# Obrnuti inženjering u razvoju softvera

**Vrban, Marko**

**Undergraduate thesis / Završni rad**

**2020**

*Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj:* **University of Zagreb, Faculty of Organization and Informatics / Sveučilište u Zagrebu, Fakultet organizacije i informatike**

*Permanent link / Trajna poveznica:* https://urn.nsk.hr/urn:nbn:hr:211:092873

*Rights / Prava:* Attribution-NoDerivs 3.0 Unported/Imenovanje-Bez prerada 3.0

*Download date / Datum preuzimanja:* **2024-03-29**




*Repository / Repozitorij:*

Faculty of Organization and Informatics - Digital Repository









**U ZAGREBU**

**FAKULTET ORGANIZACIJE I INFORMATIKE**

**V A R A Ž D I N**

**Marko Vrban**

# Obrnuti inženjering u razvoju softvera

**ZAVRŠNI RAD**

**Varaždin, 2020.**

**SVEUČILIŠTE U ZAGREBU**

**FAKULTET ORGANIZACIJE I INFORMATIKE**

**V A R A Ž D I N**

**Marko Vrban**

**Matični broj: 45204/16–R**

**Studij: Informacijski sustavi**

# Obrnuti inženjering u razvoju softvera

**ZAVRŠNI RAD**

**Mentor/Mentorica:**

Izv. prof. dr. sc. Ivan Magdalenić

**Varaždin, kolovoz 2020**

*Marko Vrban*

**Izjava o izvornosti**

Izjavljujem da je moj završni/diplomski rad izvorni rezultat mojeg rada te da se u izradi istoga nisam koristio drugim izvorima osim onima koji su u njemu navedeni. Za izradu rada su korištene etički prikladne i prihvatljive metode i tehnike rada.

*Autor/Autorica potvrdio/potvrdila prihvaćanjem odredbi u sustavu FOI-radovi*

# Sažetak

Kako sam naslov rada govori u radu će biti obrađena tema obrnutog inženjeringa softvera. Kroz rad biti će napravljeni kratku uvod u obrnuti inženjering softvera. Potom će biti prikazani razni alati pomoću kojih se može napraviti obrnuti inženjering te alati i načini koji pružaju zaštitu softvera od obrnutog inženjeringa. Biti će prikazana programska sučelja alata IDA Pro, OllyDbg, Javap – Java Class Decompiler, JDProject.

U radu će biti prikazan algoritam mjehurićastog sortiranja u dva programska jezika, C++ i Java. Tada će se pomoću dva alata, IDA Pro(Evaluation Edition) i Javap – Java Class Decompiler, napraviti obrnuti inženjering algoritma. Dobiveni izvorni assembly i bytecode će biti analizirati, analizirani kod biti će interpretirani u viši programski jezik te će biti uspoređeni sa početnim kodom izvršnih datoteka.

**Ključne riječi:** assembler, obrnuti inženjering, strojni jezik, bytecode, OllyDbg, Javap, decompiler, disassembler

# Sadržaj

1. Uvod ..... 1
2. Metode i tehnike rada ..... 3
3. Alati za obrnuti inženjering ..... 5
3.1. Pristupi obrnutom inženjeringu ..... 6
3.1.1. „Offline code“ analiza ..... 6
3.1.2. „Live code“ analiza ..... 6
3.2. Disassembler alati ..... 7
3.3. Decompiler alati ..... 7
3.4. IDA Pro ..... 8
3.5. OllyDbg ..... 9
3.6. Javap – Java Class Decompiler ..... 10
3.7. JDProject ..... 12
4. Tehnike protiv obrnutog inženjeringa ..... 13
4.1. Osnovni pristupi zaštiti od obrnutog inženjeringa ..... 14
4.2. Enkripcija koda ..... 15
4.3. Prigušivanje koda ..... 16
5. Primjer obrnutog inženjering na metodi mjehurićastog sortiranja ..... 17
5.1. Obrnuti inženjering C++ programa ..... 17
5.1.1. Izvorni C++ kod programa ..... 17
5.1.2. Zbirni kod C++ programa ..... 18
5.1.3. Analiza C++ zbirnog koda ..... 20
5.1.4. Dijagram toka algoritma mjehurićastog sortiranja kod C++ programa ..... 24
5.1.5. Usporedba dobivenog zbirnog i izvornog C++ koda ..... 25
5.2. Obrnuti inženjering Java programa ..... 25

5.2.1. Izvorni Java kod programa ..... 25
5.2.2. Zbirni kod Java programa ..... 25
5.2.3. Analiza zbirnog koda Java programa ..... 27
5.2.4. Usporedba dobivenog zbirnog koda i izvornog koda ..... 29
6. Zaključak ..... 30
Popis literature ..... 32
Popis slika ..... 33
Popis tablica ..... 34

# 1. Uvod

Obrnuti inženjering je proces u kojem se proizvod izrađen putem inženjeringa, npr. auto, avionski pogon, softver, rastavlja, odnosno de konstruira tako da se kao rezultat dobiju njegovi unutarnji detalji, kao što su dizajn i arhitektura. U softverskome svijetu obrnuti inženjering se svodi na uzimanje gotovog programa za koji ne postoji izvorni kod ili popratna dokumentacija i pokušaju se dobiti informacije o dizajnu i implementacije gotovog softvera. Kao softverski inženjering obrnuti inženjering softvera je virtualni proces koji uključuje samo centralnu procesorsku jedinicu računala, odnosno CPU, i čovjeka. [1]

Obrnuti inženjering je posebno koristan u analizi modernih softvera zbog brojnih razloga. Neki od razloga zašto je obrnuti inženjering koristan su sljedeći:

- **Pronalaženje zlonamjernog koda**. Mnogi programski virusi i zlonamjeran kod koriste obrnuti inženjering kako bi razumjeli postojeći kod programa i kako bi zaobišli zaštite anti-virusnih programa. [1]
- **Pronalaženje neočekivanih propusta i grešaka**. Svaki softver koliko god dobro bio dizajniran može sadržavati greške. Obrnuti inženjering u ovoj situaciji omogućuje pronalaženje tih grešaka prije nego one načine neku ozbiljnu štetu. [1]
- **Pronalaženje kako se neki kod koristi**. U ovome slučaju obrnuti inženjering omogućuje pronalaženje ako neki kritični dio aplikacije postoji i kako taj kod radi. [1]
- **Učenje od drugih proizvoda s različite domene ili namjene**. Ovo je korisno svojstvo obrnutog inženjeringa jer omogućuje učenje tehnika naprednih pristupa razvoja softvera. Primjer tome je da su mnoge web stranice napravljene tako da se gledalo kako su neke druge stranice napravljene. [1]
- **Otkrivanje svojstava kojih prvotni developer nije bio svjestan**. Obrnuti inženjering u ovome slučaju omogućuje pronalaženje novih otkrića u vezi softvera koje pružaju nove prilike za inovaciju. [1]

Iako postoje mnogi razlozi za korištenje obrnutog softvera aplikacija, koje su bile nabrojene iznad, obrnuti inženjering se generalno koristi u dvije svrhe: u sigurnosne svrhe i u svrhe razvoja softvera. [1]

Obrnuti inženjering je uključen u brojne aspekte računalne sigurnosti. Proces obrnutog inženjeringa se koristi u kriptografskom istraživanju. U tome procesu istraživač pomoću procesa obrnutog inženjeringa procjenjuje sigurnost nekog kriptiranog proizvoda. Jednako tako obrnuti inženjering se koristi u procesu nastajanja zlonamjernog softvera i u procesu sprečavanja zlonamjernog softvera od strane developera anti-virusnog rješenja. Jedno od najpopularnijih slučajeva korištenja obrnutog inženjeringa je kreking, gdje se analiziraju „copy protection“ zaštite i eventualno se iste zaobilaze. [1]

# 2. Metode i tehnike rada

U ovom dijelu rada biti će prikazani i ukratko objašnjeni alati koji su bili korišteni kod pisanja ovog završnog rada. Za razradu teme i pisanje rada korišteni su sljedeći alati:

- **Microsoft Word 2019** – alat koji je bio korišten za pisanje samog rada, oblikovanje teksta i slika

- **Opera browser** – alat koji je bio korišten za pristupanje online sadržaja i literature potrebne za izradu rada

- **JDeveloper** – alat koji je bio korišten za pisanje Java koda i izradu Java izvršne datoteke programa

- **Dev-C++** - alat koji je bio korišten za pisanje C++ koda i izradu C++ izvršne datoteke programa

- **OllyDbg** – alat koji je bio korišten za dobivanje izvornog koda C++ izvršne datoteke i alat koji je bio analiziran kao alat za obrnuti inženjering

- **Javap – Java Class Decompiler** - alat koji je bio korišten za dobivanje izvornog koda Java izvršne datoteke i alat koji je bio analiziran kao alat za obrnuti inženjering

- **IDA Pro** - alat koji je bio analiziran kao alat za obrnuti inženjering C++ izvršnih datoteka

- **JDProject-** alat koji je bio analiziran kao alat za obrnuti inženjering Java datoteka

- **WinDbg -** alat koji je bio analiziran kao alat za obrnuti inženjering C++ izvršnih datoteka

