SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 6831

IZRADA ALGORITMA ZA IZRAČUN JEDINSTVENOG OTISKA AUTORA NEKOG PROGRAMSKOG KODA

Robert Benić

Zagreb, lipanj 2020.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 6831

IZRADA ALGORITMA ZA IZRAČUN JEDINSTVENOG OTISKA AUTORA NEKOG PROGRAMSKOG KODA

Robert Benić

Zagreb, lipanj 2020.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 13. ožujka 2020.

ZAVRŠNI ZADATAK br. 6831

Pristupnik:	**Robert Benić (0036508373)**
Studij:	Računarstvo
Modul:	Računarska znanost
Mentor:	doc. dr. sc. Stjepan Groš

Zadatak: **Izrada algoritma za izračun jedinstvenog otiska autora nekog programskog koda**

Opis zadatka:

Prilikom analize zloćudnog koda vrlo korisna informacija bila bi o broju osoba koje su radile na kodu te na kojim dijelovima su radili. Na taj način bi se mogla provesti analiza o raspoloživim resursima napadača. Također, informacija koja govori koje sve zloćudne kodove je napisala pojedina osoba, ili sudjelovala u pisanju, bi omogućila povezivanje naizgled nepovezanih aktivnosti izvora prijetnji u jednu cjelinu. Međutim, radi se o dosta složenom problemu jer na raspolaganju nije dostupan izvorni kod, a i binarni kod koji je dostupan je značajno obfusciran. Prema tome, rješenje ovog problema potrebno je provesti u manjim koracima. U sklopu završnog rada potrebno je pozabaviti se sa dva - međusobno povezana - koraka. Prvo, potrebno je na temelju koda kojeg je napisao jedan autor definirati način izračuna otiska (engl. fingerprint). Zatim je potrebno provjeriti koliko je jedinstven otisak za različite autore te može li se otisak upotrijebiti da se identificira autor u zadanom kodu. Radu priložiti izvorni kod razvijenih i korištenih programa. Citirati korištenu literaturu i navesti dobivenu pomoć.

Rok za predaju rada: 12. lipnja 2020.

Srdačno zahvaljujem svom mentoru, doc. dr. sc. Stjepanu Grošu, i zavodskoj suradnici, mag. ing. Ivoni Brajdić, na iskazanoj pomoći i podršci u izradi ovog završnog rada.

Sadržaj

1. Uvod

Zloćudni programi (engl. *malware*) su jedna od osnovnih prijetnji protiv sigurnosti računalnih sustava. Za potrebe zaštite, poželjno je saznati što više informacija moguće o određenom zloćudnom programu, pa tako i o njegovom autoru ili autorima. Postoje mnoge metode za rasuđivanje o autorstvu izvornog kôda pisanog u nekom višem programskom jeziku (Burrows, 2010; Tennyson, 2013). Međutim, kod zloćudnih programa nije dostupan izvorni kôd, nego samo binarni kôd (strojni kôd, kôd virtualnog stroja itd.). Prevođenjem izvornog kôda u binarni kôd se gube mnoge značajke autorovog stila koje su dostupne u izvornom kôdu, kao što su nazivi lokalnih varijabli, komentari, razmaci, stil uvlačenja itd. Stoga nedvojbeno nije moguće dokazivati ili opovrgavati autorstvo za binarni kôd s jednakom sigurnošću kao što bi bilo moguće za izvorni kôd. Ipak, postojeća istraživanja na ovom području su ustanovila da se pojedine značajke stila ne gube prevođenjem u binarni kôd (Rosenblum, 2011), pa je u određenoj mjeri moguće dokazivati ili opovrgavati autorstvo i samo na osnovu binarnog kôda. Za ovaj problem su potrebni drugačiji algoritmi od onih koji se koriste u analizi izvornog kôda, jer se algoritmi za analizu izvornog kôda u velikoj mjeri zasnivaju upravo na onim značajkama koje u binarnom kôdu nisu dostupne (Kalgutkar, 2019).

Cilj ovog završnog rada je izraditi i implementirati algoritam koji bi omogućio programsko razlikovanje autora binarnih kôdova tako da se za svakog autora na osnovu njegovih kôdova konstruira otisak (engl. *fingerprint*). Otisak bi trebao biti takav da za dan programski kôd ili drugi otisak sastavljen na isti način bude moguće na osnovu tog otiska procijeniti vjerojatnost da se radi o istom autoru. Rad je proveden u četiri faze: analiza problema na određenim primjerima i prijedlog rješenja, programska implementacija predloženog rješenja, razvoj ispitnih programa i ispitivanje algoritma te analiza dobivenih rezultata.

Budući da značajke stila programera proizlaze iz elemenata izvornog kôda u višem programskom jeziku, ideja ovog rješenja je procijeniti strukturu izvornog kôda na temelju binarnog kôda pa iz te strukture izdvojiti značajke stila.

Za ovu svrhu bi čitatelj mogao doći na ideju da primijeni unazadno prevođenje (dekompiliranje), pa da se analiza stila radi na tako dobivenom kôdu u višem programskom jeziku. Iako bi u unazadno prevedenom kôdu sve značajke izvornog kôda bile prisutne, unazadni prevoditelj ne može identično rekonstruirati one značajke koje u binarnom kôdu nedostaju, pa se te značajke iznova sastavljaju na temelju konfiguracijskih postavki unazadnog prevoditelja. Prema tome, takve značajke onda neće odražavati autorov stil, nego isključivo karakteristike unazadnog prevoditelja, pa je njih potrebno izuzeti iz analize. Metode koje bi se mogle primijeniti na preostalim značajkama unazadno prevedenog kôda su vrlo slične onima koje se primjenjuju na binarne kôdove, pa je zaključak da unazadno prevođenje nije nužno primjenjivati u analizi autorovog stila. Također, efikasno unazadno prevođenje zahtijeva dobro poznavanje jezika na kojemu je izvorni kôd napisan i/ili prevoditelja kojime je preveden, dok općenito ti podaci nisu u svakom slučaju poznati, a osobito u domeni zloćudnih programa, gdje je autorima u interesu da ostanu anonimni. U ovom završnom radu se želi postići efikasno utvrđivanje stila autora uz manju količinu znanja o izvornom jeziku ili o prevoditelju, pa je u programskom rješenju ovog završnog rada odlučeno da se unazadno prevođenje neće koristiti.

2. Problem razlikovanja autora programa

Viši programski jezici dopuštaju mnogo različitih načina kako postići istovjetnu funkcionalnost u programu, npr. većina programskih jezika ima više od jednog oblika naredbe uvjetnog izvođenja (`if`/`else`, `switch` i sl.) i više od jednog oblika naredbe petlje (`for`, `while`, `do`/`while` i sl.). Različiti programeri imaju različite osobne preferencije prema pojedinim značajkama programskog jezika, pa će prema tome istu ili sličnu funkcionalnost implementirati korištenjem različitih konstrukata u izvornom kôdu, što redovito rezultira različitim konstruktima u binarnom kôdu.

U tablici 2.1. su prikazana dva primjera izvornog kôda u programskom jeziku C koji ostvaruju istu funkcionalnost, a pisana su od različitih autora. Radi kratkoće, ovdje su navedena samo dva kôda za primjer i ti kôdovi su na istom izvornom, odnosno binarnom jeziku, ali zaključci koji se izvode u nastavku vrijede vrlo općenito i mogu se potvrditi na mnogim drugim primjerima, pa i na drugim programskim jezicima.

Tablica 2.1. Dva primjera izvornog koda različitih autora koji obavljaju istu funkcionalnost (HONI, 2016)

```
long long int ls = 0, rs = 0;
int l = 0, r = n - 1, sol = 0;
for(int i = 0; i < n; i++) {
    if(ls < rs) {
        if(ls)
            sol++;
        ls += arr[l++];
    }
    else {
        if(rs)
            sol++;
        rs += arr[r--];
    }
    if(ls == rs)
        ls = rs = 0;
}
if(ls && rs)
    sol++;
```

```
int i = 0, j = n-1, prei = -1, prej = n;
int zbp = 0, zbk = 0;
for (; i <= j; ++i) {
    zbp += A[i];
    while (zbp > zbk && j > i) {
        zbk += A[j--];
    }
    if (zbk == zbp) {
        sol += (i-prei-1)+(prej-j-2);
        prei = i;
        prej = j+1;
        zbp = 0;
        zbk = 0;
    }
}
if (zbk != 0 || zbp != 0) {
    sol += prej-prei-2;
}
```

Prevoditelj gcc 7.5.0 za procesorsku arhitekturu x86-64 prevodi prethodno navedene odsječke izvornog kôda (srednji stupci) u nizove instrukcija u asemblerskom jeziku (desni stupci) u tablicama 2.2. i 2.3.

(Radi preglednosti, na nekim mjestima u izvornim ili u asemblerskim kôdovima su promijenjeni razmaci, uvlake ili proredi, što ni u kojem slučaju ne utječe na binarne kôdove koji se dobiju u konačnoj fazi prevođenja.)

