

UNIVERZITET U BEOGRADU  
MATEMATIČKI FAKULTET



Lazar S. Mladenović

# **AUTOMATSKO ISPRAVLJANJE GREŠAKA DETEKTOVANIH POMOĆU ALATA MEMCHECK**

master rad

Beograd, 2020.

**Mentor:**

doc. dr Milena VUJOŠEVIĆ JANIČIĆ, docent  
Univerzitet u Beogradu, Matematički fakultet

**Članovi komisije:**

prof. dr Miodrag ŽIVKOVIĆ, redovan profesor  
Univerzitet u Beogradu, Matematički fakultet

prof. dr Filip MARIĆ, vanredni profesor  
Univerzitet u Beogradu, Matematički fakultet

**Datum odbrane:**

*Familiji*

**Naslov master rada:** Automatsko ispravljanje grešaka detektovanih pomoću alata Memcheck

**Rezime:** Verifikacija softvera zauzima bitno mesto u procesu razvoja softvera. Dinamička verifikacija softvera zasniva se na proveru ispravnosti softvera koja se vrši tokom njegovog izvršavanja. Postoje razni alati za dinamičku verifikaciju softvera, a jedan od njih je *Valgrind*. Alati njegove distribucije mogu dati profile programa koji mogu pomoći u poboljšanju rada i performansa programa, a takođe mogu i ukazati na greške u okviru programa koji nije lako otkriti. Jedan od alata jeste *Memcheck*, koji analizira i ukazuje na greške u radu sa memorijom. Te greške mogu biti korišćenje neinicijalizovane vrednosti, pristup oslobođenoj, odnosno nedozvoljenoj memoriji i sl., a koje mogu da izazovu pad programa.

Cilj ovog rada jeste konstrukcija alata koji bi koristeći izveštaj koji je dao *Valgrind*, odnosno *Memcheck*, otkrio greške u programu koji je analiziran, a zatim koristeći implementirane šablone za ispravljanje grešaka iste i ispravio. Dakle, osnovna namera rada jeste razumevanje izveštaja koji dobijamo iz alata *Memcheck* i njegova primena za automatsko ispravljanje grešaka koristeći precizno tipizirane šablone za ispravljanje istih, koji će takođe biti implementirani.

**Ključne reči:** Verifikacija softvera, dinamička analiza, profajliranje, Valgrind, Memcheck

# Sadržaj

<b>1</b>	<b>Uvod</b>	<b>1</b>
<b>2</b>	<b>Alat Valgrind</b>	<b>3</b>
2.1	Arhitektura i princip funkcionisanja . . . . .	4
2.2	Prijavljivanje grešaka . . . . .	9
2.3	Memcheck . . . . .	10
2.4	Cachegrind . . . . .	25
2.5	Callgrind . . . . .	28
2.6	Helgrind . . . . .	30
2.7	DRD . . . . .	34
2.8	Massif . . . . .	36
2.9	DHAT . . . . .	38
2.10	BBV . . . . .	40
<b>3</b>	<b>Regularni izrazi</b>	<b>42</b>
3.1	Proces kompilacije . . . . .	43
3.2	Teorija jezika . . . . .	44
3.3	Regularni izrazi i jezici . . . . .	46
3.4	Modul re . . . . .	49
<b>4</b>	<b>Alat Koronka</b>	<b>51</b>
4.1	Korišćenje alata . . . . .	51
4.2	Klasa greške . . . . .	52
4.3	Mehanizam praćenja istorije . . . . .	54
4.4	Algoritam izvršavanja . . . . .	55
4.5	Šabloni za ispravljanje grešaka . . . . .	56
4.6	Primer rada alata . . . . .	60

## *SADRŽAJ*

---

<b>5 Zaključak</b>	<b>63</b>
<b>Literatura</b>	<b>64</b>

# Glava 1

## Uvod

Razvoj softvera predstavlja složen proces koji obuhvata razne aktivnosti.

Napisano je pravilo da se prilikom pisanja programa može desiti da se provuče neka nenamerna greška kao npr. neinicijalizacija promenljive, ili neoslobađanje alocirane memorije, koja u tom trenutku izgleda bezazleno. Međutim, kako razvoj tog programa odmiče, dodatne greške koje ta početna, naizgled bezazlena greška izaziva, bivaju krucijalne i od vitalnog značaja. Na gore pomenute greške, nažalost kompilator ne može da nam ukaže. Ovo nije veliki problem ukoliko su naši programi mali, ali u kompleksnijim programima, onim koji se sastoje od više hiljada linija koda greške i bagove je teško naći.

Verifikacija softvera u cilju ispravljanja tih grešaka zauzima bitno mesto u procesu razvoja softvera. Dinamička verifikacija softvera predstavlja tehniku ispitivanja napisanog koda u toku njegovog izvršavanja, dok statička verifikacija softvera izvršava analizu ispravnosti programa bez njegovog izvršavanja, tj. vrši analizu izvornog koda [13]. Binarna analiza obuhvata analizu na nivou mašinskog koda, snimljenog kao objektni kod (nepovezan) ili kao izvršni kod (povezan) [12]. Primer alata koji vrši dinamičku binarnu analizu koda jeste *Valgrind*.

Alat *Valgrind* pruža brojne alate za debugovanje i profajliranje koji vam pomažu da brže i pravilnije otklonite bagove svojih programa. Najpopularniji od ovih alata zove se *Memcheck*. Može otkriti mnoge greške u vezi sa memorijom koje su česte u programima C i C++, a koje mogu dovesti do rušenja i nepredvidivog ponašanja.

Alat *Valgrind* sa svim svojim alatima samo ukazuje na greške, kao i na to šta ih je izazvalo. Ispravljanje istih ostaje programeru. Cilj ovog rada jeste da automatizuje ispravljanje grešaka na koje ukazuje alat *Valgrind*, tako što će imati

implementirane šablone po kojima će ispravljati razne greške na koje *Valgrind* ukazuje. U realizaciji te ideje će se primenjivati regularni izrazi kojima će biti opisani izrazi koji mogu da prouzrokuju greške na koje nam ukazuje *Memcheck*.

U Glavi 2 alat *Valgrind* će biti detaljno opisan. Biće pobrojani i obrađeni svi alati *Valgrind*-a. Značajniji akcenat će biti na alatu *Memcheck*, dok će ostali biti obrađeni letimično.

U Glavi 3 biće ukratko reći o procesu kompilacije programa i njenim fazama. Centralno mesto zauzeće teorija o jezicima i regularnim izrazima.

U Glavi 4 biće predstavljen sam alat, opisana arhitektura, algoritam izvršavanja i njegove funkcionalnosti. Takođe, čitaoci će biti bliže upoznati sa načinom upotrebe i rada alata.

Na kraju, biće iznet osvrt na ceo rad, kao zaključak na celu priču. Kako tema to afirmiše, a i sam alat napravljen tako da može da se stalno unapređuje, biće iznete i ideje i mogućnosti za dalji razvoj ovog projekta.



## Glava 2

# Alat Valgrind

*Valgrind* je platforma za pravljenje alata za dinamičku analizu. Ime *Valgrind* potiče iz nordijske mitologije. Dolazi sa setom alata, od kojih svaki obavlja neku vrstu debugovanja, profajliranja ili slične zadatke koji vam pomažu da poboljšate svoje programe. *Valgrind* se može koristiti i kao alat za pravljenje novih alata, bez narušavanja postojeće strukture. *Valgrind* distribucija trenutno broji sledeće alate:

1. *Memcheck* - alat za detekciju memorijskih grešaka.
2. *Cachegrind* - optimizator keš memorije i skokova.
3. *Callgrind* - generator grafa skrivene memorije i predikcije skoka. Ima nekih preklapanja sa *Cachegrind* - om, ali takođe pruža neke informacije koje *Cachegrind* ne.
4. *Helgrind* - alat za detekciju grešaka niti.
5. *DRD* – takođe alat za detekciju grešaka niti. Sličan je *Helgrind*-u, ali koristi različite tehnike analize i tako može naći različite probleme.
6. *Massif* - optimizator korišćenja dinamičke memorije.
7. *DHAT* – predstavlja drugu vrstu profajlera hipa. Pomaže vam da razumete probleme sa životnim vekom bloka, upotrebom bloka i neefikasnošću rasporeda.
8. *BBV* - eksperimentalni *SimPoint* osnovni generator vektorskih blokova. Koristan je ljudima koji se bave istraživanjem i razvojem računarske arhitekture.

*Valgrind* alat radi na sledećim arhitekturama:

- Linux: x86, AMD64, ARM, ARM64, PPC32, PPC64, PPC64LE, S390X, MIPS32, MIPS64.
- Solaris: x86, AMD64.
- Android: ARM (2.3.x i novije), ARM64, x86 (4.0 i novije), MIPS32
- Darwin: x86, AMD64 (Mac OS X 10.12)[10]

### 2.1 Arhitektura i princip funkcionisanja

Arhitekturu alata *Valgrind* možemo najjednostavnije opisati pomoću formule:

$$\text{jezgro Valgrind-a} + \text{alat koji se dodaje} = \text{alat Valgrind-a.}$$

Jezgro *Valgrind*-a omogućava izvršavanje klijentskog programa, kao i snimanje izveštaja koji su nastali prilikom analize samog programa. Alat za dinamičku analizu koda se kreira kao dodatak na jezgro *Valgrind*-a, pisan u programskom jeziku C.

Alati *Valgrind*-a koriste metodu bojenja vrednosti. Princip funkcionisanja je takav da svaki registar i memorijsku vrednost "boje" (zamenjuju) sa vrednošću koja govori nešto dodatno o originalnoj vrednosti.

Svi *Valgrind* alati rade na istoj osnovi, s tim da se informacije koje se emituju variraju. Te informacije mogu se iskoristiti za otklanjanje grešaka, optimizaciju koda ili bilo koju drugu svrhu za koju je alat dizajniran.

Sledeća komanda ilustruje način pokretanja alata *Valgrind*:

```
valgrind -tool=alat [argumenti alata] ./izvršniProgram [argumenti izvršnog programa].
```

Svaki *Valgrind*-ov alat je statički povezana izvršna datoteka koja sadrži kod alata i kod jezgra. Izvršna datoteka *valgrind* predstavlja program omotač koji na osnovu *-tool* opcije bira alat koji treba pokrenuti. Sem opcije *-tool* postoji mnoštvo opcija koje precizno definišu rad *Valgrind* alata, kao npr.:

- *-track-origins* koja daje bliže informacije i detaljniji izveštaj o greškama koje su izazvane,

- *-leack-check* koja svako curenje memorije prikazuje detaljno,
- *-show-leack-kinds* koja prikazuje sve vrste curenja koje su definitivne, indirektne, moguće i dostupne u celom izveštaju,
- *-log-file* koja će izlaz alata *Valgrind* proslediti u navedeni log-file (podrazumevano *Valgrind* svoj izlaz ispisuje na terminal) , itd..

Sve ove komande će biti detaljnije objašnjene u sekcijama koje slede. Sad ćemo samo ilustrovati primer pokretanja *Valgrind*-a sa dodatnim argumentima.

```
valgrind -track-origins=yes -leack-check=full -show-leak-kinds=all
-log-file=LOG.txt ./a.out.
```

Podrazumevani alat je *Memchek*, tako da je u primeru navedenom gore poziva alat *Memchek* nad izvršnim programom *a.out*.

Izvršna datoteka alata statički je linkovana tako da se učitava počev od neke adrese koja je obično dosta iznad adresnog prostora koji koriste klasičan korisnički program (na *x86/Linux* i *MIPS/Linux* koristi se adresa 0x38000000). *Valgrind*-ovo jezgro prvo inicijalizuje podsistem kao što su menadžer adresnog prostora i njegov unutrašnji alokator memorije, a zatim učitava klijentovu izvršnu datoteku. Potom se inicijalizuju *Valgrind*-ovi podsistemi kao što su translaciona tabela, aparat za obradu signala, raspoređivač niti i učitavaju se informacije za debugovanje klijenta, ukoliko postoje. Od tog trenutka *Valgrind* ima potpunu kontrolu i počinje sa prevođenjem i izvršavanjem klijentskog programa. Može se reći da *Valgrind* vrši *JIT* (*Just In Time*) kompajliranje mašinskog koda programa u mašinski kod programa dopunjen instrumentacijom. Time postizemo da se nijedan deo koda klijenta ne izvršava u svom izvornom obliku. To funkcioniše tako što alat u originalni kod umeće operacije u svrhu instrumentacije, zatim se takav kod prevodi. Prevođenje se vrši dinamički, prevođenje bloka po potrebi. Proces prevođenja se sastoji iz raščlanjivanja originalnog mašinskog koda u odgovarajuću međureprezentaciju (engl. intermediate representation, skraćeno IR) koji se kasnije instrumentalizuje sa alatom i ponovo prevodi u novi mašinski kod.

Rezultat svega ovoga se naziva translacija, koja se čuva u memoriji i koja se izvršava po potrebi. Jezgro troši najviše vremena na sam proces pravljenja, pronalaženja i izvršavanja translacije. Originalni kod se nikada ne izvršava. Jedini problem koji se ovde može dogoditi je ako se vrši translacija koda koji se menja tokom izvršavanja programa.

Postoje mnoge komplikacije koje nastaju prilikom smeštanja dva programau jedan proces (klijentski program i program alata). Mnogi resursi se dele između ova dva programa, kao što su registri ili memorija. Takođe, alat *Valgrind*-a ne sme da se odrekne totalne kontrole nad izvršavanjem klijentskog programa prilikom izvršavanja sistemskih poziva, signala i niti.

### Osnovni blok

*Valgrind* deli originalni kod u sekvence koje se nazivaju osnovni blokovi. Osnovni blok je pravolinijska sekvenca mašinskog koda, na čiji se početak skače, a koja se završava sa skokom, pozivom funkcije ili povratkom u funkciju pozivaoca. Svaki kod programa koji se analizira ponovo se prevodi na zahtev, pojedinačno po osnovnim blokovima, neposredno pre samog izvršavanja samog bloka. Ako uzmemo da su osnovni blokovi klijentskog koda OB1, OB2, ... , onda prevedene osnovne blokove obeležavamo sa  $t(OB1)$ ,  $t(OB2)$ , itd.. Veličina osnovnog bloka je ograničena na maksimalno šesdeset mašinskih instrukcija.

### Sistemske pozivi

Usluge koje operativni sistem može da pruži aplikativnim programima realizuju se pomoću sistemskih poziva (eng. system calls). Programi uz pomoć sistemskih poziva komuniciraju sa jezgrom i pomoću njega dobijaju mogućnost da izvrše osetljive operaciju u sistemu. Praktično, sistemski pozivi su skup funkcija koji predstavlja interfejs ka operativnom sistemu.

Procesori savremenih računarskih sistema imaju mogućnost rada u dva režima – korisničkom (eng. user mode) i sistemskom (eng. supervisor, kernel mode). U sistemskom režimu moguće je izvršiti sve instrukcije, dok je broj instrukcija koje možemo izvršiti u korisničkom redukovano. Naime, instrukcije za osetljive operacije poput pristupa U/I uređajima, zaštićenim delovima memorije itd., moguće je izvršiti samo u sistemskom režimu rada procesora. Aplikativni programi se veći deo vremena izvršavaju u korisničkom režimu, dok je sistemski režim, kao što je već napomenuto, zadužen za posebno osetljive operacije koje izvodi OS. Pri korišćenju sistemskog poziva prelazi se iz korisničkog u sistemski režim, a dalju kontrolu preuzima operativni sistem. Ključni deo operativnog sistema koji reaguje u ovakvim situacijama je jezgro.

Sistemi pozivi se realizuju pomoću sistema prekida. Korisnički program postavlja parametre sistemskog poziva na određene memorijske lokacije ili registre procesora, inicira prekid, operativni sistem preuzima kontrolu, uzima parametre, izvršava tražene radnje, rezultat stavlja na određene memorijske lokacije ili u registre i vraća kontrolu korisničkom programu.

Nakon svega, zaključujemo da sistemski pozivi obezbeđuju spregu između programa koji se izvršava i operativnog sistema. Generalno, realizuju se na assembler-skom jeziku, ali viši programski jezici, poput jezika C i C++, takođe omogućavaju realizaciju sistemskog poziva. Program koji se izvršava može proslediti parametre operativnom sistemu na više načina, prosleđivanje parametara u registrima procesora, postavljanjem parametara u memorijskoj tabeli. Adresa tabele se prosleđuje u registru procesora, postavljanjem parametara na vrh steka (eng. push), koje operativni sistem "skida" (eng. pop).

Sistemi pozivi se izvršavaju bez posredstva *Valgrind*-a, zato što jezgro *Valgrind*-a ne može da prati njihovo izvršavanje u samom jezgru operativnog sistema [15].

## Translacije

U nastavku su opisani koraci kroz koje *Valgrind* prolazi prilikom analize programa. Postoji osam faza translacije. Sve faze osim instrumentacije obavlja jezgro *Valgrind*-a. Instrumentaciju obavlja alat *Valgrind*-a.

1. *Disasembliranje (razgradnja)* - prevodenje mašinskog koda u ekvivalentni interni skup instrukcija koje se nazivaju međukod instrukcije. U ovoj fazi međukod je predstavljen stablom. Ova faza je zavisna od arhitekture na kojoj se izvršava.
2. *Optimizacija 1* - prva faza optimizacije linearizuje prethodno izgrađeni međukod. Primenuju se neke standardne optimizacije programskih prevodilaca kao što su uklanjanje redundantnog koda, eliminacija podizraza itd..
3. *Instrumentacija* - blok međukoda se prosleđuje alatu, koji može proizvoljno da ga transformiše. Prilikom instrumentalizacije alat u zadati blok dodaje dodatne međukod operacije, kojima proverava ispravnost rada programa. Treba napomenuti da ubačene instrukcije ne narušavaju konzistentno izvršavanje originalnog koda.

4. *Optimizacija 2* - jednostavnija faza optimizacije od prve. Uključuje izračunavanje matematičkih izraza koji se mogu izvršiti pre faze izvršavanja i uklanjanje mrtvog koda.
5. *Izgradnja stabla* - linearizovani međukod se konvertuje natrag u stablo radi lakšeg izbora instrukcija.
6. *Odabir instrukcija* - Stablo međukoda se konvertuje u listu instrukcija koje koriste virtualne registre. Ova faza se takode razlikuje u zavisnosti od arhitekture na kojoj se izvršava.
7. *Alokacija registara* - zamena virtualnih registara stvarnim. Po potrebi se uvode prebacivanja u memoriju. Ne zavisi od platforme. Koristi se poziv funkcija koje pronalaze iz kojih se registara vrši čitanje i u koje se vrši upis.
8. *Asembliranje* - kodiranje izabranih instrukcija na odgovarajući način i smestaju u blok memorije. Ova faza se takode razlikuje u zavisnosti od arhitekture na koji se izvršava [12] [14].

Kada koristite *Valgrind* alate, bilo bi korisno rekompajlirati vašu aplikaciju i podržati biblioteke sa omogućenim informacijama o otklanjanju grešaka (opcija -g). Bez ovih informacija, najbolje što će *Valgrind*-ovi alati moći da urade je da pogode kojoj funkciji pripada određen komad koda, što čini i poruke o greškama i profilisanje rezultata gotovo beskorisnim. Pomoću -g dobićete poruke koje vode direktno do odgovarajuće linije izvornog koda.

