metoda prima argumente u obliku polja znakovnih nizova, a ona se nalazi u razredu *com.googlecode.dex2jar.tools.Dex2jarCmd*. Razred *Dex2jarWrapper* omata funkcionalnost dex2jar alata. Ispis 4.7. prikazuje isječak koda iz navedenog razreda.

```
1  public class Dex2JarWrapper {
2    private String dexFile;
3    private String outFile;

4    public Dex2JarWrapper(String dexFile, String outFile) {
5         this.dexFile = dexFile;
6         this.outFile = outFile;
7    }

9    public void dex2jar(String... arguments) {
10        String[] args = null;
11        args = new String[3 + arguments.length];
12        args[0] = dexFile;
13        args[1] = "-o";
14        args[2] = outFile;

15        for(int i = 0; i < arguments.length; i++) {
16            args[3 + i] = arguments[i];
17        }

18        com.googlecode.dex2jar.tools.Dex2jarCmd.main(args);
19    }
      ...
20 }
```

Ispis 4.7. Isječak koda razreda koji omata funkcionalnost alata dex2jar

U konstruktoru se definiraju ulazna DEX datoteka i izlazna JAR datoteka pri čemu je za izlaznu datoteku potrebno navesti puni put do nje i njezino ime (linije 5 i 6). Metoda *dex2jar* prima varijabilni broj argumenata. Na početku metode se konstruira polje znakovih nizova koje predstavlja argumente, prvo se dodaje definirana DEX datoteka, a zatim se sa opcijom *-o* definira izlazna JAR datoteka (linije 10-14). Na kraju dolaze opcionalni argumenti i poziva se *dex2jar* alat koji će obaviti transformaciju (linije 16 i 18). Kako bi se ovaj kod mogao koristiti potrebno je referencirati *dex2jar* biblioteke u projektu. Primjer korištenja dan je u ispisu 4.8.

```
Dex2JarWrapper wrapper = new Dex2JarWrapper(dexFile, outJar);
wrapper.dex2jar(); //sa argumentima: wrapper.dex2jar(arguments);
```

Ispis 4.8. Primjer korištenje Dex2JarWrapper razreda

Navedeni kod je dostupan na Github repozitoriju razvijenog dodatka [23].

5. Dodatci za Android Studio

U ovome poglavlju je opisano kako razviti dodatak za Android Studio i IntelliJ IDEA integrirano razvojno okruženje. Na početku je ukratko opisana IntelliJ platforma otvorenog koda. Zatim je navedeno kako podesiti IntelliJ IDEA razvojno okruženje za razvoj dodataka. Slijedi kratak opis kako razviti jednostavan dodatak koristeći osnovne komponente. Na kraju su navedene i objašnjene najbitnije komponente i postupci koji se koriste prilikom razvoja dodataka u IntelliJ IDEA razvojnom okruženju.

5.1. IntelliJ platforma

IntelliJ platforma [25] je platforma otvorenog koda koju je razvila tvrtka JetBrains i služi za izgradnju integriranih razvojnih okruženja. Neki od razvojnih okruženja temeljenih na IntelliJ platformi su: IntelliJ IDEA, WebStorm, RubyMine te Android Studio. Platforma nudi uređivač teksta koji podržava automatsku nadopunu koda, označavanje koda i mnoge druge funkcionalnosti uređivanja teksta. Osim toga daje komponente s kojima se mogu izgraditi različiti dijelovi korisničkog sučelja. Također nudi aplikacijsko sučelje za izgradnju standardnih funkcionalnosti razvojnih okruženja poput projekata i njihovog modela i sustava za izgradnju koda u aplikacije. PSI sučelje (*engl. Program Structure Interface*) omogućava parsiranje datoteka i izgrađivanje sintaktičkih i semantičkih stabala koda i indeksiranje tih podataka. Ovo sučelje se često koristi prilikom napredne navigacije po kodu i refaktoriranja koda.

Dodatci su jedan dio IntelliJ platforme koji omogućava nadogradnju postojećih razvojnih okruženja s dodatnom funkcionalnosti. Veliki broj funkcionalnosti u IntelliJ IDEA razvojnom okruženju su razvijene u obliku dodataka koji se mogu uključiti ili isključiti po potrebi. Dodatci nadograđuju funkcionalnosti razvojnog okruženja na različite načine, primjerice mogu dodati neki jednostavni meni koji implementira neki mali dio dodatne funkcionalnosti do podrške za nove programske jezike i proces izgradnje aplikacija u tom jeziku. Najčešći oblici dodataka su slijedeći:

- Podrška za nove programske jezike
- Integracija postojećih radnih okvira
- Integracija postojećih alata
- Nadogradnja korisničkog sučelja

Podrška za nove programske jezike uključuje implementaciju prepoznavanja novih vrsti datoteka, označavanje sintakse koda, formatiranje koda, automatsko popunjavanje koda te leksičku analizu koda. Slijedeći oblik dodataka je integracija postojećih alata i radnih okvira. Radni okviri i alati se integriraju na način da se omogućava korištenje njihove funkcionalnost direktno u razvojnom okruženju. Nadogradnja korisničkog sučelja dodaje nove komponente u korisničko sučelje koje mogu pružati nove funkcionalnosti ili mogu biti samo vizualne modifikacije korisničkog sučelja.

5.2. Podešavanje IntelliJ IDEA razvojnog okruženja za razvoj dodataka

Prije nego što je moguće razvijati vlastite dodatke u IntelliJ IDEA razvojnom okruženju potrebno je podesiti nekoliko stvari. Koraci su slijedeći:

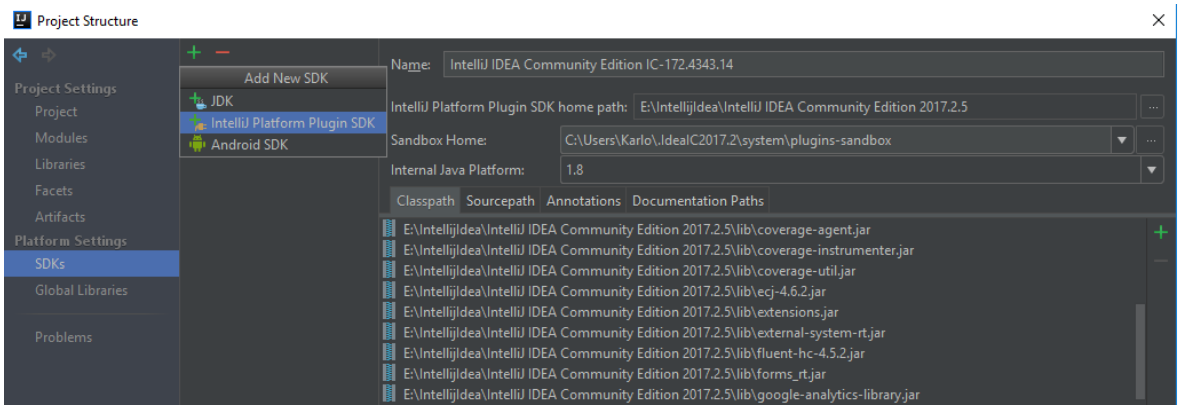
- Instalacija IntelliJ IDEA razvojnog okruženja
- Preuzimanje izvornog koda IntelliJ IDEA Community
- Omogućavanje dodatka *Plugin Devkit* IntelliJ IDEA-i
- Podesiti *IntelliJ Platform SDK* (*engl. Software Development Kit*) u projektu u kojem se razvija dodatak