Tablica 2.2. Usporedni prikaz izvornog kôda i prevedenog asemblerskog kôda, 1. primjer

1	`long long int ls = 0, rs = 0;`		`mov QWORD PTR -16[rbp], 0` `mov QWORD PTR -8[rbp], 0`

2	`int l = 0, r = n - 1, sol = 0;`		`mov DWORD PTR -32[rbp], 0`
			`mov eax, DWORD PTR n[rip]`
			`sub eax, 1`
			`mov DWORD PTR -28[rbp], eax`
			`mov DWORD PTR -24[rbp], 0`

3	`for(int i = 0; i < n; i++) {`		`mov DWORD PTR -20[rbp], 0` `jmp .L4`

4	`    if(ls < rs) {`	`.L10:`	`mov rax, QWORD PTR -16[rbp]`
			`cmp rax, QWORD PTR -8[rbp]`
			`jge .L5`

5	`        if(ls)`		`cmp QWORD PTR -16[rbp], 0` `je .L6`
6	`            sol++;`		`add DWORD PTR -24[rbp], 1`

7	`        ls += arr[l++];`	`.L6:`	`mov eax, DWORD PTR -32[rbp]`
			`lea edx, 1[rax]`
			`mov DWORD PTR -32[rbp], edx`
			`cdqe`
			`lea rdx, 0[0+rax*4]`
			`lea rax, arr[rip]`
			`mov eax, DWORD PTR [rdx+rax]`
			`cdqe`
			`add QWORD PTR -16[rbp], rax`
			`jmp .L7`

8	`    }`		
9	`    else {`		
10	`        if(rs)`	`.L5:`	`cmp QWORD PTR -8[rbp], 0` `je .L8`
11	`            sol++;`		`add DWORD PTR -24[rbp], 1`

12	`        rs += arr[r--];`	`.L8:`	`mov eax, DWORD PTR -28[rbp]`
			`lea edx, -1[rax]`
			`mov DWORD PTR -28[rbp], edx`
			`cdqe`
			`lea rdx, 0[0+rax*4]`

			lea	rax, arr[rip]
			mov	eax, DWORD PTR [rdx+rax]
			cdqe	
			add	QWORD PTR -8[rbp], rax
13	}			
14	if(ls == rs)	.L7:	mov	rax, QWORD PTR -16[rbp]
			cmp	rax, QWORD PTR -8[rbp]
			jne	.L9
15	ls = rs = 0;		mov	QWORD PTR -8[rbp], 0
			mov	rax, QWORD PTR -8[rbp]
			mov	QWORD PTR -16[rbp], rax
16	}	.L9:	add	DWORD PTR -20[rbp], 1
		.L4:	mov	eax, DWORD PTR n[rip]
			cmp	DWORD PTR -20[rbp], eax
			jl	.L10
17	if(ls && rs)		cmp	QWORD PTR -16[rbp], 0
			je	.L11
			cmp	QWORD PTR -8[rbp], 0
			je	.L11
18	sol++;		add	DWORD PTR -24[rbp], 1
		.L11:		

Tablica 2.3. Usporedni prikaz izvornog kôda i prevedenog asemblerskog kôda, 2. primjer

1	int i = 0, j = n-1, prei = -1, prej = n;		mov	DWORD PTR -24[rbp], 0
			mov	eax, DWORD PTR n[rip]
			sub	eax, 1
			mov	DWORD PTR -20[rbp], eax
			mov	DWORD PTR -16[rbp], -1
			mov	eax, DWORD PTR n[rip]
			mov	DWORD PTR -12[rbp], eax
2	int zbp = 0, zbk = 0;		mov	DWORD PTR -8[rbp], 0
			mov	DWORD PTR -4[rbp], 0
3	for (; i <= j; ++i) {		jmp	.L4
4	zbp += A[i];	.L9:	mov	eax, DWORD PTR -24[rbp]
			cdqe	
			lea	rdx, 0[0+rax*4]
			lea	rax, A[rip]
			mov	eax, DWORD PTR [rdx+rax]
			add	DWORD PTR -8[rbp], eax
			jmp	.L5
5	while (zbp > zbk && j > i) {			
6	zbk += A[j--];	.L7:	mov	eax, DWORD PTR -20[rbp]
			lea	edx, -1[rax]
			mov	DWORD PTR -20[rbp], edx
			cdqe	
			lea	rdx, 0[0+rax*4]
			lea	rax, A[rip]
			mov	eax, DWORD PTR [rdx+rax]

			add	DWORD PTR -4[rbp], eax
7	}	.L5:	mov	eax, DWORD PTR -8[rbp]
			cmp	eax, DWORD PTR -4[rbp]
			jle	.L6
			mov	eax, DWORD PTR -20[rbp]
			cmp	eax, DWORD PTR -24[rbp]
			jg	.L7
8	if (zbk == zbp) {	.L6:	mov	eax, DWORD PTR -4[rbp]
			cmp	eax, DWORD PTR -8[rbp]
			jne	.L8
9	sol += (i-prei-1) +(prej-j-2);		mov	eax, DWORD PTR -24[rbp]
			sub	eax, DWORD PTR -16[rbp]
			lea	edx, -1[rax]
			mov	eax, DWORD PTR -12[rbp]
			sub	eax, DWORD PTR -20[rbp]
			sub	eax, 2
			add	edx, eax
			mov	eax, DWORD PTR sol[rip]
			add	eax, edx
			mov	DWORD PTR sol[rip], eax
10	prei = i;		mov	eax, DWORD PTR -24[rbp]
			mov	DWORD PTR -16[rbp], eax
11	prej = j+1;		mov	eax, DWORD PTR -20[rbp]
			add	eax, 1
			mov	DWORD PTR -12[rbp], eax
12	zbp = 0;		mov	DWORD PTR -8[rbp], 0
13	zbk = 0;		mov	DWORD PTR -4[rbp], 0
14	}			
15	}	.L8:	add	DWORD PTR -24[rbp], 1
		.L4:	mov	eax, DWORD PTR -24[rbp]
			cmp	eax, DWORD PTR -20[rbp]
			jle	.L9
16	if (zbk != 0 || zbp != 0) {		cmp	DWORD PTR -4[rbp], 0
			jne	.L10
			cmp	DWORD PTR -8[rbp], 0
			je	.L11
17	sol += prej-prei-2;	.L10:	mov	eax, DWORD PTR -12[rbp]
			sub	eax, DWORD PTR -16[rbp]
			lea	edx, -2[rax]
			mov	eax, DWORD PTR sol[rip]
			add	eax, edx
			mov	DWORD PTR sol[rip], eax
18	}			
		.L11:		

Jedna instrukcija u asemblerskom jeziku u pravilu, izuzev makroinstrukcija, koje su izostavljene iz ovih primjera, odgovara jednoj instrukciji u binarnom (strojnom) kôdu. Kao što je vidljivo na prethodnim primjerima, svaka naredba je prevedena s

odsječkom koji sadrži najviše 10 instrukcija u asemblerskom, odnosno u binarnom kôdu, a poredak tih odsječaka približno odgovara poretku naredbi u izvornom programu. Općenito, jedna osnovna operacija (pri čemu naredba može sadržavati više od jedne osnovne operacije) na višem programskom jeziku se tipično prevodi s jednom, dvije ili tri binarne instrukcije. Prema tome, moguće je zaključiti da svaka pojedinačna instrukcija u binarnom kôdu ne daje osobito korisnu informaciju u kontekstu određivanja autora, ali *n*-grami binarnog kôda, tj, kratki odsječci iz niza binarnih instrukcija relativno dobro aproksimiraju konstrukte izvornog kôda, pa daju bolje informacije.

Nadalje, usporedbom dobivena dva asemblerska odsječka je moguće uočiti kako se pojedine odluke programera koji programira u višem programskom jeziku odražavaju u dobivenom binarnom kôdu. Tako je npr. autor prvog odsječka koristio dvije lokalne varijable podatkovnog tipa `long long int`, koji prevoditelj gcc za x86-64 arhitekturu definira kao 64-bitni cjelobrojni tip s predznakom, pa se u binarnom kôdu za pristup tim varijablama koriste 64-bitne varijante instrukcija (kojima odgovaraju instrukcije s argumentima tipa `QWORD PTR` u asemblerskom jeziku). Autor drugog odsječka je na istom mjestu koristio podatkovni tip `int`, koji gcc za x86-64 definira kao 32-bitni cjelobrojni tip s predznakom, pa se naredbe tog odsječka prevode s 32-bitnim instrukcijama (kojima odgovaraju instrukcije s argumentima tipa `DWORD PTR` u asemblerskom jeziku). Nadalje, autori oba odsječka su nazvali varijablu za rješenje zadatka `sol`, ali autor prvog odsječka je deklarirao varijablu `sol` lokalno, unutar istog odsječka (2. redak), dok je autor drugog odsječka deklarirao varijablu `sol` globalno. To se na dobivene binarne kôdove odražava tako što se u prvom odsječku varijabla `sol` nalazi na stogu, pa se za pristup njoj koriste instrukcije s neizravnim adresiranjem (pomoću pokazivača na okvir stoga – registra `rbp` – s pomakom), a u drugom odsječku se nalazi na fiksnoj memorijskoj lokaciji, pa se koriste instrukcije s izravnim adresiranjem (u asemblerskom kôdu je zadržana oznaka `sol`, koja će u posljednoj fazi prevođenja asemblerskog kôda u binarni kôd biti zamijenjena nekom memorijskom adresom).

Općenito nije moguće tvrditi da su preferencije programera binarne, tj. da programer neke konstrukte izvornog jezika neće koristiti ni u kojem slučaju ili da će ih koristiti u svakom programu bez iznimke, ali njegove preferencije se očituju u frekvenciji korištenja. U slučajevima gdje mu je prepuštena odluka, programer će se za određene varijante odlučivati često, a za druge rijetko ili gotovo nikad. Na primjer, u slučaju da programeri rješavaju problem koji zahtijeva računanje s cjelobrojnim vrijednostima većim od $2^{32} - 1$, velika većina programera bi primijenila 64-bitni cjelobrojni tip podataka, ali bi ga neki programeri primjenjivali češće, a neki bi ga nastojali izbjeći. Prema tome, ovisno o problemu koji programer rješava, pojedine značajke ne moraju odražavati samo njegov stil, nego i funkcionalnost koju program obavlja. Dakle, za precizno određivanje značajki stila programera iz programa su bitne i informacije o funkcionalnosti tog programa. Iz ovoga proizlazi da ovdje korištene metode postižu veću preciznost u određivanju otiska ako je za analizu raspoloživa veća količina programa istog programera koji obavljaju različite funkcionalnosti.

Unutar nekog odsječka binarnog kôda, svaka instrukcija se može zapisati kao niz bitova. Unutar takvog niza bitova je moguće razlikovati bitove koji pripadaju operacijskom kôdu instrukcije i bitove koji čine argumente i/ili opcije instrukcije. Na primjer, instrukcija `add eax, edx` u prethodnom primjeru se može zapisati pomoću 16 bitova (2 bajta) u binarnom zapisu `00000001 11010000`, ili u heksadekadskom zapisu `01 D0`. Prvih 6 bitova čine operacijski kôd za zbrajanje (`000000`), sljedeća 4 bita (`0111`) čine opcije koje određuju da se zbrajaju dva 32-bitna registra, sljedeća 3 bita (`010`) predstavljaju adresu drugog operanda (registar `edx`), a posljednja 3 bita (`000`) predstavljaju adresu prvog operanda, na koju se ujedno sprema i rezultat (registar `eax`).