- **Mendeley desktop** – alat koji je bio korišten za pravilno navođenje literature i citiranje

# 3. Alati za obrnuti inženjering

Obrnuti inženjering nemoguće je obaviti bez pravih alata. Postoje brojni dostupni alati koji se mogu koristiti za obrnuti inženjering. Ključno je razumjeti razliku između njih kako bi se odabrao pravi za obrnuti inženjering željenog programa. Kako postoje dvije platforme na kojima rade izvršne datoteke programa, Java Virtual Machine i Common Language Runtime, jednako kao što je potrebno koristiti različite Compilere za njih da bi se dobila izvršna datoteka tako je potrebno koristiti različite alate za obrnuti inženjering kako bi se dobio izvorni kod tih programa. Korištenjem alata za obrnuti inženjering Java izvršnih datoteke, ekstenzije .jar, rezultat dobivenog izbornog koda je „bytecode“. Prilikom obrnutog inženjeringa i analize bytecode-a promatra se stanje na stogu kako bi se izvukle željene informacije iz izvršne datoteke. S druge strane, prilikom obrnutog inženjeringa izvršne datoteke nekog drugog jezika, npr. C++ jezika kao u ovome radu, rezultat je dobiveni u obliku assembly koda, odnosno asembler programski jezik. Kod analize assembly koda promatra se stanje u registrima procesora, kojih u 32-bitnom sustavu ima ukupno 8. [1]

Navedeni registri i njihovo značenje su sljedeći:

- EAX, EBX, EDX – registri koji služe za integer, bool, logičke i memorijske operacije [1]
- ECX – registar brojač, ponekad služi kao brojač za iteracijske instrukcije [1]
- ESI, EDI – registri koji služe za pokazivače izvora i destinacije kod instrukcija koje kopiraju podatke [1]
- EBP – registar koji služi kao pokazivač na stogu, u kombinaciji s ESP registrom služi oblikuje okvir stogu [1]
- ESP – registar koji služi kao pokazivač na stog CPU-a, u njega se sprema trenutka adresa na stogu [1]

Prilikom obrnutog inženjeringa, proces analize bytecodea potpuno je drugačije od analize izvornog assembly koda. Spomenuta činjenica rezultat je toga što je bytecode puno detaljniji nego izvorni assembly kod. Zbog toga teško je razvijačima programa zaštiti program od obrnutog inženjeringa i tu dolazi do potrebe za zaštitom koda što će kasnije biti detaljnije obrađeno u radu. [1]

## 3.1. Pristupi obrnutom inženjeringu

Postoje mnogi pristupi obrnutom inženjeringu, odabir pravog pristupa ovisi o ciljanom programu, platformi na kojoj radi program i na kojoj je razvijeni program te o vrsti informacije koja se želi dobiti iz izvršnog programa. Generalno ‚pristupi obrnutom inženjeringu su „Offline Code(Dead Listing)“ analiza i „Live Analiza“.[1]

Navedeni slučajevi će biti detaljnije objašnjeni u nastavku.

### 3.1.1. „Offline code“ analiza

Offline analiza koda je pristup kod kojeg se uzme izvršna datoteka programa te se pomoći disassembler ili decompiler alata pretvori u oblik razumljiv čovjeku, odnosno bytecode ili assembly code. Dobiveni rezultata se potom interpretira i analizira kako bi se došlo do informacija. Navedeni pristup je dobar jer prikazuje jasan kod i lako je pronaći specifične funkcije. Loša strana ovog pristupa je za što zahtjeva bolje razumijevanje koda što je posljedica toga jer se ne može pratiti tok podataka u programu. Jednako tako kod ovog pristupa treba pogađati kojim tipovima podataka program upravlja. Zaključno tome offline analiza koda smatra se naprednim pristupom obrnutom inženjeringu. [1]

### 3.1.2. „Live code“ analiza

Live kod analiza koristi jednaki način pretvorbe izvršne datoteke programa u izvorni assembly kod kao što je to kod offline kod analize kako bi program bio razumljiv čovjeku. Ovaj pristup sadrži puno više informacija jer se mogu promatrati podaci kojima program upravlja i kako podaci utječu na tijek samog koda. Jednako tako mogu se promatrati koje podatke određene varijable posjeduju i što se događa kada program čita i modificira te podatke. Pristup live analize se preporučuje za početnike u obrnutom inženjeringu jer pruža puno više podataka. [1]

## 3.2. Disassembler alati

Disassembler alat jedan je od najvažnijih alata za obrnuti inženjering. Zadatak disassembler alata je pretvaranje mašinskog binarnog koda, koji je niz brojeva, u čovjeku čitljivi asembler programski kod. Proces pretvorbe sličan je onome što se dešava u CPU-u kada program radi. Razlika je u tome što umjesto obavljanja zadataka specificiranih u kodu, disassembler dekodira svaku pojedinu instrukciju i stvara tekstualnu reprezentaciju instrukcije. [1]

Proces pretvorbe instrukcije u tekstualni reprezentaciju specifičan je platformi na kojoj program radi, budući da svaka platforma podržava različite instrukcije i različiti set registara, stoga svaki disassembler definiran je platformom na kojoj radi. Ujedno postoje disassembleri koji podržavaju više od jedne platforme. [1]

Disassembler pretvara mašinski kod u tekst tako da uzme opcode instrukcije i traži taj opcode u translacijskoj tablici koja sadrži tekstualni naziv svake instrukcije, zajedno sa njenim formatom. Navede instrukcije slične su funkcijama u programu tako da svaka ima različiti set parametara koje prima. Disassembler potom analizira koji parametri se koriste u pojedinoj instrukciji kako bi ustanovila koja točna instrukcija se koristi. [1]

## 3.3. Decompiler alati

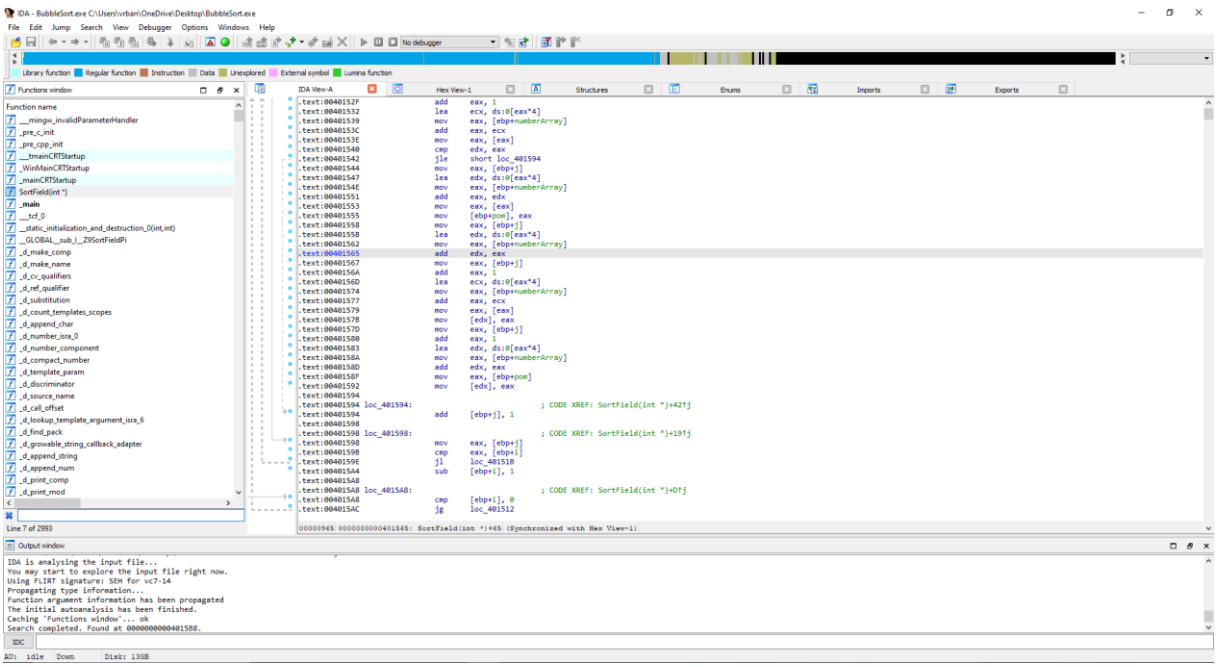
Decompiler alati su naprednija verzija alata za obrnuti inženjering. Kako navodi autor Eldad Eilam u vlastitoj knjizi „Reversing“ ti alati su san svakog pojedinca koji se bavi obrnutim inženjeringom. Za razliku od disassembler alata, decompiler alati iz strojnog koda pokušaju dati što točniju reprezentaciju izvršnog koda datoteke u jeziku visoke razine u kojem je program originalno napisani. Naravno nikad nije moguće vratiti kod u prvotni oblik jer compiler prilikom stvaranja izvršne datoteke uklanja neke informacije iz koda. Količina zadržanih informacija u koda ovisi o programskom jeziku u kojem je program napisani te zbirnom jeziku u koji se program prevađa pomoću compilera i o tome koji se compiler koristi. Kao primjer, .NET programi napisani u nekom od .NET kompatibilnih programskih jezika i preveden u MSIL(Microsoft Instruction Language) može se pretvoriti u zbirni jezik s vrlo dobrim rezultatima. Dok s druge strane programi napisani u nekim drugim jezicima se ne mogu pretvoriti u

reprezentaciju visokog jezika pomoću decompiler alata jer sadrže puno manje informacija iz programskog jezika visoke razine u kojem je program napisan. [1]