Prvo je potrebno preuzeti i instalirati IntelliJ IDEA razvojno okruženje [26], instalacija se sastoji od nekoliko koraka i jednostavna je. Izvorni kod IntelliJ IDEA Community je potreban jer se mnoge komponente dodataka baziraju na tom kodu. Kod se nalazi na Github repozitoriju i najlakše ga je preuzeti sa slijedećom git naredbom:

```
git clone --depth 1 git://git.jetbrains.org/idea/community.git <odredište>
```

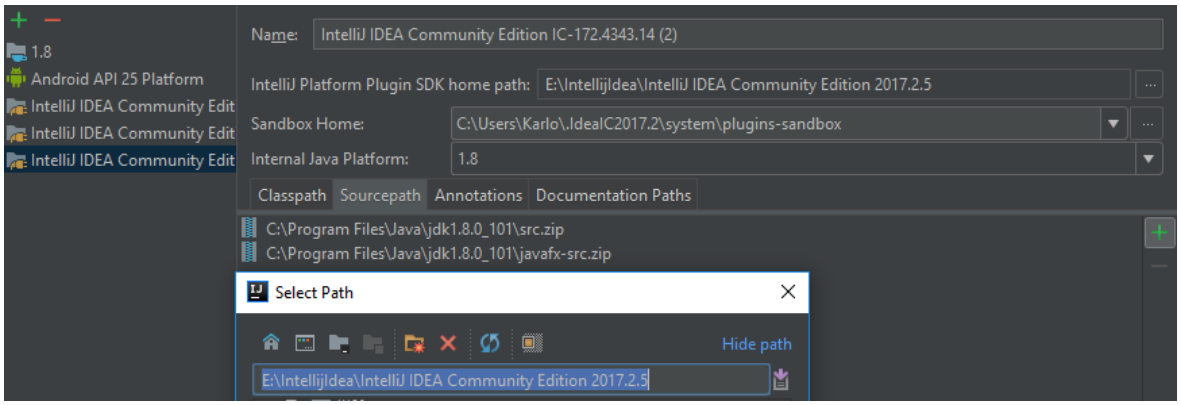
Opcionalno preuzeti kod se može izgraditi pokretanjem *ant* skripte build.xml, ali taj korak se može preskočiti.

U slijedećem koraku je potrebno definirati IntelliJ Platform SDK koji će se koristiti u projektima u kojima se razvijaju dodatci. U razvojnom okruženju se odabere *File* > *Project Structure* i zatim unutar menija *Platform Settings* > *SDK* dodati novi *IntelliJ Platform Plugin SDK*. Postupak je prikazan na slici 5.1.



Slika 5.1. Postupak definiranja IntelliJ Platform SDK-a

Zatim je u slijedeća dva koraka prvo potrebno odabrati za korijenski direktorij SDK-a direktorij u kojem je preuzeti kod IntelliJ IDEA Community te u drugom koraku za internu Java platformu odabrati predefinirani JDK (*engl. Java Development Kit*). Ukoliko JDK nije definiran analogno ga se definira tako da se odabere *Add New SDK* > *JDK* i za korijenski direktorij se odabere instalacija JDK-a na računalu. Zadnji korak u definiranju SDK-a je dodavanje puta do izvornog koda u *Sourcepath* meniju gdje je ponovo potrebno odabrati direktorij u kojem je preuzeti IntelliJ IDEA Community kod. Postupak je prikazan na slici 5.2.



Slika 5.2. Dodavanje puta do izvornog koda SDK-a

Dodatak *Plugin Devkit* se uključuje u postavkama razvojnog okruženja tako da se odabere *File* > *Settings* i u meniju *Plugins* uključi odabirom kućice sa strane.

Sada je prilikom kreiranja novog projekta za razvoj dodatka potrebno odabrati potrebno odabrati za projetki SDK kreirani IntelliJ Platform Plugin SDK. Projekt se kreira odabirom *File* > *New* > *Project...* i zatim odabere IntelliJ Platform Plugin.

5.3. Razvoj dodataka u IntelliJ IDEA razvojnom okruženju

Dodatci unapređuju razvojno okruženje implementirajući neku novu funkcionalnost. Te nove funkcionalnosti, nazvane akcije su u razvojnom okruženju predstavljene u obliku različitih menija i elemenata u alatnim trakama čijim odabirom se one izvršavaju. Akcije se organiziraju u grupe koje opet mogu unutar sebe sadržavati druge grupe akcija. Na ovaj način one mogu tvoriti više različitih podmenija unutar nekog menija ili alatne trake. Akcije se implementiraju nasljeđivanjem razreda *AnAction* koji sadrži metodu *actionPerformed* koja izvršava novu funkcionalnost svaki puta kada se odabere dodani meni ili element [28]. Kako bi se implementirala nova funkcionalnost potrebno je obaviti slijedeća dva koraka:

- Definirati vlastitu akciju naslijeđivanjem razreda *AnAction* i implementiranjem metode *actionPerformed*
- Registrirati akciju programski ili u datoteci *plugin.xml*

Primjer definicije jednostavne akcije dan je u ispisu 5.1.

```
1  public class Test extends AnAction {
2      public Test() {
3          super("_Test","Item description",
            IconLoader.getIcon("/package/icon.png"));
4          // super("_Test"); name only
5      }

6      public void actionPerformed(AnActionEvent event) {
7         Project project = event.getData(PlatformDataKeys.PROJECT);
8         // or event.getProject()
9         String txt= Messages.showInputDialog(project, "What's 2 + 2?",
            "2 + 2", Messages.getQuestionIcon());
10        int ans = Integer.parseInt(txt);
11        String message;
12        if(ans == 4) {
13            message = "Correct.";
14        } else {
15            message = "Wrong.";
16        }
17        Messages.showMessageDialog(project,message, "Answer",
            Messages.getInformationIcon());
18    }
19 }
```

Ispis 5.1. Primjer implementacije jednostavne akcije

U navedenom primjeru se u konstruktoru definira ime, opis i ikona akcije (linije 2-5). Metoda *actionPerformed* dohvaća projekt iz kojeg je pozvana akcija te prikazuje korisniku upit u dijalogu (linija 6). Na temelju korisničkog unosa prikazuje poruku opet u obliku dijaloga (linija 17).

Metoda *actionPerformed* prima kao argument objekt tipa *AnActionEvent*. Taj objekt sadrži informacije o trenutnom kontekstu razvojnog okruženja u kojem je akcija pozvana. Kontekst sadrži informacije poput trenutnog projekta u kojem je akcija pozvana, datoteci koja je trenutno odabrana, odabranog teksta u sučelju. Detaljnije informacije o komponentama poput *AnActionEvent* razreda dane su u poglavlju 5.4.

Drugi korak implementacije akcije je registracija akcije kako bi razvojno okruženje moglo smjestiti tu akcija na odgovarajuće mjesto u sučelju. Akcije je moguće opisati u konfiguracijskoj datoteci dodatka *plugins.xml* ili programski primjerice da se registracija obavlja prilikom pokretanja razvojnog okruženja. Ovdje će biti opisano kako registrirati akcije u datoteci *plugins.xml* i ta metoda je korištena dalje u radu. Datoteka *plugins.xml* je konfiguracijska datoteka koja sadrži metapodatke o dodatku te registrirane akcije. Primjer konfiguracijske datoteke je dan u ispisu 5.2.

```
1   <idea-plugin>
2     <id>com.your.company.unique.plugin.id</id>
3     <name>Plugin display name here</name>
4     <version>1.0</version>
5     <vendor email="support@yourcompany.com"
            url="http://www.yourcompany.com">YourCompany</vendor>

6   <description><![CDATA[
      Enter short description for your plugin here.<br>
      <em>most HTML tags may be used</em>
    ]]></description>
7   <change-notes><![CDATA[
      Add change notes here.<br>
      <em>most HTML tags may be used</em>
    ]]>
8   </change-notes>
9   <idea-version since-build="145.0"/>
10  <!-- uncomment to enable plugin in all products
    <depends>com.intellij.modules.lang</depends>
    -->
11  <extensions defaultExtensionNs="com.intellij">
      <!-- Add your extensions here -->
12  </extensions>
13  <actions>
14    <group id="MyPlugin.TestMenu" text="_Test Menu"
            description="Test menu">
15      <add-to-group group-id="MainMenu" anchor="first"  />
16      <action id="Myplugin.Test" class="Test" text="Test"
            description="A test menu item" />
17    </group>
18  </actions>
19  </idea-plugin>
```