U kontekstu pridruživanja binarnih instrukcija konstruktima izvornog jezika, operacijski kôd svake pojedine instrukcije čini bitnu informaciju jer instrukcije različitih operacijskih kôdova obavljaju različitu funkcionalnost, pa će se pojedini konstrukt u izvornom kôdu u pravilu prevoditi s istim ili sličnim slijedom operacijskih kôdova instrukcija u binarnom kôdu. S druge strane, argumenti

instrukcija su najčešće adrese registara, memorijske adrese ili numeričke konstante, koje programer u višem programskom jeziku uglavnom ne piše, nego se programer služi identifikatorima (varijablama ili imenovanim konstantama), a pridruživanje tih identifikatora adresama ili konstantama je u nadležnosti prevoditelja. Stoga će iste ili slične naredbe u izvornom jeziku biti prevedene u iste ili slične operacijske kôdove instrukcija, dok će se argumenti razlikovati. Tako se u prethodna dva primjera na više mjesta pojavljuju odsječci sljedećeg oblika:

```
oznaka1:  mov registar, lokacija1
          cmp registar, lokacija2
          jne oznaka2
```

Ovo je uobičajeni slijed instrukcija u koji se prevodi naredba uvjetnog izvođenja s uvjetom na jednakost varijabli, tipa `if(a == b) { ... }`: u neki registar se učita vrijednost varijable `a` iz memorije, vrijednost koja se onda nalazi u istom registru se potom usporedi s vrijednošću varijable `b` iz memorije pa ako one nisu jednake, izvede se uvjetni skok na oznaku koja se nalazi nakon bloka naredbi na koji se odnosi naredba `if`, čime je preskočeno izvođenje tog bloka u slučaju da uvjet nije zadovoljen. U ovom odsječku je semantički bitno da se dva puta adresira isti registar, a različite memorijske lokacije: kad bi se adresirali različiti registri, s vrijednošću varijable `b` se ne bi uspoređivala upravo učitana vrijednost varijable `a`, nego neka ranije prisutna vrijednost u registru, a kad bi se adresirale jednake memorijske lokacije, jedna varijabla bi se uspoređivala sama sa sobom umjesto s drugom varijablom.

Međutim, pojedine posebne vrijednosti argumenata mogu činiti značajnu informaciju za analizu autorstva. Na primjer, instrukcije s operacijskim kôdom `add` (zbrajanje) se pojavljuju u odsječcima koji odgovaraju operatorima `+`, `+=` ili `++` jezika C u prethodna dva primjera. Međutim, moguće je primijetiti da se takve instrukcije pojavljuju u nekoliko različitih oblika:

```
add lokacija, 1          (1)
add lokacija, registar   (2)
add registar, lokacija   (3)
```

```
add registar1, registar2
```
(4)

Samo u obliku (1) se u asemblerskom kôdu instrukcije pojavljuje konstantna vrijednost 1. Čitatelj može uočiti da se oblik (1) pojavljuje na onim mjestima u asemblerskom, odnosno u binarnom kôdu na kojima se u izvornom kôdu pojavljuje operator ++, a oblici (2), (3) ili (4) se pojavljuju na mjestima gdje se u izvornom kôdu pojavljuju operatori + ili +=. Prema tome, u kontekstu elemenata izvornog jezika, vrijednost argumenta 1 u instrukciji za zbrajanje ima posebno značenje, jer ta vrijednost odgovara različitoj operaciji (++, inkrement) od operacija s općenitim vrijednostima argumenata (+ ili +=, zbrajanje u općem slučaju).

Također, za pojedini slijed instrukcija je bitna informacija ponavlja li se ista vrijednost argumenta na više mjesta, jer to znači da taj slijed instrukcija odgovara naredbi koja izvodi više slijednih operacija nad istim podatkom, umjesto nad različitim podacima. Na primjer, u prethodnim kôdovima se pojavljuju odsječci sljedećeg oblika:

```
cmp lokacija1, 0
je oznaka
cmp lokacija2, 0
je oznaka
```

U ovom obliku programskog odsječka je bitna informacija da obje instrukcije uvjetnog skoka (`je`) imaju istu odredišnu lokaciju, jer to znači da se radi o uvjetnom izvođenju sa složenim uvjetom, odnosno o naredbi oblika `if(a && b) { ... }` u jeziku C. Ako bi oznake u naredbama uvjetnog skoka bile različite, onda bi se u kontrolnom toku programa vršilo grananje na tri različite grane, što bi u jeziku C odgovaralo npr. nizu naredbi `if(a) { ... } else if(b) { ... }` ili naredbi `switch(...) { ... }` s više različitih `case` blokova.

Nadalje, uz određene informacije o programu prevoditelju kojim je izvorni kôd preveden u promatrani binarni kôd, moguće je izgraditi rječnik odsječaka u binarnom kôdu koji odgovaraju elementima izvornog jezika. Binarni programi mogu sadržavati metapodatke o alatima koji su korišteni u postupcima prevođenja i povezivanja, a u mnogim slučajevima se za tu svrhu koriste isti, široko dostupni

(komercijalni ili slobodni) alati, pa je analizom tih alata moguće izgraditi vrlo kvalitetan rječnik. U drugim slučajevima nije poznato mnogo podataka o korištenom prevoditelju (npr. ako su za tu svrhu korištena interna rješenja neke tvrtke), ali mogu biti dostupne velike količine programskog kôda koji je preveden pomoću istog prevoditelja ili drugog prevoditelja sličnih karakteristika, na temelju čega je također moguće konstruirati rječnik dobre kvalitete. Tako se u oba prethodna primjera može uočiti da se kod petlje oblika `for(int i = 0; i < n; i++) { ... }` u jeziku C prevedene prevoditeljem gcc na kraju bloka pojavljuje karakterističan odsječak oblika:

```
                add lokacija1, 1
oznaka1:        mov registar, lokacija1
oznaka2:        cmp registar, lokacija2
                jl oznaka3
```

Korisnik koji raspolaže ovakvim znanjem o prevoditelju koji programer koristi može sastaviti rječnik odsječaka takvog tipa, što omogućuje točniju programsku analizu od uspoređivanja velikog broja *n*-grama (odsječaka), metodom „grube sile”.

Neki drugi znanstveni radovi iz područja analize autorstva binarnog kôda su koristili i drugačije metode kod analize osim ovdje navedenih, ali ustanovljeno je da najbolje rezultate daju navedene ili njima slične metode (Rosenblum, 2011; Alrabaee, 2014; Caliskan-Islam, 2015), pa su za ovaj završni rad odabrane upravo iste metode.

3. Programska implementacija izračuna otiska

Programsko rješenje u sklopu ovog završnog rada je zamišljeno tako da se, uz odgovarajuće ulazne postavke, može primijeniti na binarne kôdove u širokom rasponu jezika, što uključuje strojne jezike različitih arhitektura procesora, jezike virtualnih strojeva i dr.

Programsko rješenje je izvedeno u dvije varijante. Prva varijanta koristi metodu analize *n*-grama, a druga varijanta koristi metodu rječnika.

Obje varijante programa mogu raditi s proizvoljno mnogo ulaznih datoteka koje sadrže odsječke binarnog kôda. Pritom se svaka ulazna datoteka obrađuje kao zaseban program, tj. kôdovi iz ulaznih datoteka se ne spajaju u jedan, već se računa zaseban otisak na temelju svakog kôda pa se dobiveni otisci spajaju u jedan kumulativni otisak programera.

Također, obje varijante čitaju iz posebne datoteke formate instrukcija u konkretnom binarnom jeziku koji se koristi. Instrukcije mogu biti proizvoljnih duljina i rasporeda bitova, što omogućuje primjenu i na binarnim jezicima sa složenim skupom instrukcija kao što je npr. strojni jezik arhitekture Intel x86.

Elementi na temelju kojih se sastavlja otisak programera su u obje varijante odsječci binarnog kôda koji se sastoje od određenog broja uzastopnih instrukcija. Za potrebe programske analize odsječaka, definira se sljedeći kriterij usporedbe: dva odsječka se smatraju jednakima ako i samo ako su jednakih duljina, svaki par pripadajućih instrukcija ima jednake operacijske kôdove te za svaki par argumenata u prvom odsječku vrijedi da su oni jednaki ako i samo ako su pripadajući argumenti u drugom odsječku također jednaki. Ovako definiran kriterij omogućuje razlikovanje samo onih značajki koje je odabrao programer – koje se operacije izvode i nad kojim podacima, a ne onih koje određuje program prevoditelj – na kojim mjestima u memoriji se nalaze ti podaci.

Prva varijanta radi s minimalnom količinom znanja o izvornom jeziku u kojemu su pisani ili o prevoditelju kojim su prevedeni promatrani kôdovi. U toj varijanti se uzastopne instrukcije grupiraju u *n*-grame (odsječke) za svaku vrijednost *n* (broj instrukcija) između 2 i 10, a koji se pritom smiju preklapati. Nakon što su prikupljeni podaci o *n*-gramima, za svaki *n*-gram se računa frekvencija pojavljivanja jednakih *n*-grama prema prethodno navedenom kriteriju između svih *n*-grama za istu vrijednost *n*. Otisak dobiven na temelju jednog binarnog programskog odsječka čini unija ovako sastavljenih skupova podataka za svaku vrijednost *n*.

Druga varijanta zahtijeva i koristi dostupno znanje o karakteristikama izvornog jezika, odnosno načina na koji se izvorni kôd prevodi u binarni kôd. Tom znanju odgovara rječnik poznatih odsječaka binarnog kôda kakvi se dobivaju prevođenjem izvornog kôda. U ovoj varijanti se iz programa izdvajaju odsječci koji odgovaraju rječniku, na način da se nijedna dva odsječka ne preklapaju, a instrukcija koje se ne mogu grupirati ni u jedan odgovarajući odsječak se pretpostavlja da neće biti mnogo pa se takve odbacuju. Za svaki učitani odsječak se ponovo računa frekvencija pojavljivanja prema navedenom kriteriju usporedbe.

U obje varijante, nakon što su na ovaj način prikupljeni podaci za svaki ulazni program, isti podaci se akumuliraju tako da se za svaki odsječak izračunaju srednja vrijednost i varijanca frekvencije pojavljivanja u svim programima:

$$\bar{f} = \frac{f_1 + f_2 + \ldots + f_n}{n}$$

$$v^2 = \frac{(f_1 - \bar{f})^2 + (f_2 - \bar{f})^2 + \ldots + (f_n - \bar{f})^2}{n - 1}$$

Kako bi se smanjio utjecaj značajki vezanih uz programsku funkcionalnost i dr., kao značajni za određivanje stila autora se uzimaju samo oni elementi (odsječci) za koje je varijanca frekvencije manja ili jednaka određenoj vrijednosti praga. Na ovaj način se može očekivati veća preciznost algoritma za veći skup ulaznih programa. Naime, na velikom skupu programa će se oni odsječci koji su značajni

za autorov stil pojavljivati sa sličnim frekvencijama u većini ulaznih podataka pa će njihove varijance frekvencija biti manje. Kod onih odsječaka čije korištenje nije značajka autorovog stila, frekvencije će se pak znatno razlikovati od programa do programa pa će varijance tih frekvencija na većem uzorku biti veće.