Ako planirate da koristite *Memcheck*, važno je znati da je u retkim prilikama primećeno da su optimizacije kompajlera (na -O2 i više, a ponekad i -O1) generisale kod koji zavara *Memcheck* da pogrešno prijavljuje greške vezane za neinicijalizovane vrednosti. Razvojni tim *Valgrind*-a je detaljno razmotrio kako to popraviti i nažalost došli su rezultata da bi to učinilo dalje značajno usporavanje onog što je već sporo sredstvo. Dakle, najbolje rešenje je potpuno isključiti optimizaciju. Budući da ovo često stvari čini nekontrolisano sporima, razuman kompromis je korišćenje opcije -O. Ovo vam donosi većinu prednosti viših nivoa optimizacije, dok istovremeno držite relativno male šanse za lažno pozitivne ili lažno negativne podatke od *Memcheck*-a. Takođe, trebali biste kompajlirati svoj kod sa -Wall jer on može identifikovati neke ili sve probleme koje *Valgrind* može propustiti na višim nivoima optimizacije. (Korišćenje -Wall je takode dobra ideja u celini.) Nivo optimizacije ne utiče na sve ostale alate, a za alate za profilisanje

kao što je *Cachegrind* bolje je da svoj program kompajlirate na normalnom nivou optimizacije [9].

## 2.2 Prijavljivanje grešaka

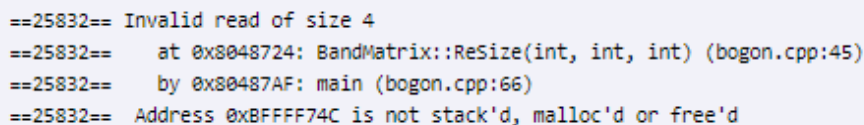
Alati *Valgrind*-a pišu komentare koji sadrže detaljne izveštaje o greškama i drugim značajnim događajima. Svi redovi u komentaru imaju sledeći oblik:

`==12345== odgovarajuća-poruka-alata-Valgrind.`

12345 je ID procesa. Ova šema olakšava razlikovanje izlaznih rezultata programa od komentara *Valgrind*-a, a takođe i razlikovanje komentara od različitih procesa koji su se spojili iz bilo kog razloga.

*Valgrind*-ovi alati podrazumevano pišu u komentar samo ključne poruke, kako bi izbegli da vas preplave informacijama od sekundarne važnosti. Ako želite više informacija o tome šta se događa, pokrenite ponovo *Valgrind* uz opciju `-v`.

Kada alat za proveru greške otkrije da se nešto loše događa u programu, u komentar se upisuje poruka o grešci. Na slici 2.1 prikazan je izlaz iz alata *Memcheck*.



```
==25832== Invalid read of size 4
==25832==    at 0x8048724: BandMatrix::ReSize(int, int, int) (bogon.cpp:45)
==25832==    by 0x80487AF: main (bogon.cpp:66)
==25832== Address 0xBFFFFFF4C is not stack'd, malloc'd or free'd
```

Slika 2.1: Primer izlaza alata Memcheck, izvor: [9]

Ova poruka kaže da je program izvršio ilegalno 4-bajtno čitanje adrese 0xBFFFFFF4C, koja, koliko *Memcheck* može reći, nije važeća adresa steka niti odgovara bilo kojem trenutnom bloku hipa ili nedavno oslobođenim blokovima hipa. Čitanje se događa u redu 45 *bogon.cpp*, pozvanog iz reda 66 iste datoteke itd.. Za greške povezane sa identifikovanim (trenutnim ili oslobođenim) blokom hipa, npr. čitanje oslobođene memorije, *Valgrind* izveštava ne samo o lokaciji na kojoj dogodila se greška, već i gde je pridruženi blok hipa dodeljen/oslobođen.

*Valgrind* pamti sve izveštaje o greškama. Kada se otkrije greška, ona se upoređuje sa starim izveštajima da bi se videlo da li je duplikat. Ako je tako, greška

je zabeležena, ali dalji komentari se ne emituju. Ovo izbegava da vas zatrpaju milionima duplikata izveštaja o greškama.

Zanimljivo je da se greške prijavljuju pre nego što se pridružena operacija zaista dogodi. Na primer, ako koristite *Memcheck* i vaš program pokušava da čita sa adrese nula, *Memcheck* će poslati poruku u tom smislu, a vaš program će onda verovatno pući sa greškom *Segmentation fault*. Generalno, dobra je praksa da pokušate da ispravite greške redosledom u kojem su prijavljene. Ako to ne učinite, to može da zbuni. Na primer, program koji kopira neinicijalizovane vrednosti na nekoliko memorijskih lokacija i kasnije ih koristi, generisaće nekoliko poruka o grešci kada se pokrene na *Memcheck*-u. Prva takva poruka o grešci može sasvim direktno dati osnovni uzrok problema.

Proces otkrivanja duplikata grešaka prilično je skup i koji može prouzrokovati značajno usporenje ako vaš program generiše ogromne količine grešaka. Da bi izbegao ozbiljnije probleme, *Valgrind* će jednostavno zaustaviti prikupljanje grešaka nakon što se vide 1.000 različitih grešaka ili ukupno 10.000.000 grešaka. U ovoj situaciji možete zaustaviti svoj program i popraviti ga, jer vam *Valgrind* neće reći ništa drugo korisno nakon ovoga. Imajte na umu da se ograničenja od 1.000 / 10.000.000 primenjuju nakon uklanjanja potisnutih grešaka. Ova ograničenja su definisana u *m\_errormgr.c* i mogu se povećati ako je potrebno [9].

## 2.3 Memcheck

*Memcheck* je detektor grešaka u memoriji. Kako vrši analizu nad mašinskim, a ne nad izvornim kodom, *Memcheck* ima mogućnost analize programa pisanih u bilo kom jeziku. Može da otkrije sledeće probleme koji su česti u programima C i C++:

- Pristup memoriji kojoj ne biste smeli, tj. prekoračenje i potkopavanje blokova hipa, prekoračenje vrha steka i pristup memoriji nakon što se oslobodi.
- Korišćenje nedefinisanih vrednosti, tj. vrednosti koje nisu inicijalizovane ili su izvedene iz drugih nedefinisanih vrednosti.
- Pogrešno oslobađanje hip memorije, poput dvostrukog oslobađanja blokova hipa, ili neusklađena upotreba *malloc/new/new[]* naspram *free/delete/delete[]*.



- Preklapanje *src* i *dst* pokazivača u *memcpy* i srodnim funkcijama.
- Prosleđivanje sumnjive (verovatno negativne) vrednosti parametru veličine funkcije dodeljivanja memorije.
- Curenje memorije.

Problemi poput ovih teško se mogu pronaći drugim sredstvima, često ostajući neotkriveni tokom dužih perioda, a zatim uzrokujući povremene, teško dijagnostikovane padove. Upravo iz tih razloga memorijske greške spadaju u grupu najteže detektujućih grešaka pa je samim tim i njihovo otklanjanje težak zadatak.

Da biste koristili ovaj alat, ispišite `-tool = memcheck` u komandnoj liniji prilikom pokretanja *Valgrind*-a.

```
valgrind -tool=memcheck [argumenti memchecka] program [argumenti programa]
```

Bez obzira, niste u obavezi s obzirom da je *Memcheck* podrazumevani alat.

Program koji radi pod kontrolom *Memcheck*-a obično je dvadeset do sto puta sporiji nego kada se izvršava samostalno, što je posledica translacije koda. Više o tome biće rečeno u sekcij koja sledi. Izlazni program biće povećan za izlaz koji daje *valgrind* i sam alat *Memcheck*, koji se ispisuje na standardnom izlazu za greške ukoliko se ne definiše drugačije [8].

## Princip funkcionisanja alata Memcheck

### Bitovi valjane vrednosti (V bitovi)

Najjednostavnije je razmišljati o *Memcheck*-u kao alatu koji implementira sintetički CPU koji je identičan stvarnom CPU, osim jednog ključnog detalja. Svaki bit (doslovno) podataka koje obrađuje, čuva i njima rukuje stvarni CPU, dok u sintetičkom CPU ima pridruženi bit "valjane vrednosti" (eng. valid-value bit), koji govori da li prateći bit ima legitimnu vrednost ili ne. U daljem tekstu ovaj bit se naziva V bitom (važeće vrednosti).

Svaki bajt u sistemu iz tog razloga ima i 8V bitova koji ga prate gde god bio. Na primer, kada CPU učitava reč veličine 4 bajta iz memorije, takođe učitava odgovarajućih 32V bita iz bitmape koja čuva V bitove za čitav adresni prostor procesa. Ako bi CPU kasnije trebalo da upiše celu ili neki deo te vrednosti u memoriju na drugoj adresi, relevantni V bitovi će se sačuvati nazad u V-bitnoj bitmapi.

Ukratko, svaki bit u sistemu ima (konceptualno) pridruženi V bit, koji ga prati svuda, čak i unutar CPU-a. Svi registri CPU-a (celi brojevi, registri sa pokretnom tačkom, vektori i uslovi) imaju svoje V bitne vektore. Da bi ovo uspelo, *Memcheck* koristi veliku količinu kompresije da bi kompaktno prikazao V bitove.

Kopiranje vrednosti okolo ne dovodi do toga da *Memcheck* proverava greške ili izveštava o njima. Međutim, kada se vrednost koristi na način koji bi mogao da utiče na spoljno-vidljivo ponašanje vašeg programa, pridruženi V bitovi se odmah proveravaju. Ako bilo koji od njih ukazuje da je vrednost nedefinisana, čak i delimično, prijavice se greška.

Većina operacija na niskom nivou, kao što je sabiranje, uzrokuje da *Memcheck* koristi V bitove za operande za izračunavanje V bitova za rezultat. Čak i ako je rezultat delimično ili u potpunosti nedefinisan, on se ne žali. Provere definisanosti dešavaju se samo na tri mesta - kada se vrednost koristi za generisanje memorijske adrese, kada treba doneti odluku o kontrolnom toku i kada se detektuje sistemski poziv, *Memcheck* proverava definisanost parametara prema potrebi.

Ako bi provera detektovala nedefinisanost, izdala bi se poruka o grešci. Dobijena vrednost bi se naknadno smatrala dobro definisanom. Zašto je to tako? Ako postupite drugačije, to bi dalo duge lance poruka o greškama. Drugim rečima, kada *Memcheck* prijavi grešku nedefinisane vrednosti, pokušava da izbegne prijavljivanje daljih grešaka izvedenih iz te iste nedefinisane vrednosti. Ovo zvuči prekomplikovano što prirodno nameće sledeće pitanje. Zašto jednostavno ne proveriti sva čitanja iz memorije i žaliti se ako je nedefinisana vrednost učitana u registar procesora? Osnovni razlog je taj što to ne funkcioniše dobro, jer savršeno legitimni C programi rutinski kopiraju neinicijalizovane vrednosti u memoriji i ne želimo beskrajne žalbe zbog toga. Da bi to razjasnili, pogledajmo primer sa slike 2.2.

```
1  struct S { int x; char c; };
2  struct S s1, s2;
3  s1.x = 2020;
4  s1.c = 'a';
5  s2 = s1;
```

Slika 2.2: Primer kratkog segmenta koda

Pitanje na koje treba dati odgovor je kolika je struktura S, u bajtovima? Int je 4 bajta, a char jedan bajt, pa možda struktura S zauzima 5 bajtova? Pogrešno.

Svi kompajleri za koje znamo zaokružiće veličinu struct S na ceo broj reči, u ovom slučaju 8 bajtova. Ukoliko to nije slučaj, kompajleri se prisiljavaju da generišu zaista zastrašujući kod za pristup nizovima struct S-a na nekim arhitekturama. Dakle s1 zauzima 8 bajtova, ali će samo njih 5 biti inicijalizovano. Za zadatak  $s2 = s1$ , GCC generiše kod za kopiranje svih 8 bajtova u s2 bez obzira na njihovo značenje. Ako bi Memcheck jednostavno proveravao vrednosti redom kako su izašle iz memorije, “zaurlalo” bi svaki put kad bi se desilo ovakvo dodeljivanje strukture. Dakle, neophodno je komplikovanije ponašanje opisano gore. Ovo omogućava GCC-u da kopira s1 u s2 na bilo koji način, a upozorenje će se emitovati samo ako se kasnije koriste neinicijalizovane vrednosti [8].

### Bitovi važeće adrese (A bitovi)

Prethodna sekcija opisuje kako se uspostavlja i održava valjanost vrednosti bez potrebe da se kaže da li program ima ili nema pravo na pristup bilo kojoj određenoj memorijskoj lokaciji. U ovoj sekciji ćemo razmotriti to pitanje.

Kao što je gore opisano, svaki bit u memoriji ili u procesoru ima pridruženi bit valjane vrednosti (V). Pored toga, svi bajtovi u memoriji, ali ne i u procesoru, imaju pridruženi bit valjane adrese (A, eng. valid-address bit). Ovo ukazuje na to da li program može legitimno čitati ili pisati u tu lokaciju. To ne daje nikakve naznake valjanosti podataka na toj lokaciji, kako je to zadatak V bitova već samo da li se toj lokaciji može pristupiti ili ne. Svaki put kada vaš program čita ili upisuje u memoriju, *Memcheck* proverava A bitove povezane sa adresom. Ako bilo koji od njih označi neispravnu adresu, prijavljuje se greška. Bitno je reći da sama čitanja i pisanja ne menjaju A bitove, već ih samo konsultuju.

Princip postavljanja, odnosno čišćenja A bitova je sledeći:

- Kada se program pokrene, sva globalna područja podataka su označena kao pristupačna.
- Kada program izvrši *malloc/new*, A bitovi za tačno dodeljeno područje, ni bajt više, označeni su kao dostupni. Po oslobađanju područja A bitovi se menjaju kako bi ukazali na nepristupačnost.
- Kada se registar pokazivača steka (SP) pomeri gore ili dole, postavljaju se A bitovi. Pravilo je da je područje SP od vrha do podnožja steka označeni kao dostupno, a ispod SP nedostupno.

- Pri obavljanju sistemskih poziva, A bitovi se odgovarajuće menjaju. Na primer, *mmap* "magično" čini da se datoteke pojavljuju u adresnom prostoru procesa, pa se A bitovi moraju ažurirati ako *mmap* uspe [8].

Nakon svega navedenog dolazimo do rezimea *Memcheck*-ovog principa funkcionisanja.

- Svaki bajt u memoriji ima 8 pridruženih V bitova (važee vrednosti), govoreći da li bajt ima definisanu vrednost i jedan bit A (valjane adrese), odnosno govoreći da li program trenutno ima pravo da čita/piše na tu adresu. Kao što je gore pomenuto, velika upotreba kompresije znači da troškovi obično iznose oko 25%.
- Kada se memorija čita ili se u nju piše, pregledaju se odgovarajući A bitovi. Ako označe nevažecu adresu, *Memcheck* emituje grešku neispravno čitanje (eng. invalid read) ili neispravno pisanje (eng. invalid write).
- Kada se memorija čita u registre CPU-a, relevantni V bitovi se preuzimaju iz memorije i čuvaju u simuliranom CPU-u. Oni se ne konsultuju.
- Kada se registar ispiše u memoriju, V bitovi za taj registar se takođe zapisuju nazad u memoriju.
- Kada se vrednosti u registrima CPU koriste za generisanje memorijske adrese ili za određivanje ishoda uslovne grane, V bitovi za te vrednosti se proveravaju i emituje se greška ako je bilo koja od njih nedefinisana. Kada se vrednosti u registrima procesora koriste u bilo koju drugu svrhu, *Memcheck* izračunava V bitove za rezultat, ali ih ne proverava.
- Jednom kada se provere V bitovi za vrednost u CPU, oni se postavljaju da ukazuju na validnost. Ovo izbegava dugačke lance grešaka.
- Kada se vrednosti učitaju iz memorije, *Memcheck* proverava A bitove za tu lokaciju i izdaje upozorenje o nevalidnoj adresi ako je potrebno. U tom slučaju su učitani V bitovi primorani da označe validno stanje, uprkos tome što je lokacija nevalidna. Ovaj naizgled čudan izbor smanjuje količinu zbunjujućih informacija koje se predstavljaju korisniku. Izbegava neprijatnu pojavu u kojoj se memorija čita sa mesta koje je i nedostupno i sadrži nevažee vrednosti, a kao rezultat toga dobićete ne samo grešku sa nevažecom adresom

(čitanje / pisanje), već i potencijalno veliki skup greškaka neinicijalizovanih vrednosti - jedna za svaki put kada se vrednost koristi.

*Memcheck* presreće pozive funkcija *malloc*, *calloc*, *realloc*, *valloc*, *memalign*, *free*, *new*, *new[]*, *delete* i *delete[]*. Ponašanje koje se dobija usled toga je sledeće:

- *malloc/new/new[]*: vraćena memorija je označena kao adresabilna, ali nema važeće vrednosti. To znači da u nju morate pisati pre nego što je pročitate.
- *calloc*: vraćena memorija označena je i adresabilnom i važećom, jer *calloc* postavlja područje na nulu.
- *realloc*: ako je nova veličina veća od stare, novi odeljak je adresiran, ali nevažeći, kao kod *malloc*-a. Ako je nova veličina manja, odloženi odeljak je označen kao nedostupan. *Realloc*-u možete proslediti samo pokazivač koji vam je prethodno izdao *malloc/calloc/realloc*.
- *free/delete/delete[]*: ovim funkcijama možete proslediti samo pokazivač koji vam je prethodno izdala odgovarajuća funkcija dodeljivanja. Inače, *Memcheck* se žali. Ako je pokazivač zaista važeći, *Memcheck* označava celo područje na koje pokazuje kao da ga nije moguće adresirati i postavlja blok u red oslobođenih blokova. Cilj je odložiti što je duže moguće realokaciju ovog bloka. Dok se to ne dogodi, svi pokušaji da mu se pristupi izazvaće grešku nevažeće adrese, kao što biste se nadali [8].

## Greške nevalidnog čitanja/pisanja (eng. invalid read/write errors)

```
Invalid read of size 4
at 0x40F68BCC: (within /usr/lib/libpng.so.2.1.0.9)
by 0x40F68B04: (within /usr/lib/libpng.so.2.1.0.9)
by 0x40B07FF4: read_png_image(QImageIO *) (kernel/qpngio.cpp:326)
by 0x40AC751B: QImageIO::read() (kernel/qimage.cpp:3621)
Address 0xBFFFFFF0E0 is not stack'd, malloc'd or free'd
```

Slika 2.3: Primer ispisa greške nevalidnog čitanja, izvor: [8]

Ove greške se javljaju kada vaš program čita ili piše u memoriju na mestu za koje *Memcheck* smatra da ne bi trebalo. U ovom primeru, program je izvršio

čtetvorobajtno čitanje na adresi 0xBFFFF0E0, negde unutar sistemske biblioteke *libpng.so.2.1.0.9*, koja je pozvana negde drugde u istoj biblioteci, pozvana iz linije 326 *qpngio.cpp*, i tako dalje.

*Memcheck* će svakako pokušati da odredi na šta se ilegalna adresa može odnositi, jer je to često korisno. Dakle, ako pokazuje na blok memorije koji je već oslobođen, bićete obavešteni o tome, kao i o tome gde je blok oslobođen. Isto tako, ako se ispostavi da se nalazi na kraju hip bloka, što je greška, tj. greška prekoračenja niza, bićete obavešteni o ovoj činjenici, kao i o tome gde je blok dodeljen. Ako koristite opciju *-read-var-info*, *Memcheck* će raditi sporije, ali može dati detaljniji opis bilo koje nelegalne adrese.

U ovom primeru *Memcheck* ne može da identifikuje adresu. Zapravo se adresa nalazi na steku, ali iz nekog razloga ovo nije važeća adresa steka - nalazi se ispod pokazivača steka i to nije dozvoljeno. U ovom konkretnom slučaju verovatno je uzrokovano GCC-om koji generiše nevažeći kod, što je poznata greška u nekim drevnim verzijama GCC-a.