## 3.4. IDA Pro

IDA (Interactive Disassembler) od strane Hex Rays-a je ekstremno moćan alat za obrnuti inženjering koji podržava mnoge procesne arhitekture, kao npr. IA-32, AMD64 i mnoge druge. Pomoću IDA alata jednako tako podržava razne vrste izvršnih datoteka kao što su PE(Portable Executable, korišten kod Windowsa), ELF(Executable and Linking Format, korišten kod Linuxa) i čak XBE format, koji se koristi kod Microsoftove Xbox konzole. [1], [2]

Za razliku od ostalih programa koje ću opisati u radu koji su besplatni, IDA Pro je isključivo komercijalan alat. Standarda verzija ovog programa počinje s cijenom od 399$, dok Advance verzija programa počinje s cijenom od 795$. No unatoč cijeni Hex Rays daje besplatnu Evaluation verziju programa koja je dostupna tvrtkama, tako za ovaj rad sam dobio tu verziju preko FOI-ja kao student, koja podržava sve opcije kao i Standard verzija. Na slici 1. prikazano je sučelje programa IDA Pro. [2]



Slika 1. Sučelje programa IDA Pro [vlastita izrada]

Što se tiče opcija koje IDA pro nudi, nema im manjka. Softver ima u sebi uključen debugger te omogućava postavljanje breakpointa u kodu i pokretanje koda kako bi se mogao pratiti tijek izvršavanja funkcija u „Live“ analizi koda. Dodatno, IDA Pro daje jasan prikaz memorije i registara procesora. Uz to, IDA pro vraća iznimno detalju reprezentaciju zbirnog koda dobivenog iz izvršne datoteke. IDA Pro uz sve nabrojene mogućnosti još izrađuje dijagrame toka za svaku pojedinu funkciju. Ti grafovi predstavljaju vizualnu reprezentaciju tijeka koda i kako logičke operacije utječu na tijek koda u programu. [2]

## 3.5. OllyDbg

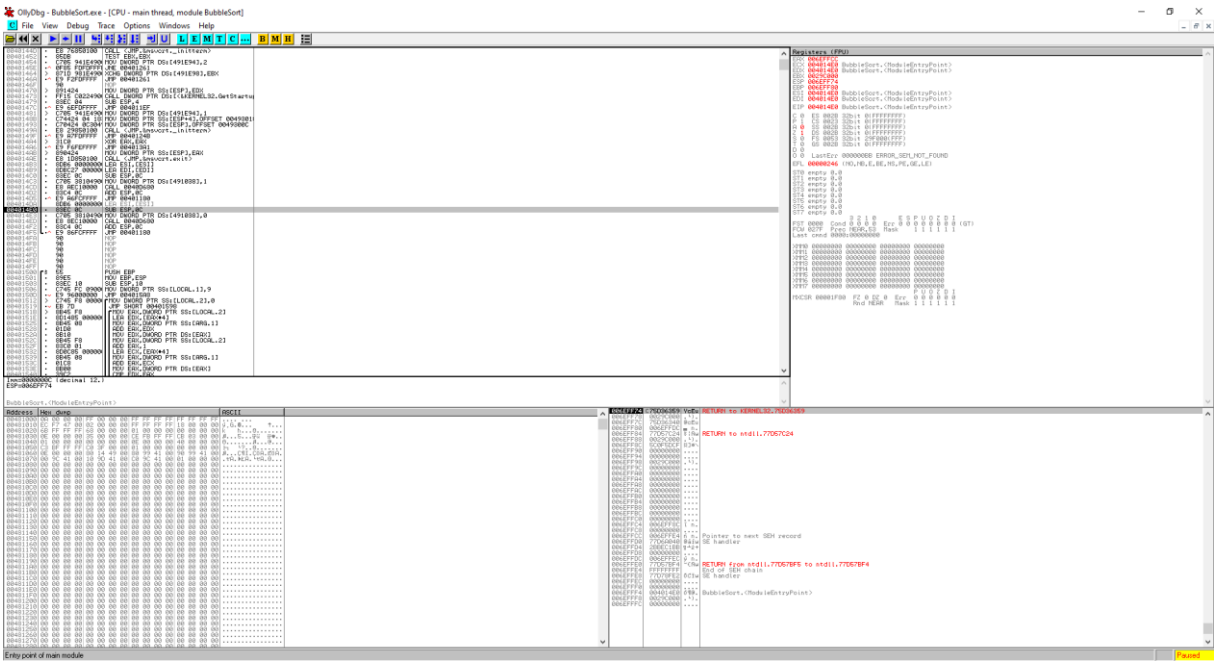
Kako navodi Eldad Eilam u vlastitoj knjizi „Reversing“ za pojedince koji se bave obrnutim inženjeringom alat OllyDbg, razvijen od strane Oleha Yuschuka, je najbolji alat za obrnuti inženjering i on je potpuno besplatan alat. Činjenica da je alat OllyDbg kompletno razvijen kao alat za obrnuti inženjering ima u sebi ugrađen vrlo moćan disassembler. Kao i alat IDA Pro, OllyDbg daje prikaz memorije i registara procesora te omogućava postavljanje breakpointa unutar programa i pokretanje programa kako bi se pratio tijek programa učitanog u OllyDbg alat. [1], [3]

Najjača strana alata OllyDbg je njegov disassembler koji omogućava odlične opcije za analizu dobivenog zbirnog koda iz izvršne datoteke. Prilikom analize koda OllyDbg identificira petlje, switch blokove u kodu i druge ključne strukture unutar koda. Pritom alat prikazuje nazive za sve poznate funkcije i API-je, ujedno podržava i pretraživanje dobivenog koda u oba smjera. To čini OllyDbg jednim od najboljih alata s disassembly mogućnostima. [1], [3]

Osim moćnih disassembly mogućnosti, OllyDbg podržava razne poglede, što uključuje prikaz uvoza i izvoza u modelu programa, listu sučelja i objekata koje koristi izvršna datoteka učitana u OllyDbg alat, listu upravitelja pogrešaka u kodu, uveze biblioteke u izvršnu datoteku programa i ostale informacije. [1], [3]

OllyDbg ujedno podržava assembly i opcije za zakrpu programa, što je korisno kod krekiranja programa te ga to čini omiljenim alatom kod probijanja zaštita nekih programa. Moguće je pomoću OllyDbg programa u zbirni kod dopisati vlastiti kod i

promjene vratiti u izvršnu datoteku ako je to potrebno te tako promijeniti postojeću izvršnu datoteku programa. Na slici 2. prikazano je sučelje OllyDbg programa.



Slika 2. Sučelje OllyDbg alata [vlastita izrada]

## 3.6. Javap – Java Class Decompiler

Javap – Java Class Decompiler je alat koji služi za obrnuti inženjering klasa Java izvršnih datoteka ekstenzije .jar. Alat radi na način da zbirni bytecode kod pretvara u čovjeku čitljivi Java kod. Iz izvršne Java datoteke do klase se može doći na jednostavan način da se .jar raspakira nekim od alata, kao što je npr. WinRar. Javap – Java Class Decompiler je alat koji dolazi u sklopu Java Development Kita i nije ga potrebno posebno instalirati na računalo kako bi se koristio. Za razliku od ostalih alata Javap nema vlastito korisničko sučelje nego se on pokreče pomoću naredbenog retka. Način pokretanja Javap – Java Class Decompilera je da se u naredbenom retku postavi u bin direktorij postavi gdje je instalirana Java i doda se putanja do Java klase. Način pokretanja Javap-a bi izgledao na sljedeći način:

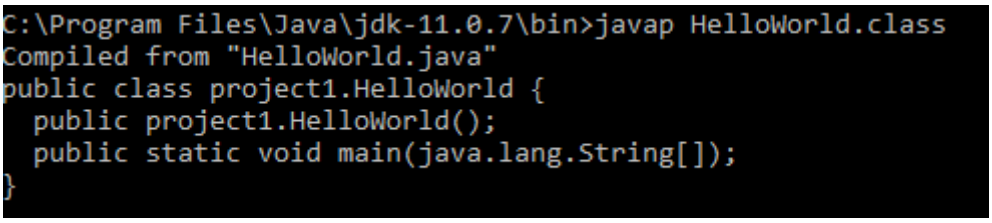
Javap NazivJavaKlase.class

Dobiveni rezultat obrnutog inženjeringa pomoću Javap – Java Class Decompilera ovisi o tome koja opcija se postavi nakon ključne riječi Javap u naredbenome retku. Na sljedećem primjeru jednostavnog HelloWorld programa budu prikazani različiti rezultata pokretanja programa Javap bez i s postavljenom opcijom.