Nakon što je algoritam razvijen, potrebno je pripremiti binarne programe na kojima će se on izvoditi. Za tu svrhu je implementirano izdvajanje binarnog kôda iz izvršnih ili objektnih datoteka u nekoliko različitih formata. Detalji ove implementacije izlaze izvan okvira ovog završnog rada.

Konačno, u fazi ispitivanja je prepoznato nekoliko problema u implementaciji opisanog algoritma pa je implementacija dopunjena dodatnim značajkama koje rješavaju spomenute probleme.

Prvi problem je u tomu što su srednje vrijednosti i varijance frekvencija zavisne o konkretnom binarnom jeziku na koji je program preveden. Zbog toga je kao alternativa konstantnoj maksimalnoj varijanci uvedena i mogućnost ograničenja broja elemenata koji čine otisak, pri čemu se u slučaju viška elemenata biraju oni s najmanjim varijancama frekvencija.

Drugi problem leži u činjenici da prevoditelj na pojedina mjesta u programu dodaje elemente koji ne čine značajke autorovog stila i nisu korisni za analizu autorstva. Neki takvi elementi se pojavljuju na velikom broju mjesta u programu (npr. na početku ili na kraju svake funkcije), a drugi se pojavljuju samo jednom ili još rjeđe. Kao rješenje ovom problemu su uvedene mogućnosti minimalnog i maksimalnog ograničenja na srednje vrijednosti frekvencija elemenata koji budu uvršteni u otisak, tako da korisnik može ograničiti frekvencije elemenata na neke vrijednosti koje bolje odgovaraju frekvencijama dijelova izvornog koda koje programer piše.

4. Ispitivanje podudarnosti otisaka autora

Nakon što je algoritam za računanje otiska izrađen i implementiran, potrebno je ispitati njegovu točnost. Cilj ovog rada je da algoritam bude takav da se otisci izračunati na temelju kôdova istog autora međusobno razlikuju što manje, a otisci izračunati na temelju kôdova dvoje različitih autora što više. U tu svrhu, napisan je program za uspoređivanje dva otiska dobivena kao rezultat provedbe algoritma.

Ispitivanje se vrši na dva načina: (1) usporedbom dva otiska koji su izvedeni iz različitih skupova programa ili (2) usporedbom jednog otiska s jednim izabranim programom.

Kod oba načina se razlikuju slučajevi da otisci, odnosno otisak i program, pripadaju istom programeru ili da pripadaju različitim programerima. Cilj algoritma je da u slučaju istog programera, podudarnost bude što veća, a u slučaju različitih programera, podudarnost bude što manja. Za ovu svrhu je potrebno definirati mjere podudarnosti kod oba načina ispitivanja.

U prvom načinu ispitivanja, razlikujemo elemente (elemente rječnika ili *n*-grame) koji se pojavljuju u oba ili samo u jednom od ispitivanih otisaka. Za sve elemente koji se pojavljuju u oba otiska, što je manja razlika u frekvencijama pojavljivanja elementa u prvom i u drugom otisku, to je veća vjerojatnost da se radi o istom programeru. Za element koji se pojavljuje samo u jednomu od dva otiska, postoje dvije mogućnosti zašto se ne pojavljuje u drugom otisku: (1) element se ne pojavljuje ni u jednom programu od onih prema kojima je sastavljen drugi otisak, tj. programer kojemu odgovara drugi otisak ga ne koristi, ili (2) element se pojavljuje, ali ga je algoritam odbacio kao stilski neznačajan element, jer se pojavljuje s velikom varijancom frekvencije. Međutim, čest je slučaj da programer pojedinu značajku u višem programskom jeziku, pa tako i odgovarajući element u binarnom kôdu, nastoji primjenjivati što manje. Stoga je moguće da se na nekom skupu programa jednog programera neki element ne pojavljuje uopće, a na nekom drugom skupu programa istog programera se pojavljuje rijetko, jer je to uvjetovano

funkcionalnošću programa. Na primjer, velika većina programera u jeziku C izbjegava korištenje naredbe `goto`, ali ona je često najbolje rješenje ako je potrebno izaći iz više ugniježđenih petlji ili `switch` naredbi. S druge strane, malo je vjerojatno da bi isti programer na nekom drugom programu upotrebljavao isti element vrlo često, pa u slučaju (1), što je veća frekvencija tog elementa ondje gdje se pojavljuje, to je vjerojatnije da su u pitanju programi različitih autora.

U slučaju (2) ova tvrdnja ne vrijedi, ali čitatelj može uočiti da je za velike skupove programa manja vjerojatnost da isti element na jednom skupu programa bude stilski značajan, a na drugomu ne bude stilski značajan, neovisno o programeru, jer svaki programer ima jasne preferencije prema elementima izvornog kôda gdje god mu je omogućen izbor. Prema tome, za dovoljno veliku količinu programskog kôda različitih funkcionalnosti će vjerojatnost slučaja (2) biti zanemariva, pa ispitni programi u sklopu ovog završnog rada zanemaruju taj slučaj.

Valja napomenuti da su prethodni zaključci izvedeni uz pretpostavku da se funkcionalnosti promatranih programa istog autora dovoljno razlikuju. Intuitivno je jasno da ukoliko se analiza zasniva na nekoliko vrlo sličnih programa, dobiveni rezultati neće dobro odražavati stil programera, jer će biti uvjetovani funkcionalnošću istih programa. Eliminacija funkcionalnih značajki iz analize je problem koji je detaljnije istražen u nekim prethodnim znanstvenim radovima iz područja analize autorstva (Rosenblum, 2011), no izlazi iz okvira ovog završnog rada.

Dakle, za prvi način ispitivanja valja pronaći mjeru podudarnosti takvu da je podudarnost veća što je manja razlika u frekvencijama nekog elementa, pri čemu se za element koji nedostaje uzima 0. Ovdje se kao prikladna mjera pokazuje korijen zbroja kvadratnih pogrešaka po frekvencijama. Ako bi pritom svakom elementu bila pridružena jednaka težina u zbroju kvadratnih pogrešaka, onda bi ukupna težina pridružena zajedničkim elementima u oba otiska bila neproporcionalno manja od ukupne težine ostalih elemenata, jer postoji samo jedan skup zajedničkih elemenata u oba otiska, dok se ostali elementi dijele na

dva skupa ovisno o tome u kojem otisku se pojavljuju. Zbog toga je bolje zajedničkim elementima pridružiti dvostruko veću težinu nego ostalima, jer je na taj način omjer težina zajedničkih elemenata i ostalih elemenata u zbroju kvadratnih pogrešaka jednak srednjoj vrijednosti njihovih omjera u oba otiska.

Ovako konstruirana mjera poprima uvijek nenegativne vrijednosti, tj. najmanja moguća vrijednost mjere podudarnosti je 0, pri čemu manja vrijednost znači veću podudarnost. Za potrebe izračuna, korisno je ograničiti i maksimalnu vrijednost mjere podudarnosti, pa se stoga dobivena vrijednost dijeli korijenom zbroja kvadrata frekvencija svih elemenata u oba otiska. Ovako se dobiva mjera čije su vrijednosti nužno unutar intervala $[0, 1]$, budući da za bilo koje dvije vrijednosti frekvencija f_1 i f_2 vrijedi nejednakost:
$(f_1 - f_2)^2 = f_1^2 - 2f_1f_2 + f_2^2 \leq f_1^2 + f_2^2$

Slična ideja se primjenjuje na drugi način ispitivanja. Ovdje je situacija jednostavnija, jer je za svaki element koji se ne pojavljuje u programu njegova frekvencija očito 0. Za elemente koji se ne pojavljuju u otisku, a pojavljuju se u programu, prema prethodno navedenoj argumentaciji, može se pretpostaviti da nisu stilski značajni, pa se ne trebaju razmatrati. Za sve elemente koji se pojavljuju u otisku je vjerojatnije da je program pisao isti programer ukoliko se pojavljuju sa sličnim frekvencijama. Stoga će se ovdje kao mjera podudarnosti koristiti korijen zbroja kvadratnih pogrešaka podijeljen korijenom zbroja kvadrata frekvencija po onim elementima koji se pojavljuju u otisku, pri čemu su svim elementima pridružene jednake težine u zbroju kvadratnih pogrešaka i ako se element ne pojavljuje u programu, onda se za njegovu frekvenciju u programu uzima 0.

5. Analiza rezultata algoritma izračuna otiska

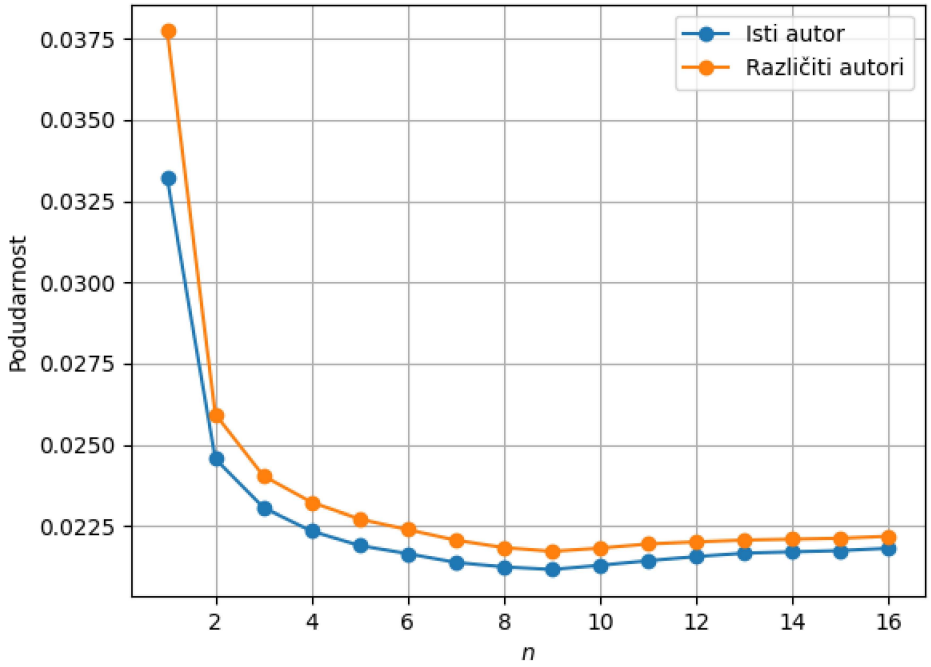
Programsko rješenje razvijeno u sklopu ovog završnog rada je ispitano na skupu od 1000 programa u jezicima C ili C++ preuzetih s natjecanja u programiranju Google Code Jam. Programi su preuzeti u obliku izvornog kôda i razvrstani po korisničkim imenima autora. Izdvojeni su oni autori koji su napisali više od 20 programa svaki, između njih je slučajno odabrano 50 autora i za svakoga od tih 50 autora je slučajno odabrano 20 programa.