Imajte na umu da vam *Memcheck* samo govori da će vaš program uskoro pristupiti memoriji na nedozvoljenoj adresi i da pritom taj pristup ne može sprečiti. Dakle, ako vaš program pristupi, što bi obično rezultiralo greškom (segmentation fault), vaš program će i dalje imati isto ponašanje - ali od *Memcheck*-a ćete dobiti poruku neposredno pre toga. U ovom konkretnom primeru čitanje "smeća" na steku nije fatalno, a program ostaje živ [8].

## Korišćenje neinicijalizovanih vrednosti

Greška pri korišćenju neinicijalizovane vrednosti se prijavljuje kada vaš program koristi vrednost koja nije inicijalizovana - drugim rečima, nije definisana. Ovde se nedefinisana vrednost koristi negde unutar *printf* mašine C biblioteke. Ova greška je prijavljena prilikom pokretanja programa sa slike 2.4, dok je njen ispis dat na slici 2.5.

Važno je shvatiti da vaš program može kopirati neželjene (neinicijalizovane) podatke koliko god želi. *Memcheck* to posmatra i evidentira podatke, ali se ne žali. Žalba se izdaje samo kada vaš program pokušava da koristi neinicijalizovane podatke na način koji može uticati na spolja vidljivo ponašanje vašeg programa. U ovom primeru, *x* je neinicijalizovano. *Memcheck* primećuje vrednost koja se prosleđuje *\_IO\_printf* i odatle *\_IO\_vfprintf*, ali ne daje komentar. Međutim,

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(){
5      int x;
6      printf("%d\n", x);
7      exit(EXIT_SUCCESS);
8  }
```

Slika 2.4: Program koji izaziva grešku korišćenja neinicijalizovanih vrednosti

```
==5430== Memcheck, a memory error detector
==5430== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==5430== Using Valgrind-3.14.0 and LibVEX; rerun with -h for copyright info
==5430== Command: ./program
==5430==
==5430== Conditional jump or move depends on uninitialised value(s)
==5430==   at 0x48E2E40: __vfprintf_internal (vfprintf-internal.c:1644)
==5430==   by 0x48CD8D7: printf (printf.c:33)
==5430==   by 0x109162: main (program.c:6)
==5430==
==5430== Use of uninitialised value of size 8
==5430==   at 0x48C732E: _itoa_word (_itoa.c:179)
==5430==   by 0x48E29EF: __vfprintf_internal (vfprintf-internal.c:1644)
==5430==   by 0x48CD8D7: printf (printf.c:33)
==5430==   by 0x109162: main (program.c:6)
==5430==
==5430== Conditional jump or move depends on uninitialised value(s)
==5430==   at 0x48C7339: _itoa_word (_itoa.c:179)
==5430==   by 0x48E29EF: __vfprintf_internal (vfprintf-internal.c:1644)
==5430==   by 0x48CD8D7: printf (printf.c:33)
==5430==   by 0x109162: main (program.c:6)
==5430==
==5430== Conditional jump or move depends on uninitialised value(s)
==5430==   at 0x48E348B: __vfprintf_internal (vfprintf-internal.c:1644)
==5430==   by 0x48CD8D7: printf (printf.c:33)
==5430==   by 0x109162: main (program.c:6)
==5430==
==5430== Conditional jump or move depends on uninitialised value(s)
==5430==   at 0x48E2B5A: __vfprintf_internal (vfprintf-internal.c:1644)
==5430==   by 0x48CD8D7: printf (printf.c:33)
==5430==   by 0x109162: main (program.c:6)
==5430==
0
```

Slika 2.5: Primer ispisa greške korišćenja neinicijalizovanih vrednosti

`_IO_vfprintf` mora ispitati vrednost kako bi je mogao pretvoriti u odgovarajući ASCII niz i *Memcheck* se u ovom trenutku žali.

Izvori grešaka neinicijalizovanih vrednosti su sledeći:

- Lokalne promenljive u procedurama koje nisu inicijalizovane, kao u primeru iznad.
- Sadržaj blokova hipa (dodeljen *malloc*-om, *new* ili sličnom funkcijom) pre nego što se nešto u tom bloku ispiše, odnosno pre inicijalizacije.

Da biste videli informacije o izvorima neinicijalizovanih podataka u svom programu, koristite opciju `-track-origins=yes`. Kao što je već rečeno, ovo čini *Memcheck* sporijim, ali može znatno olakšati pronalaženje osnovnih uzroka neinicijalizovanih grešaka [8].

## Korišćenje neinicijalizovane ili neadresirane vrednosti u sistemskom pozivu

*Memcheck* prati i proverava sve parametre sistemskih poziva. Svaki parametar se proverava pojedinačno, bio on inicijalizovan ili ne. Ukoliko sistemski poziv treba da čita iz bafera koji je obezbeđen programu, *Memcheck* proverava da li je ceo bafer adresiran i da li je njegov sadržaj inicijalizovan. Takođe, ako sistemski poziv treba da piše u bafer, *Memcheck* proverava da li je bafer moguće adresirati. Nakon sistemskog poziva, *Memcheck* ažurira sve parametre koje je pratio kako bi tačno odražavale sve promene u stanju memorije izazvane sistemskim pozivom.

```
1  #include <stdlib.h>
2  #include <unistd.h>
3
4  int main( void ){
5      char* arr = malloc(10);
6      int* arr2 = malloc(sizeof(int));
7      write( 1 /* stdout */, arr, 10 );
8      exit(arr2[0]);
9  }
```

Slika 2.6: Program koji izaziva grešku korišćenja neinicijalizovane ili neadresirane vrednosti u sistemskom pozivu, izvor: [8]

Na slikama 2.6 i 2.7 dat je primer programa koji ilustruje sistemski poziv sa neispravnim parametrima, kao i izveštaj koji dobijamo nakon analize pomenutog programa. Možemo da vidimo da je *Memcheck* prikazao dve greške sa informacijama o korišćenju neinicijalizovanih vrednosti u sistemskim pozivima. Prva greška prikazuje da parametar *arr* sistemskog poziva *write()* pokazuje na neinicijalizovanu vrednost. Druga greška prikazuje da je podatak koji se prosleđuje sistemskom pozivu *exit()* nedefinisan. Takođe, prikazane su i linije u samom programu gde se ove vrednosti koriste [8].



```
==5695== Conditional jump or move depends on uninitialised value(s)
==5695==    at 0x48E2E40: __vfprintf_internal (vfprintf-internal.c:1644)
==5695==    by 0x48CD8D7: printf (printf.c:33)
==5695==    by 0x109162: main (1.c:7)
==5695==    Uninitialised value was created by a stack allocation
==5695==    at 0x109145: main (1.c:5)
==5695==
==5695== Use of uninitialised value of size 8
==5695==    at 0x48C732E: _itoa_word (_itoa.c:179)
==5695==    by 0x48E29EF: __vfprintf_internal (vfprintf-internal.c:1644)
==5695==    by 0x48CD8D7: printf (printf.c:33)
==5695==    by 0x109162: main (1.c:7)
==5695==    Uninitialised value was created by a stack allocation
==5695==    at 0x109145: main (1.c:5)
==5695==
==5695== Conditional jump or move depends on uninitialised value(s)
==5695==    at 0x48C7339: _itoa_word (_itoa.c:179)
==5695==    by 0x48E29EF: __vfprintf_internal (vfprintf-internal.c:1644)
==5695==    by 0x48CD8D7: printf (printf.c:33)
==5695==    by 0x109162: main (1.c:7)
==5695==    Uninitialised value was created by a stack allocation
==5695==    at 0x109145: main (1.c:5)
==5695==
==5695== Conditional jump or move depends on uninitialised value(s)
==5695==    at 0x48E348B: __vfprintf_internal (vfprintf-internal.c:1644)
==5695==    by 0x48CD8D7: printf (printf.c:33)
==5695==    by 0x109162: main (1.c:7)
==5695==    Uninitialised value was created by a stack allocation
==5695==    at 0x109145: main (1.c:5)
==5695==
==5695== Conditional jump or move depends on uninitialised value(s)
==5695==    at 0x48E2B5A: __vfprintf_internal (vfprintf-internal.c:1644)
==5695==    by 0x48CD8D7: printf (printf.c:33)
==5695==    by 0x109162: main (1.c:7)
==5695==    Uninitialised value was created by a stack allocation
==5695==    at 0x109145: main (1.c:5)
```

Slika 2.7: Primer ispisa greške korišćenja neinicijalizovane ili neadresirane vrednosti u sistemskom pozivu

## Nedopušteno oslobađanje memorije

*Memcheck* prati sve blokove memorije koje je program alocirao pomoću *malloc/new*, tako da može tačno znati da li je argument prosleđen *free/delete* legitiman ili ne. Slika 2.8 ilustruje program koji je oslobodio isti blok dva puta.

Kao i kod grešaka nevalidnog čitanja/pisanja, *Memcheck* pokušava da dokuči oslobođenu adresu. Ako je, kao ovde, slučaj da je adresa koja je prethodno oslobođena, to će biti prijavljeno i učiniće duplikate oslobađanja istog bloka lako uočljivim. Ovu poruku ćete dobiti i ako pokušate da oslobodite pokazivač koji ne pokazuje početak hip bloka. Slika 2.9 ilustruje izlaz dobijen puštanjem programa sa slike 2.8 na analizu alatom *Memcheck* [8].

## Preklapanje izvornog i odredišnog bloka

Sledeće C bibliotečke funkcije kopiraju neke podatke iz jednog memorijskog bloka u drugi - *memcpy*, *strcpy*, *strncpy*, *strcat*, *strncat*. Blokovi na koje ukazuju

```
1  #include <stdlib.h>
2  #include <stdio.h>
3
4  int main(){
5      char* p;
6      p = (char) malloc(5);
7      p = (char) malloc(7);
8      free(p);
9      free(p);
10     p = (char) malloc(9);
11     exit(EXIT_SUCCESS);
12 }
```

Slika 2.8: Program koji oslobađa isti blok dva puta

```
==6215== Memcheck, a memory error detector
==6215== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==6215== Using Valgrind-3.14.0 and LibVEX; rerun with -h for copyright info
==6215== Command: ./3
==6215==
==6215== Invalid free() / delete / delete[] / realloc()
==6215==    at 0x483997B: free (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==6215==    by 0x10918C: main (3.c:8)
==6215==    Address 0xffffffffffff90 is not stack'd, malloc'd or (recently) free'd
==6215==
==6215== Invalid free() / delete / delete[] / realloc()
==6215==    at 0x483997B: free (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==6215==    by 0x109198: main (3.c:9)
==6215==    Address 0xffffffffffff90 is not stack'd, malloc'd or (recently) free'd
==6215==
==6215== HEAP SUMMARY:
==6215==    in use at exit: 21 bytes in 3 blocks
==6215==    total heap usage: 3 allocs, 2 frees, 21 bytes allocated
==6215==
==6215== LEAK SUMMARY:
==6215==    definitely lost: 21 bytes in 3 blocks
==6215==    indirectly lost: 0 bytes in 0 blocks
==6215==    possibly lost: 0 bytes in 0 blocks
==6215==    still reachable: 0 bytes in 0 blocks
==6215==    suppressed: 0 bytes in 0 blocks
==6215== Rerun with --leak-check=full to see details of leaked memory
==6215==
==6215== For counts of detected and suppressed errors, rerun with: -v
==6215== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
```

Slika 2.9: Primer ispisa greške nedopuštenog oslobađanja memorije

njihovi pokazivači *src* i *dst* ne smeju se preklapati. *POSIX* standardi imaju sledeću formulaciju: "Ako se kopiranje vrši između objekata koji se preklapaju, ponašanje je nedefinisano". Stoga *Memcheck* ovo proverava. Ukoliko dođe do preklapanja, prijavice grešku. Ne želite da se dva bloka preklapaju, jer bi jedan od njih mogao

```
==27492== Source and destination overlap in memcpy(0xbffff294, 0xbffff280, 21)
==27492==    at 0x40026CDC: memcpy (mc_replace_strmem.c:71)
==27492==    by 0x804865A: main (overlap.c:40)
```

Slika 2.10: Primer ispisa greške preklapanja izvornog i odredišnog bloka, izvor: [8]

delimično da se prepíše kopiranjem. Može delovati da *Memcheck* previše pedantno prijavljuje ovo u slučaju kada je *dst* manji od *src*. Na primer, očigledan način implementacije *memcpy* je kopiranje iz prvog bajta u poslednji. Međutim, vodiči za optimizaciju nekih arhitektura preporučuju kopiranje od poslednjeg bajta do prvog. Pouka priče je da ako zaista želite da napišete prenosivi kod, ne pravite nikakve pretpostavke o implementaciji jezika [8].

## Sumnjive vrednosti argumenata

Sve funkcije za alokaciju memorije uzimaju argument koji određuje veličinu memorijskog bloka koji treba dodeliti. Jasno je da tražena veličina treba da bude pozitivna vrednost koja obično nije preterano velika. Na primer, krajnje je neverovatno da veličina zahteva za dodelu premašuje  $2^{63}$  bajta na 64-bitnoj mašini. Mnogo je verovatnije da je takva vrednost rezultat pogrešnog izračunavanja veličine i da je u stvari negativna vrednost (koja se čini preterano velikom jer se obrazac bita tumači kao nepotpisani celi broj). Takva vrednost se naziva "sumnjivom vrednošću". Argument veličine sledećih funkcija dodeljivanja biće proveren da li je sumnjiv: *malloc*, *calloc*, *realloc*, *memalign*, *new*, *new[]*, *\_builtin\_new*, *\_builtin\_vec\_new*. Za *calloc* se proveravaju oba argumenta.

Ukoliko dođe do ovakve situacije, prijaviće se greška sa slike 2.11 [8].

```
==32233== Argument 'size' of function malloc has a fishy (possibly negative) value: -3
==32233== at 0x4C2CFA7: malloc (vg_replace_malloc.c:298)
==32233== by 0x400555: foo (fishy.c:15)
==32233== by 0x400583: main (fishy.c:23)
```

Slika 2.11: Primer ispisa greške sumnjive vrednosti argumenata, izvor: [8]

## Detekcija curenja memorije

*Memcheck* evidentira podatke o svim blokovima hipa koji su alocirani tokom izvršavanja programa pozivom funkcija *malloc/new* i dr.. Dakle, kada program prekine sa radom, *Memcheck* zna koji blokovi nisu oslobođeni. Ukoliko je opcija *-leak-check* adekvatno podešena, *Memcheck* će za svaki neoslobođeni blok odrediti da li mu je moguće pristupiti preko pokazivača. Postoje dva načina da pristupimo sadržaju nekog memorijskog bloka preko pokazivača. Prvi način je preko pokazivača koji pokazuje na početak memorijskog bloka (eng. start-pointer), dok je drugi

način preko pokazivača koji pokazuje na sadržaj unutar memorijskog bloka (eng. interior-pointer). Na nekoliko načina možemo saznati da li postoji pokazivač koji pokazuje na unutrašnjost nekog memorijskog bloka. Može se desiti da je postojao pokazivač koji je pokazivao na početak bloka, ali je namerno (ili nenamerno) pomeren da pokazuje na unutrašnjost bloka. Drugi slučaj jeste postojanje neželjene vrednosti u memoriji, koja je u potpunosti nepovezana i slučajna.

Imajući to na umu, na slici 2.12 navešćemo devet mogućih slučajeva kada pokazivači pokazuju na neke memorijske blokove.

	Lanac pokazivača	Slučaj curenja AAA	Slučaj curenja BBB
(1)	RRR -----> BBB		DR
(2)	RRR ---> AAA ---> BBB	DR	IR
(3)	RRR ----- BBB		DL
(4)	RRR AAA ---> BBB	DL	IL
(5)	RRR -----?-----> BBB		(y)DR, (n)DL
(6)	RRR ---> AAA -?-> BBB	DR	(y)IR, (n)DL
(7)	RRR -?-> AAA ---> BBB	(y)DR, (n)DL	(y)IR, (n)IL
(8)	RRR -?-> AAA -?-> BBB	(y)DR, (n)DL	(y,y)IR, (n,y)IL, (_,n)DL
(9)	RRR AAA -?-> BBB	DL	(y)IL, (n)DL

Legenda lanca pokazivača:

- RRR: skup pokazivača
- AAA, BBB: memorijski blokovi u dinamičkoj memoriji
- --->: (startni) pokazivač
- -?->: unutrašnji pokazivač

Legenda curenja memorije:

- DR: direktno dostupan
- IR: indirektno dostupan
- DL: direktno izgubljen
- IL: indirektno izgubljen
- (y)XY: XY ako je unutrašnji pokazivač pravi pokazivač
- (n)XY: XY ako unutrašnji pokazivač nije pravi pokazivač
- (\_)XY: XY u svakom slučaju

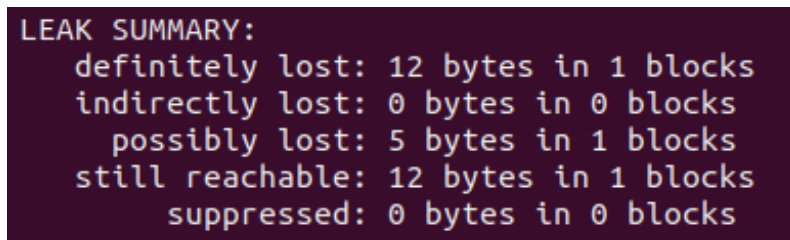
Slika 2.12: Primer pokazivača na memorijski blok

Svaki mogući slučaj može se svesti na jedan od gore navedenih devet. *Memcheck* objedinjuje neke od ovih slučajeva, što rezultira u sledeće četiri kategorije curenja memorije.

- **Još uvek dostupan (eng. still reachable)** – Ova klasa pokriva slučajeve 1 i 2 navedene na slici 2.12. Pronađen je pokazivač na početak bloka ili više takvih pokazivača. Kako postoje pokazivači koji pokazuju na memorijsku lokaciju koja nije oslobođena, programer može da oslobodi memorijsku lokaciju neposredno pre završetka izvršavanja programa. Blokovi iz ove kategorije su vrlo česti i verovatno nisu problem. Dakle, *Memcheck* podrazumevano neće pojedinačno prijavljivati takve blokove.

- **Definitivno izgubljen (eng definitely lost)** – Ova klasa pokriva slučaj 3 naveden na slici 2.12. To znači da se ne može naći pokazivač na blok. Blok je klasifikovan kao "izgubljen", jer oslobađanje istog nije moguće na izlazu iz programa, kako na njega ne postoji pokazivač. Ovo je verovatno simptom gubitka pokazivača u nekom ranijem trenutku programa. Takve slučajeve treba da popravi programer.
- **Indirektno izgubljen (eng indirectly lost)** – Ova klasa pokriva slučajeve 4 i 9 navedene na slici 2.12. To znači da je blok izgubljen, ne zato što na njega nema pokazivača, već zato što su svi blokovi koji na njega upućuju sami izgubljeni. Na primer, ako imate binarno stablo i koren se izgubi, svi njegovi čvorovi-potomci će se indirektno izgubiti. Budući da će problem nestati ako se popravi definitivno izgubljeni blok koji je prouzrokovao indirektno curenje, *Memcheck* po default-u neće pojedinačno prijavljivati takve blokove sem ako nije podešena opcija *-show-reachable=yes*.
- **Moguće izgubljen (eng. possibly lost)** – Ova klasa obuhvata slučajeve 5-8 navedene na slici 2.12. To znači da je pronađen jedan ili više pokazivača na memorijski blok, ali bar jedan od pokazivača je pokazuje na unutrašnjost memorijskog bloka. To bi mogla biti slučajna vrednost u memoriji koja pokazuje na unutrašnjost bloka, što ne treba smatrati validnim sve dok se ne razreši slučaj pokazivača koji pokazuje na unutrašnjost bloka.

Treba napomenuti da ovo mapiranje devet mogućih slučajeva na četiri vrste curenja nije nužno najbolji način za prijavljivanje curenja. Naročito se prema unutrašnjim pokazivačima postupa nedosledno. Moguće je da će se kategorizacija u budućnosti poboljšati. Na slici 2.13 dat je rezime curenja memorije koji ispisuje *Memcheck*.