Ispis 5.2. Primjer *plugin.xml* konfiguracijske datoteke

Element *id* definira jedinstveni identifikator dodatka. Elementi *name*, *version* i *vendor* sadrže informacije o imenu, verziji i autoru dodatka. *Description* sadrži kratki opis dodatka i njegovih funkcionalnosti. Unutar elementa *change-notes* se nalaze informacije o promjenama u toj verziji dodatka. Element *idea-version* sadrži informacije o tome od koje verzije do koje verzije IntelliJ IDEA razvojnog okruženja je dodatak podržan (linija 9). U elementu *depends* su definirani moduli IntelliJ platforme o kojima ovisi taj dodatak (linija 10). Različita razvojna okruženja sadržavaju različite module, stoga je ovdje moguće navesti module koji su potrebni za normalan rad dodatka. Dodatak će biti instaliran u razvojno okruženje koje sadrži sve module navedene u *depends* elementu. Element *extensions*

omogućava definiciju mjesta u dodatku na kojima drugi dodatci mogu unaprijediti njegovu funkcionalnost (linije 11 i 12).

Unutar elementa *actions* se registriraju akcije i grupe akcija (linije 13-18). U navedenom primjeru je definirana grupa koja definira meni nazvan *Test Menu* i ta grupa je sa elementom *add-to-group* dodana u grupu menija glavne alatne trake sučelja, atribut *anchor* određuje gdje će se u alatnoj traci nalaziti novi meni. Element *action* registrira definiranu akciju sa slijedećim atributima:

- *id* – jedinstveni identifikator akcije
- *class* – razred koji sadrži implementaciju akcije
- *text* – tekst koji je prikazan u sučelju za tu akciju
- *description* – kratak opis akcije

Dodatke je moguće testirati direktno iz grafičkog sučelja standardnim pokretanjem projekta. Prilikom pokretanja dodatka pokrenuti će se nova instanca IntelliJ IDEA okruženja s instaliranim dodatkom. Na kraju, da bi se dodatak mogao prenositi i instalirati potrebno ga je izgraditi. Prvo je potrebno u razvojnom okruženju odabrati opciju *Build > Make Project* kako bi se projekt izgradio i zatim *Build > Prepare Plugin Module for Deployment* kako bi se generirala JAR ili ZIP datoteka. Navedena datoteka se nalazi u projektnom direktoriju dodatka i može se instalirati. U slučaju kada dodatak ne koristi vanjske biblioteke biti će zapakiran u JAR datoteku, a kada koristi vanjske biblioteke onda će se skupa s njima zapakirati u ZIP datoteku. Dodatak se instalira odabirom *File > Settings* te zatim u meniju *Plugins* odabirom opcije *Install plugin from disk...* i na kraju se odabere datoteka dodatka.

5.4. Komponente IntelliJ IDEA dodataka

Ovdje su opisane neke od najbitnijih komponenti IntelliJ IDEA dodataka koje su bitne za razumijevanje ovoga rada. Većina tih komponenti će biti korištena prilikom implementacije neke akcije. Metoda *actionPerformed* razreda *AnAction* je objašnjena u poglavlju 5.3. *AnAction* sadrži još jednu bitnu metodu *update* koju nije potrebno implementirati, ali će je često biti potrebno implementirati kako bi se ovisno o kontekstu konkretna akcija omogućila ili onemogućila. Metodu *update* poziva razvojno okruženje periodički i omogućava promjenu stanja akcije poput toga je li ona omogućena ili vidljiva ovisno o kontekstu. Metoda prima objekt tipa *AnActionEvent* koji sadrži kontekst u kojem akcija pozvana.

Komponenta *AnActionEvent* kao što je već navedeno sadrži informacije o kontekstu u kojem je pozvana akcija. Kako bi se dohvatila neka od tih informacija koristi se metoda *getData* razreda *AnActionEvent*. Ta metoda prima ključ prema kojem se određuje koju se informaciju želi dohvatiti. Lista nekih od ključeva se nalazi u razredima *CommonDataKeys*, *PlatformDataKeys* te *DataKeys*. Primjerice ako se želi dohvatiti projekt iz kojeg je pozvana akcija:

```
Project project = event.getData(PlatformDataKeys.PROJECT);
```

Komponenta *Project* predstavlja projekt unutar razvojnog okruženja. Ona sadrži informacije poput apsolutnog puta do direktorija tog projekta te imena projekta. Osim toga sadrži metodu za spremanje promjena projekta.

S obzirom da se svaka akcija može pojavljivati na više mjesta, tj. grupa unutar sučelja, svako to mjesto ima drugačiji prikaz, primjerice može imati drugačiji tekst i drugačiju ikonu. Prikaz je definiran objektom tipa *Presentation*. Objekt tipa *Presentation* možemo dohvatiti unutar akcije iz objekta tipa *AnActionEvent* pozivom metode *getPresentation*. Tim objektom je moguće upravljati je li neka akcija omogućena ili vidljiva, na način da se pozovu metode *setVisible, setEnabled* ili *setEnabledAndVisible*. Osim toga *Presentation* sadrži metode za dohvaćanje i postavljanje opisa, ikona i teksta te prezentacije akcije. Ispis 5.3. prikazuje primjer uporabe objekta tipa *Presentation* u metodi *update* neke akcije:

```
1  public void update(AnActionEvent event) {
2         Presentation presentation = event.getPresentation();
3         // some complex condition is defined here ...
4         if(condition)
5                 presentation.setEnabledAndVisible(false);
6  }
```

Ispis 5.3. Primjer korištenja razreda *Presentation*

Komponenta *VirtualFile* predstavlja apstraktnu reprezentaciju datoteke na disku unutar razvojnog okruženja. IntelliJ platforma koristi virtualni datotečni sustav [29] koji reprezentira datotečni sustav operacijskog sustava. Virtualni datotečni sustav pruža programsko sučelje za operacije s datotekama neovisno o njihovoj lokaciji u datotečnom sustavu operacijskog sustava, prati promjene datoteka i nudi mogćnost pridruživanja dodatnih podataka datotekama unutar virtualnog datotečnog sustava. Virtualni datotečni sustav ne mora uvijek biti sinkroniziran s datotečnim sustavom operacijskog sustava, već se periodički sinkronizira ili se to obavlja na temelju nekog događaja nastalog unutar razvojnog okruženja. Objekti tipa *VirtualFile* postoje dok god postoji aktivan IDEA proces, tj. dok je razvojno okruženje aktivno. Iz akcije je moguće dohvatiti odabranu datoteku na slijedeći način:

```
VirtualFile file = event.getData(PlatformDataKeys.VIRTUAL_FILE);
```

Ukoliko se želi dohvatiti datoteka iz datotečnog sustava koristeći njezinu putanju:

```
VirtualFile file =
            LocalFileSystem.getInstance().findFileByPath(filePath);
```

Ukoliko je potrebno stvoriti neku datoteku na datotečnom sustavu nije preporučeno koristiti razred *VirtualFile* već je preporučeno koristiti standardno programsko sučelje Jave.