Osnovni kod programa je sljedeći:

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

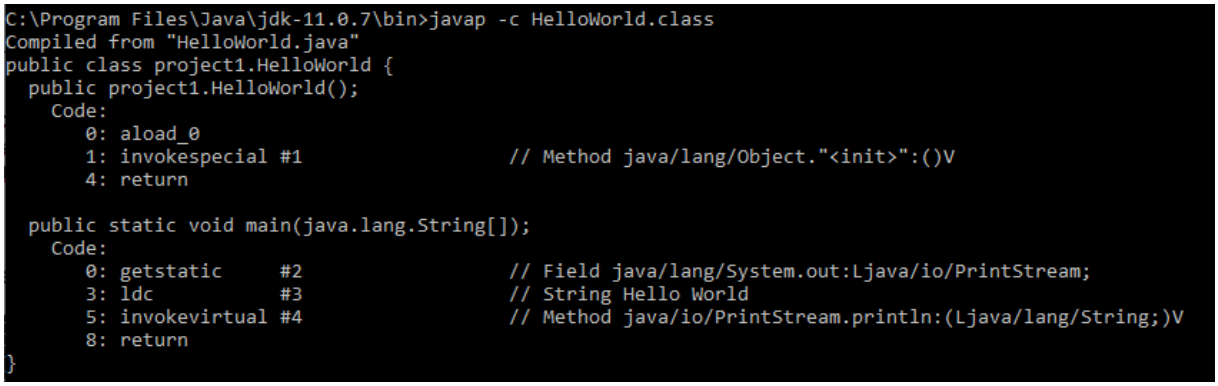
Pokretanjem Javap – Java Class Decompilera dobiveni rezultat je prikazani na slici 3.




Slika 3. rezultat Javap - Java Class Decompilera [vlastita izrada]

Kao što je vidljivo na slici dobiveni rezultat sadrži samo nazive klasa koji se nalaze unutar klase HelloWorld.class bez ikakvih dodatnih informacija.

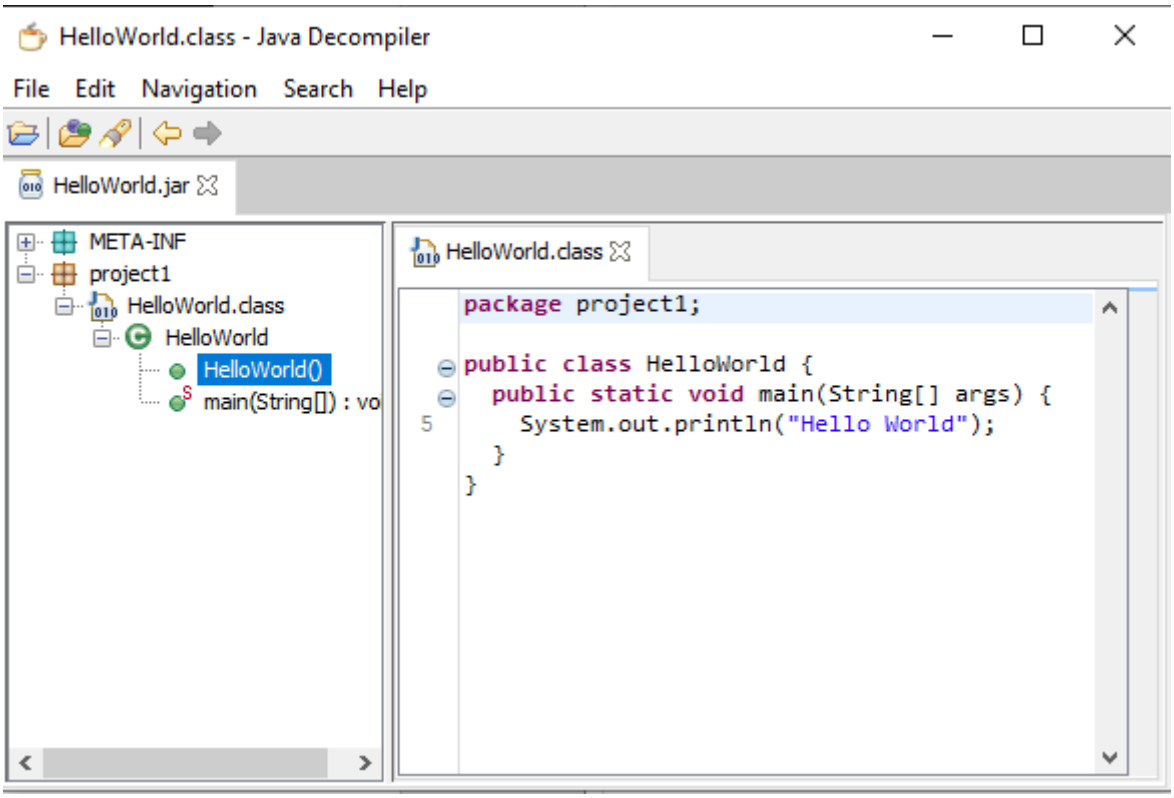
Sljedeće će biti pokrenuta ista naredba samo će biti dodana opcija -c tako da će naredba za pokretanje Javap-a izgledati „javap -c HelloWorld.class“. Postavljanjem opcije „-c“ Javap će vratiti rezultat u klase u obliku bytecoda, što je prikazano na slici 4 u nastavku.




Slika 4. rezultat Javap - Java Class Decompilera [vlastita izrada]

## 3.7. JDProject

JDProject je najkorišteniji alat za obrnuti inženjering Java izvršnih datoteka. Alat je razvijen kako bi omogućio obrnuti inženjering Java koda verzije 5 ili novije verzije. JDProject je besplatan alat te ga se može skinuti na stranici va-decompiler.hithub.io. Verzija programa JD-GUI je samostojeći alat koji ima vlastito grafičko sučelje i on omogućava prikaz izvornog Java koda .class datoteka. Alat omogućava pregled cijelog rekonstruiranog Java koda za laki pronalazak metoda i polja. Dobiveni rezultat rekonstrukcije koda je hijerarhijski raspoređen jednako kako je to napravljeno prilikom razvoja programa. Za alat JD-GUI ujedno postoji JD-Eclipse dodatak. Dodatak JD-Eclipse omogućava korisniku prikaz Java izvornog koda tijekom procesa uklanjanja grešaka(debugging). Slika 5. prikazuje grafičko sučelje programa JDProject. [5], [6]




Slika 5. grafičko sučelje programa JDProject [vlastita izrada]

# 4. Tehnike protiv obrnutog inženjeringa

Postoje brojne situacije kada je poželjno razviti softver koji je imun na proces obrnutog inženjeringa. Takav pristup razvoja programa koji će biti potpuno imuni je nemoguće, no što se može napraviti je usporiti pojedinca koji pokuša obrnutim inženjeringom doći do podataka unutar programa u nadi da se oteža sam proces i da pojedinac odustane. Efektivnost zaštite ovisi o tome koliko je netko voljan platiti za zaštitu jer alati koji pomažu kod toga nisu besplatni iako neki nude probne verzije. Svaki način zaštite programa od obrnutog inženjeringa nosi neku cijenu za sobom, bilo to kapacitet korištenja procesora računala, veličina koda i ponekad pouzdanost i robusnost samog programa. [1], [7]

Zanemarujući cijenu uključenu kako bi se program zaštitio, proces zaštite uvijek ima smisla. Razlog tome je da korisnik koji nije u razvojnom krugu ne može doći do izvornog koda programa ako je program namijenjen komercijalnoj primjeni, a nije besplatan. Treba uzeti u obzir kako svaki program nije vrijedan procesa zaštite od obrnutog inženjeringa ako se sastoji od jednostavnog koda, takav kod je isplativije razviti samostalno nego prolaziti kroz proces obrnutog inženjeringa. Neke aplikacije zahtijevaju zaštitu od obrnutog inženjeringa kako bi se spriječio plagijat razvijenog programa i narušila digitalna prava pojedinca koji je razvio program. Sprječavanje pojedinca kako bi vidio izvorni kod programa koji je zaštićen patentom važan je postupak kako bi se spriječilo narušavanje prava i da bi se program dodatno zaštitio od krađe. [1], [7]

Naime, neke platforme za razvoj programa zahtijevaju zaštitu od obrnutog inženjeringa, jer bi u suprotnome pojedinac mogao lako doći do skoro jednake reprezentacije izvornog koda. Navedena činjenica veže se za sve platforme koje se baziraju na bytecode zbirnom jeziku, kao što su Java i .NET. Jedan od takvih alata koji omogućavaju zaštitu izvornog koda je obfuscator o kojem će biti malo više rečeno u nastavku poglavlja. [1]

## 4.1. Osnovni pristupi zaštiti od obrnutog inženjeringa

Postoji nekoliko pristupa, svaki sa svojim dobrim i lošim stranama. Programi koji namjeravaju koristiti neku vrstu zaštite od obrnutog inženjeringa većinom će koristiti kombinaciju više takvih pristupa. [1]

Osnovni pristupi su sljedeći:

- **Eliminacija simboličnih informacija** – prvi i najočitiji korak u sprečavanju obrnutog inženjeringa je eliminacija svih tekstualnih informacija iz programa. U regularnom programu koji se ne bazira na bytecode zbirnog jeziku to znači brisanje svih simboličnih informacija iz izvršne datoteke programa. U programima koji se baziraju na bytecode zbirnom jeziku izvršna datoteka programa sadrži veliki broj informacija kao što su to nazivi klasa, nazivi članova klasa i inicijaliziranih globalnih varijabli i objekata. To se primarno odnosi na programe napisane u Java ili nekom od .NET programskih jezika. Navedene informacije mogu biti iznimno korisne pojedincu kako bi olakšale proces obrnutog inženjeringa, zato se one moraju obavezno maknuti iz programa prilikom izrade izvršne datoteke. Iz tog razloga postoje alati prigušivači(eng. Obfuscator) koji u tome slučaju sve simbolične nazive preimenuju u niz beznačajnih znakova. [1]
- **Prigušivanje programa** – zbunjivanje(eng. „Obfuscating“) je naziv za broj načina kojima je cilj smanjiti ranjivost programa bilo kakvoj statičnoj analizi prilikom procesa obrnutog inženjeringa. Zbunjivanje se postiže tako da se modificira raspored unutar programa, logika programa, sadržaj koji program koristi i organizacija koda tako da program ostaje funkcionalan, ali se prilikom procesa obrnutog inženjeringa smanjuje čitljivost koda. [1]
- **Ugradnja koda protiv obrnutog inženjeringa** – ovaj način sprječavanja procesa obrnutog inženjeringa programa specifično se odnosi na sprečavanje „live“ analize koda koja je prije u radu bila objašnjena. Ideja je u ovome slučaju u program ugraditi dio koda koji će štetiti ili onemogućiti alat za obrnuti inženjering. Neki od tih postupaka rade

jednostavno tako da detektiraju rad alata za obrnuti inženjering i zaustavljaju njegov rad, dok postoje i napredniji načini sprečavanja rada alata za obrnuti inženjering. Postoje brojni načini unutar ovog postupka koji su mnogi specifični za određenu platformu i čak za pojedini alat za obrnuti inženjering.

## 4.2. Enkripcija koda

Enkripcija koda je najraširenija metoda sprečavanje statične analize koda prilikom procesa obrnutog inženjeringa. Postiže se tako da se program u jednome trenutku nakon izrade izvršne datoteke kriptira i ugrađuje se kod za dekriptiranje unutar izvršne datoteke programa. Ovaj postupak nije pretjerano efektivan jer za iskusnog pojedinca koji se bavi obrnutim inženjeringom predstavljat će samo manju neugodnost jer sve potrebno za dekriptiranje izvršne datoteke programa se nalazi unutar same izvršne datoteke, tako da osoba lako može doći do logike i ključa potrebnih za dekriptiranje programa. [1]

Kriptiranje koda svejedno je najučestalija korištena tehnika za sprječavanje statične analize programa, jer komplicira postupak analize koda programa. Loša strana je što u mnogo slučajeva kriptirana izvršna datoteka programa se može programski dekriptirati pomoću nekih alata koji su upoznati s korištenim algoritmom kriptiranja. Takvi alati za dekriptiranje kriptirane izvršne datoteke će stvoriti novu izvršnu datoteku programa koja neće biti kriptirana, putem koje je lako onda doći do izvornog koda programa procesom obrnutog inženjeringa. [1]

Jedini efikasan način sprečavanje dekriptiranja izvršne datoteke je pokušaj sakrivanja ključa za dekriptiranje unutar programa. Jedan takav način je korištenje ključa za dekriptiranje programa koji se generira tijekom pokretanja programa na temelju nekog algoritma. Takav pristup može se postići korištenjem više globalnih varijabli kojima pristupaju različiti dijelovi programa i mijenjaju njihovu vrijednost. Analizom koda uživo(eng. „Live“ analizom koda) lako bi se mogao pronaći takav ključ u programu no ideja je koristiti puno globalnih varijabli da bi pojedincu trebalo puno vremena da pronađe sve vrijednosti varijabli te na posljetku da odustane. [1]

## 4.3. Prigušivanje koda

Iako je prije ukratko bilo spomenuto prigušivanje programa, pravo prigušivanje programa uključuje promjenu koda tako da on postane manje razumljiv čovjeku dok još uvijek zadrži svoju funkcionalnost. Ovaj postupak radi tako da sakriva početnu namjenu programa i utapa pojedinca koji pokuša postupkom obrnutog inženjeringa doći do informacija o programu s brdo beskorisnih informacija. Razina kompleksnosti dodana u program pomoću nekog alata za zagušivanje programa mjeri se u potentnosti programa. Potentnost programa izračunava se na temelju toga koliko izraza program sadrži i dubini ugniježđenosti unutar određene sekvence u kodu. [1]

Osim kompleksnosti nadodane pomoću dodatne logike i aritmetike programa proces zagušenja koda mora biti otporan. Razlog za postojanje otpornosti je taj što postoje alati za zagušivanje koda programa tako postoje alati koji kod vraćaju u početni oblik. Alati koji vraćaju kod u početni oblik analiziraju zagušeni kod, mogu prepoznati nadodanu logiku i aritmetiku u program na temelju analize toka podataka te vraćaju kod programa u početni oblik. [1], [7]

Tipično proces zagušivanja koda će imati neku cijenu. To može biti u obliku većeg koda odnosno veće izvršne datoteke programa, sporije vrijeme izvršavanja koda, ili povećanu količinu potrošnje memorije prilikom izvršavanja. Važno je za prepoznati kako neće svako zagušenje programa imati povezanu cijenu jer će neki procesi zagušenja samo reorganizirati kod koji je transparentan stroju na kojem program radi te će kod učini samo manje čitljiv čovjeku. [1], [7]

Primjer jednog alata za zagušivanje izvršne datoteke programa je ConfuserEx koji je potpuno besplatan alat i njegov izvorni kod je dostupan korisnicima. [7]

# 5. Primjer obrnutog inženjering na metodi mjehurićastog sortiranja

U ovome poglavlju rada biti će prikazan primjer obrnutog inženjeringa dva jednostavna programa koji izvršavaju algoritam mjehurićastog sortiranja polja. Programi koji će biti analizirani su napisani u C++ i Java programskog jeziku. Dobiveni zbirni kodovi biti će detaljno analizirani u nastavku rada te ću pokušati doći pomoću analize do izvornog koda programa te ću na kraju usporediti dobiveni izvorni kod analizom sa originalnim izvornim kodom.

## 5.1. Obrnuti inženjering C++ programa

Zbirni kod C++ programa procesom obrnutog inženjeringa izvršne datoteke dobiven je pomoću alata IDA Pro (Evaluation Edition).

### 5.1.1. Izvorni C++ kod programa

```
#include <iostream>
#define MAX_ELEMENTS 10

using namespace std;

void SortField(int* numberArray){
    for (int i = MAX_ELEMENTS - 1; i > 0; i--) {
        for (int j = 0; j < i; j++)
            if (numberArray[j] > numberArray[j + 1]) {
                float pom = numberArray[j];
                numberArray[j] = numberArray[j + 1];
                numberArray[j + 1] = pom;
            }
    }
}

int main() {

    int numberArray[MAX_ELEMENTS] = { 4,10,123,12,3,7,13,6,20,1 };

      SortField(numberArray);

    for (int k = 0; k < MAX_ELEMENTS; k++) {
        cout << *(numberArray + k) << " ";
    }
    cout << endl;
    system("pause");
    return 0;
}
```