Izabrani programi su podijeljeni na dvije slučajne particije od 10 programa svakog autora, kako bi se moglo usporediti otiske sastavljene na osnovu dva različita skupa programa istog autora. Nadalje, budući da bi uspoređivanje otiska sa skupom programa ili s drugim otiskom za svaki par različitih autora bilo previše dugotrajno, za svakoga od 50 ispitanih autora je slučajno odabrano 5 drugih autora s kojima se uspoređuje. Otisak dobiven na temelju programa nekog autora iz prve particije se uspoređuje s programima pripadnih 5 autora iz druge particije, odnosno s otiscima dobivenim na temelju njih.

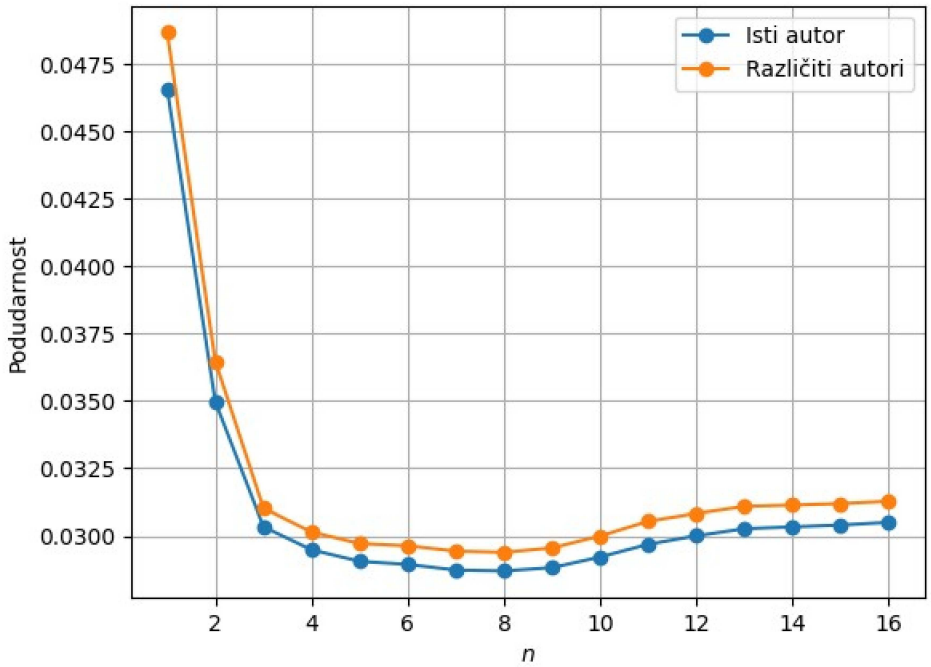
Za pripremu programa u binarnom obliku su korišteni prevoditelj `gcc`/`g++` 7.5.0 za operacijski sustav Linux na procesorskoj arhitekturi ARM i programi za izdvajanje instrukcija iz izvršnih datoteka u sklopu programske podrške ovog završnog rada. U posljednjoj fazi ispitivanja, gdje su ispitani učinci različitih prevoditelja i operacijskih sustava na algoritam, dodatno su korišteni prevoditelji `gcc`/`g++` 7.5.0 i `clang` 10.0.0 za operacijske sustave Linux i Windows na procesorskoj arhitekturi Intel x86.

U prvoj fazi je ispitana metoda izračuna otiska pomoću *n*-grama za različite vrijednosti *n*. Kako bi se skratilo vrijeme izvođenja, broj elemenata *m* u svakom otisku je ograničen na 2000. Grafovi 5.1. i 5.2. prikazuju srednje vrijednosti mjere podudarnosti parova otisaka, odnosno otisaka s programima, za sve ispitane kombinacije istog autora i dvoje različitih autora, u ovisnosti o vrijednosti *n*.

Graf 5.1. Srednje vrijednosti podudarnosti dva otiska u ovisnosti o n, $m = 2000$



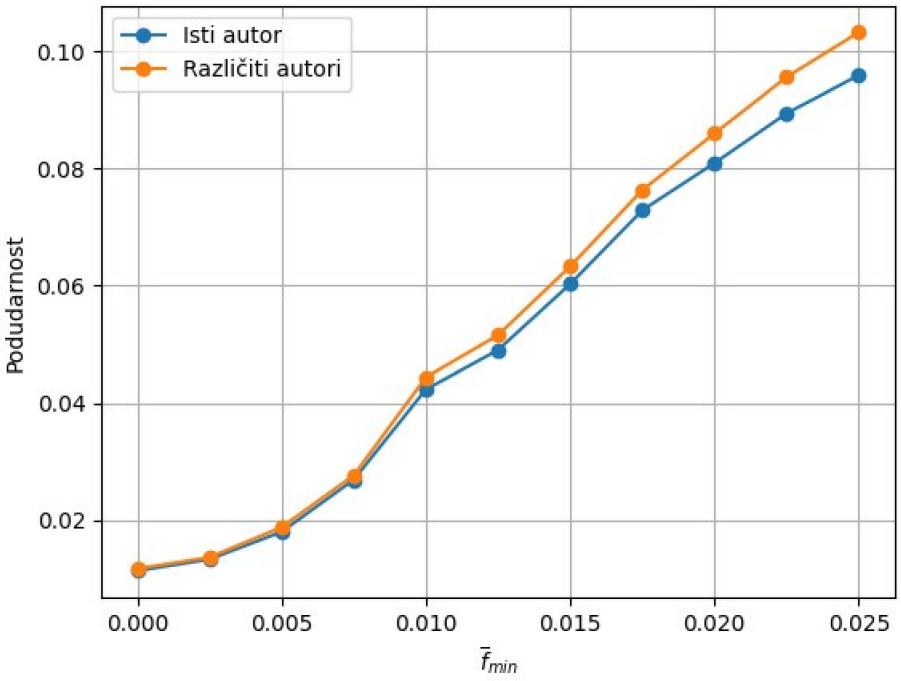
Graf 5.2. Srednje vrijednosti podudarnosti otiska i programa u ovisnosti o n, $m = 2000$



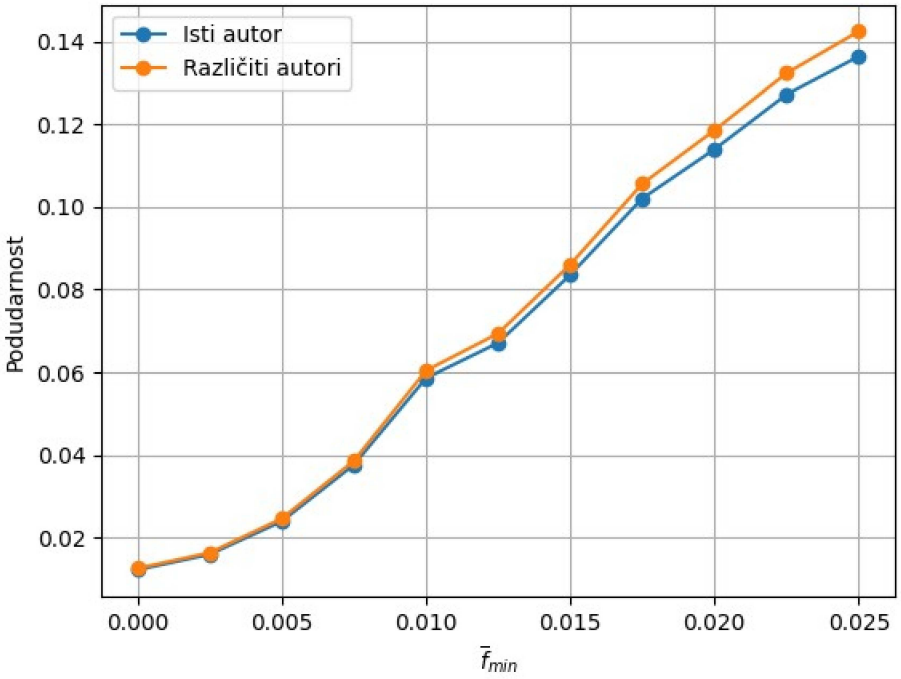
Vidljivo je da se s povećanjem vrijednosti *n* razlike između srednjih podudarnosti za istog, odnosno za različite autore smanjuju, pa ovi rezultati potvrđuju prethodni zaključak da su za stil programera karakteristični relativno kratki programski odsječci. U daljnjim ispitivanjima metode *n*-grama su korištene sve vrijednosti *n* manje ili jednake 10.

U drugoj fazi su uvedena i ispitana ograničenja na frekvencije elemenata. Grafovi 5.3. i 5.4. prikazuju srednje vrijednosti mjere podudarnosti za kombinacije istog autora i različitih autora u ovisnosti o minimalnom ograničenju $\bar{f}_{min}$ na srednju vrijednost frekvencije. Grafovi 5.5. i 5.6. prikazuju iste metrike u ovisnosti o maksimalnom ograničenju $\bar{f}_{max}$ na srednju vrijednost frekvencije, a grafovi 5.7. i 5.8. u ovisnosti o maksimalnom ograničenju v^2_{max} na varijancu frekvencije.