Prilikom razvijanja vlastitih dodataka nekada je potrebno implementirati vlastite komponente grafičkog sučelja. Za implementaciju tih komponenti se preporuča korištenje gotovih Swing komponenti IntelliJ platforme [30]. Implementacija komponenti se obavlja naslijeđivanjem odgovarajućih razreda. Korištenje tih komponenti osigurava da će se razvijene grafičke komponente pravilno izvršavati unutar sučelja i da će njihov izgled biti usklađen s ostatkom sučelja. Primjer jedne takve komponente dan je u poglavlju 6.

Implementacija i izvorni kod navedenih komponenti su dostupni na Github repozitoriju IntelliJ IDEA Community alata [27].

6. Razvijeni dodatak za Android Studio

U ovome poglavlju je opisan razvijeni dodatak za Android Studio koji integrira funkcionalnost navedenih dekompajlera. Osim toga opisane su i dodatne funkcionalnosti koje dodatak ima implementirano, poput navigacije po kodu i usporedbe izlaza dekompajlera. Za svaku opisanu funkcionalnost biti će prikazan i objašnjen isječak najbitnijeg dijela koda koji ju implementira. Uz to prikazano je gdje su te funkcionalnosti smještene unutar razvojnog okruženja i kako ih koristiti.

Razvijeni dodatak iskorištava opciju profiliranja i otklanjanja greški Android aplikacija koja je dostupna u Android Studio-u. Prilikom stvaranja novog projekta potrebno je odabrati *Profile or Debug APK* i zatim odabrati APK datoteku aplikacije. Pritom će Android Studio raspakirati APK datoteku, dekodirati manifest datoteku te transformirati DEX kod u smali kod. Osim toga moguće je dodati izvorni Java kod u projekt odabirom opcije *Attach Java Sources* opcije prilikom čega će se sav smali kod zamijeniti izvornim Java kodom.

Razvijeni dodatak nadograđuje navedeni tip projekta integracijom dekompajlera i dodatnih funkcionalnosti. Dodatak je omogućen samo u tom tipu projekta, odnosno dodatak provjerava postojanje APK datoteke u projektnom direktoriju. Dodatak omogućuje odabir dekompajlera koji se žele koristiti i definiranje dodatnih argumenata dekompajlera kroz grafičko sučelje. Definiranje argumenata je implementirano u akciji *DefineSettings*. Isječak koda razreda *DefineSettings* je prikazan u ispisu 6.1.

```
1   public class DefineSettings extends AnAction {
2     public static DecompilerSettings settings;
      ...
3     @Override
4     public void actionPerformed(AnActionEvent event) {
5         Project project = event.getData(PlatformDataKeys.PROJECT);
6         SettingsForm settingsForm = new SettingsForm(
            "Define decompilers and arguments", project);
7        settingsForm.show();

8         String jadxArgs = null;
9         String procyonArgs = null;
10        String fernFlowerArgs = null;

11        try {
12            jadxArgs = settingsForm.getJadxArgs();
13        } catch(Exception e) {};
14        try {
15            procyonArgs = settingsForm.getProcyonArgs();
16        } catch(Exception e) {};
17        try {
18            fernFlowerArgs = settingsForm.getFernFlowerArgs();
19        } catch(Exception e) {};

20        settings = new DecompilerSettings(settingsForm.jadxSelected(),
      settingsForm.procyonSelected(), settingsForm.fernFlowerSelected(),
      jadxArgs, procyonArgs, fernFlowerArgs, false);
21    }
22  }
```

Ispis 6.1. Isječak koda akcije *DefineSettings*

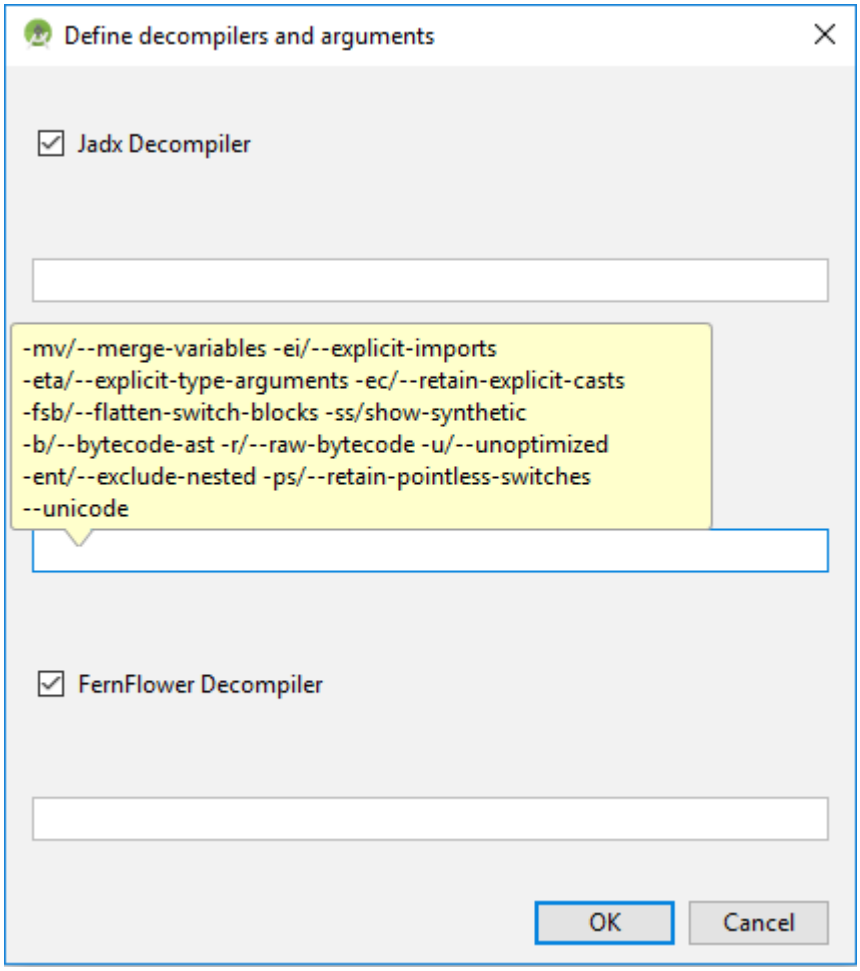
Akcija *DefineSettings* prikazuje korisniku formu u kojoj je moguće odabrati dekompajlere i definirati opcionalne argumente dekompajlera (linije 6 i 7). Zatim se informacije o odabranim dekompajlerima i argumentima pohranjuju u objekt tipa *DecompilerSettings* (linija 20). *DecompilerSettings* je pomoćni razred koji služi za pohranu i proslijeđivanje navedenih informacija.

Forma u kojoj korisnik upisuje parametre dekompajlera je implementirana u razredu *SettingsForm* i kao što je objašnjeno u poglavlju 5.4. ta forma nasljeđuje razred *DialogWrapper* čime je osigurano da će se ispravno izvršavati u sučelju. Prilikom naslijeđivanja grafičkih komponenti IntelliJ platforme potrebno je napraviti dvije stvari. Prvo je potrebno u konstruktoru pozvati konstruktor originalne klase i u njemu predati objekt tipa *Project*, te pozvati metodu *init*. Zatim je potrebno implementirati metodu *createCenterPanel* koja vraća glavnu grafičku komponentu koja predstavlja tu formu. Primjer kako je to napravljeno u razredu *SettingsForm* je dan u ispisu 6.2.

```
1  public class SettingsForm extends DialogWrapper {
2    private JPanel decompilersPanel;
     ...
3    public SettingsForm(String title, Project project) {
4        super(project);
5        init();
6        setTitle(title);
         ...
7    }
     ...
8    protected JComponent createCenterPanel() {
9        return decompilersPanel;
10   }
11 }
```

Ispis 6.2. Primjer implementacije grafičke komponente IntelliJ Platforme

Prikaz forme za odabir dekompajlera i definiranje njihovih argumenata dan je na slici 6.1.