## 5.1.2. Zbirni kod C++ programa

```
.text:00401500                push    ebp
.text:00401501                mov     ebp, esp
.text:00401503                sub     esp, 14h
.text:00401506                mov     [ebp+i], 9
.text:0040150D                jmp     loc_4015B9
.text:00401512                mov     [ebp+j], 0
.text:00401519                jmp     loc_4015A9
.text:0040151E                mov     eax, [ebp+j]
.text:00401521                lea     edx, ds:0[eax*4]
.text:00401528                mov     eax, [ebp+numberArray]
.text:0040152B                add     eax, edx
.text:0040152D                mov     edx, [eax]
.text:0040152F                mov     eax, [ebp+j]
.text:00401532                add     eax, 1
.text:00401535                lea     ecx, ds:0[eax*4]
.text:0040153C                mov     eax, [ebp+numberArray]
.text:0040153F                add     eax, ecx
.text:00401541                mov     eax, [eax]
.text:00401543                cmp     edx, eax
.text:00401545                jle     short loc_4015A5
.text:00401547                mov     eax, [ebp+j]
.text:0040154A                lea     edx, ds:0[eax*4]
.text:00401551                mov     eax, [ebp+numberArray]
.text:00401554                add     eax, edx
.text:00401556                mov     eax, [eax]
.text:00401558                mov     [ebp+var_14], eax
.text:0040155B                fild    [ebp+var_14]
.text:0040155E                fstp    [ebp+pom]
.text:00401561                mov     eax, [ebp+j]
.text:00401564                lea     edx, ds:0[eax*4]
.text:0040156B                mov     eax, [ebp+numberArray]
.text:0040156E                add     edx, eax
.text:00401570                mov     eax, [ebp+j]
.text:00401573                add     eax, 1
.text:00401576                lea     ecx, ds:0[eax*4]
.text:0040157D                mov     eax, [ebp+numberArray]
.text:00401580                add     eax, ecx
.text:00401582                mov     eax, [eax]
.text:00401584                mov     [edx], eax
.text:00401586                mov     eax, [ebp+j]
.text:00401589                add     eax, 1
.text:0040158C                lea     edx, ds:0[eax*4]
.text:00401593                mov     eax, [ebp+numberArray]
.text:00401596                add     edx, eax
.text:00401598                mov     eax, [ebp+pom]
.text:0040159B                mov     [ebp+var_14], eax
.text:0040159E                cvttss2si eax, [ebp+var_14]
.text:004015A3                mov     [edx], eax
.text:004015A5                add     [ebp+j], 1
.text:004015A9                mov     eax, [ebp+j]
.text:004015AC                cmp     eax, [ebp+i]
.text:004015AF                jl      loc_40151E
.text:004015B5                sub     [ebp+i], 1
.text:004015B9                cmp     [ebp+i], 0
.text:004015BD                jg      loc_401512
.text:004015C3                leave
.text:004015C4                retn

.text:004015C5                lea     ecx, [esp+4]
```

```
.text:004015C9                and     esp, 0FFFFFFF0h
.text:004015CC                push    dword ptr [ecx-4]
.text:004015CF                push    ebp
.text:004015D0                mov     ebp, esp
.text:004015D2                push    ecx
.text:004015D3                sub     esp, 44h
.text:004015D6                call    ___main
.text:004015DB                mov     [ebp+numberArray], 4
.text:004015E2                mov     [ebp+var_30], 0Ah
.text:004015E9                mov     [ebp+var_2C], 7Bh ; '{'
.text:004015F0                mov     [ebp+var_28], 0Ch
.text:004015F7                mov     [ebp+var_24], 3
.text:004015FE                mov     [ebp+var_20], 7
.text:00401605                mov     [ebp+var_1C], 0Dh
.text:0040160C                mov     [ebp+var_18], 6
.text:00401613                mov     [ebp+var_14], 14h
.text:0040161A                mov     [ebp+var_10], 1
.text:00401621                lea     eax, [ebp+numberArray]
.text:00401624                mov     [esp], eax      ; numberArray
.text:00401627                call    __Z9SortFieldPi ; SortField(int *)
.text:0040162C                mov     [ebp+var_C], 0
.text:00401633                jmp     short loc_40166A
.text:00401635                mov     eax, [ebp+var_C]
.text:00401638                lea     edx, ds:0[eax*4]
.text:0040163F                lea     eax, [ebp+numberArray]
.text:00401642                add     eax, edx
.text:00401644                mov     eax, [eax]
.text:00401646                mov     [esp], eax
.text:00401649                mov     ecx, offset __ZSt4cout ; std::cout
.text:0040164E                call    __ZNSolsEi      ;
std::ostream::operator<<(int)
.text:00401653                sub     esp, 4
.text:00401656                mov     dword ptr [esp+4], offset asc_489000
; " "
.text:0040165E                mov     [esp], eax      ;
std::ostream::sentry *
.text:00401661                call
__ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc ;
std::operator<<<std::char_traits<char>>(std::ostream &,char const*)
.text:00401666                add     [ebp+var_C], 1
.text:0040166A
.text:0040166A loc_40166A:                            ; CODE XREF:
_main+6E↑j
.text:0040166A                cmp     [ebp+var_C], 9
.text:0040166E                jle     short loc_401635
.text:00401670                mov     dword ptr [esp], offset
__ZSt4endlIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_ ;
std::endl<char,std::char_traits<char>>(std::ostream &)
.text:00401677                mov     ecx, offset __ZSt4cout ; std::cout
.text:0040167C                call    __ZNSolsEPFRSoS_E ;
std::ostream::operator<<(std::ostream & (*)(std::ostream &))
.text:00401681                sub     esp, 4
.text:00401684                mov     dword ptr [esp], offset Command ;
"pause"
.text:0040168B                call    _system
.text:00401690                mov     eax, 0
.text:00401695                mov     ecx, [ebp+var_4]
.text:00401698                leave
.text:00401699                lea     esp, [ecx-4]
.text:0040169C                retn
```

### 5.1.3. Analiza C++ zbirnog koda

Sa analizom koda započinjemo od memorijske adrese 004015D6, gdje kako vidimo po nazivu instrukcije se prvi put poziva glavna funkcija(eng. „main“) programa, što znači da tu počinje izvršavanje koda funkcije.

Odmah nakon vidimo instrukciju mov koja se ponavlja 10 puta:

```
mov     [ebp+numberArray], 4,
```

instrukcija uvećava vrijednost registra ebp za veličinu varijable numberArray i postavlja njegovu vrijednost na 4, po tome i činjenici da se instrukcija ponavlja 10 puta možemo zaključiti da se u tome dijelu inicijalizira polje brojeva. Kako su vrijednosti koje smo dobili procesom obrnutog inženjeringa u heksadecimalnom brojevnom sustavu da bi znali točne vrijednosti tih brojeva trebamo ih pretvoriti u dekadski brojevni sustav. Nakon što to napravimo znamo da su brojevi u polju sljedeći:

Tablica 1. Vrijednosti brojeva u polju

| Heksadecimalna vrijednost | Dekadska vrijednost |
|---|---|
| 4 | 4 |
| 0A | 10 |
| 7B | 123 |
| 0C | 12 |
| 3 | 3 |
| 7 | 7 |
| 0D | 13 |
| 6 | 6 |
| 14 | 20 |
| 1 | 1 |

Tu već možemo zaključiti kako je prva instrukcija u main funkciji koja se izvršava jednaka:

```
int numberArray = {4,10,123,12,3,7,13,6,20,1};
```

koja inicijalizira polje brojeva koje program tada sortira. Naziv polja nije nasumično odabran nego tu informaciju isto možemo iščitati iz instrukcije mov.

Sljedeća bitna instrukcija se nalazi na adresi 00401627, to je adresa gdje se poziva funkcija unutar programa pod nazivom SortField(int*) i znamo po vraćenome

rezultatu da ona prima jedan parametar koji je pointer na polje brojeva. Po nazivu funkcije možemo zaključiti kako je to funkcija koja sortira polje brojeva inicijalizirano na početku. Na početku SortField(int*) funkcije prvo se izvršavaju sljedeće instrukcije:

```
push    ebp
mov     ebp, esp
sub     esp, 14h
mov     [ebp+i], 9
jmp     loc_4015B9
```

Prva instrukcija koja se izvodi postavlja vrijednost ebp registra na vrh stoga, tada instrukcija mov postavlja vrijednost ebp na vrijednost registra esp te potom se od vrijednosti registra esp oduzima vrijednost 14 u heksadecimalnom sustavu. Iako je ovaj dio koda čini nerazumljivim ta sekvenca instrukcija sa stoga oslobađa količinu 14 heksadecimalno. Sljedeće funkcija mov na adresi registra ebp uz pomak za vrijednosti „i“ postavlja vrijednost 9, u drugim riječima tu se inicijalizira lokalna varijabla i njena vrijednost iznosi 9 te program nastavlja dalje sa skokom na adresu 004015B9.

Na toj adresi izvršavaju se sljedeće instrukcije:

```
cmp     [ebp+i], 0
jg      loc_401512
```

Tu vidimo da program jednostavno provjerava ako je vrijednosti varijable „i“ veća od nule i nastavlja rad skokom na adresu 00401512. Na toj adresi izvršavaju se dvije instrukcije:

```
mov     [ebp+j], 0
jmp     loc_4015A9
```

Izvršavanjem ovih dviju funkcija koje su jednake kao i prije kod inicijalizacije varijable „i“ ovdje se inicijalizira varijabla pod nazivom „j“ i njena vrijednost je 0 te program skače i nastavlja rad na adresi 004015A9.

Na toj adresi izvršavaju se sljedeće instrukcije:

```
mov     eax, [ebp+j]
cmp     eax, [ebp+i]
jl      loc_40151E
sub     [ebp+i], 1
```

prva instrukcija mov postavlja vrijednost registra eax na vrijednost varijable „j“, u nastavku cmp instrukcija uspoređuje vrijednost registra eax, odnosno varijable „j“ sa varijablom „i“. Ako je vrijednost varijable „j“ manja od vrijednosti varijable „i“ program nastavlja radom na adresi 0040151E, a u suprotnome umanjuje vrijednost varijable „i“ za 1 i nastavlja sa radom na adresi 00415B9 koju smo već objasnili do sada.