```
LEAK SUMMARY:
  definitely lost: 12 bytes in 1 blocks
  indirectly lost: 0 bytes in 0 blocks
    possibly lost: 5 bytes in 1 blocks
  still reachable: 12 bytes in 1 blocks
    suppressed: 0 bytes in 0 blocks
```

Slika 2.13: Rrezime curenja memorije

Ako je uključena opcija `-leak-check = full`, *Memcheck* će dati detaljan izveštaj za svaki definitivno izgubljeni ili moguće izgubljeni blok, uključujući i mesto gde je alociran. Međutim, *Memcheck* vam ne može reći kada, kako ili zašto se izgubio pokazivač na iscureli blok, to morate sami da rešite. Generalno, treba težiti tome da vaši programi nemaju definitivno izgubljene ili moguće izgubljene blokove na izlazu.

Opcija `-show-leak-kind = <set>` kontroliše skup vrsta curenja koje će se prikazati kada je uključena opcija `-leak-check = full`. Inače, podrazumevana vrednost opcije `-leak-check` je *summary*.

`<set>` se definiše na jedan od sledećih načina:

1. Lista opcija odvojenih zarezom. Moguće opcije su: *definite*, *indirect*, *possible*, *reachable*.
2. *All* - za specifikaciju svih vrsta curenja.
3. *None* - za prazan skup.

Podrazumevana vrednost za vrste curenja koje se prikazuju je `-show-leak-kinds = definite, possible`.

Na slici 2.14 prikazan je izveštaj koji nam daje *Memcheck* o definitivnom gubitku bloka veličine 12 bajta, još uvek dostupnom bloku veličine 12 bajta, moguće izgubljenom bloku veličine 5 bajta, kao i liniju u programu gde su oni alocirani.

```
HEAP SUMMARY:
  in use at exit: 29 bytes in 3 blocks
  total heap usage: 4 allocs, 2 frees, 33 bytes allocated

5 bytes in 1 blocks are possibly lost in loss record 1 of 3
   at 0x483874F: malloc (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-and64-linux.so)
   by 0x109166: main (4.c:5)

12 bytes in 1 blocks are still reachable in loss record 2 of 3
   at 0x483874F: malloc (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-and64-linux.so)
   by 0x109185: main (4.c:9)

12 bytes in 1 blocks are definitely lost in loss record 3 of 3
   at 0x483874F: malloc (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-and64-linux.so)
   by 0x109193: main (4.c:10)
```

Slika 2.14: Izveštaj o curenju memorije

Budući da postoje različite vrste curenja memorije različitih težina, postavlja se pitanje koja curenja treba računati kao istinske "greške", a koja ne. *Memcheck* koristi sledeći kriterijum:

- *Memcheck* smatra da je curenje memorije "greška" samo ako je uključena opcija `-leak-check=full`. Drugim rečima, ako podaci o curenju memorije nisu prikazani, smatra se da to curenje nije "greška".

- Definitivno i moguće izgubljeni blokovi se smatraju za pravu "grešku", dok indirektno izgubljeni i još uvek dospuni blokovi se ne smatraju kao greška [8].

## 2.4 Cachegrind

Kako vremenom razlika između brzine RAM memorije i performansi koje obezbeđuje procesor eksponencijalno raste, merenje performansi keša postaje važna disciplina. Keš je zadužen da amortizuje tu razliku, a da bismo utvrdili koliko je on sposoban za to, prate se promašaji i pogodci u kešu. Jedna od ideja za poboljšanje jeste da se smanji broj promašaja u kešu, prevashodno na nivou *LL*.

*Cachegrind* je alat koji simulira i prati pristup keš memoriji mašine na kojoj se program, koji se analizira, izvršava. Takođe, može se koristiti i za profajliranje izvršavanja grana. On simulira memoriju mašine, koja ima prvi nivo keš memorije podeljene u dve odvojene nezavisne sekcije: *I1* - sekcija keš memorije u koju se smeštaju instrukcije i *D1* - sekcija keš memorije u koju se smeštaju podaci, uz podršku objedinjenog drugog nivoa keša - *L2*. Ovaj način konfiguracije odgovara mnogim modernim mašinama.

Međutim, neke moderne mašine imaju tri ili četiri nivoa keš memorije. Za ove mašine (u slučajevima kada *Cachegrind* može automatski otkriti konfiguraciju keš memorije) *Cachegrind* simulira pristup kešu prvog i poslednjeg nivoa, tj. *Cachegrind* simulira *I1*, *D1* i *LL*. Razlog za ovaj izbor je taj što keš memorija poslednjeg nivoa (*LL*) ima najveći uticaj na vreme izvršavanja, jer maskira pristupe glavnoj memoriji. Dalje, *L1* keš memorije često imaju malu asocijativnost, pa se njihovom simulacijom mogu otkriti slučajevi kada kod loše interaguje sa ovom keš memorijom.

*Cachegrind* prikuplja sledeće statističke podatke o programu koji analizira:

- Podaci o čitanjima instrukcija iz keš memorije uključuju sledeće statistike:
  - Ir* - ukupan broj izvršenih instrukcija
  - I1mr* - broj promašaja čitanja instrukcija iz keš memorije nivoa *I1*
  - ILmr* - broj promašaja čitanja instrukcija iz keš memorije nivoa *LL*
- Podaci o čitanjima brze memorije uključuju sledeće statistike:
  - Dr* - ukupan broj čitanja memorije

$D1mr$  - broj promašaja čitanja nivoa keš memorije  $D1$

$DLmr$  - broj promašaja čitanja nivoa keš memorije  $LL$

- Podaci o pisanjima u brzu memoriju uključuju sledeće statistike:  
 $Dw$  - ukupan broj pisanja u memoriji  
 $D1mw$  - broj promašaja pisanja u nivo keš memorije  $D1$   
 $DLmw$  - broj promašaja pisanja u nivo keš memorije  $LL$
- Broj uslovno izvršenih grana -  $Bc$  i broj promašaja uslovno izvršenih grana -  $Bcm$
- Broj indirektno izvršenih grana -  $Bi$  i broj promašaja indirektno izvršenih grana -  $Bim$ .

Imajte na umu da ukupni pristup  $D1$  jednak zbiru  $D1mr + D1mv$ , a da je ukupni pristup  $LL$  jednak zbiru  $ILmr + DLmr + DLMv$ . Ova statistika se prikuplja na nivou celog programa, kao i pojedinačno na nivou funkcija. Postoji mogućnost da se dobije i broj pristupa skrivenoj memoriji za svaku liniju koda u originalnom programu. Na modernim mašinama  $L1$  promašaj košta oko 10 procesorskih ciklusa,  $LL$  promašaj košta oko 200 procesorskih ciklusa, a promašaji uslovno i indirektno izvršene grane od 10 do 30 procesorskih ciklusa. Detaljno profajliranje keša i grana može biti vrlo korisno za razumevanje načina na koji vaš program komunicira sa mašinom, a samim tim i na taj način kako da to učinite bržim.

Što se rutine upotrebe *Valgrind* alata tiče, uobičajeno je da kompajlirate program sa za debug mod (opcija  $-g$ ). Međutim, za razliku od uobičajene upotrebe ostalih *Valgrind* alata, sa *Cachegrind*-om želimo još i uključenu optimizaciju, jer biste trebali profilisati svoj program onako kako će se normalno pokretati.

Nakon kompilacije treba pokrenuti sam *Cachegrind* da biste prikupili informacije o profilisanju, a zatim pokrenuti *cg\_annotate*, koji je u okviru paketa *Valgrind*, da biste dobili detaljan prikaz tih informacija. Kao opcionalni međukorak, možete da koristite *cg\_merge* za sumiranje izlaza više pokretanja *Cachegrind*-a u jednu datoteku koju zatim koristite kao ulaz za *cg\_annotate*. Alternativno, možete koristiti *cg\_diff* za razlikovanje više izlaza iz alata *Cachegrind*, koje kasnije koristimo kao ulaz za *cg\_annotate*.

Da biste koristili alat *Cachegrind* neophodno je podesiti opciju  $-tool=cachegrind$ . To ilustruje sledeća komanda.



*valgrind -tool=cachegrind [argumenti cachegrinda] program [argumenti programa].*

Izvršavanje programa trajeće veoma sporo, a po završetku štampaće se sažete statistički podaci prikazani na slici 2.15. Statistika predviđanja grana se ne

```

==3665== Cachegrind, a cache and branch-prediction profiler
==3665== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.
==3665== Using Valgrind-3.14.0 and LibVEX; rerun with -h for copyright info
==3665== Command: ./1
==3665==
--3665-- warning: L3 cache found, using its data for the LL simulation.
sum = 10000.000
==3665==
==3665== I   refs:      25,216,564
==3665== I1 misses:      1,174
==3665== LLi misses:      1,167
==3665== I1 miss rate:      0.00%
==3665== LLi miss rate:      0.00%
==3665==
==3665== D   refs:      14,067,419 (12,053,566 rd + 2,013,853 wr)
==3665== D1 misses:      253,351 ( 127,675 rd + 125,676 wr)
==3665== LLd misses:      253,046 ( 127,416 rd + 125,630 wr)
==3665== D1 miss rate:      1.8% ( 1.1% + 6.2% )
==3665== LLd miss rate:      1.8% ( 1.1% + 6.2% )
==3665==
==3665== LL refs:      254,525 ( 128,849 rd + 125,676 wr)
==3665== LL misses:      254,213 ( 128,583 rd + 125,630 wr)
==3665== LL miss rate:      0.6% ( 0.3% + 6.2% )

```

Slika 2.15: Izveštaj alata *Cachegrind*

prikuplja po default-u. Da biste to omogućili, dodajte opciju *-branch-sim = yes*.

Pored štampanja rezimea na standardni izlaz, *Cachegrind* u datoteku upisuje i detaljnije informacije o profajliranju. Podrazumevano se ova datoteka naziva *cachegrind.out.<pid>* (gde je *<pid>* ID procesa koji se izvršio), ali njeno ime se može promeniti pomoću opcije *-cachegrind-out-file*. Ova datoteka je čitljiva za ljude, ali je pre svega namenjena tumačenju pratećeg programa *cg\_annotate*.

Da biste dobili rezime funkciju po funkciju, pokrenite komandu

*cg\_annotate <filename>*,

gde je *<filename>* *Cachegrind* izlazna datoteka.

Prvi deo izveštaja prikazan je na slici 2.16. On obuhvata informacije o konfiguraciji keša (*I1*, *D1*, *LL*), komandu pokretanja programa koji se ispituje, koji su događaji praćeni itd..

Nakon toga sledi izveštaj na nivou celog programa prikazan na slici 2.17. On je sličan rezimeu datom kada *Cachegrind* završi sa radom uz dodatnu statistiku fukcija po funkcija. Svaka funkcija je identifikovana parom *ime\_datoteke: ime\_funkcije*. Ako kolona sadrži samo tačku, to znači da funkcija nikada ne izvodi taj događaj. Ime *???* koristi se ako se ime datoteke i/ili ime funkcije ne može utvrditi iz

```
I1 cache:      32768 B, 64 B, 8-way associative
D1 cache:      32768 B, 64 B, 8-way associative
LL cache:      3145728 B, 64 B, 12-way associative
Command:       ./1
Data file:      cachegrind.out.3665
Events recorded: Ir I1mr ILmr Dr D1mr DLMr Dw D1mw DLMw
Events shown:   Ir I1mr ILmr Dr D1mr DLMr Dw D1mw DLMw
Event sort order: Ir I1mr ILmr Dr D1mr DLMr Dw D1mw DLMw
Thresholds:     0.1 100 100 100 100 100 100 100 100
Include dirs:
User annotated:
Auto-annotation: off
```

Slika 2.16: Prvi deo izveštaja dobijenog primenom `cg_annotate`

informacija o otklanjanju grešaka. Ako većina unosa ima oblik `???: ???` program verovatno nije kompajliran sa opcijom `-g` [2].

```
-----
Ir  I1mr ILmr      Dr  D1mr  DLMr      Dw  D1mw  DLMw
25,210,564 1,174 1,107 12,053,566 127,674 127,415 2,013,853 125,670 125,630  PROGRAM TOTALS
-----
Ir  I1mr ILmr      Dr  D1mr  DLMr      Dw  D1mw  DLMw  file:function
14,007,012 0 0 7,003,004 125,000 125,000 1,001,004 0 0 /home/student/Desktop/EXAMPLES/example1.c:array_sun
11,007,024 3 3 5,003,006 2 1 1,001,007 125,003 125,003 /home/student/Desktop/EXAMPLES/example1.c:main
60,317 10 10 14,235 1,101 1,097 16 2 1 /build/glibc-KRRMSn/glibc-2.29/elf/dl-addr.c:dl_addr
20,201 22 22 9,170 263 194 3,115 11 6 /build/glibc-KRRMSn/glibc-2.29/elf/dl-lookup.c:do_lookup_x
-----
```

Slika 2.17: Drugi deo izveštaja dobijenog primenom `cg_annotate`

## 2.5 Callgrind

*Callgrind* je alat za profajliranje koji generiše istoriju poziva funkcija korisničkog programa u vidu grafa. Bazično, sakupljeni podaci sastoje se od broja izvršenih instrukcija, njihov odnos sa linijama izvršnog koda, odnos pozivaoc/pozvan između funkcija, kao i broj takvih poziva. Opciono, simulacija keš memorije i/ili profajliranje grana (slično *Cachegrind*-u) mogu dati dodatne informacije o ponašanju aplikacije u toku izvršavanja.

Podaci dobijeni profajliranjem ispisuju se u datoteku po završetku rada alata i programa. Za prezentaciju podataka i interaktivnu kontrolu profajliranja obezbeđena su dva alata komandne linije.

- **callgrind\_annotate** - Ovaj alat čita generisani fajl i ispisuje sortirane liste funkcija, opciono sa anotacijom izvora. Za grafičku vizuelizaciju podataka, preporučuju se dodatni alat *KCachegrind*, GUI zasnovan na KDE / Qt, koji olakšava navigaciju velikom količinom podataka koje proizvodi *Callgrind*.

- **callgrind\_control** - Ovaj alat vam omogućava da interaktivno posmatrate i kontrolirate status programa koji se trenutno izvodi pod kontrolom *Callgrind*-a, bez zaustavljanja programa. Možete dobiti statističke podatke kao npr. stanje steka, a u svakom trenutku se takođe može generisati i profil.

Alat *Cachegrind* sakuplja podatke, odnosno broji događaje koji se dešavaju direktno u jednoj funkciji. Ovaj mehanizam sakupljanja podataka se naziva ekskluzivnim. Alat *Callgrind* proširuje ovu funkcionalnost tako što propagira cenu funkcije preko njenih granica. Na primer, ako funkcija *foo* poziva funkciju *bar*, cena funkcije *bar* se dodaje funkciji *foo*. Kada se ovaj mehanizam primeni na celu funkciju, dobija se slika takozvanih inkluzivnih poziva, gde cena svake funkcije uključuje i cene svih funkcija koje ona poziva, direktno ili indirektno.

Zahvaljujući grafu poziva, može da se odredi, počevši od main funkcije, koja funkcija ima najveću cenu poziva. Pozivaoc/pozvan cena je izuzetno korisna za profilisanje funkcija koje imaju više poziva iz raznih funkcija, i gde imamo priliku za optimizaciju našeg programa menjajući kod u funkciji koja je pozivaoc, tačnije redukovanjem broja poziva. Važno je napomenuti da mogućnost detektovanja svih poziva funkcija, kao i zavisnost instrukcija alata *Callgrind* zavisi od platforme na kojoj se izvršava. Ovaj alat najbolje radi na *x86* i *amd64*, ali nažalost ne daje najtačnije rezultate na platformama *PowerPc*, *ARM* i *MIPS*. Razlog je taj što kod navedenih platformi ne postoji eksplicitan poziv ili instrukcija u skupu instrukcija, pa *Callgrind* mora da se oslanja na heuristike da bi detektovao pozive ili instrukcije.

Kao i kod *Cachegrind*-a, da bi uspešno analizirali program koristeći *Callgrind*, program kompajliramo za debug mod (opcija *-g*) i sa uključenom optimizacijom. Da biste koristili ovaj alat, morate navesti *-tool = callgrind* u komandnoj liniji *Valgrind*-a. To ilustruje sledeća komanda.

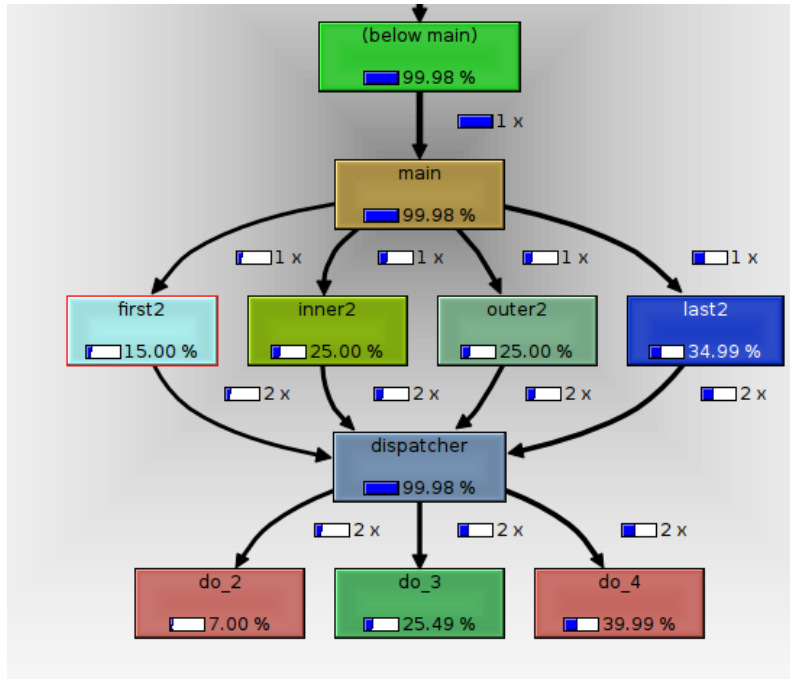
```
valgrind -tool=callgrind [opcije callgrinda] program [argumenti programa].
```

Nakon završetka programa i rada alata, generiše se datoteka podataka profila koja se zove *callgrind.out.<pid>*, gde je *<pid>* ID procesa programa koji se profajlira. Da biste generisali rezime funkciju po funkciju koristeći pomenutu datoteku, iskoristite sledeću komandu:

```
callgrind_annotate [dodatni argumenti] callgrind.out.<pid>.
```

Ovaj rezime je sličan izlazu koji dobijete pokretanjem *Cachegrind*-a sa *cg\_annotate* - lista funkcija je poredana po ekskluzivnim troškovima funkcija, koje su takođe prikazane [3].

Na slici 2.18 prikazana vizuelizacija izveštaja *Callgrind*-a korišćenjem alata *KCachegrind*.



Slika 2.18: Primer vizuelizacije izveštaja, izvor: Stack Overflow

## 2.6 Helgrind

*Helgrind* je *Valgrind*-ov alat za otkrivanje grešaka sinhronizacije u programima C, C++ i Fortran prilikom upotreba modela niti *POSIX*. Glavne apstrakcije modela niti *POSIX* su: skup niti koji dele zajednički adresni prostor, formiranje niti, spajanje niti, izlaz iz funkcije niti, muteksi (katanci), uslovne promenljive (obaveštenja o događajima među nitima), čitaj-piši zaključavanje, spinloкови, semafori i barijere.

*Helgrind* može otkriti sledeća tri tipa grešaka:

1. Pogrešna upotreba *POSIX* API-ja
2. Potencijalno blokiranje prouzrokovano redosledom zaključavanja

3. Trka sa podacima - pristup memoriji bez odgovarajućeg zaključavanja ili sinhronizacije

Problemi poput ovih često rezultiraju ponovljivim padovima, vremenski zavisnim padovima, mrtvim tačkama i drugim nepravilnim ponašanjem, a koji se teško mogu pronaći drugim sredstvima.

*Helgrind* najbolje funkcioniše kada vaša aplikacija koristi samo interfejs za rad sa nitima *POSIX*. Međutim, ako želite da koristite korisnički prilagođene niti, možete opisati njihovo ponašanje *Helgrind*-u koristeći makro *ANNOTATE\_\** definisan u *helgrind.h*.

Da biste koristili ovaj alat, morate navesti *-tool = helgrind* u komandnoj liniji *Valgrind*-a.

### Pogrešna upotreba POSIX API-ja

*Helgrind* presreće pozive ka funkcijama biblioteke *pthread*, i zbog toga je u mogućnosti da otkrije veliki broj grešaka. Za sve *pthread* funkcije koje *Helgrind* presreće generiše se podatak o grešci ako funkcija vrati kod greške, i ako *Helgrind* nije našao greške u kodu. Ovakve greške mogu da dovedu do nedefinisanog ponašanja programa i do pojave grešaka u programima koje je kasnije veoma teško otkriti.

Greške koje mogu biti otkrivene su:

- Greške u otključavanju muteksa – slučaj nevažećeg muteksa, nezaključanog muteksa, ili muteksa zaključanog od strane druge niti.
- Greške u radu sa zaključanim muteksom – uništavanje nevažećeg ili zaključanog muteksa, rekurzivno zaključavanje nerekurzivnog muteksa, oslobađanje memorije koja sadrži zaključan muteks.
- Prosleđivanje muteksa kao argumenta funkcije koja očekuje kao argument reader-writer lock i obrnuto.
- Greške sa *pthread barrier* - nevažeća ili dupla inicijalizacija, uništavanje *pthread barrier* koji nikada nije inicijalizovan ili koga niti čekaju ili čekanje na objekat koji nije nikada inicijalizovan.
- Greške prilikom korišćenja funkcije *pthread\_cond\_wait* - prosleđivanje nezaključanog, nevažećeg ili muteksa koga je zaključala druga nit.

- *Pthread* funkcija vrati kod greške koji je potrebno dodatno obraditi, kada se nit uništi, a da još drži zaključanu promeljivu.
- Nekonzistentne veze između uslovnih promeljivih i njihovih odgovarajućih muteksa.

Provere koje se odnose na mutekse se takođe primenjuju i na reader-writer lock. Prijavljena greška prikazuje i primarno stanje steka koje pokazuje gde je detektovana greška. Takođe, ukoliko je moguće ispisuje se i broj linije u izvornom kodu gde se greška nalazi. Ukoliko se greška odnosi na muteks, *Helgrind* će prikazati i gde je prvi put detektovao problematični muteks.

### Potencijalno blokiranje niti

*Helgrind* nadgleda redosled kojim niti zaključavaju promenljive. To mu omogućava detektovanje potencijalnih zastoja koji bi mogli nastati formiranjem ciklusa zaključavanja. Na ovaj način je moguće detektovati greške koje se nisu javile tokom samog procesa testiranja programa, već se javljaju kasnije tokom korišćenja istog.

Šta tačno predstavlja ovakav tip greške, ilustrovaćemo primerom: Zamislimo da imamo zajednički resurs *R*, kome da bi pristupili moramo da zaključamo dve promenljive *L1* i *L2*. Pretpostavimo zatim da dve niti *N1* i *N2* žele da pristupe deljenoj promenljivoj *R*. Do blokiranja niti dolazi kada nit *N1* zaključa *L1*, a u istom trenutku *N2* zaključa *L2*. Nakon toga nit *N1* ostane blokirana jer čeka da se otključa *L2*, ne bi li je zaključao, a analogno tome nit *N2* ostane blokirana jer čeka da se otključa *L1*.

*Helgrind* kreira usmereni graf koji predstavlja sve promenljive koje se mogu zaključati, a koje je otkrio u prošlosti. Kada nit naiđe na novu promenljivu koju zaključava, graf se ažurira i proverava se da li graf sadrži ciklus u kome se nalaze zaključane promenljive. Postojanje kruga u kome se nalaze zaključane promenljive je signal da je moguće da će doći do blokiranja niti tokom izvršavanja.

Generalno, *Helgrind* će odabrati dve zaključane promenljive uključene u ciklus i pokazati vam kako je narušena konzistentnost. To čini tako što prikazuje programske tačke koje su prvo definisale redosled i programske tačke koje su ga kasnije prekršile. Ako postoje više od dve zaključane promenljive u krugu problem je još ozbiljniji.

## Trka sa podacima

Trka sa podacima (eng. data races) nastaje, odnosno može nastati usled korišćenja deljene memorije bez adekvatnog zaključavanja ili sinhronizacije, a u cilju omogućavanja pristupa samo jednoj niti. Obezbeđivanje programa bez trke sa podacima je jedna od centralnih poteškoća programiranja sa nitima.

Primer sa slike 2.19 ilustruje program u kom se koristi promenljiva bez adekvatne sinhronizacije.

```
#include <pthread.h>

int var = 0;

void* child_fn ( void* arg ) {
    var++; /* Unprotected relative to parent */ /* this is line 6 */
    return NULL;
}

int main ( void ) {
    pthread_t child;
    pthread_create(&child, NULL, child_fn, NULL);
    var++; /* Unprotected relative to child */ /* this is line 13 */
    pthread_join(child, NULL);
    return 0;
}
```

Slika 2.19: Primer programa koji koristi promenljivu sa neadekvatnom sinhronizacijom, izvor: [6]

U ovom programu je nemoguće znati kolika je vrednost *var* na kraju rada programa. Da li je 2 ili 1? Razlog tome je što ništa ne sprečava da obe niti (roditelj i dete) da pristupe i promene vrednost promenljive *var*. Ispravan program bi zaštitio *var* katancem tipa *pthread\_mutex\_t*, koja se dobija pre svakog pristupa i oslobađa nakon toga. Izlaz *Helgrind*-a za program sa slike 2.19 prikazan je na slici 2.20:

U izveštaju koji je prikazan na slici 2.20 možemo tačno da vidimo koje niti pristupaju promenljivoj bez sinhronizacije, gde se vrši sam pristup promenljivoj, ime i veličinu same promenljive kojoj niti pristupaju. Prikazivanje osnovnih informacija promenljivih nema dodatnih troškova tokom izvršavanja, nakon što se *Helgrind* pokrene i pokrene vaš program. Međutim, *Helgrind* će potrošiti znatno više vremena i memorije prilikom pokretanja programa kada je aktivna opcija čitanja informacija za debugovanje. Stoga je ova opcija podrazumevano onemogućena. Da biste je omogućili, trebate *Helgrind*-u dati opciju *-read-var-info = yes*.

```
Thread #1 is the program's root thread

Thread #2 was created
  at 0x511C08E: clone (in /lib64/libc-2.8.so)
  by 0x4E333A4: do_clone (in /lib64/libpthread-2.8.so)
  by 0x4E33A30: pthread_create@@GLIBC_2.2.5 (in /lib64/libpthread-2.8.so)
  by 0x4C299D4: pthread_create@* (hg_intercepts.c:214)
  by 0x400605: main (simple_race.c:12)

Possible data race during read of size 4 at 0x601038 by thread #1
Locks held: none
  at 0x400606: main (simple_race.c:13)

This conflicts with a previous write of size 4 by thread #2
Locks held: none
  at 0x4005DC: child_fn (simple_race.c:6)
  by 0x4C29AFF: mythread_wrapper (hg_intercepts.c:194)
  by 0x4E3403F: start_thread (in /lib64/libpthread-2.8.so)
  by 0x511C0CC: clone (in /lib64/libc-2.8.so)

Location 0x601038 is 0 bytes inside global var "var"
declared at simple_race.c:3
```

Slika 2.20: Izveštaj *Helgrind*-a za korišćenje promenljive bez adekvatne sinhronizacije, izvor: [6]

Algoritam za detekciju pristupa promenljivoj bez sinhronizacije baziran je na "desilo se pre" relaciji (eng. happens-before relation). Kada kažemo da su dva pristupa promenljivoj od strane dve različite niti u "relaciji desilo se pre", to znači da postoji neki lanac operacija sinhronizacije među nitima zbog kojih se ti pristupi dešavaju u određenom redosledu, bez obzira na stvarne brzine napretka pojedinačnih niti. Ovo je neophodno svojstvo da bi program koji radi sa nitima bio pouzdan, zbog čega ga *Helgrind* proverava. *Helgrind* presreće precizno definisan skup događaja na kojima počiva relacija "desilo se pre" i kreira usmereni aciklični graf koji predstavlja sve "desilo se pre" relacije u programu. Pored toga, on prati i sve pristupe memoriji u programu. Ako promenljivoj pristupaju dve različite niti, ali *Helgrind* ne može da pronađe nijednu putanju od prvog pristupa do drugog kroz promenuti graf, onda izveštava grešku o trci sa podacima [6].

## 2.7 DRD

*DRD* je *Valgrind*-ov alat za otkrivanje grešaka u višenitnim programima C i C++. Alat radi za bilo koji program koji koristi niti bazirane na modelu *POSIX*.

U zavisnosti od toga koja se višenitna paradigma koristi u programu, može se pojaviti jedna ili više sledećih grešaka:

- Trka sa podacima;



- Zadržavanje katanaca - jedna nit blokira napredovanje jedne ili više drugih niti držeći predugo katanac zaključanim;
- Pogrešna upotreba *POSIX* API-ja;
- Potencijalno blokiranje prouzrokovano redosledom zaključavanja;
- Lažno deljenje - ako niti koje rade na različitim procesorskim jezgrima često pristupaju različitim promenljivim smeštenim u istoj liniji keš memorije, to će usporiti uključene niti zbog česte razmene linija keš memorije.

*DRD* i *Helgrind* ne koriste iste algoritme za otkrivanje grešaka, pa samim tim ne otkrivaju iste tipove grešaka. Što se tiče grešaka vezanih za pogrešnu upotrebu *POSIX* API-ja, kao i trke sa podacima, greške koje *DRD* otkriva su identične onim koje otkriva *Helgrind*, a koje su opisane u sekciji 2.6, uz određene minimalne razlike.

Kada je u pitanju greška vezana za trku sa podacima, *DRD* ispisuje poruku svaki put kada otkrije da je došlo do trke podataka. Svakako treba imati na umu sledeće činjenice prilikom tumačenja izlaza *DRD*-a. Alat *DRD* svakoj niti dodeljuje jedinstveni ID broj. Brojevi koji se dodeljuju nitima u svojstvu identifikatora počinju od jedinice i dodeljuju se jednokratno. Termin segment se odnosi na uzastopni niz operacija učitavanja, skladištenja i sinhronizacije - sve izvršene u istoj niti. Segment uvek započinje i završava se operacijom sinhronizacije. Analiza trke sa podacima vrši se između segmenata, a ne između pojedinačnih operacija učitavanja i skladištenja iz razloga boljih performansi. Uvek postoje najmanje dva pristupa memoriji uključena u trku podataka. Pristupi memoriji uključeni u trku podataka nazivaju se sukobljenim pristupima memoriji. *DRD* štampa izveštaj za svaki pristup memoriji koji je u sukobu sa prethodnim pristupom memoriji. Na slici 2.21 dat je izlaz iz alata *DRD* nakon što je došlo do trke sa podacima u programu.

Niti moraju biti u stanju da napreduju, a da ih druge niti predugo ne blokiraju. Ponekad nit mora da sačeka dok muteks ili objekat sinhronizacije reader-writer ne otključa druga nit. Pojava u kojoj jedna nit ne može da nastavi sa radom zbog blokiranja drugih niti naziva se zadržavanje katanaca (eng. lock contention). Ovakva pojava je nepoželjna u višenitnim sistemima, a u njenom otklanjanju pomaže alat *DRD* koji otkriva ovaj tip problema. Zadržavanje katanaca uzrokuje kašnjenja. Takva kašnjenja treba da budu što kraća. Dve opcije komandne li-

```
$ valgrind --tool=drd --read-var-info=yes drd/tests/rwlock_race
...
==9466== Thread 3:
==9466== Conflicting load by thread 3 at 0x006020b8 size 4
==9466==   at 0x400B6C: thread_func (rwlock_race.c:29)
==9466==   by 0x4C291DF: vg_thread_wrapper (drd_pthread_intercepts.c:186)
==9466==   by 0x4E3403F: start_thread (in /lib64/libpthread-2.8.so)
==9466==   by 0x53250CC: clone (in /lib64/libc-2.8.so)
==9466== Location 0x6020b8 is 0 bytes inside local var "s_racy"
==9466== declared at rwlock_race.c:18, in frame #0 of thread 3
==9466== Other segment start (thread 2)
==9466==   at 0x4C2847D: pthread_rwlock_rdlock*
(drd_pthread_intercepts.c:813)
==9466==   by 0x400B6B: thread_func (rwlock_race.c:28)
==9466==   by 0x4C291DF: vg_thread_wrapper (drd_pthread_intercepts.c:186)
==9466==   by 0x4E3403F: start_thread (in /lib64/libpthread-2.8.so)
==9466==   by 0x53250CC: clone (in /lib64/libc-2.8.so)
==9466== Other segment end (thread 2)
==9466==   at 0x4C28B54: pthread_rwlock_unlock*
(drd_pthread_intercepts.c:912)
==9466==   by 0x400B84: thread_func (rwlock_race.c:30)
==9466==   by 0x4C291DF: vg_thread_wrapper (drd_pthread_intercepts.c:186)
==9466==   by 0x4E3403F: start_thread (in /lib64/libpthread-2.8.so)
==9466==   by 0x53250CC: clone (in /lib64/libc-2.8.so)
...
```

Slika 2.21: Izveštaj *DRD*-a o trci sa podacima, izvor: [5]

nije *-exclusive-threshold=<n>* i *-shared-threshold=<n>* omogućavaju otkrivanje prekomernog zadržavanja katanaca tako što *DRD* prijavljuje svako zaključavanje koje je zadržano duže od navedenog praga [5].

Da biste koristili ovaj alat, morate navesti *-tool = drd* u komandnoj liniji *Valgrind-a*.

## 2.8 Massif

*Massif* je alat za analizu hip memorije korisničkog programa. Meri koliko memorije hipa vaš program koristi i na koji način. To obuhvata memoriju koju korisnik može pristupiti, kao i memoriju koja se koristi za dodatne koncepte poput book-keeping bajtova i prostora za poravnanje. Takođe, može izračunati i veličinu stek memorije vašeg programa, s tim da ova opcija nije podrazumevana, već se mora eksplicitno navesti.

Profajliranje hipa vam može pomoći da smanjite količinu memorije koju vaš program koristi. Na modernim mašinama sa virtuelnom memorijom to će doprineti određenim pogodnostima. To može ubrzati vaš program, kako manji programi imaju bolju iskorišćenost keša i izbegavaju straničenje. S druge strane, ako vaš program zahteva puno memorije, dobra iskorišćenost hipa će smanjiti šansu za izglednjivanjem prostora za razmenu (eng. swap space) korisničke mašine.

Postoje određeni tipovi curenja memorije koji ne mogu biti otkriveni klasičnim alatima koji vrše analizu curenja memorije, poput *Memcheck*-a. To je zato što memorija zapravo nikada nije izgubljena, pokazivač na nju i dalje postoji, ali nije u upotrebi. Programi koji imaju ovakvo curenje mogu s vremenom nepotrebno povećati količinu memorije koju koriste. *Massif* može pomoći u identifikovanju ovakvog tipa curenja memorije.

Važno je znati da vam *Massif* ne daje samo informaciju koliko hip memorije koristi vaš program, već takođe daje vrlo detaljne informacije koje ukazuju na to koji su delovi vašeg programa odgovorni za alociranje te memorije.

Pre svega, kao i za ostale *Valgrind* alate, trebalo bi da kompajlirate program za debug mod (opcija -g). U ovom slučaju nije presudno važno sa kojim nivoom optimizacije kompajlirate svoj program, jer je malo verovatno da će to uticati na upotrebu memorije hipa.

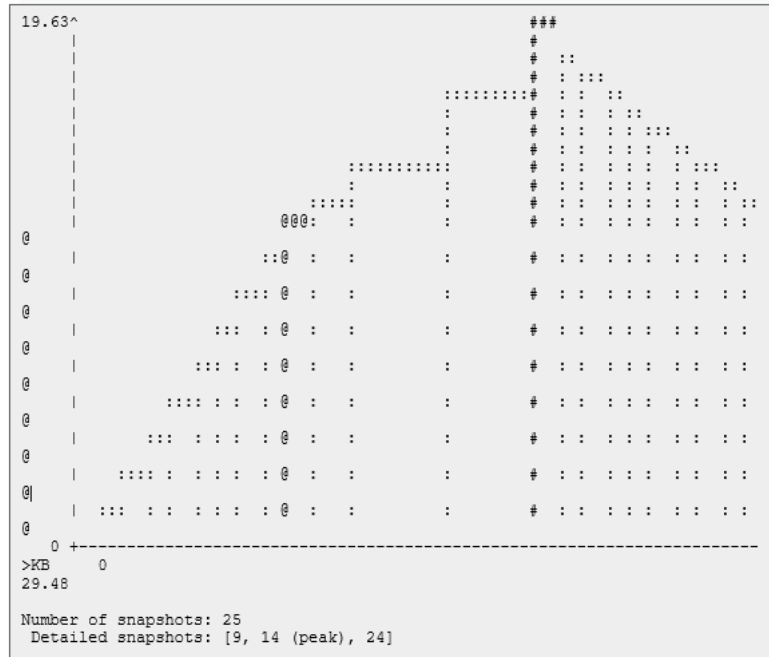
Nakon kompilacije željenog programa treba pokrenuti sam *Massif* da biste prikupili informacije o profajliranju, nakon čega treba pokrenuti *ms\_print* da biste ih predstavili na čitljiv način. Da biste koristili ovaj alat, morate navesti *-tool = massif* u komandnoj liniji *Valgrind*-a.

```
valgrind -tool=massif [argumenti massif] program [argumenti programa]
```

Program koji se izvršava pod alatom *Massif* će raditi veoma sporo. Po završetku rada programa i alata, svi podaci o profajliranju zapisuju se u datoteku. Podrazumevano se ova datoteka naziva *massif.out.<pid>*, gde je *<pid>* ID procesa, iako se ovo ime datoteke može promeniti pomoću opcije *-massif-out-file*. Kao što je već rečeno, da bi informacije koje je *Massif* sakupio mogli da vidimo u čitljivom formatu, koristimo *ms\_print* nad izlaznom datotekom alata *Massif*, npr. *massif.out.12345*.