Slika 6.1. Forma za odabir dekompajlera i argumenata

Prelaskom preko polja za unos argumenata prikazuju se potencijalni argumenti koji se mogu zadati konkretnom dekompajleru.

Dekompajliranje je implementirano u akciji *DecompilerAPK*. Ta akcija obavlja nekoliko stvari prije nego što započne s dekompajliranjem. Prvo provjerava je li korisnik definirao dekompajlere i argumente, ukoliko nije obavještava ga i pita želi li nastaviti s pretpostavljenim postavkama. Pretpostavljene postavke su da su svi dekompajleri odabrani bez dodatnih argumenata. Zatim se provjerava postoji li već dekompajlirani kod te ukoliko postoji, pita se korisnika želi li nastaviti s dekompajliranjem pri čemu će sav prijašnji dekompajlirani kod biti obrisan. U slijedećem koraku se dohvaća APK datoteka iz direktorija projekta. Zatim se kreiraju direktoriji u koji će se pohraniti dekompajlirani kod te se kreira direktorij za log datoteke i pomoćni direktorij koji služi za pohranu međurezultata. Slijedi raspakiravanje APK datoteke i pozivanje alata dex2jar koji će transformirati DEX datoteku u JAR. Zatim se pozivaju pojedini dekompajleri i ako u procesu nije bilo grešaka, korisniku se pokazuje poruka o uspješnosti, inače mu se prikazuje poruka o grešci koja se dogodila. Cijeli navedeni proces se obavlja unutar asinkronong pozadinskog posla u razvojnom okruženju kako razvojno okruženje ne bi bilo blokirano. Isječak koda iz metode *actionPerformed* razreda *DecompileAPK* s prikazom funkcionalnosti dekompajliranja dan je u ispisu 6.3.

```
1  public void actionPerformed(AnActionEvent event) {
   ...
2   ProgressManager.getInstance().run(new Task.Backgroundable(project,
     "Decompilation") {
3       public void run(ProgressIndicator indicator) {
4          if (DefineSettings.settings.isJadxSelected()) {
5              indicator.setText("Decompilation in progress:
                   running Jadx...");
               ...
6              PrintStream ps = null;
               ...
7              System.setOut(ps);
8              File jadxout = new File(projectDir + Utils.jadxOutput);
9              if (jadxArgs == null)
10                jadx = new JadxWrapper(apkFile, jadxout);
11             else
12                jadx = new JadxWrapper(apkFile, jadxout, jadxArgs);
13            try {
14                jadx.decompile();
15             } catch (Exception e) {
16                jadxFailed = e.getMessage();
17             }
18             System.setOut(console);
19          }
20          indicator.setFraction(0.5);
21       }
22    }
23  }
```

Ispis 6.3. Isječak koda koji obavlja funkcionalnost dekompajliranja

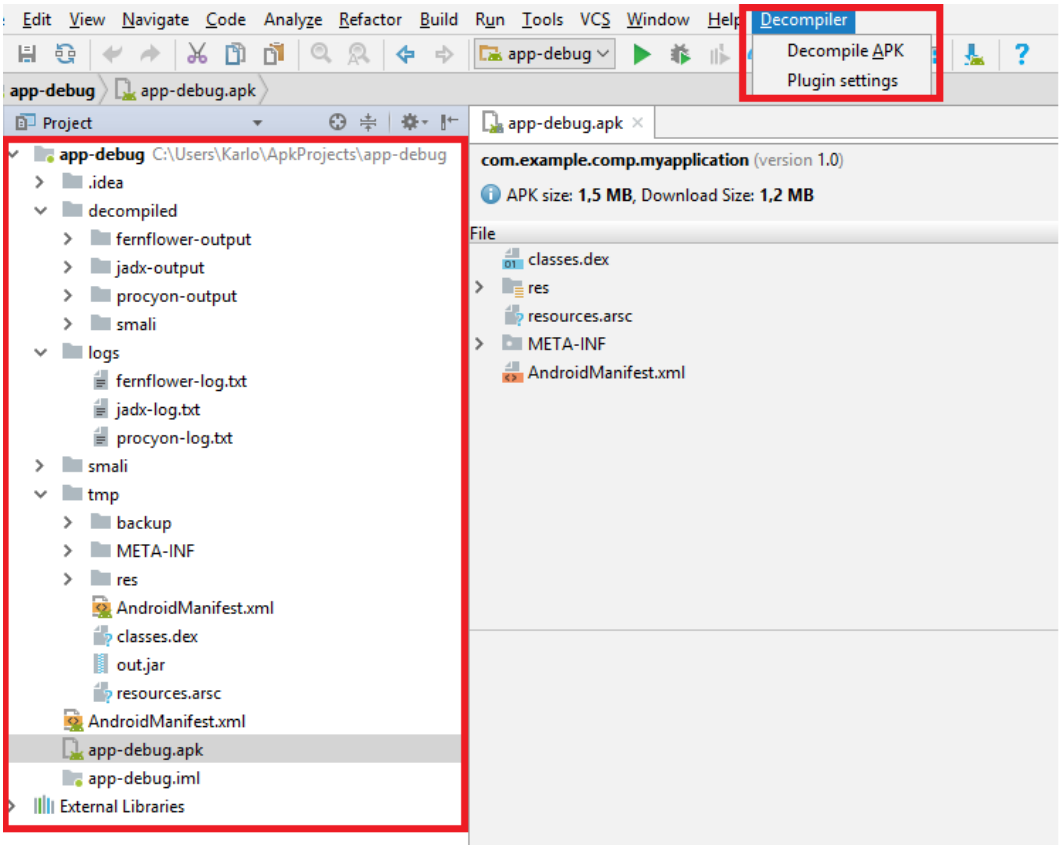
Objekt tipa *ProgressManager* pokreće novi pozadinski posao kroz anonimni razred koji je tipa *Task.Backgroundable*. U objektu tipa *Task.Backgroundable* potrebno je implementirati metodu *run* unutar koje se nalazi kod koji će obaviti dekompajliranje (linije 2 i 3). U isječku je konkretno pokrenuto dekompajliranje Jadx dekompajlera (linija 14). Metoda *run* prima objekt tipa *ProgressIndicator* kojim je moguće u grafičkom sučelju prikazivati napredak obavljenog posla u tekstualnom i grafičkom obliku pozivom metoda *setFraction* i *setText*. U navedenom isječku su izostavljene definicije varijabli *jadxLog*, *jadxArgs* i *ps* zbog preglednosti. *jadxLog* je put do datoteke u koju se zapisuju log podatci, *jadxArgs* su argumenti dekompajlera te *ps* je izlazni tok koji omata datoteku za log podatke. Na početku procesa se metodom *setText* signalizira da se obavlja dekompajliranje (linija 5). Zatim se standardni izlaz preusmjerava u datoteku koja je definirana varijablom *jadxLog* jer dekompajleri na standardni izlaz ispisuju detalje dekompajliranja pa se te informacije prikupljaju u obliku log datoteka. Na kraju se pokreće dekompajliranje i standarni izlaz se preusmjerava nazad na konzolu (linije 14 i 18). Analogan proces se obavlja za druga dva dekompajlera.