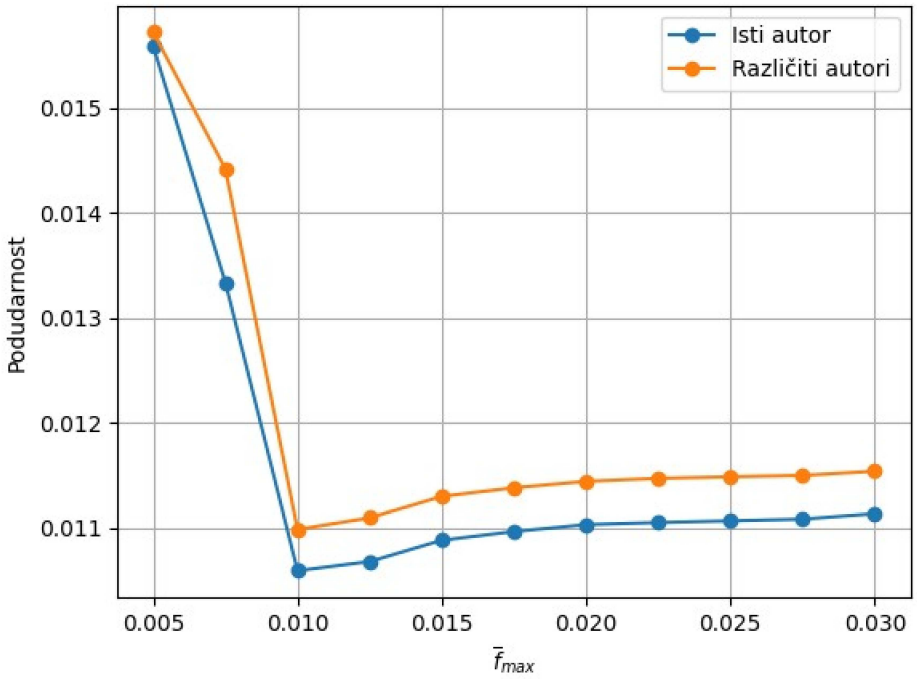
Graf 5.3. Srednje vrijednosti podudarnosti dva otiska u ovisnosti o $\bar{f}_{min}$, $1 \leq n \leq 10$, $m = 2000$



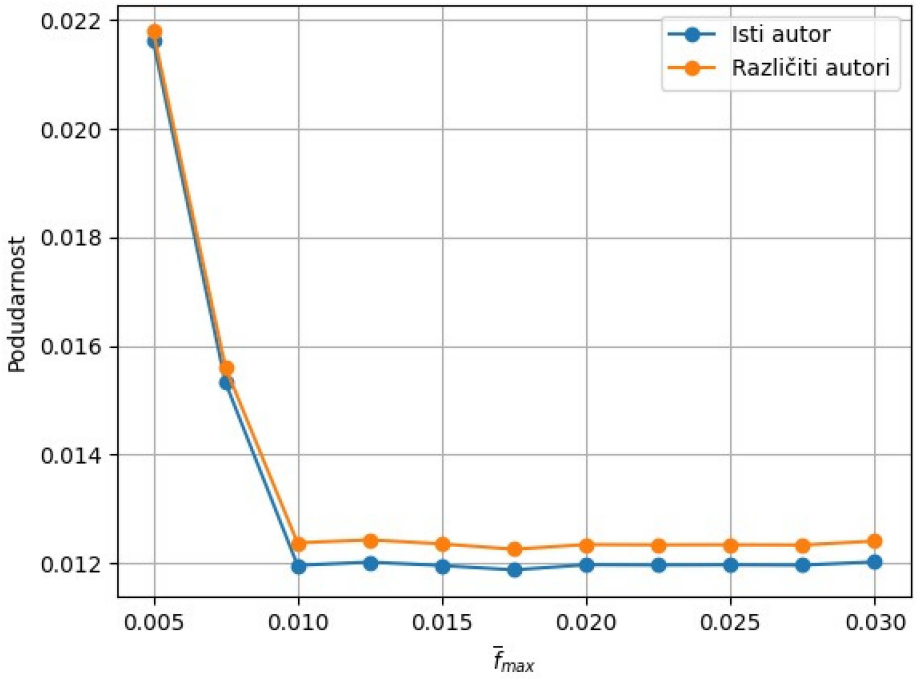
Graf 5.4. Srednje vrijednosti podudarnosti otiska i programa u ovisnosti o $\bar{f}_{min}$, $1 \leq n \leq 10$, $m = 2000$



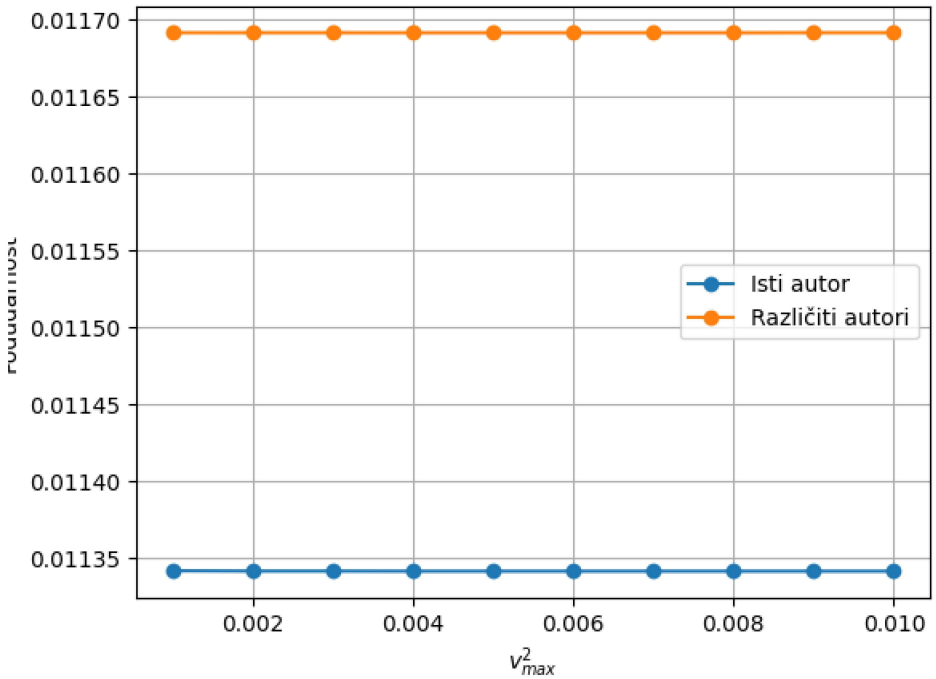
Graf 5.5. Srednje vrijednosti podudarnosti dva otiska u ovisnosti o $\bar{f}_{max}$, $1 \leq n \leq 10$, $m = 2000$



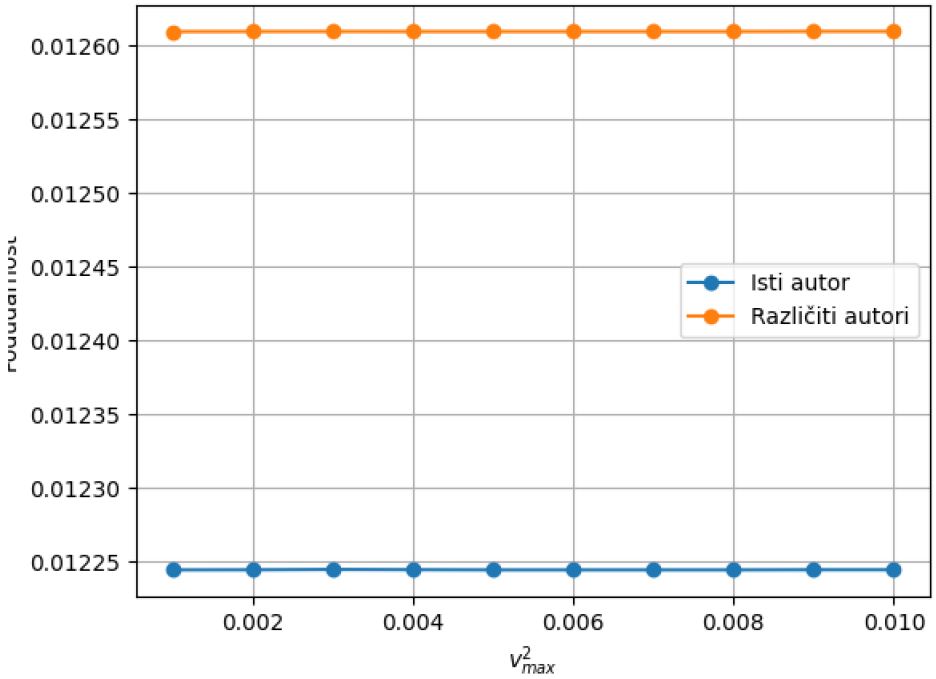
Graf 5.6. Srednje vrijednosti podudarnosti otiska i programa u ovisnosti o $\bar{f}_{max}$, $1 \leq n \leq 10$, $m = 2000$



Graf 5.7. Srednje vrijednosti podudarnosti dva otiska u ovisnosti o v^2_{max}, $1 \leq n \leq 10$, $m = 2000$



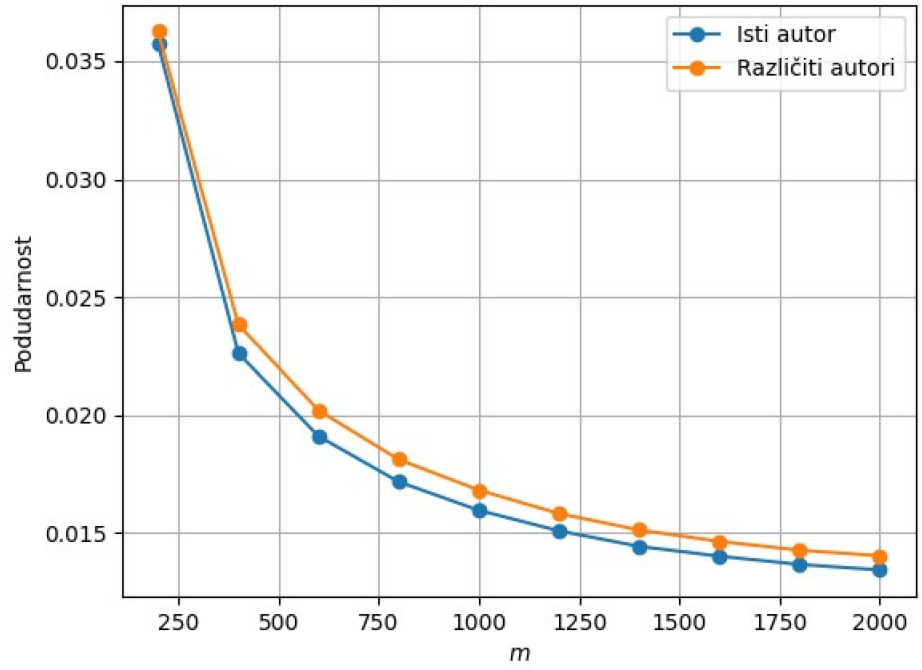
Graf 5.8. Srednje vrijednosti podudarnosti otiska i programa u ovisnosti o v^2_{max}, $1 \leq n \leq 10$, $m = 2000$