```
ms_print massif.out.12345
```

*ms\_print* proizvodi graf koji prikazuje potrošnju memorije tokom izvršavanja programa, kao i detaljne informacije o različitim tačkama programa koje su odgovorne za alokaciju memorije. Korišćenje različitih skripti za prezentaciju rezultata je namerno, u cilju odvajanja sakupljanja podataka od prezentacije, što znači da je moguće dodati nov način prikaza podataka u svakom trenutku. Na slici 2.22 vidimo izlaz iz alata *Massif* prikazan pomoću *ms\_print*.

Slika 2.22: Primer opterećenja hipa korišćenjem alata *Massif*, izvor: [7]

Važno je naglasiti da alat *Massif* meri samo hip memoriju, odnosno memoriju koja je alocirana koristeći *malloc*, *calloc*, *realloc*, *memalign*, *new*, *new[]* i nekoliko drugih sličnih funkcija. Ovo znači da *Massif* ne meri memoriju koja je alocirana sistemskim pozivima nižeg nivoa kao što su *mmap*, *mremap* i *brk*. Iako u programu mogu postojati sistemski pozivi za alociranje memorije, *Massif* neće uzeti u obzir tu memoriju tokom analize programa. Ukoliko iz nekog razloga želimo da se uzme u obzir sva alocirana memorija u našem programu, to možemo učiniti uključivanjem opcije *-pages-as-heap=yes*. Uključivanjem ove opcije, *Massif* neće profajlirati hip memoriju, već stranice u memoriji [7].

## 2.9 DHAT

*DHAT* je alat za ispitivanje kako programi koriste alociranu memoriju na hipu. Prati dodeljene blokove i pregleda svaki pristup memoriji da bi pronašao kom bloku, ako takav blok postoji, se pristupa. Na osnovu tačke dodeljivanja predstavlja informacije o tim blokovima, kao što su veličina, životni vek, broj čitanja i pisanja i obrasce čitanja i pisanja. Korišćenjem ovih informacija moguće je identifikovati mesta alokacije sa sledećim karakteristikama:

- potencijalna curenja tokom životnog veka procesa: blokovi koji se dodeljuju se samo akumuliraju i oslobađaju se tek na kraju izvršavanja.
- prekomerni promet: tačke koje zauzmu veliki deo hipa, čak i ako ga ne zadržavaju jako dugo.
- prekomerno prolazne: tačke koje dodeljuju vrlo kratkotrajne blokove.
- beskorisne ili nedovoljno iskorišćene alokacije: blokovi koji su dodeljeni, ali nisu u potpunosti popunjeni, ili su popunjeni, ali se iz njih naknadno nije čitalo.
- blokovi sa neefikasnim rasporedom - segmenti kojima se nikada nije pristupilo ili sa poljima raštrkanim po bloku.

Slično kao kod alata *Callgrind* i *Cashegrind*, da bi uspešno izvršili analizu alatom *DHAT* neophodno je da kompajlirati program sa opcijom za debug mod (opcija -g) i uključenom optimizacijom, jer nema smisla profajlirati kod koji je drugačiji od onoga koji će se normalno izvršavati. Nakon kompilacije, morate pokrenuti svoj program pod *DHAT*-om da biste prikupili informacije o profajliranju. Možda ćete morati da smanjite vrednost *-num-callers* da biste dobili izlazne datoteke razumne veličine, posebno ako profajlirate veliki program. Na kraju, trebate da koristite *DHAT*-ov prikazivač (u veb pregledaču) da biste dobili detaljnu prezentaciju tih informacija.

Da bismo pokrenuli alat *DHAT* neophodno je navesti *-tool=dhat* u komandnoj liniji *Valgrind*-a.

*valgrind -tool=dhat [argumenti dhata] program [argumenti programa]*

Analiza programa alatom *DHAT* će se izvršavati prilično sporo, a po završetku, šampaće se sažeti statistički podaci, kao što je prikazano na slici 2.23.

```
==11514== Total:      823,849,731 bytes in 3,929,133 blocks
==11514== At t-gmax: 133,485,082 bytes in 436,521 blocks
==11514== At t-end:   258,002 bytes in 2,129 blocks
==11514== Reads:     2,807,182,810 bytes
==11514== Writes:    1,149,617,086 bytes
```

Slika 2.23: Primer izlaza alata *DHAT*, izvor: [4]

Prva linija pokazuje koliko je blokova hipa i bajtova alocirano tokom celog izvršenja. Drugi red pokazuje koliko blokova i bajtova hipa je bilo živo u trenutku

$t-gmax$ , tj. u trenutku kada je veličina hipa dostigla svoj globalni maksimum (mereno bajtovima). Treći red pokazuje koliko je blokova i bajtova hipa bilo živo u trenutku  $t-end$ , odnosno na kraju izvršenja. Drugim rečima, daje informaciju koliko blokova i bajtova nije izričito oslobođeno. Četvrti i peti red pokazuju koliko je bajtova unutar blokova hipa pročitano i napisano tokom čitavog izvršavanja.

Ove linije su u najboljem slučaju umerenog značaja. Korisnije informacije mogu se videti sa *DHAT*-ovim prikazivačem. To omogućava činjenica da pored štampanja rezimea, *DHAT* u datoteku upisuje i detaljnije informacije o profajljanju. Podrazumevano se ova datoteka naziva *dhat.out.<pid>*, gde je *<pid>* ID procesa programa, ali njeno ime se može promeniti pomoću opcije *-dhat-out-file*. Ova datoteka je u JSON formatu i namenjena je pregledu *DHAT*-ovog pregledača. Treba imati na umu da ova datoteka može biti dosta velika u zavisnosti od veličine aplikacije. *DHAT*-ov prikazivač se može pokrenuti u veb pregledaču učitavanjem datoteke *dh\_viev.html*. Pomoću dugmeta "Učitaj" (eng. load) neophodno je odabrati *DHAT* izlaznu datoteku za prikaz. Ako učitavanje traje dugo, preporučuje se ponovno pokretanje *DHAT*-a sa manjom vrednošću *-num-callers* da bi se smanjile dubine steka, jer to može značajno smanjiti veličinu izlaznih datoteka *DHAT*-a [4].

## 2.10 BBV

Osnovni blok je linearni presek koda sa jednom ulaznom tačkom i jednom izlaznom tačkom. Vektor osnovnog bloka (eng. basic block vector - BBV) je lista svih osnovnih blokova unetih tokom izvršavanja programa i broj koliko je puta svaki osnovni blok pokrenut.

*BBV* je alat koji generiše osnovne blok vektore za upotrebu sa alatom za analizu *SimPoint*. *SimPoint* metodologija omogućava ubrzavanje arhitektonskih simulacija pokretanjem samo malog dela programa i ekstrapolacijom ukupnog ponašanja iz ovog malog dela. Većina programa pokazuje ponašanje zasnovano na fazama, što znači da će u različito vreme tokom izvršavanja program naići na vremenske intervale u kojima se kod ponaša slično prethodnom intervalu. Na kraju, to znači da ako možemo da detektujemo ove intervale i grupišemo ih, približna vrednost ukupnog ponašanja programa može se dobiti samo simulacijom najmanjeg broja intervala, a zatim skaliranjem rezultata. U istraživanju računarske arhitekture, pokretanje testova na simulatoru sa tačnošću ciklusa može prouzrokovati usporavanje reda veličine oko 1000 puta, što znači da su potrebni dani,

nedelje ili čak duže da se izvrše kompletni testovi. Korišćenjem *SimPoint*-a ovo se može značajno smanjiti, obično za 90-95%, uz zadržavanje razumne tačnosti.

Da bismo kreirali datoteku osnovnog vektorskog bloka, neophodno je u komandnoj liniji *Valgrind*-a navesti opciju *-tool=exp-bbv*.

```
valgrind -tool=exp-bbv [argumenti bbva] program [argumenti programa]
```

Podrazumevano će biti kreirana datoteka zvana *bb.out.PID*, gde je PID zamenjen ID-om procesa koji se izvršava. Ova datoteka sadrži vector osnovnog bloka. Za programe koji se dugo izvršavaju ova datoteka može biti prilično velika, pa bi bilo pametno da je kompresujete sa *gzip*-om ili nekim drugim programom za kompresiju. Da biste kreirali stvarne *SimPoint* rezultate, biće vam potreban uslužni program *SimPoint*, dostupan na *SimPoint* veb stranici. Pod pretpostavkom da ste preuzeli *SimPoint 3.2* i kompajlirali ga, kreirajte *SimPoint* rezultate naredbom poput sledeće:

```
./SimPoint.3.2/bin/simpoint -inputVectorsGzipped -loadFVFile bb.out.1234.gz  
-k 5 -saveSimpoints results.simpts -saveSimpointWeights results.weights,
```

gde je *bb.out.1234.gz* kompresovana datoteka vektora osnovnog bloka koju generiše *BBV*.

Uslužni program *SimPoint* vrši slučajnu linearnu projekciju pomoću 15 dimenzija, a zatim k-dimenziono grupisanje kako bi izračunao koji intervali su od interesa. U ovom primeru definisano je 5 intervala pomoću opcije *-k 5*. Izlazi nakon završetka rada *SimPoint*-a su datoteke *results.simpts* i *results.weights*. Prva sadrži 5 najrelevantnijih intervala programa. Druga sadrži težinu za skaliranje svakog intervala prilikom ekstrapolacije ponašanja u celom programu. Intervali i težine mogu se koristiti zajedno sa simulatorom koji podržava brzo premotavanje unapred. To u bukvalnom smislu znači da premotavate unapred do intervala od interesa, prikupljate statistiku za željenu dužinu intervala, a zatim koristite statistike prikupljene zajedno sa težinama za izračunavanje rezultata [1].

## Glava 3

# Regularni izrazi

Računari su mašine koje vrše transformaciju informacija iz jednog oblika u drugi. Te informacije se računaru predstavljaju u posebnom formatu označenim kao podatak, a sam računar će zadatu transformaciju nad informacijama obaviti bez razumevanja značenja same informacije, kao ni transformacije koja se nad njom vrši.

U skladu sa izrečenim zaključujemo da je program specifikacija transformacije koju računar izvršava, dok je programski jezik notacija koja omogućava zapisivanje programa. Program kao specifikacija transformacije koja će biti izvršena na računaru definiše šta će program raditi, ali ne i način na koji će to uraditi. Jasnno, jedan program može biti napisan različitim notacijama, odnosno različitim programskim jezicima. Uloga kompilatora jeste da transformiše tekst programa iz reprezentacije na visokom programskom jeziku (eng. high-level programming language) u evivalentnu reprezentaciju na mašinskom jeziku, tj. jeziku razumljivom računaru u cilju izvršavanja samog programa. U tom smislu, kompilacija se sastoji u čitanju niske karaktera koja je sastavljena u skladu sa pravilima, a sa ciljem da se generiše druga reprezentacija informacije koju ti karakteri predstavljaju.

Program napisan na nekom programskom jeziku nazivamo izvornim kodom (eng. source code), dok rezultat rada kompilatora nazivamo objektnim kodom (eng. object code). Objektni kod može biti izražen na mašinskom jeziku, na assembleru ili na nekom drugom programskom jeziku [16].



## 3.1 Proces kompilacije

Transformacija teksta programa iz izvornog u objektni kod, koju izvršava kompilator, naziva se proces kompilacije.

Proces kompilacije obično se razlaže na dve etape:

1. etapa analize izvornog koda – tokom ove etape izvorni program se prevodi u odgovarajuću posrednu reprezentaciju koja prikazuje strukturu izvornog programa u obliku drveta.
2. etapa sinteze izvornog koda – tokom ove etape se koristeći rezultate iz prethodne etape na osnovu posredne reprezentacije generiše objektni kod.

Svaka od etapa se kasnije deli na faze. U ovom tekstu biće detaljnije pojašnjena analiza izvornog koda, sa akcentom na regularnim izrazima, na kome ovaj rad počiva.

Tokom analize, kompilator analizira izvorni program da bi odredio njegovu strukturu i značenje. Etapa analize izvornog koda se razlaže na tri posebne faze:

1. leksička analiza
2. sintaksička analiza
3. semantička analiza

Leksička analiza je faza u kojoj se karakteri iz ulazne struje karaktera rastavljaju na leksičke jedinice ili lekseme analiziranog jezika. Lekseme su minimalni, nedeljivi konstituenti jezika. Sintaksička analiza je faza u kojoj se ispituje i rekonstruiše struktura programa kao celine na osnovu niza tokena koje prosleđuje leksički analizator. Sintaksičkom analizom se utvrđuje da li je program saglasan sa gramatičkim pravilima programskog jezika, ali ne i njegova semantička konzistentnost. Opis semantike programskog jezika obično se zadaje polazeći od značenja koju pojedinačni iskazi programskog jezika vrše. Shodno tome se opis dejstva pojedinačnih iskaza nekog programskog jezika naziva semantika programskog jezika. Faza semantičke analize, koja sledi nakon sintaksičke analize, najčešće se sastoji u izvršavanju akcija koje nalaže sintaksički analizator, a koje se sastoje u kreiranju i ažuriranju informacija u tabeli simbola [16].

## 3.2 Teorija jezika

Da bi ispitivanje izvornog koda bilo moguće, potrebno je precizno definisati izvorni jezik. Jedna klasa jezika, koja je od značaja za leksičku analizu programskih jezika, opisuje se posebnom notacijom, tj. regularnim izrazima.

Definicija izvornog jezika mora obuhvatiti opis alfabeta ili azbuke jezika, tj. konačan skup simbola čija je upotreba dozvoljena u jeziku, a zatim i opis svih onih niski simbola (reči) koje pripadaju tom jeziku uključujući i njihovo značenje, odnosno sintaksu i semantiku jezika.

Neka je  $\Sigma$  konačan skup. Niska (eng. string) ili reč (eng. word) nad  $\Sigma$  je svaki konačan niz

$$x = (a_1, a_2, \dots, a_n),$$

gde je  $n \geq 0$ , a  $a_i \in \Sigma$  za svako  $i \in [1, n]$ . Broj  $n$  se naziva dužina reči i obeležava se sa  $|x|$ . Ako je  $n = 0$ , reč se naziva prazna niska ili prazna reč, i obeležava se simbolom  $\varepsilon$ , odnosno važi  $|\varepsilon| = 0$ . Ako je

$$y = (b_1, b_2, \dots, b_m)$$

neka druga reč nad  $\Sigma$ , tada je proizvod dopisivanja, tj. konkatencije reči  $x$  i  $y$  reč

$$xy = (a_1, a_2, \dots, a_n, b_1, b_2, \dots, b_m).$$

Operacija dopisivanja je asocijativna. Skup  $\Sigma$  nazivamo alfabet ili azbuka, a njegove elemente slova ili simboli.

Skup svih niski nad azbukom  $\Sigma$  obeležava se sa  $\Sigma^*$ . Ovaj skup se naziva Klinijevo zatvorenje azbuke  $\Sigma$ . Na primer, ukoliko je  $\Sigma = \{a, b\}$ , onda je

$$\Sigma^* = \varepsilon, a, b, aa, ab, ba, bb, \dots$$

Prazna reč  $\varepsilon$  je neutralni element u odnosu na dopisivanje, tj. za  $\forall x \in \Sigma, x\varepsilon = \varepsilon x = x$ . Skup svih reči bez prazne reči označava se kao  $\Sigma^+$ , tj.  $\Sigma^+ = \Sigma^* - \varepsilon$ . Na skupu  $\Sigma^*$  nije definisan pojam inverznog elementa, tj. ako  $x \in \Sigma^+$ , onda ne postoji  $y \in \Sigma^+$ , tdj.  $xy = \varepsilon$ .

Elementi azbuke  $\Sigma$  se ne mogu razložiti na druge elemente, tj. za bilo koje  $a \in \Sigma$ , ne postoje  $b, c \in \Sigma^+$ , takvi da  $a = bc$ , iz čega sledi da se bilo koja reč  $x \in \Sigma^*$  rastavlja na jedinstven način na elemente azbuke  $\Sigma$ . Imajući u vidu ova svojstva zaključujemo da je  $\Sigma^*$  slobodni monoid sa skupom generatora  $\Sigma$ .

**Lema 1.** (Levijsva lema) Neka su  $t, u, v, w \in \Sigma^*$ . Ako je  $tu = vw$ , onda  $\exists!$  reč  $z \in \Sigma^*$ , tako da važi

- ili je  $t = vz$  i  $zu = w$
- ili je  $v = tz$  i  $zw = u$ .

Neka je  $v \in \Sigma^*$ . Za reč  $u \in \Sigma^*$  se kaže da je faktor reči  $v$  ako postoje reči  $v_1, v_2 \in \Sigma^*$ , takve da je  $v = v_1uv_2$  i to:

- levi faktor ili prefiks, ako je  $v = uv_2$
- desni faktor ili sufiks, ako je  $v = v_1u$
- pravi faktor ili infiks, ako je  $v = v_1uv_2, v_1 \neq \varepsilon, v_2 \neq \varepsilon$ .

Faktorizacija jedne reči predstavlja zapis te reči kao proizvod dopisivanja faktora. Podreč reči  $v \in \Sigma^*$  je svaki podniz slova koja čine reč  $v$ .

**Definicija 1.** (Formalni) jezik nad azbukom  $\Sigma$  je bilo koji podskup skupa  $\Sigma^*$ :  $L \subset \Sigma^*$ . Ako niska  $x \in L$ , onda  $x$  predstavlja rečenicu jezika  $L$ .

Neka je data azbuka simbola  $\Sigma = \{a, b, c\}$ . Nad ovom azbukom, neki od jezika predstavljeni su sledećim skupovima:

$L_1 = \{a_n | n > 0\}$  – jezik koji se sastoji od niski u kojima se proizvoljan broj puta pojavljuje simbol  $a$ .

$L_2 = \{a_nb_m | m, n > 0\}$  – jezik koji se sastoji od niski u kojima se proizvoljan broj puta pojavljuje simbol  $a$  iza čega sledi pojavljivanje simbola  $b$  proizvoljan broj puta.

Jezici mogu biti konačni ili beskonačni. Konačni jezici se najjednostavnije definišu eksplicitnim navođenjem niski koje u njima predstavljaju rečenice. Problem je kako opisati na konačan način rečenice koje pripadaju jeziku koji nije konačan. Tu se dolazi do opšteg problema svakog jezika, a to je kako za datu nisku  $x \in \Sigma^*$  odrediti da li je ona rečenica jezika  $L$  ili ne. Najopštiju klasu algoritama koja rešava ovaj problem opisuju Tjuringove mašine. Prevelika opštost ovog mehanizma ne dozvoljava da se izrazi specifičnost koju zahtevaju programski jezici. Uvodeći razna ograničenja, rešavanje problema pripadanja niske  $x$  jeziku može se realizovati na znatno efikasniji način. Iz ugla kompilacije programskih jezika, dve klase takvih jezika su regularni jezici i kontekstno slobodni jezici.

Pre nego što uvedemo formalnu priču o regularnim izrazima, pomenućemo neke od skupovnih operacijama koje se mogu primeniti nad jezicima. Nad jezicima se mogu izvršiti operacije unije, preseka i razlike, kao i operacija proizvoda [16].

**Definicija 2.** Proizvod jezika  $L_1$  i  $L_2$  nad azbukom  $\Sigma$  u oznaci  $L_1L_2$  je jezik

$$L_1L_2 = \{xy | x \in L_1, y \in L_2\}.$$

Proizvod jezika je asocijativna operacija, a  $\varepsilon$  je njen neutralni element.

**Definicija 3.** Za jezik  $L$ ,  $n$ -ti stepen jezika  $L$  je jezik:

$$L_0 = \{\varepsilon\}$$

$$L_1 = L$$

$$L_n = LL_{n-1}, \text{ za } n > 1.$$

**Definicija 4.** Iteracija ili (Klinijevo) zatvorenje jezika  $L$ , u oznaci  $L^*$ , je jezik:

$$L^* = \bigcup_{n \geq 0} L^n.$$

Pozitivno zatvorenje jezika  $L$ , u oznaci  $L^+$ , je jezik:

$$L^+ = \bigcup_{n \geq 1} L^n = L^* - \{\varepsilon\}.$$

### 3.3 Regularni izrazi i jezici

**Definicija 5.** Regularni ili racionalni izrazi nad azbukom  $\Sigma$  opisuju se rekursivno na sledeći način:

1. Prazan skup je regularni izraz koji se prikazuje simpolom  $\emptyset$ ;
2. Regularni izraz  $\varepsilon$  predstavlja jezik  $\{\varepsilon\}$ ;
3. Ako je  $a \in \Sigma$ , onda regularni izraz  $a$  predstavlja jezik  $\{a\}$ ;
4. Ako su  $p$  i  $q$  regularni izrazi jezika  $L(p)$  i  $L(q)$ , onda je:
  - $(p + q)$  regularni izraz koji predstavlja jezik  $L(p) \cup L(q)$ .
  - $(pq)$  regularni izraz koji predstavlja jezik  $L(p)L(q)$ .

- $(p)^*$  regularni izraz koji predstavlja jezik  $(L(p))^*$ .
- $(p)$  regularni izraz koji predstavlja jezik  $L(p)$ .

Poslednje pravilo ukazuje na to da se zagrade mogu izostaviti iz regularnog izraza, a da se pritom opisani jezik ne promeni. Prilikom izostavljanja zagrada, treba znati činjenicu da su operatori  $*$ ,  $+$  i konkatencija levo asocijativni, a da im je prioritet određen sa:

$$\text{prioritet}(*) > \text{prioritet}(\text{konkatencija}) > \text{prioritet}(+).$$

Operacije sa regularnim izrazima nad jezikom  $\Sigma$  imaju sledeća svojstva:

- Operacija  $+$  je idempotentna, asocijativna i komutativna;
- Operacija konkatencije je asocijativna i distributivna u odnosu na  $+$ ;
- Sledeće jednakosti važe za svaki regularni izraz:

$$\emptyset + p = p + \emptyset = p$$

$$\varepsilon p = p\varepsilon = p$$

$$\emptyset p = p\emptyset = \emptyset;$$

- $\emptyset^* = \varepsilon^* = \varepsilon$ .

**Definicija 6.** Za jedan jezik se kaže da je regularan ukoliko se može predstaviti regularnim izrazom.

Na osnovu gore navedenih svojstava regularnih izraza, skup regularnih izraza sa uvedenim operacijama predstavlja jednu algebru koju ćemo obeležiti sa  $I(\Sigma)$ . Važnost ove notacije ogleda se u tome što se slovo azbuke, reč koja se sastoji od tog slova, regularni izraz koji predstavlja to slovo, i na kraju jezik predstavljen regularnim izrazom beleže na isti način.

Preslikavanje  $L$  sa algebre  $I(\Sigma)$  u skup regularnih izraza nad  $\Sigma$

$$L : I(\Sigma) \rightarrow P(\Sigma^*)$$

definisano je sledećim rekursivnim pravilima:

$$L(\emptyset) = \emptyset, L(\varepsilon) = \varepsilon, L(a) = \{a\},$$

$$L(p + q) = L(p) \cup L(q), L(pq) = L(p)L(q), L(p^*) = L(p)^*.$$

Kažemo da je za neki regularni izraz  $p$ , jezik  $L(p)$  jezik opisan regularnim izrazom  $p$ . Dva regularna izraza  $p$  i  $q$  su ekvivalentna, u oznaci  $p \approx q$ , ako opisuju isti jezik, tj. ako važi  $L(p) = L(q)$ . O tome govori naredna teorema.

**Teorema 3.1.** *Za bilo koje regularne izraze  $p$  i  $q$  važi sledeća ekvivalencija:*

$$(pq)^* = \varepsilon + p(qp)^*q$$

Kao što smo se već uverili, regularni izrazi se grade polazeći od drugih regularnih izraza. To može dovesti do nepreglednosti njihovog zapisa, što rešavamo uvođenjem regularnih definicija koje omogućuju da se određenim regularnim izrazima dodeli ime. Regularne definicije zapisujemo u obliku:

$$D_1 \rightarrow R_1$$

$$D_2 \rightarrow R_2$$

...

$$D_n \rightarrow R_n,$$

gde je svako  $d_i$  niska nad azbukom koja je disjunktna sa azbukom  $\Sigma$ , različita od  $d_1, d_2, \dots, d_{i-1}$ , a svako  $r_i$  regularni izraz nad azbukom  $\Sigma \cup \{d_1, d_2, \dots, d_{i-1}\}$ .

Radi pojednostavljenja zapisivanja regularnih izraza, definicija regularnih izraza se dopunjuje dodavanjem sledećih konvencija:

- Neka je  $r$  regularni izraz koji opisuje jezik  $L(r)$ . Tada je  $(r)^+$  regularni izraz koji opisuje jezik  $(L(r))(L(r))^*$ , a  $(r)?$  regularni izraz koji opisuje jezik  $L(r) \cup \{\varepsilon\}$ ;
- Ako su  $c_1, c_2, \dots, c_n$  karakteri, tada se regularni izraz  $c_1 + c_2 + \dots + c_n$  može obeležiti sa  $[c_1, c_2, \dots, c_n]$ . Izraz  $[c_1 - c_2]$  označava sekvencu svih karaktera  $c$  tdj.  $c_1 \leq c \leq c_2$ .

Tako na primer slova engleske abecede označavamo sa  $[A - Z a - z]$ , dok cifre označavamo sa  $[0 - 9]$ , itd..

Klasa jezika koji se mogu opisati regularnim izrazima je dovoljna da se opišu lekseme jednog programskog jezika. Treba imati u vidu da se složeniije konstrukcije u programskom jeziku ne mogu izraziti regularnim izrazima [16].

## 3.4 Modul *re*

U današnje vreme, programski jezici imaju veoma dobru podršku za rad sa regularnim izrazima kao i da imaju širok dijapazon upotrebe. Alat koji ovaj rad prati razvijen je u programskom jeziku Python, a *re* je modul koji pruža podršku za rad sa regularnim izrazima u Python programima.

Ovaj modul pruža operacije podudaranja regularnih izraza. Važno je napomenuti da je većina operacija sa regularnim izrazima dostupna u vidu funkcija i metoda na nivou modula sa kompajliranim regularnim izrazima. Funkcije su prečice koje ne zahtevaju da prvo kompajlirate objekat regularnog izraza, ali je mana što im nedostaju neki parametri za fino podešavanje.

Regularni izraz (ili RE) navodi niz karaktera koji mu odgovara, dok vam funkcije u ovom modulu omogućavaju da proverite da li se određeni niz karaktera sa datim regularnim izrazom (ili da li se regularni izraz podudara sa određenim nizom karaktera, što se svodi na istu stvar). Neke od funkcija koje ovaj modul pruža, a koji su iskorišćeni u alatu koji ovaj rad prati su sledeći:

- *re.search(pattern, string, flags=0)* - Skenira kroz *string* tražeći prvu lokaciju na kojoj obrazac regularnog izraza daje podudaranje i vraća odgovarajući objekt podudaranja. Vraće *None* ako nijedna pozicija u nizu karaktera ne odgovara obrascu. Imajte na umu da se ovo razlikuje od pronalaženja podudaranja nulte dužine u nekom trenutku niza.
- *re.match(pattern, string, flags=0)* - Ako se nula ili više znakova na početku niza karaktera *string* podudara sa regularnim izrazom, vraće odgovarajući objekt podudaranja. Vraće *None* ako se niz ne podudara sa obrascem.
- *re.split(pattern, string, maxsplit=0, flags=0)* - Podeliće nisku *string* pojavljivanjima koja se poklapaju sa *pattern*.
- *re.findall(pattern, string, flags=0)* - Vraća sva podudaranja *pattern*-a u *string*-u kao listu stringova, tj. niski. Niska u kojoj se traže preklapanja - *string*, skenira se s leva na desno, a poklapanja se vraćaju u redosledu u kom se i javljaju.
- *re.sub(pattern, repl, string, count=0, flags=0)* - Vraća string dobijen zamenu krajnjeg levog poklapanja sa *pattern*-om u *string*-u sa *repl*. Ako

poklapanje nije pronađeno, *string* se vraća nepromenjen. *Repl* može biti niz ili funkcija.

Kompajlirani objekti regularnog izraza podržavaju slične metode i attribute, s tim da je pozivanje istih u formi

*Pattern.funkcija(argumenti).*

Objekti podudaranja uvek imaju logičku vrednost *True*. Budući da *match()* i *search()* vraćaju *None* kada nema podudaranja, možete testirati da li je postojalo podudaranje jednostavnim *if* izrazom.

Objekti podudaranja podržavaju mnoge metode i attribute, a najiskorišćenija u alatu koji se razvija, a u cilju izvlačenja potrebnih informacija iz poklapanja sa regularnim izrazom je:

- *Match.group([group1, ...])* - Vraća jednu ili više podgrupa podudaranja. Ako postoji jedan argument, rezultat je jedan string, dok ako postoji više argumenata, rezultat je t-orka sa jednom stavkom po argumentu. Bez argumenata, *group1* je podrazumevano nula, tj. vraća se ceo *Match* objekat. Ako je indeks *group* u inkluzivnom opsegu [1..99], to je string koji odgovara odgovarajućoj grupi u zagradama na toj poziciji. Ako je broj grupe negativan ili veći od broja grupa definisanih u obrascu, biće izbačen izuzetak *IndexError*. Ako je grupa sadržana u delu uzorka koji se ne podudara, odgovarajući rezultat je *None*. Ako je grupa sadržana u delu uzorka koji se podudara više puta, vraća se poslednje podudaranje [11].

Primer korišćenja funkcije *group* dat je na slici 3.1