Akcije *DefineSettings* i *DecompileAPK* su dostupne u novom meniju nazvanom Decompiler u glavnoj alatnoj traci razvojnog okruženja. Osim toga akcija *DecompileAPK* je dostupna u istoimenom meniju pritiskom na desni klik miša. Ispis 6.4. prikazuje kako su obje akcije opisane u datoteci *plugin.xml*.

```
1  <group id="Decompiler.DecompileMenu" text="_Decompiler"
       description="Decompiler plugin">
2       <add-to-group group-id="MainMenu" anchor="last"  />
3       <action id="Decompiler.DecompileAPK"
                class="hr.fer.decompiler.plugin.action.DecompileAPK"
       text="Decompile _APK" description="Action for apk decompilation" />
4       <action id="Decompiler.DefineSettings"
                class="hr.fer.decompiler.plugin.action.DefineSettings"
                text="Plugin settings" description=
                "Action for plugin arguments definition"/>
5  </group>
```

Ispis 6.4. Opis akcija u datoteci *plugin.xml*

Iz ispisa je vidljivo da je definiran novi meni nazvan *Decompiler* koji je dodan u grupu glavnog menija (linija 2) i sastoji se od navedene dvije akcije (linije 3 i 4). Slika 6.2. prikazuje navedeni meni te izlaze koje producira dodatak.



Slika 6.2. Prikaz menija i izlaza dodatka

U lijevom dijelu slike vidljivo je da dodatak stvara slijedeće direktorije u projektu:

- *decompiled* – sadrži izlaze svih dekompajlera i kopirani smali kod
- *logs* – sadrži log datoteke svih dekompajlera
- *tmp* – sadrži raspakiranu APK datoteku, JAR koji je dobiven iz DEX datoteke te *backup* direktorij koji se koristi u funkcionalnosti zamjene izlaza dekompajlera

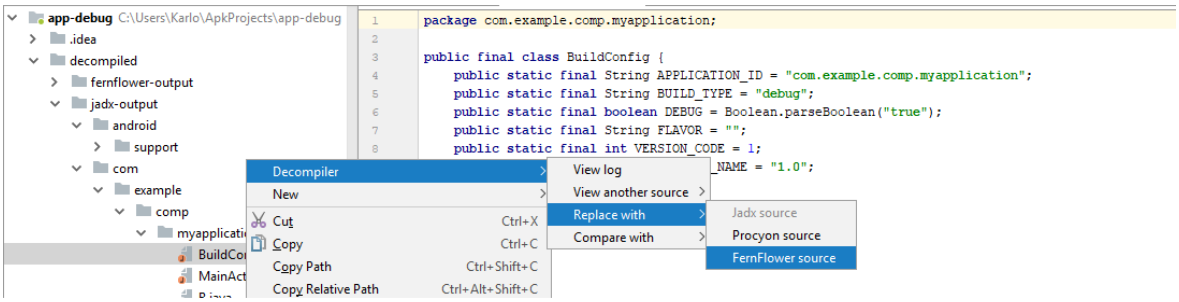
Slijedeća funkcionalnost koju nudi dodatak je zamjena izlaza jednog dekompajlera s izlazom drugoga. Funkcionalnost se može pozvati nad jednom Java datotekom ili nad direktorijem unutar izlaza dekompajlera. Ukoliko se radi o direktoriju, dodatak će zamijeniti cijeli direktorij i sav njegov sadržaj rekurzivno s direktorijom drugog dekompajlera. Originalna datoteka odnosno direktorij se pohranjuju u direktorij *tmp/backup* kako bi se datoteke kasnije mogle vratiti. Navedena funkcionalnost je implementirana u pozadinskom poslu jer kopiranjem strukture direktorija može potrajati. Ispis 6.5. daje prikaz isječka koda iz akcije *ReplaceWithFernFlower* koja implementira tu funkcionalnost za FernFlower dekompajler.

```
1  if (!selectedFile.isDirectory()) {
2    if (!Utils.hasBackup(file, backupPath)) {
3        Utils.backupFile(file, backupPath, originalFilePath);
4    }

5    String fileToBeCopied = projectPath + Utils.fernflowerOutput +
           "/" + filePath;
6    try {
7       Files.copy(Paths.get(fileToBeCopied),Paths.get(originalFilePath),
           StandardCopyOption.REPLACE_EXISTING);
8    } catch (IOException e) {
9        JOptionPane.showMessageDialog(null,
           "Something went wrong when copying files: " + e.getMessage(),
           "Error", JOptionPane.INFORMATION_MESSAGE);
10    }
11    project.getBaseDir().refresh(false, true);
12    VirtualFile newFile =
         LocalFileSystem.getInstance().findFileByPath(originalFilePath);
14    FileEditorManager.getInstance(project).openFile(newFile, true);
15  } else {
    ...
16    String backupDirPath = backupPath + "/" + file;
17    if (!Utils.hasBackup(file, backupPath)) {
18        new File(backupDirPath).mkdirs();
19        Utils.copyDirectory(originalFilePath, backupDirPath);
20    }
21    String dirToBeCopied = projectPath + Utils.fernflowerOutput +
      "/" + filePath;
22    Utils.copyDirectory(dirToBeCopied, originalFilePath);
23  }
```

Ispis 6.5. Isječak koda iz akcije *ReplaceWithFernflower*

Ukoliko se radi o datoteci akcija prvo radi kopiju originalne datoteke definirane varijablom *file* (linija 3). Zatim se konstruira put do datoteke u varijabli *fileToBeCopied* koja će zamijeniti originalnu datoteku (linije 5-8). Na kraju se ta datoteka otvara u uređivaču razvojnog okruženja (linija 14). Analogno ako se radi o direktoriju on se prvo rekurzivno kopira i zatim zamjenjuje s direktorijom definiranim varijablom *dirToBeCopied* (linije 16-22). Za operacije kopiranja je razvijen razred *Utils* koji sadrži metode za manipulacije datotekama i direktorijima, te ostale pomoće metode koje se koriste u akcijama. Za ostale dekompajlere postoji analogna akcija. Akcija je definirana u podgrupi *Replace with* grupe *Decompiler* na desnom kliku miša. Omogućena je samo kod odabira datoteke ili direktorija koji se nalaze unutar direktorija izlaza nekog dekompajlera. Slika 6.3. prikazuje meni definirane akcije i kako se ona koristi.



Slika 6.3. Primjer funkcionalnosti zamjene izlaza dekompajlera

Nakon što je neki izlaz zamijenje s izlazom drugog dekompajlera, akcija vraćanja originalnog izlaza postaje dostupna. Akcija je implementirana u razredu *RevertFile* koji obavlja obrnut proces od zamjene. Kod je vrlo sličan kodu gore navedene akcije stoga nije prikazan ovdje. Akcija prvo dohvaća originalnu datoteku ili direktorij iz direktorija u kojem se pohranjuju kopije. Zatim se originalna datoteka ili direktorij kopiraju na originalno mjesto i na kraju se briše kopija iz *backup* direktorija. Akcija je dostupna u grupi *Decompiler* na desnom kliku miša u meniju nazvanom *Revert file*.