Na temelju ovog analiziranog koda možemo doći do zaključka da se radi o dvije „for“ petlje, gdje je jedna ugniježđena unutar druge, jednako tako budući da je vrijednost prve varijable veća od vrijednosti druge varijable znamo da će se polje sortirati uzlazno jer će nakon svake iteracije najveći broj u preostalom polju isplivati na kraj polja. Tu već možemo doći do zaključka kako izgleda početak SortField(int*), koji bi bio sljedeći:

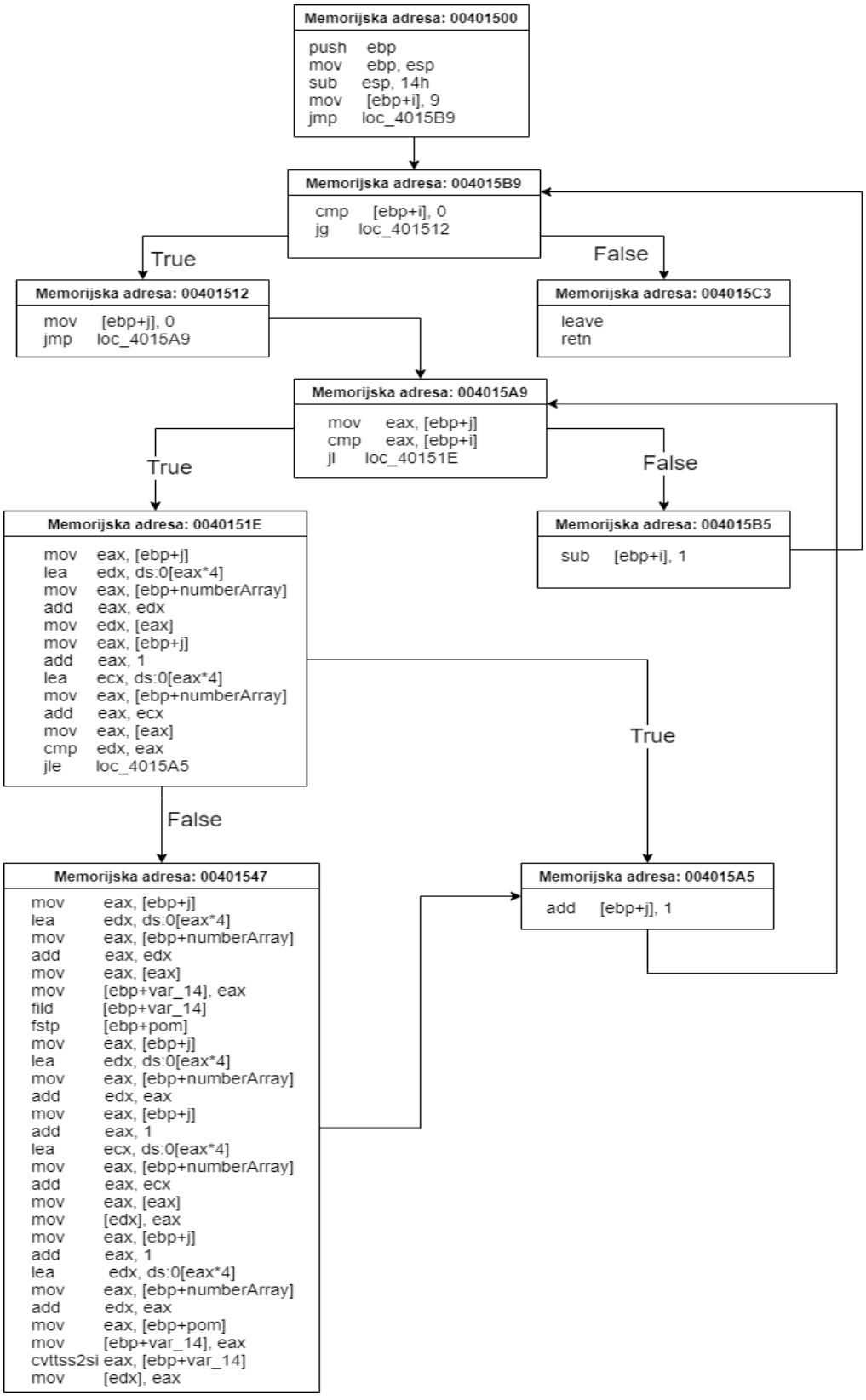
```
void SortField(int* Field){
	int i=9;
	int j=0;
	for(;i>0;i-1){
		for(;j<i;j++){
		}
	}}
```

Postojećim znanjem bez da nastavljamo analizu koda možemo doći do zaključka kako izgleda ostatak funkcije SortField(int*), što bi po pretpostavci bilo sljedeće:

```
void SortField(int* Field){
	int i=9;
	int j=0;
	for(;i>0;i-1){
		for(;j<i;j++){
			if(Field[j]>Field[j+1]){
				int k = Field[j]
				Field[j]=Field[j+1]
				Field[j+1]=k;
			}
		}
	}
}
```

Ako se sad vratimo unazad, daljnjom analizom main() funckije programa, po informacijama koje smo dobili obrnutim inženjeringom vidimo da se više nikakva funkcija ne poziva te se samo izvršava ispis sortiranom polja numberArray[10] na ekran korisnika putem programa.

### 5.1.4. Dijagram toka algoritma mjehurićastog sortiranja kod C++ programa




Slika 6. Dijagram toka algoritma mjehurićastog sortiranja [vlastita izrada]

### 5.1.5. Usporedba dobivenog zbirnog i izvornog C++ koda

Analizom zbirnog koda dobivenog procesom obrnutog inženjeringa izvršne datoteke programa koji nije imao nikakvu zaštitu, vidimo koliko zapravo podataka možemo dobiti iz jedne izvršne datoteke. Ako usporedimo početni izvorni kod i onaj rekonstruirani putem analize zbirnog koda vidimo da zapravo možemo na taj način doći do početnog koda nekog programa. Iz zbirnog koda uspješno je dobivena van informacija o nazivu funkcija koje se pozivaju unutar programa, nazivu varijabli unutar main i SortField(int*) funkcija, te čak podaci kojima program upravlja.

## 5.2. Obrnuti inženjering Java programa

Zbirni bytecode kod Java izvršne datoteke dobiven je pomoću alata Javap – Java Class Decompiler.

### 5.2.1. Izvorni Java kod programa

```
package bubblesort;

public class BubbleSort {
    public static void main(String[] args){
         int numberArray[] = {4,10,123,12,3,7,13,6,20,1};

         for (int i = 10 - 1; i > 0; i--) {
             for (int j = 0; j < i; j++)
                 if (numberArray[j] > numberArray[j + 1]) {
                     int pom = numberArray[j];
                     numberArray[j] = numberArray[j + 1];
                     numberArray[j + 1] = pom;
                 }
         }

         for (int k = 0; k < 9; k++) {
             System.out.print(numberArray[k]+" ");
         }
         System.out.print(numberArray[9]+" ");
         System.out.println();
     };
}
```

### 5.2.2. Zbirni kod Java programa

```
public class bubblesort.BubbleSort {
  public bubblesort.BubbleSort();

  public static void main(java.lang.String[]);
    Code:
       0: bipush        10
       2: newarray       int
```

```
 4: dup
 5: iconst_0
 6: iconst_4
 7: iastore
 8: dup
 9: iconst_1
10: bipush        10
12: iastore
13: dup
14: iconst_2
15: bipush        123
17: iastore
18: dup
19: iconst_3
20: bipush        12
22: iastore
23: dup
24: iconst_4
25: iconst_3
26: iastore
27: dup
28: iconst_5
29: bipush        7
31: iastore
32: dup
33: bipush        6
35: bipush        13
37: iastore
38: dup
39: bipush        7
41: bipush        6
43: iastore
44: dup
45: bipush        8
47: bipush        20
49: iastore
50: dup
51: bipush        9
53: iconst_1
54: iastore
55: astore_1
56: bipush        9
58: istore_2
59: iload_2
60: ifle          113
63: iconst_0
64: istore_3
65: iload_3
66: iload_2
67: if_icmpge     107
70: aload_1
71: iload_3
72: iaload
73: aload_1
74: iload_3
75: iconst_1
76: iadd
77: iaload
78: if_icmple     101
81: aload_1
82: iload_3
```

```
 83: iaload
 84: istore        4
 86: aload_1
 87: iload_3
 88: aload_1
 89: iload_3
 90: iconst_1
 91: iadd
 92: iaload
 93: iastore
 94: aload_1
 95: iload_3
 96: iconst_1
 97: iadd
 98: iload         4
100: iastore
101: iinc          3, 1
104: goto          65
107: iinc          2, -1
110: goto          59
113: iconst_0
114: istore_2
115: iload_2
116: bipush        9
```

### 5.2.3. Analiza zbirnog koda Java programa

Analizom programa od početka main() funkcije već u drugoj liniji vidimo da se inicijalizira novo polje, dok u prvoj linija koda vidimo da je njegova veličina 10. Od linije koda pod brojem 5 pa sve do linije koda 54 vidimo kako se jedan po jedan broj dodaju u polje, na početku odmah instrukcija iconst_0 stavlja vrijednost 0 na stog, sljedeća linija iconst_4 stavlja vrijednost na stog te sljedeća linija koda iastore sprema vrijednost 4 u polje na mjesto indeksa 1. Taj postupak se ponavlja sve do linije 54, gdje se 3 jednake funkcije ponavljaju kako bi se vrijednost na određenome indeksu dodala u polje. Ako pogledamo te vrijednosti one nisu u heksadecimalnom obliku kao kod C++ programa nego u dekadskome pa odmah znamo koje vrijednosti se nalaze u polju no u ovome slučaju ne znamo sam naziv polja, stoga zaključujemo da se odmah na početku main() funkcije inicijalizira neko polje sa 10 brojeva koji su sljedeći:

Int field[10] = {4, 10, 123, 12, 3, 7, 13, 6, 20, 1};

U nastavku na liniji 56 do 60 vidimo da se inicijalizira još jedna varijabla i njena vrijednost postaje 9, tada se vrijednost te varijable provjerava instrukcijom ifle ako vrijednost nije manja ili jednaka nuli, ako jest tada program skače na liniju koda 113. Na linija koda 113 gdje se vrijednost varijable postavlja na nulu. U suprotnome program

nastavlja sa radom sa sljedećom linijom koda 63 gdje se ponovno na isti način inicijalizira nova varijabla i njena vrijednost je 0. Potom se putem instrukcija iload_3 i iload_2 učitavaju varijable pod rednim brojem 3 i 2, tako da je na stogu varijabla 2 na vrhu odnosno ona je prva varijabla u usporedbi. Sljedeća funkcija if_icmpge provjerava ako je prva vrijednost veća ili jednaka drugoj te u tome slučaju čini uvjetni skok na liniju koda 107. Na liniji koda 107, instrukcija iinc 2, -1 uvećava varijablu 2 za vrijednost -1 te se program vraća na liniju koda 59 gdje se provjerava ako varijabla nije manja ili jednaka nuli, što je već bilo objašnjeno. U suprotnome varijabla nastavlja se izvršavanje koda na liniji 70. Instrukcija aload_1 učitava referencu iz polje 1, koja označava prije inicijalizirano polje, u nastavku se učitava varijabla 3, te na kraju se učitava vrijednost iz polja koja se nalazi na indeksu koji je jednak varijabli 3. Ponovno se tada izvršavaju instrukcije koje učitavaju referencu polja i vrijednost varijable 3, tada na liniji 75. se na stog dodaje broj 1, sljedeća funkcija iadd zbraja dva integera, kako znamo da na stogu postoje varijabla 3 i vrijednost 1, zaključujemo da se varijabla 3 uvećava za jedan, te se dohvaća vrijednost polja koja se nalazi na indeksu u polju novo dobivene vrijednosti dobivene zbrajanjem prethodne dvije. Instrukcija na broju 78, if_icmple provjerava ako je prva varijabla manja od druge i ako jest radi uvjetni skok na liniju 101, gdje se učitava vrijednost varijable 3 i onda se uveća za jedan pomoću instrukcije iinc 2, 1 te se izvršavanje programa vraća na liniju 65.

Sa dobivenim informacijama koje smo do sada stekli, osim inicijaliziranog polja vidimo da se ne poziva nikakva funkcija već se izvršava neki algoritam jer smo identificirali pomoću uvjetnih skokova petlju i dodatne dvije inicijalizirane varijable. Tako bi kod nakon inicijalizacije polja bio sljedeći:

```
Int var1 = 9;
Int var2 = 0;
for(;var1>0;var1-1){
        for(;var2<var1;var2+1){
                if(Field[var2]>Field[var2+1]{
                }
        }
}
```

Sada kada smo dobili izgled petlje možemo zaključiti da se radi o algoritmu sortiranja te da se unutar „if“ uvjeta izvršava jednostavna zamjena mjesta vrijednostima unutar polja, a sa znanjem o vrijednostima varijabli možemo zaključiti da će sortirano polje biti sortirano uzlazno.

### 5.2.4. Usporedba dobivenog zbirnog koda i izvornog koda

Usporednom koda dobivenog na temelju analize i početnog izvornog koda, vidimo opet kako je lako doći do informacija putem obrnutog inženjeringa. Iako u ovome slučaju nemamo točan naziv varijabli kao kod prethodnog primjera, svejedno smo došli do bitnih otkrića unutar izvršne datoteke programa, a to je kako izgleda sam algoritam i koji podaci se koriste u programu. Iako u ovome slučaju nismo došli do toliko informacija kao kod prethodnog primjera postupak je koristan jer smo dobili bitne informacije iz zbirnog koda i približno rekonstruirali izvorni kod.

# 6. Zaključak

U radu bila je obrađena tema obrnutog inženjeringa u razvoju softvera. U početku bio je napravljeni uvod u sami pojam obrnutog inženjeringa te čemu on služi. Bilo to pronalaženje grešaka u kodu ili provjera da program ne sadrži nikakav zlonamjeran kod koji bi naštetio pojedincu ili računalu.

U nastavku rada bili su opisani pristupi obrnutom inženjeringu kao što su to „Live“ analiza koda ili „Offline“ analiza koda (eng. Dead Listing). Kako za razvoj nekog programa tako i za obrnuti inženjering neophodni su pravi alati koji su bili isprobani i ukratko objašnjena njihova moć i namjena. Kako postoje dvije platforme na kojima rade programi tako postoje i dvije vrste alata za obrnuti inženjering, oni koji će biti korisni kod obrnutog inženjeringa programa baziranih na assembly zbirnom jeziku i oni koji će biti korisni kod obrnutog inženjeringa programa baziranih na bytecode zbirnom jeziku. Tako su u radu bili detaljnije objašnjeni programi IDA Pro i OllyDbg koji služe za obrnuti inženjering .exe datoteka programa baziranih na assembly zbirnom jeziku te Javap i JDProject koji služe za obrnuti inženjering Java izvršnih datoteka.

Nakon toga bili su objašnjeni načini kako zaštiti izvršnu datoteku programa od obrnutog inženjeringa kako bi se pojedincu otežali ili onemogućilo. Detaljnije su bila opisana dva načina, enkripcija samog koda izvršne datoteke koja ima svoje nedostatke i lako ju je razbiti te prigušivanje koda kako bi se otežala čitljivost u nadi da pojedinac odustane od procesa obrnutog inženjeringa.

Na kraju rada napravio sam implementaciju mjehurićastog sortiranja polja u C++ i Java programskom jeziku te pomoću alata IDA Pro i Javap izvršio proces obrnutog inženjeringa nad njima. Dobivene zbirne kodove sam analizirao do neke razine liniju po liniju da provjerim koliko informacija mogu izvući iz assembly i bytecode zbirnih jezika te koliko točno mogu dobivene informacije interpretirati da dobijem izvorni kod programa. Po dobivenim rezultatima sam zaključio da se može dobiti skoro jednaki kod izvornome kodu procesom analize zbirnog jezika dobivenog procesom obrnutog inženjeringa.

Tema rada bila mi je zabavna te sam sretan što sam naučio nešto ne svakidašnje u razvoju softvera i nešto što možda svaki programer ne zna. Obradom

teme dobio sam uvid u to koliko je bitno zaštiti program i kako je nekome iskusnome lako doći do informacija iz nečega što se čini kao da nema puno čovjeku čitljivih dijelova. Smatram da je za obrnuti inženjering pojedincu potrebno osnovno znanje zbirnog i programskog jezika, kao i puno volje te želja za rješavanjem zagonetki kako bi se došlo do informacija. Iako je u radu puno toga objašnjeno ima još puno alata za obrnuti inženjering i puno načina zaštite izvršnih datoteka od procesa obrnutog inženjeringa koji u ovome radu nisu zahvaćeni jer bi lako bilo napisati podugu knjigu na ovu temu.

# Popis literature


[1]. E. Eldad, *Reversing: Secrets of Reverse Engineering,* Wiley Publishing, Inc., Indianapolis, IN, USA, 2005.

[2]. „IDA Pro in a nutshell“ [Online],

Available: https://www.hex-rays.com/products/ida.

[Accessed: 1-August-2020].

[3]. „OllyDbg“ [Online],

Available: http://www.ollydbg.de.

[Accessed: 1-August-2020]

[4]. „javap – The Java Class File Disassembler“ [Online],

Available: https://docs.oracle.com/javase/7/docs/technotes/tools/windows/javap.html,

[Accessed: 5-August-2020]

[5]. „Java Decompiler - Yet another fast Java decompiler“ [Online],

Available: http://java-decompiler.github.io,

[Accessed: 5-August-2020]

[6]. „8 Best Java Decompilers in 2020“ [Online],

Available:https://javahungry.blogspot.com/2018/12/8-best-java-decompiler-in-2019.html,

[Accessed: 10-August-2020]

[7]. „Protect your source code from decompiling or reverse engineering“ [Online],

Available: http://arunendapally.com/post/protect-your-source-code-from-decompiling-or-reverse-engineering,

[Accessed: 10-August-2020]

# Popis slika

Slika 1. Sučelje programa IDA Pro [vlastita izrada] ..... 8
Slika 2. Sučelje OllyDbg alata [vlastita izrada] ..... 10
Slika 3. rezultat Javap - Java Class Decompilera [vlastita izrada] ..... 11
Slika 4. rezultat Javap - Java Class Decompilera [vlastita izrada] ..... 11
Slika 5. grafičko sučelje programa JDProject [vlastita izrada] ..... 12
Slika 6. Dijagram toka algoritma mjehurićastog sortiranja [vlastita izrada] ..... 24

# Popis tablica

Tablica 1. Vrijednosti brojeva u polju.....20