```
>>> import re
>>> m = re.match(r"(\w+) (\w+)", "Petar Petrovic, student")
>>> m.group(0)
'Petar Petrovic'
>>> m.group(1)
'Petar'
>>> m.group(2)
'Petrovic'
>>> m.group(1,2)
('Petar', 'Petrovic')
>>> ■
```

Slika 3.1: Primer korišćenja funkcije *group* modula *re*



# Glava 4

## Alat Koronka

Alat *Koronka* predstavlja alat koji automatski otkriva greske u kodu napisanom u C-u koristeći alat *Valgrind*-ove distribucije *Memcheck*, a zatim ih ispravlja ukoliko je to u njegovoj moći. Razvijan pod *Linux* okruženjem, a implementiran u programskom jeziku *Python*. Alat je nazvan *Koronka*, zbog situacije u kojoj se ceo svet našao početkom 2020. godine. Osnovna svrha alata je demonstracija rada alata *Valgrind*, kao i tumačenje izveštaja o greškama koje *Valgrind* daje i njihovo uspešno otklanjanje. Alat je slobodno dostupan i nalazi se na linku [https://github.com/LMladenovic/Error\\_fixing\\_tool](https://github.com/LMladenovic/Error_fixing_tool). Na pomenutom linku se nalaze neophodne datoteke alata, opis sistema, kao i skup test primera i njihova pokretanja.

### 4.1 Korišćenje alata

Da biste pokrenuli alat *Koronka* neophodno je da prethodno instalirate *GCC* kompajler, sam alat *Valgrind*, kao i *Python*. Alat se pokreće sledećom komandom:

```
python koronka.py [files=[list of files]] [structures=[list of user defined  
structures]] c file/path to c file [other arguments] .
```

Navedeni arumenti podrazumevaju sledeće:

- *files* - sadrži dodatne fajlove glavnog programa (*c file*) koji se analizira (npr. *.h* fajlove). Ukoliko se vaš program sastoji iz više fajlova, neophodno je da ih navedete kao argument. Bitni su iz razloga sto će *Koronka* svo vreme svog rada imati niz *files* u memoriji koji će imati podatak o fajlovima nad kojima

može da vrši promene. Ako se ne navede argument *files*, niz *files* će sadržati samo glavni program (*c file*) i biće u mogućnosti da vrši promene samo nad njim.

- *structures* - predstavlja korisnički definisane strukture u okviru programa. Kako alat podrazumevano radi samo sa primitivnim tipovima podataka, ukoliko u programu postoje korisnički definisane strukture, a ne navedu se kao argument, *Koronka* neće biti u mogućnosti da ispravi greške povezane sa tim strukturama, naravno, ukoliko postoje.
- *program args* - argumenti programa koji se analizira.

Takođe, alat *Koronka* sadrži i opciju *-help* čijim će navođenjem na komandnu liniju biti ispisano uputstvo za upotrebu alata.

Sve argumente koji se navedu prilikom pokretanja alata *Koronka* obradiće funkcija za obrađivanje argumenata komandne linije i rasporediće sve podatke tamo gde treba. Nakon pokretanja alata, u direktorijumu alata će se kreirati direktorijum u formatu *datum pokretanja-vreme pokretanja*. U njega će se kopirati svi podaci navedeni kao argument *files*, kao i glavni program nad kojim se vrši analiza i nad njima će biti vršene odgovarajuće promene u skladu sa otkrivenim greškama. Originalni fajlovi ostaće nepromenjeni, tamo gde su i perzistirani. Po završetku rada alata, u pomenutom direktorijumu će se naći još dva fajla. Prvi je *ExecutionReport* koji sadrži detaljan izveštaj o radu alata, koje greške je našao, na koji način ih je rešio, koje fajlove, odnosno linije koda u njima je izmenio. Drugi, pod nazivom *ValgrindLOG*, predstavlja izlaz alata *Valgrind*, koji će se u procesu rada alata koristiti za parsiranje grešaka koje nađe alat *Memcheck*. Ukoliko analizirate ispravljeni program nakon završetka rada alata alatom *Memcheck*, dobićete isti izlaz koji možete videti u datoteci *ValgrindLOG*. Način i logika rada alata biće opisani u sekcijama koje slede.

## 4.2 Klasa greške

Prilikom ispravljanja grešaka, greška koja se trenutno obrađuje biće predstavljena kao objekat klase *ErrorInfo*. Na slici 4.1 je predstavljen potpis pomenute klase.

Podaci koje klasa sadrži su sledeći:

- *errorType* - sadrži informaciju o tipu greške, npr. nevalidno oslobađanje memorije, nedozvoljeno čitanje/pisanje itd..
- *valgrindOutput* - sadrži kompletan sadržaj sa slike 4.3, za potrebe dodatnog parsiranja informacija o grešci.
- *files* - sadrži niz fajlova koji su povezani sa greškom, a koji bi eventualno bili ispravljani.
- *changedFile* - sadrži ime fajla koji će biti promenjen.
- *changedLine* - sadrži tačnu liniju koda u fajlu *changedFile* koja će biti promenjena.
- *problemLines* - sadrži linije u kodu koje su izazvale grešku koja se ispravlja.
- *errorReason* - sadrži razlog koji je izazvao grešku, npr. upotreba neinicijalizovane vrednosti sa steka, odnosno hipa.
- *bug* - sadrži bag u kodu, tj. sadržaj linije koda koji je izazvao grešku.
- *bugFix* - sadrži ispravku бага, tj. sadržaj kojim treba zameniti *changedLine* da bi se greška uspešno ispravila.

```
class ErrorInfo:

    # Initialisation with errorType
    def __init__(self, errorType, valgrindOutput, files):
        self.errorType = errorType
        self.valgrindOutput = valgrindOutput
        self.files = files
        self.changedFile = ''
        self.changedLine = -1
        self.problemLines = []
        self.errorReason = []
        self.bug = ''
        self.bugFix = ''
```

Slika 4.1: Potpis klase *ErrorInfo*

Za sve attribute klasa sadrži odgovarajuće *get*-ere i *set*-ere, kao i funkciju koja će proveravati da li je razlog greške poslat kao arument, zaista razlog koji je izazvao grešku koja se obrađuje. Prikaz funkcije dat je na slici 4.2.

```
def isKnownReason(self, newReason):  
    for reason in self.errorReason:  
        if reason.find(newReason) >= 0:  
            return True  
            break  
  
    return False
```

Slika 4.2: Prikaz funkcije koja proverava validnost razloga greške

Na slici 4.3 prikazan je segment izveštaja alata *Valgrind* koji predstavlja jednu grešku koju treba ispraviti. Žutom bojom je obeležen deo koji predstavlja tip greške, zelenom bojom deo koji predstavlja razlog greške, dok su plavom bojom obeležene sumnjive(problematične) linije. Ti podaci će redom biti smešteni u *errorType*, *errorReason* i *problemLines*. Sve potrebne informacije od značaja iz datog iz datog segmenta izveštaja dobićemo upotrebom regularnih izraza.