Osim akcije zamjene izlaza dodatak omogućava pregledavanja iste dekompajlirane datoteke drugog dekompajlera ili pregledavanje smali koda te datoteke. Slično kao i kod zamjene datoteka dodatak dohvaća datoteku drugog dekompajlera i zatim ju prikazuje u uređivaču razvojnog okruženja. Kod metode *actionPerformed* akcije *ViewProcyonSource* dan je u ispisu 6.6.

```
1  public void actionPerformed(AnActionEvent event) {
2          Project project = event.getProject();
3          VirtualFile selectedFile = event.getData(DataKeys.VIRTUAL_FILE);
4          if(selectedFile != null) {
5              String fileSuffix =
                       Utils.preparePath(selectedFile.getCanonicalPath(),
                       project.getBasePath());
6              String filePath = project.getBasePath() + "/"
                       + Utils.procyonOutput + "/" + fileSuffix;
7              VirtualFile file =
                       LocalFileSystem.getInstance().findFileByPath(filePath);
8              if(file != null)
9                  FileEditorManager.getInstance(project).openFile(file,
                                                                   true);
10         }
11 }
```

Ispis 6.6. Kod akcije *actionPerformed* akcije *ViewProcyonSource*

Akcija prvo dohvaća odabranu datoteku u varijablu *selectedFile* i zatim priprema put do datoteke Procyon dekompajlera u varijabli *filePath* (linije 5 i 6). Na kraju ukoliko ta datoteka

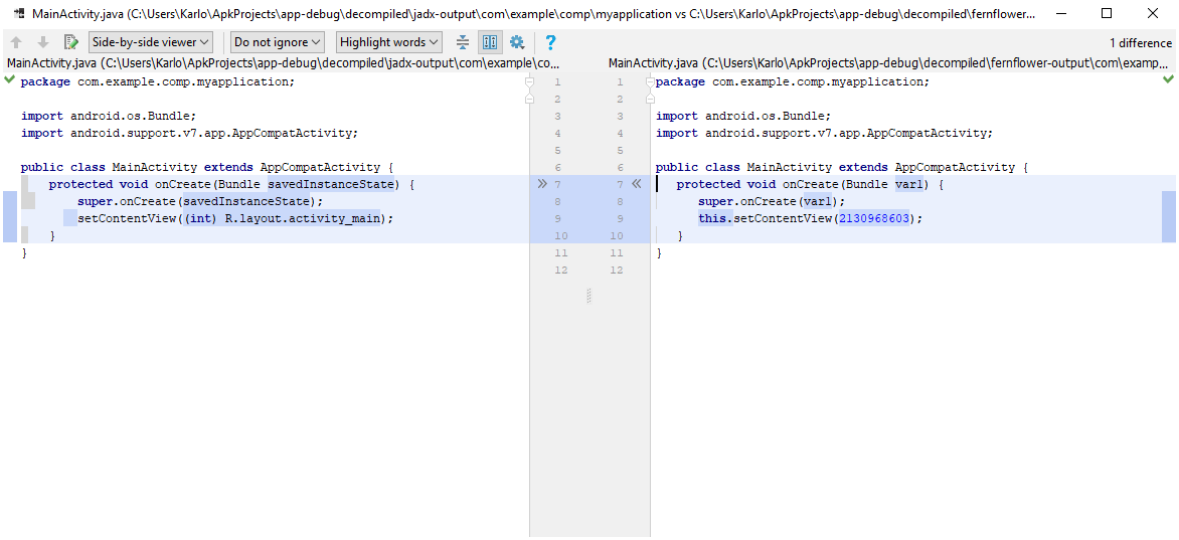
postoji, prikazuje ju u uređivaču (linija 9). Akcija je registrirana u podgrupi *View another source* koja se nalazi u grupi *Decompiler* na desnom kliku miša. Omogućena je samo nad Java datotekama unutar direktorija dekompajliranih izlaza.

Zadnja akcija koja radi manipulacije s izlazim dekompajlera je akcija za usporedbu dvije iste datoteke dva različita dekompajlera. Usporedba datoteka je već implementirana u razvojnom okruženju, pa navedena akcija iskorištava neke od komponenti IntelliJ platforme. Isječak koda metode *actionPerformed* akcije *CompareWithJadx* dan je u ispisu 6.7.

```
1  public void actionPerformed(AnActionEvent event) {
2        Project project = event.getProject();
3        VirtualFile firstFile = event.getData(DataKeys.VIRTUAL_FILE);
4        String fileSuffix =
5        Utils.preparePath(firstFile.getCanonicalPath(),
            project.getBasePath());
6        String filePath = project.getBasePath() + "/" +
            Utils.jadxOutput + "/" + fileSuffix;
7        VirtualFile secondFile =
            LocalFileSystem.getInstance().findFileByPath(filePath);
8        DiffRequestFactoryImpl factory = new DiffRequestFactoryImpl();
9        DiffRequest diffRequest =
            factory.createFromFiles(project, firstFile, secondFile);
10        DiffManager.getInstance().showDiff(project, diffRequest);
11  }
```

Ispis 6.7. isječak koda metode *actionPerformed* akcije *CompareWithJadx*

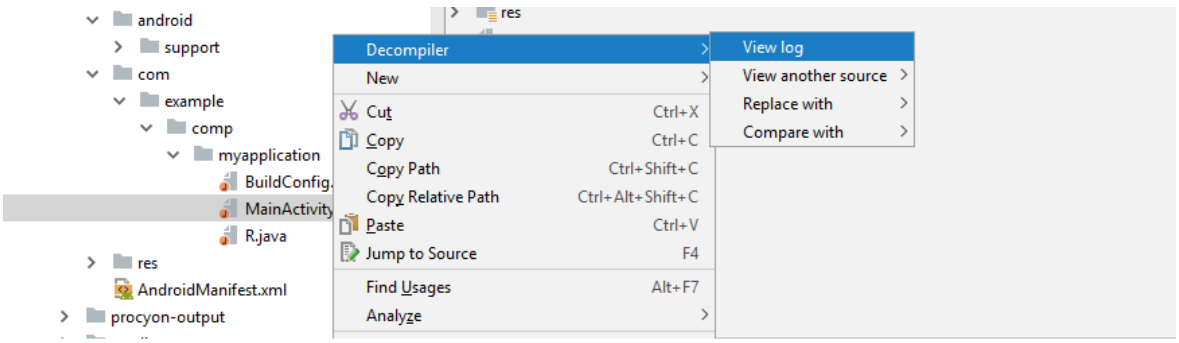
Metoda dohvaća odabranu datoteku te dohvaća datoteku Jadx dekompajlera s kojom ju uspoređuje (linije 3-7). Zatim se koriste komponenete IntelliJ platforme za usporedbu datoteka. Razred *DiffRequest* predstavlja zahtjev za usporedbom dvije datoteke. Objekt tipa *DiffRequest* se kreira koristeći metodu *createFromFiles* razreda *DiffRequestFactoryImpl* koji implementira oblikovni obrazac tvornica za kreiranja objekata (linije 8 i 9). Na kraju se usporedba vrši pozivom metode *showDiff* objekta tipa *DiffManager* kojoj je potrebno predati trenutni projekt i zahtjev za usporedbom. Slika 6.4. prikazuje primjer usporedbe dva izlaza dekompajlera.



Slika 6.4. Primjer usporedbe dva izlaza različitih dekompajlera

Akcija je registrirana u podgrupi *Compare with* grupe *Decompiler* na desnom kliku miša. Akcija je omogućena samo za Java datoteke koje se nalaze unutar direktorija dekompajliranih izlaza.

Posljednja funkcionalnost dodatka je pregledavanje log datoteka. Ova jednostavna funkcionalnost je implementirana u akciji *ViewLog*. Akcija je registrirana u grupi *Decompiler* na desnom kliku miša. Akcija je omogućena samo za direktorije i datoteke koje se nalaze unutar direktorija dekompajliranih izlaza. Kod akcije slično kao i kod već opisanih akcija dohvaća odgovarajuću log datoteku dekompajlera i prikazuje ju u uređivaču razvojnog okruženja. Slika 6.5. prikazuje meni te akcije i kako se ona koristi.



Slika 6.5. Primjer korištenja funkcionalnosti pregledavanja log datoteke

Sav kod prikazan u ovome poglavlju kao i ZIP datoteka dodatka koja se može instalirati dostupni su na Github repozitoriju razvijenog dodatka [23].