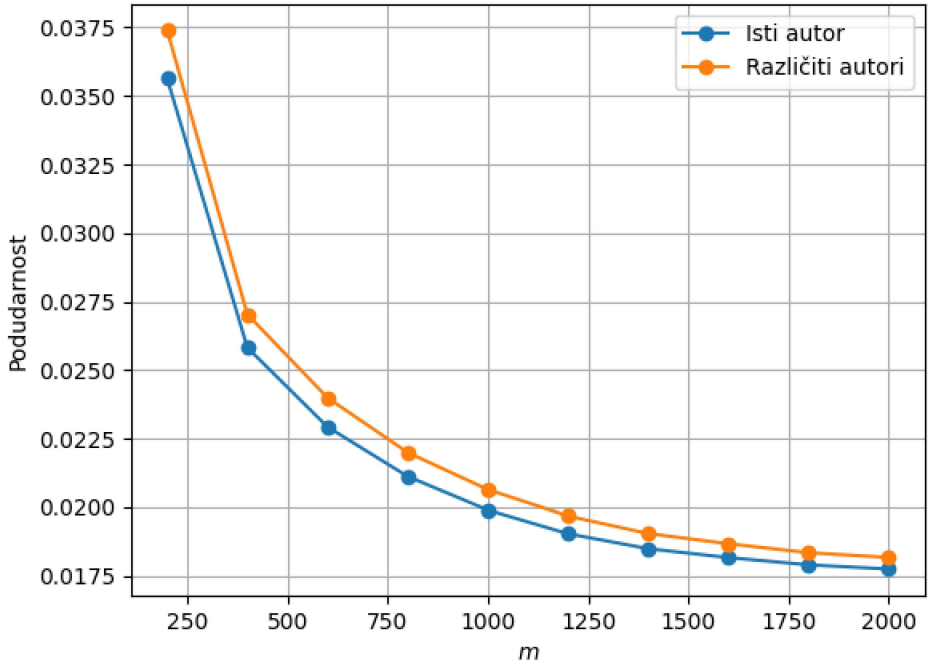
Kao što je vidljivo iz grafova, odgovarajuće minimalne i maksimalne vrijednosti frekvencija čine da budu izabrani stilski značajniji elementi, pa poboljšavaju točnost raspoznavanja autora, dok maksimalna varijanca nema bitnog utjecaja. Za nastavak ispitivanja su odabrane vrijednosti: $\bar{f}_{min} = 0{,}0025$, $\bar{f}_{max} = 0{,}01$, $v^2_{max} = 0{,}005$.

U trećoj fazi ispitivanja je ispitan utjecaj broja elemenata *m* otiska na točnost metode *n*-grama. Grafovi 5.9. i 5.10. prikazuju srednje vrijednosti mjere podudarnosti za kombinacije istog autora i različitih autora u ovisnosti o broju elemenata *m* uvrštenih u otisak.

Graf 5.9. Srednje vrijednosti podudarnosti dva otiska u ovisnosti o *m*, $1 \leq n \leq 10, \bar{f}_{min} = 0{,}0025, \bar{f}_{max} = 0{,}01, v^2_{max} = 0{,}005$



Graf 5.10. Srednje vrijednosti podudarnosti otiska i programa u ovisnosti o *m*, $1 \leq n \leq 10, \bar{f}_{min} = 0{,}0025, \bar{f}_{max} = 0{,}01, v^2_{max} = 0{,}005$



Vidljivo je da se mjere podudarnosti smanjuju s brojem elemenata u otisku, ali razlike između podudarnosti za istog ili za različite autore su približno jednake, pa je smanjenje broja elemenata korisno budući da skraćuje vrijeme izvođenja, a ne utječe bitno na rezultat.

U sljedećoj fazi je ispitana metoda rječnika, međutim nije bilo moguće provesti detaljno ispitivanje i prikaz rezultata zbog približno 100 puta duljeg vremena izvođenja programa. Na znatno manjem broju primjera su dobiveni rezultati, koji su u istom redu veličine kao i rezultati metode *n*-grama.

U posljednjoj fazi je ispitan utjecaj kombinacije procesorske arhitekture, operacijskog sustava i prevoditelja na algoritam. U tu svrhu je algoritam osim na arhitekturi računala ARM, pokrenut i na arhitekturi računala Intel x86, s operacijskim sustavima Linux i Windows. Na arhitekturi Intel x86 se ispitni postupak također izvodi u mnogostruko duljem vremenu nego na arhitekturi ARM. Razlog tomu leži u činjenici da se Intel x86 ubraja u porodicu računala sa složenim instrukcijskim skupom (*Complex Instruction Set Computer*, CISC), dok je ARM računalo sa smanjenim instrukcijskim skupom (*Reduced Instruction Set Computer*, RISC), pa je instrukcijski opis za arhitekturu Intel x86 bitno složeniji nego za ARM. Rezultati dobiveni na arhitekturi Intel x86 na manjem broju primjera su također usporedivi s onima za ARM.

Iz navedenoga je moguće zaključiti da su, uz određena poboljšanja, metode opisane i implementirane u sklopu ovog rada primjenjive općenito, neovisno o značajkama konkretnog sustava na kojemu, odnosno za koji su programi napisani.

6. Zaključak

Rezultati ovog rada potvrđuju da je problem raspoznavanja autora u binarnom programskom kôdu rješiv s određenom sigurnošću. Međutim, taj problem još nije riješen s dovoljnom sigurnošću da bi se na osnovu dobivenih rezultata moglo odgovorno tvrditi da netko jest ili nije sudjelovao u razvoju nekog zloćudnog programa, što je jedan od glavnih ciljeva koji se žele postići rješavanjem tog problema. K tomu, za postojeća rješenja ovog problema su potrebne velike količine podataka i velika procesorska snaga, veće negoli su dostupne prosječnom zainteresiranom korisniku, pa je njihova primjena još uvijek ograničena na eksperimentalne svrhe.

Osim prikupljanja dovoljne količine podataka o programu koji se proučava, važno bi bilo i odrediti kolika je kvaliteta podataka koji su dostupni u određenom slučaju. Štoviše, u kontekstu zloćudnih računalnih programa, autorima je u interesu da ostanu anonimni, pa bi mogli primjenjivati i postupke koji imaju cilj da otežaju ispravnu identifikaciju autora. Mogućnosti takvih postupaka čine dodatan problem koji je potrebno istražiti.

Može se očekivati poboljšanje rezultata pomoću naprednijih metoda, kao što su strojno učenje ili primjena posebnih zaključaka za određeni jezik, tj. arhitekturu računala ili virtualnog stroja, no i dalje je upitna rješivost ovog problema sa sigurnošću kolika bi bila poželjna u računalnoj forenzici.

7. Literatura

- Burrows S., *Source Code Authorship Attribution*, doktorski rad, RMIT University, 2010.
- Rosenblum N., Zhu X., Miller B. P., *Who Wrote This Code? Identifying the Authors of Program Binaries*, *European Symposium on Research in Computer Security 2011*, Leuven, Belgija, 2011., str. 172-189
- Tennyson M. F., *A Replicated Comparative Study of Source Code Authorship Attribution*, *3rd International Workshop on Replication in Empirical Software Engineering Research*, Baltimore, Maryland, SAD, 2013., str. 77-83
- Alrabaee S., Saleem N., Preda S., Wang L., Debbabi M., *Oba2: An Onion Approach to Binary Code Authorship Attribution*, *Digital Investigation* 11, 2014., str. S94-S103
- Caliskan-Islam A., Yamaguchi F., Dauber E., Harang R., Rieck K., Greenstadt R., Narayanan A., *When Coding Style Survives Compilation: De-Anonymizing Programmers from Executable Binaries*, *2018 Network and Distributed System Security Symposium*, San Diego, California, SAD, 2015.
- Hrvatsko otvoreno natjecanje u informatici (HONI) 2016./2017., 2. kolo, natjecateljska rješenja 5. zadatka Robert Benić, Luka Tomić
- Kalgutkar V., Kaur R., Gonzalez H., Stakhanova N., Matyukhina A., *Code Authorship Attribution: Methods and Challenges*, *ACM Computing Surveys* 52.1, 2019., str. 1-36

8. Naslov, sažetak i ključne riječi

Naslov: Izrada algoritma za izračun jedinstvenog otiska autora nekog programskog kôda

Ovaj završni rad iznosi nekoliko koncepata na osnovu kojih je moguće razlikovati stilove autora programskih kôdova u binarnom (objektnom ili izvršnom) formatu i na tim konceptima izgrađuje programsko rješenje.

Algoritam je vođen idejom da budući da programer piše izvorni kôd, a ne binarni kôd, potrebno je procijeniti strukturu izvornog kôda što je bolje moguće na osnovu binarnog kôda. Elementi na kojima se algoritam zasniva su kratki odsječci koji se dobiju prevođenjem svake naredbe iz izvornog kôda.

Od elemenata prikupljenih iz programa nekog autora se sastavlja otisak koji treba biti jedinstven za njegov vlastiti stil. Za izbor elemenata se koriste dvije različite metode: metoda *n*-grama i metoda rječnika. Metoda *n*-grama je prikladna u slučajevima u kojima je malo dostupnih podataka o korištenom programskom jeziku i prevoditelju, a metoda rječnika je prikladna u slučajevima u kojima je dostupno više podataka.

Ključne riječi: računalna forenzika, zlоćudni programi, binarni kôd, stil programiranja, otisak, identifikacija, atribucija, analiza *n*-grama, analiza rječnikom

9. Title, summary and keywords

Title: Creating an algorithm for calculating the unique footprint of an author of a program code

This final paper presents several concepts based on which it is possible to distinguish between styles of program code authors in a binary (object or executable) format and builds a software solution upon these concepts.

The algorithm is guided by the idea that because a programmer writes source code rather than binary code, one should approximate the structure of the source code as best as possible based on the binary code. The elements on which the algorithm is based are short snippets obtained by compiling each command from the source code.