7. Zaključak

Prvi korak sigurnosne analize APK datoteka je dekompajliranje prevedenog koda u izvorni Java kod. Kako bi se bolje poznavale APK datoteke objašnjena je njihova struktura, smali kod i navedeni su neki od alata koji olakšavaju njihovu manipulaciju. Za bolje razumijevanje Android aplikacija navedene su osnovne komponente aplikacija, njihova interakcija i sigurnosni model Android aplikacija. Za potrebe dekompajliranja postoje različiti dekompajerli koju su razvijeni neovisno jedan o drugome. U radu su objašnjeni neki od dekompajlera, prikazano je kako ih koristiti i kako ih integrirati programski. IntelliJ platforma se pokazala kao kvalitetan alat za razvijanje novih dodataka za razvojna okruženja. U radu je opisano kako razviti novi dodatak koristeći IntelliJ platformu. S obzirom da su dekompajleri nepovezani i često se koriste neovisno razvijen je dodatak za Android Studio razvojno okruženje koji ih integrira zajedno. Osim toga razvijeni dodatak nudi mogućnosti usporedbe rezultata dekompajlera i dodatnu navigaciju po njihovim izlazima. Kombinacija naprednih mogućnosti navigacije po kodu razvojnog okruženja Android Studio i razvijenog dodatka omogućava lakšu analizu dekompajliranog koda i APK datoteke. Razvijeni dodatak se može još nadograditi integriranjem novih dekompajlera i implementacijom novih mogućnosti.

8. Literatura

[1] Wikipedia, *Android (Operating system)*, https://en.wikipedia.org/wiki/Android_(operating_system), 28.5.2018.

[2] Wikipedia, *Android application package*, https://en.wikipedia.org/wiki/Android_application_package, 28.5.2018.

[3] Wikipedia, *Decompiler*, https://en.wikipedia.org/wiki/Decompiler, 28.5.2018.

[4] Android Development, *The build process*, https://developer.android.com/studio/build/#build-process, 29.5.2018.

[5] Android Source, *ART and Dalvik*, https://source.android.com/devices/tech/dalvik/, 29.5.2018.

[6] Android Development, *App resources overview*, https://developer.android.com/guide/topics/resources/providing-resources, 29.5.2018.

[7] Android Source, *Understand the APK structure*, https://developer.android.com/topic/performance/reduce-apk-size#apk-structure, 29.5.2018.

[8] Android Development, *App Manifest Overview*, https://developer.android.com/guide/topics/manifest/manifest-intro, 29.5.2018.

[9] Github repozitorij Apktool alata, https://ibotpeaches.github.io/Apktool/, 1.6.2018.

[10] Android Development, *Application Fundamentals*, https://developer.android.com/guide/components/fundamentals#Components, 1.6.2018.

[11] Github repozitorij baksmali alata, https://github.com/JesusFreke/smali, 2.6.2018.

[12] Dalvik tipovi podataka, https://github.com/JesusFreke/smali/wiki/TypesMethodsAndFields, 2.6.2018.

[13] Dalvik registri, https://github.com/JesusFreke/smali/wiki/Registers, 2.6.2018.

[14] Wikipedia, *Java virtual machine*, https://en.wikipedia.org/wiki/Java_virtual_machine, 4.6.2018.

[15] Oracle, *JAR File Specification*, https://docs.oracle.com/javase/8/docs/technotes/guides/jar/jar.html#Intro, 4.6.2018.

[16] CFR dekompajler, http://www.benf.org/other/cfr/, 4.6.2018.

[17] Github repozitorij Krakatau dekompajlera, https://github.com/Storyyeller/Krakatau, 4.6.2018.

[18] Github repozitorij JD-GUI dekompajlera, https://github.com/java-decompiler/jd-gui, 4.6.2018.

[19] Github repozitorij Jadx dekompajlera, https://github.com/skylot/jadx, 4.6.2018.

[20] Bitbucket repozitorij Procyon dekompajlera, https://bitbucket.org/mstrobel/procyon/overview, 4.6.2018.

[21] Bitbucket, *Procyon decompiler wiki*, https://bitbucket.org/mstrobel/procyon/wiki/Java%20Decompiler, 4.6.2018.

[22] Github repozitorij Intellij Idea Community verzije alata, *Fernflower decompiler*, https://github.com/JetBrains/intellij-community/tree/master/plugins/java-decompiler/engine, 4.6.2018.

[23] Github repozitorij razvijenog dodatka, https://github.com/kmravunac/Decompilers-plugin, 11.6.2018.

[24] Github repozitorij Dex2jar alata, https://github.com/pxb1988/dex2jar, 11.6.2018.

[25] Jetbrains, *What is the Intellij Platform?*, https://www.jetbrains.org/intellij/sdk/docs/intro/intellij_platform.html, 12.6.2018.

[26] Jetbrains, Stranica za preuzimanje Intellij IDEA okruženja, https://www.jetbrains.com/idea/download, 12.6.2018.

[27] Github repozitorij IntelliJ IDEA Community, https://github.com/JetBrains/intellij-community, 12.6.2018.

[28] IntelliJ Platform SDK DevGuide, *Creating an action*, https://www.jetbrains.org/intellij/sdk/docs/basics/getting_started/creating_an_action.html, 13.6.2018.

[29] IntelliJ Platform SDK DevGuide, *Virtual File System*, https://www.jetbrains.org/intellij/sdk/docs/basics/virtual_file_system.html, 15.6.2018.

[30] IntelliJ Platform SDK DevGuide, *User Interface Components*, https://www.jetbrains.org/intellij/sdk/docs/user_interface_components/user_interface_components.html, 15.6.2018.

Dodatak za Android Studio za podršku sigurnosne analize APK datoteka

Sažetak

Ovaj rad se bavi problematikom analize APK datoteka i dekompajliranja njihovog koda niže razine u izvorni kod. Na početku su opisane komponente Android aplikacija, njihov proces izgradnje i sigurnosni model. Zatim je objašnjena struktura APK datoteka i smali kod koji je ponekad potreban kako bi se razumio izvoran kod aplikacije. Osim toga navedeni su neki alati koji olakšavaju analizu APK datoteka. Za dekompajliranje prevedenog su korištena tri dekompajlera: Jadx, Procyon i Fernflower. U posebnom poglavlju je objašnjeno kako ih koristiti, koje su njihove mogućnosti i kako ih koristiti programski. Osim dekompajlera objašnjen je i alat dex2jar koji je korišten za transformaciju DEX datoteka u JAR datoteke. Zatim je opisano kako razviti vlastiti dodatak za Android Studio razvojno okruženje u IntelliJ IDEA alatu. Objašnjena je IntelliJ platforma, komponente dodataka i kako podesiti IntelliJ IDEA alat za razvoj. Na kraju je prikazan razvijeni dodatak za Android Studio koji integrira navedene dekompajlere i nudi dodatne mogućnosti.

Ključne riječi: Android, dekompajliranje, Android Studio, Android aplikacija, APK datoteka, IntelliJ platforma

Android Studio plugin for support of security analysis of APK files

Abstract

This thesis studies the analysis of APK file and decompilation of its bytecode into source code. At the beginning the components of Android applications, their build process and security model are described. Then the structure of APK file and smali code which is some times needed in order to understand the source code are described. Additionally some tools which make analysis of APK file easier are mentioned. For decompilation of compiled code three decompiler were used: Jadx, Procyon and FernFlower. In separate chapter it is described how to use them, which are their features and how to use them programmatically. Besides decompilers the tool dex2jar for transforming DEX files to JAR files is described. Then it is described how to develop custom plugin for Android Studio development environment in the IntelliJ IDEA tool. IntelliJ platform, components of the plugin and how to setup the IntelliJ IDEA tool for plugin development are described. At the end, the developed plugin which integrates mentioned decompilers and provides additional features is shown.

Keywords: Android, decompilation, Android Studio, Android application, APK file, IntelliJ platform