A fingerprint is constructed from elements collected from a certain author's programs, which is supposed to be unique for the author's own style. Two different methods are used for selection of elements: the *n*-gram method and the dictionary method. The *n*-gram method is suitable in cases where there is little available data about the programming language and the compiler that were used, and the dictionary method is suitable in cases where more data is available.

Keywords: computer forensics, malware, binary code, programmer style, fingerprint, identification, attribution, *n*-gram analysis, dictionary analysis

Dodatak: Upute za korištenje programske podrške

U prilogu ovog završnog rada se nalazi repozitorij `progauthfp` koji sadrži sve razvijene programe i korištene ispitne podatke u sklopu ovog završnog rada. Isti repozitorij je moguće preuzeti s web-lokacije:

`https://github.com/rbenic-fer/progauthfp.git`

U repozitoriju `progauthfp` se nalaze dvije varijante programskog rješenja, tri ispitna programa i dva pomoćna programa za izdvajanje kôda iz datoteka, prema načinima opisanim u prethodnim poglavljima. Svi programi su pisani u jeziku C++, po standardu C++11 i koriste elemente standardne biblioteke jezika C++ (C++ Standard Library). Ne koriste se biblioteke ili funkcije specifične ni za koji operacijski sustav, pa se programi mogu prevesti bilo kojim prevoditeljem s podrškom za standard C++11, neovisno o operacijskom sustavu.

Izvorni kôdovi programa se nalaze u direktoriju `src`. U direktoriju `bin` se nalaze unaprijed pripremljene izvršne datoteke programa za operativne sustave Linux i Windows na arhitekturi x86, u 32-bitnoj i u 64-bitnoj varijanti. Za izvođenje na drugim operativnim sustavima je potrebno prevesti i povezati izvorne datoteke. U svrhu jednostavnijeg prevođenja i povezivanja, u direktoriju `src` je uz programe priložena datoteka `Makefile`, koja omogućuje automatsko prevođenje i povezivanje pomoću alata `make`.

Programi `parse_elf` i `parse_exe` služe za izdvajanje programskog kôda iz izvršnih ili objektnih datoteka u formatima Executable and Linking Format (ELF), koji se koristi na većini operacijskih sustava zasnovanih na Unixu, odnosno Microsoft Portable Executable (PE), koji se koristi na operacijskom sustavu Microsoft Windows. Oba programa `parse_elf` i `parse_exe` učitavaju po jednu datoteku i izdvojen programski kôd iz nje ispisuju u drugu datoteku.

Program `progauthfp_ngram` sadrži implementaciju 1. metode konstrukcije otiska autora (metode *n*-grama), a program `progauthfp_dict` sadrži implementaciju 2. metode konstrukcije otiska autora (metode rječnika). Oba programa `progauthfp_ngram` i `progauthfp_dict` konstruiraju otisak na osnovu jedne ili više ulaznih datoteka koje sadrže programske kôdove te ga ispisuju u datoteku.

Oba programa `progauthfp_ngram` i `progauthfp_dict` zahtijevaju i koriste datoteku s opisom formata instrukcija. Program `progauthfp_dict` također zahtijeva datoteku s opisom rječnika.

U datoteci s opisom formata binarnih instrukcija, svaki redak opisuje jednu instrukciju, gdje svaki znak predstavlja jedan bit koji pripada toj instrukciji. Pritom znakovi `0` i `1` predstavljaju bitove operacijskog kôda, znakovi `-` i `+` predstavljaju bitove fiksnih vrijednosti 0, odnosno 1 koji ne pripadaju operacijskom kôdu (to mogu biti npr. neke posebne vrijednosti argumenata), a ostali znakovi predstavljaju argumente ili opcije, pri čemu se svaki argument označava s onoliko jednakih znakova koliko ima bitova. Sljedeći su primjeri ispravnih zapisa u datoteci formata:

```
0010aaaabbbbcccc
1010aaaaxxxx++--
```

Ovim zapisima odgovaraju dvije 16-bitne instrukcijske riječi s 4-bitnim operacijskim kôdovima `0010` (heksadekadski `2`), odnosno `1010` (heksadekadski `A`). Svaka instrukcija ima tri 4-bitna argumenta, pri čemu je kod druge instrukcije treći argument postavljen fiksno na `1100` (heksadekadski `C`).

U datoteci sa sadržajem rječnika, zapisi rječnika se sastoje od instrukcija u istom formatu, pri čemu pojedini zapisi trebaju biti odvojeni (barem jednim) praznim retkom. Pritom znakovi koji se koriste za argumente instrukcija ne moraju odgovarati onima u datoteci s formatima, a korištenje istog znaka označava više jednakih argumenata. Sljedeći su primjeri dva zapisa u datoteci rječnika:

```
0010iiiijjjjkkkk
1010iiiixxxx++--
```

```
0010-+--tttttttt
1010uuuuvvvv++--
```

Oba zapisa označavaju odsječke koji se sastoji od iste dvije instrukcije, pri čemu su u prvom zapisu njihovi prvi argumenti jednaki, a u drugom zapisu je prvi argument prve instrukcije postavljen fiksno na `0100` (heksadekadski `4`) te su njezi drugi i treći argument jednaki.

Program `progauthfp_compare` sadrži implementaciju ispitne metode za uspoređivanje dva otiska dobivena jednim od prethodnih programa. Program `progauthfp_compare` se na jednak može koristiti za uspoređivanje otisaka autora i kod metode *n*-grama i kod metode rječnika. Međutim, uspoređivanje otiska izračunatog jednom metodom s otiskom izračunatim drugom metodom se ne preporuča i nije primjenjivano u ovom završnom radu.

Programi `progauthfp_test_ngram` i `progauthfp_test_dict` sadrže implementacije ispitne metode za uspoređivanje otiska s programskim kôdom za otisak izračunat metodom *n*-grama, odnosno metodom rječnika. Implementacija ove ispitne metode je razdvojena na dva programa jer se, kao i kod metoda računanja otiska, razlikuju načini izdvajanja elemenata iz ulaznog programskog kôda. Programi `progauthfp_test_ngram` i `progauthfp_test_dict` zahtijevaju datoteke koje sadrže opis formata instrukcija, odnosno opis formata instrukcija i rječnik, kao što zahtijevaju programi `progauthfp_ngram`, odnosno `progauthfp_dict`.

Za ispitne programe `progauthfp_compare`, `progauthfp_test_ngram` i `progauthfp_test_dict`. ulazne datoteke s otiscima moraju biti u istom formatu kao izlazne datoteke programa `progauthfp_ngram` i `progauthfp_dict`. Ne preporučuje se ručno pisanje datoteka s otiscima.

Svi programi se pokreću iz naredbenog retka. Ako se program pokrene bez argumenata, potrebno je na standardni ulaz upisati nazive i/ili putanje svih

potrebnih ulaznih datoteka. Za potrebe automatizacije, umjesto standardnog ulaza se mogu koristiti i argumenti naredbenog retka.

Programi ispisuju određene statističke podatke na standardni izlaz. Ukoliko je to nepoželjno, količina podataka na standardnom izlazu se može smanjiti ili se standardni izlaz može potpuno isključiti.

Svi programi na koje je to primjenjivo podržavaju argumente i opcije opisane u tablici D.1.

Tablica D.1. Argumenti i opcije programa u repozitoriju `progauthfp`

Duga opcija	**Kratka opcija**	**Vrsta argumenta**	**Opis**
(bez opcije)		naziv ili putanja datoteke	datoteke koje sadrže ulazne otiske (kod ispitnih programa) ili ulazne programe (kod ostalih programa)
`--confidence`	`-c`	realan broj	maksimalna mjera pouzdanosti s kojom se pretpostavlja isti autor (kod ispitnih programa)
`--dict`	`-d`	naziv ili putanja datoteke	datoteka s opisom rječnika
`--elements`	`-e`	prirodan broj	maksimalni broj elemenata u izlaznom otisku
`--formats`	`-f`	naziv ili putanja datoteke	datoteka s opisom formata instrukcija
`--freq-max`	`-R`	realan broj	maksimalna frekvencija elementa otiska
`--freq-min`	`-r`	realan broj	minimalna frekvencija elementa otiska
`--help`	`-h`	*(bez argumenta)*	ispis pomoći
`--n-max`	`-N`	prirodan broj	maksimalna duljina *n*-grama u izlaznom otisku
`--n-min`	`-n`	prirodan broj	minimalna duljina *n*-grama u izlaznom otisku
`--output`	`-o`	naziv ili putanja datoteke	izlazna datoteka
`--program`	`-p`	naziv ili putanja datoteke	datoteka koja sadrži ulazni program (samo kod programa `progauthfp_test_ngram` i `progauthfp_test_dict`)
`--quiet`	`-q`	*(bez argumenta)*	tihi način rada (smanjuje količinu podataka na standardnom izlazu)
`--variance`	`-v`	realan broj	maksimalna varijanca frekvencije elementa u izlaznom otisku

Kod dugih opcija se argument zapisuje sa znakom = (npr. `--elements=1000`), a kod kratkih opcija s razmaknicom (npr. `-e 1000`). Oznaka `--` se može koristiti za završetak opcija.

Kod programa koji ispisuju rezultate u datoteku, podrazumijevani naziv izlazne datoteke je jednak nazivu programa s dodanim nastavkom `.out` (npr. `progauthfp_ngram.out`), a datoteka se sprema u isti direktorij u kojemu se nalazi program, ukoliko nije drugačije zadano.

Primjer pokretanja programa za izradu i ispitivanje otisaka:

```
$ ./parse_elf a.out --output=programFile1
$ ./progauthfp_ngram --formats=formatsFile --n-min=10    \
    --n-max=20 --elements=1000 --output=fingerprintFile1 \
    --quiet programFile1 programFile2 programFile3
$ ./progauthfp_ngram -f formatsFile -n 10 -N 20 -e 1000 -o \
    fingerprintFile2 -- programFile4 -programFile5         \
    --programFile6
$ ./progauthfp_compare --confidence=0.05 fingerprintFile1 \
    fingerprintFile2
```

U direktoriju `config` repozitorija `progauthfp` se nalaze primjeri konfiguracijskih datoteka (opisa formata instrukcija i opisa rječnika) za nekoliko različitih binarnih jezika.

U direktoriju `examples` repozitorija `progauthfp` se nalaze primjeri programskih kôdova koji su korišteni u izradi i u fazi ispitivanja ovog završnog rada.