```
Conditional jump or move depends on uninitialised value(s)  
at 0x48E2E40: __vfprintf_internal (vfprintf-internal.c:1644)  
by 0x48CD8D7: printf (printf.c:33)  
by 0x109162: main (1.c:7)  
Uninitialised value was created by a stack allocation  
at 0x109145: main (1.c:5)
```

Slika 4.3: Prikaz izveštaja za jednu grešku

Na osnovu *errorType* i *errorReason* biće određeno koji će šablon biti upotrebljen za ispravljanje greške. Na osnovu *problemLines* biće pronađena tačna linija koda koja je izazvala grešku, a iz nje izvučeni podaci koji su neophodni za ispravljanje greške, kao i iz ostalih linija ukoliko sadrže informacije od značaja. Kada se neophodni podaci o grešci izvuku i nađe njeno rešenje, biće postavljene vrednosti *changedFile*, *changedLine*, *bug*, *bugFix*. Kada se obezbede svi podaci, dalje izvršavanje alata prepušta se sistemu za praćenje istorije promena koji je opisan u 4.3.

## 4.3 Mehanizam praćenja istorije

Sve promene koje je alat *Koronka* izvršio nad fajlovima biće praćene pomoću mehanizma za praćenje istorije. Svaka promena biće sačuvana pojedinačno u nizu

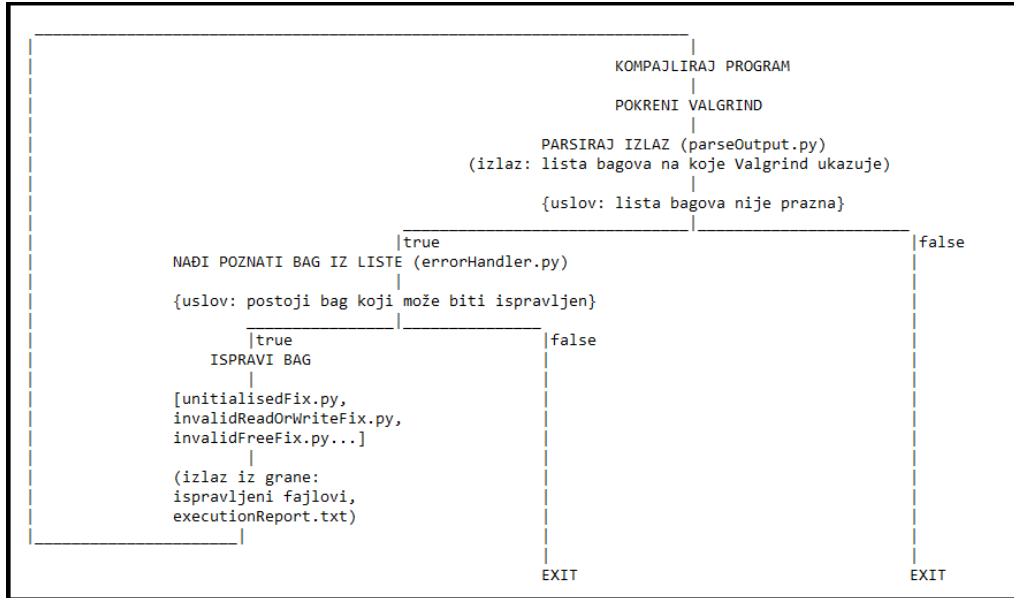
*history* u vidu uređene trojke (*izmena u kodu, promenjena linja koda, fajl nad kojim se vrši promena*). Ovakav vid praćenja istorije je dobar iz više razloga. Na primer, može se desiti da imamo istu liniju koda koja izaziva grešku koja se nalazi u dva različita fajla. Promena tih linija bez praćenja fajla nad kojim se vrši promena neće biti moguća u oba fajla, kako je ispravka greške opet ista linija koda sa određenim dodacima. Takođe, može se desiti na primer da u istom fajlu imamo dva bloka unutar kojih postoji linija koja izaziva problem. To su dakle dve greške, čije će rešenje biti dve potupuno iste linije koda, izmenjene u dve različite linije koda u istom fajlu. Ukoliko mehanizam praćenja istorije ne pamti i liniju koda u kojoj su promene izvršene, to neće biti moguće.

Prilikom pokretanja alata niz *history* sadrži samo jednu trojku, tj. ( ' ', -1, ' '). Kako se greške iterativno ispravljaju, svaka iteracija predstavlja pokušaj alata da datu grešku ispravi, što je bliže opisano u sekciji 4.4. Za mehanizam praćenja istorije je to važno jer će on biti taj koji će validirati trenutni predlog za ispravku greške koje je alat našao. Prvi slučaj je da se predlog ispravke ne nalazi u istoriji, on će biti implementiran i dodat isoriji izmena u nadi da će se greška uspešno ispraviti. U drugom slučaju, predlog ispravke se već nalazi u istoriji izmena, što je signal alatu da je predlog rešenja već implementiran i da ta implementacija nije ispravila grešku, te da alat nastavi sa traženjem optimalnog rešenja. U teoriji, ukoliko alat iskoristi sve mehanizme za ispravljanje određene greške koje poseduje, tj. implementira sve predloge rešenja koje je u mogućnosti da predloži, a greška ostane neispravljena, alat će nastaviti sa radom i pokušati da ispravi ostale greške, ukoliko postoje, a navedena greška će ostati neispravljena, a korisnik će biti obavešten da postoji greška koja nažalost nije ispravljena. Jasno, prilikom nalaženja rešenja koje će biti zaista i biti implementirano, to rešenje biva dodato u istoriju, a alat nastavlja sa iteracijama ispravljanja ostalih grešaka.

## 4.4 Algoritam izvršavanja

Kao što je već rečeno, pri pokretanju alata generiše se odgovarajući direktorijum, u njega se kopiraju svi navedeni (potrebni) fajlovi, formiraju se nizovi *structures*, *files* i *history*. Nakon početnih podešavanja sledi iterativno kompilacija programa i analiza *Valgrind*-ovim alatom *Memcheck* čiji izlaz se smešta u datoteku *ValgrindLOG.txt*. Ovaj fajl se parsira, i iz njega se čitaju informacije o greškama iterativno. Kad alat naiđe na gresku, parsira je, nakon čega proba da je ispravi

ukoliko je u mogućnosti. Ukoliko nađe resenje koje već nije primenjeno, biće implementirano nakon čega alat ide u novu iteraciju kompilacije i provere grešaka, uz prethodno ažuriranje istorije i izveštaja o ispravljenim greškama. Proces se ponavlja dokle god ima grešaka koje nisu ispravljene, a koje je *Koronka* u mogućnosti da ispravi. Grafički prikaz algoritma dat je na slici 4.4.



Slika 4.4: Algoritam izvršavanja alata *Koronka*

## 4.5 Šabloni za ispravljanje grešaka

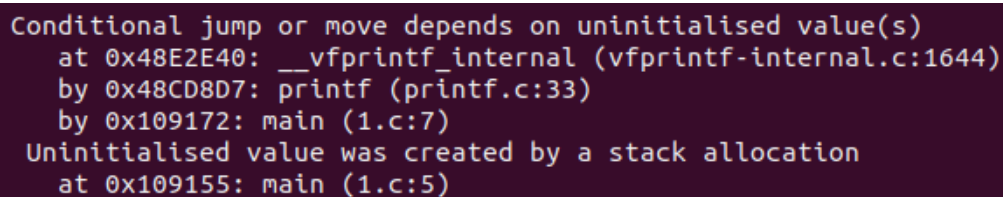
Alat *Koronka* sadrži nekoliko šablona od kojih svaki ispravlja određenu vrstu grešaka. Šablon koji će biti primenjen za ispravljanje date greške određen je vrstom greške kao i razlogom koji je datu grešku izazvao, što znači da su šabloni ekskluzivno mapirani tipom i razlogom greške. Bez obzira na optimizaciju koji *Memcheck* poseduje za ispisivanje izveštaja o grešci, i dalje ispisuje par paragrafa koji se odnose na istu grešku, sa različitim razlogom, odnosno tipom (pogledati npr. sliku 2.7). Razlog tome je taj što *Memcheck* prati izvršavanje od početka, odnosno prati indirektan put te greške. To dalje znači da ne moramo da mapiramo sve tipove i razloge grešaka, već samo one koje će nam dati dovoljno informacija za ispravljanje greške. Kada ispravimo grešku, iz izveštaja koji je sadržao par paragrafa za istu, nestaću svi paragrafi kako je greška otklonjena, a za njeno otklanjanje smo koristili samo neophodnu količinu podataka, a ne sve podatke koje smo

imali. Na taj način doprinosimo efikasnosti alata. Šabloni su smešteni u fajlove sa sufiksom *Fix*, dok u imenu sadrže i tip greške koji ispravljaju.

Kao što je već rečeno, šabloni su orijentisani prvenstveno prema tipu greške koju ispravljaju, pa će njihovo izlaganje biti realizovano kroz tipove grešaka koje ispravljaju.

## Korišćenje neinicijalizovane vrednosti

Šablon koji ispravlja ovaj tip grešaka mapiran je tipom *"Conditional jump or move depends on uninitialised value(s)"*, dok razlozi za grešku mogu biti *"Uninitialised value was created by a stack allocation"* i *"Uninitialised value was created by a heap allocation"*, u zavisnosti od toga da li je reč o statičkoj, odnosno dinamički alociranoj memoriji. Ispis alata *Valgrind* koji dobijamo za ovaj tip greške u slučaju statičke memorije, a na osnovu kog je ispravljamo, dat je na slici 4.5.



```
Conditional jump or move depends on uninitialised value(s)
at 0x48E2E40: __vfprintf_internal (vfprintf-internal.c:1644)
by 0x48CD8D7: printf (printf.c:33)
by 0x109172: main (1.c:7)
Uninitialised value was created by a stack allocation
at 0x109155: main (1.c:5)
```

Slika 4.5: Ispis greške korišćenja statičke neinicijalizovane promenljive

Iz datog izveštaja dobijamo informacije od važnosti za ispravljanje greške kao što su linije koda u kojima se koristi neinicijalizovana promenljiva, kao i liniju koda i fajl u kom počinje blok u kom je ta promenljiva definisana. Kako nemamo tačnu liniju koda u kojoj je promenljiva definisana, već blok, grešku ćemo ispraviti inicijalizovanjem svih neinicijalizovanih vrednosti u okviru tog bloka.

Analizom linija koda datog bloka poklapanjem sa regularnim izrazom  $([t]*)([a - zA - Z_+]) + ([a - zA - Z_+].*)$ ; dobijamo liniju koda gde imamo neinicijalizovanu definisanu promenljivu. Ona može biti i višedimenziona, pa pre inicijalizacije vršimo i tu proveru da bi je, ukoliko je to slučaj, adekvatno inicijalizovali. Iz rezultata dobijenih poklapanjem sa regularnim izrazima izvlačimo podatke poput tipa promenljive, imena, a u slučaju višedimenzionog niza još i dimenzije. Za inicijalizaciju promenljivih koristimo funkciju prikazanu na slici 4.6, dok u slučaju nizova, odnosno višedimenzionih nizova, koristimo pomoćnu funk-

ciju koja koristi petlje za inicijalizaciju, a koja se suštinski oslanja na funkciju sa slike 4.6.

```
def initialise(varType):
    initialisator = {
        'int': '0',
        'double': '0',
        'float': '0',
        'boolean': 'False',
        'char': '\\\\0\\'
    }

    if varType in initialisator:
        return initialisator[varType]
    else:
        return 'Invalid'
```

Slika 4.6: Funkcija inicijalizacije

U slučaju da je neinicijalizovana promenljiva korisnički definisana struktura, ukoliko je ona navedena kao argument komandne linije, alat će biti u mogućnosti da ispravi ovu grešku. Ispravljanje se sastoji u tome što će alat pronaći kako je ta struktura definisana, odnosno koji su njeni činioči. Koristeći odgovarajuću funkciju iz *userDefinedStructuresHandler*-a, u kome su definisane sve operacije nad korisničkim strukturama, inicijalizovaće celu strukturu, redom, inicijalizujući njene činioce, indirektno koristeći funkciju sa slike 4.6. Takođe, podržana je i inicijalizacija nizova, odnosno višedimenzionih nizova korisničkih struktura.

Nakon što smo opisali mehanizam inicijalizacije, ispravku greške formiramo tako što zamenimo problematičnu liniju koda linijom gde je promenljiva definisana i inicijalizovana, dok u slučaju nizova i višedimenzionih nizova, problematičnu liniju menjamo blokom koda koji se sastoji iz definicije promenljive nakon koje sledi inicijalizacija korišćenjem petlje, odnosno petlji u slučaju višedimenzionih nizova.

Ispis alata *Valgrind* koji dobijamo za ovaj tip greške u slučaju dinamički alocirane memorije, tj. upotrebe pokazivača, a na osnovu kog je ispravljamo, dat je na slici 4.7.

Za razliku od greške upotrebe neinicijalizovane promenljive koja je statički definisana, ovde, u izveštaju pored linija koda u kojima se koristi ta promenljiva dobijamo i informaciju o tačnoj liniji koda i fajlu u kom je promenljiva definisana.



```
Conditional jump or move depends on uninitialised value(s)
at 0x48E2E40: __vfprintf_internal (vfprintf-internal.c:1644)
by 0x48CD8D7: printf (printf.c:33)
by 0x1091BC: main (1.c:15)
Uninitialised value was created by a heap allocation
at 0x483874F: malloc (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
by 0x10917C: main (1.c:9)
```

Slika 4.7: Ispis greške korišćenja dinamički alocirane neinicijalizovane promenljive (pokazivača)

Način inicijalizacije je isti kao kod statički definisanih promenljivih, dok potrebne informacije dobijamo primenom nešto izmenjenim regularnim izrazima. Takođe, ukoliko je promenljiva samo pokazivač, bez alokacije memorije prilikom definicije iste, njena vrednost biće postavljena na *NULL*. Regularni izraz koji koristimo u ovom slučaju jeste *(malloc|calloc|realloc)(.+)*; dok se preciznije dobijanje informacije o grešci dobija specijalizacijom pomenutog izraza. Na osnovu rezultata primene pomenutih regularnih izraza dobijamo informacije o tipu pokazivača, imena pokazivača kao i veličini alocirane memorije. Takođe, i u ovom slučaju imamo podršku za rad sa korisnički definisanim strukturama.

Nakon sakupljanja potrebnih informacija ispravku greške formiramo na isti način kao i kod statički definisanih promenljivih, tj. problematičnu liniju menjamo blokom koji pored te linije sadrži i kod koji vrši inicijalizaciju, indirektno naslonjen na funkciju 4.6 i inicijalizacione funkcije koje koriste petlje.

## Nevalidno čitanje/pisanje

Ovakav tip grešaka mapiran je tipom greške *"Invalid read of size x"* u slučaju nevalidnog čitanja, odnosno *"Invalid write of size x"* u slučaju nevalidnog pisanja, gde je *x* broj bajtova, i razlogom greške *"Address adr is y bytes after a block of size z alloc'd"*, gde *adr* predstavlja heksadekadni zapis adrese, a *y* i *z* broj bajtova. To se jasno može videti na slici 4.8.

```
Invalid write of size 4
at 0x10917A: main (4.c:8)
Address 0x4a59054 is 4 bytes after a block of size 16 alloc'd
at 0x483874F: malloc (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
by 0x10916D: main (4.c:7)
```

Slika 4.8: Ispis greške nevalidnog čitanja

Analizom segmenta sa slike 4.8, odnosno samo razloga i tipa greške, možemo da dobijemo informaciju koliko je velika memorija kojoj pokušavamo da pristupimo,

ili da je izmenimo, a koja nije alocirana. Ispravka bi bila da memoriju koju program alocira, a sa kojom je pokušana interakcija, proširimo za vrednost koju dobijamo primenom pomenutih informacija. Takođe, na slici 4.8 vidimo da dobijamo i tačnu liniju koda, odnosno fajl gde je memorija alocirana. Baš tu liniju koda menjamo ispravnom, a ispravka se sastoji u tome da pogrešnu veličinu memorije koja se alocira zamenimo ispravnom, uvećanom za broj bitova koji dobijemo analizom greške.

## Nevalidno oslobađanje memorije

Ovakav tip grešaka mapiran je tipom greške *"Invalid free() / delete / delete[] / realloc()"*. Za ovaj tip greške razlog nije od presudnog značaja, tako da se mapiranje vrši samo nad tipom greške. Na slici 4.9 prikazan je segment ispisa greške koji koristimo da bismo je ispravili. Analizom ispisa sa slike dobijamo preciznu informaciju o liniji koda, kao i o fajlu gde je izvršeno nevalidno oslobađanje memorije. Ovu grešku ispravljamo tako što tu liniju izbrišemo, odnosno otklonimo nevalidno oslobađanje memorije.

```
Invalid free() / delete / delete[] / realloc()
at 0x483997B: free (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
by 0x10919F: main (2.c:11)
Address 0x4a590a0 is 0 bytes inside a block of size 12 free'd
at 0x483997B: free (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
by 0x109193: main (2.c:10)
Block was alloc'd at
at 0x483874F: malloc (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
by 0x109183: main (2.c:9)
```

Slika 4.9: Ispis greške nevalidnog oslobađanja memorije

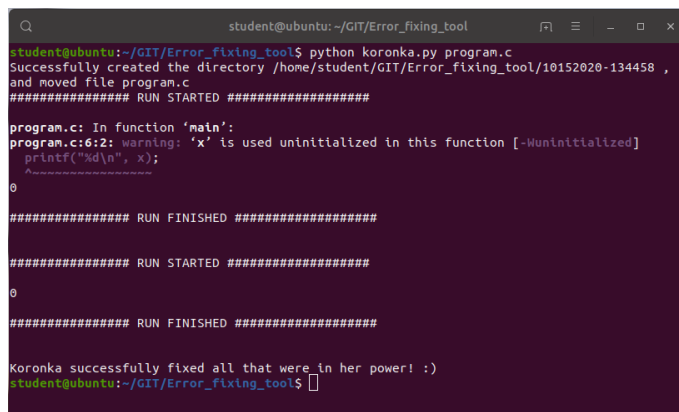
## 4.6 Primer rada alata

Analiza i ispravka programa sa slike 2.4 predstavljena je u ovoj sekciji. Dati program sadrži grešku upotrebe neinicijalizovane promenljive koja će biti ispravljena. Alat pokrećemo komandom u komandnoj liniji:

*python koronka.py program.c.*

Izgled komandne linije nakon pokretanja alata prikazan je na slici 4.10.

Direktorijum koji je formiran i dodeljen ovom konkretnom pokretanju alata i njegovom izvršavanju prikazan je na slici 4.11, dok su izmenjen program nakon izvršavanja alata *Koronka*, kao i izveštaj, prikazani na slikama 4.12 i 4.13.



```
student@ubuntu: ~/GIT/Error_fixing_tool
student@ubuntu:~/GIT/Error_fixing_tool$ python koronka.py program.c
Successfully created the directory /home/student/GIT/Error_fixing_tool/10152020-134458 ,
and moved file program.c
##### RUN STARTED #####

program.c: In function 'main':
program.c:6:2: warning: 'x' is used uninitialized in this function [-Wuninitialized]
printf("%d\n", x);
^
0

##### RUN FINISHED #####

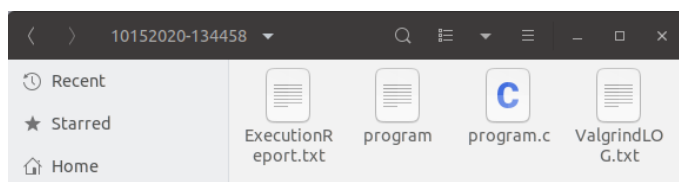
##### RUN STARTED #####

0

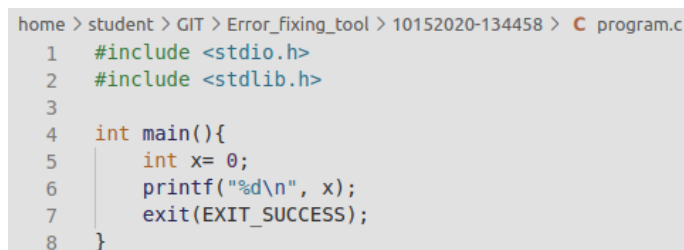
##### RUN FINISHED #####

Koronka successfully fixed all that were in her power! :)
student@ubuntu:~/GIT/Error_fixing_tool$
```

Slika 4.10: Izgled terminala nakon izvršavanja alata *Koronka*



Slika 4.11: Izgled foldera u formatu *datum pokretanja - vreme pokretanja* koji formira alat *Koronka*



```
home > student > GIT > Error_fixing_tool > 10152020-134458 > program.c
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(){
5      int x= 0;
6      printf("%d\n", x);
7      exit(EXIT_SUCCESS);
8  }
```

Slika 4.12: Izgled ispravljenog programa alatom *Koronka*

Prikazani primer deo je skupa test primera nalazi se u direktorijumu *Example* u okviru glavnog direktorijuma alata. Podeljeni su po folderima, a takođe se mogu i naći pokretanja i ispravke koje je *Koronka* implementirala nad tim test primerima. Skup test primera dostupan je na linku [https://github.com/LMladenovic/Error\\_fixing\\_tool/tree/master/Example](https://github.com/LMladenovic/Error_fixing_tool/tree/master/Example).

```
home > student > GIT > Error_fixing_tool > 10152020-134458 > ExecutionReport.txt
1  ##### Based on Valgrind output: #####
2
3  Conditional jump or move depends on uninitialised value(s)
4  at 0x48E2E40: __vfprintf_internal (vfprintf-internal.c:1644)
5  by 0x48CD8D7: printf (printf.c:33)
6  by 0x109162: main (program.c:6)
7  Uninitialised value was created by a stack allocation
8  at 0x109145: main (program.c:4)
9
10 ##### Koronka made following changes in program.c #####
11
12 Changed 5. line
13     int x;
14     with
15     int x= 0;
```

Slika 4.13: Izgled izveštaja nakon izvršavanja alata *Koronka*

## Glava 5

# Zaključak

Analiza i otkrivanje grešaka koje možemo otkriti alatima iz distribucije *Valgrind* ručno obično može biti mukotrpan i ne tako retko neefikasan postupak kad su u pitanju veliki programi. Srećom, alat *Valgrind* je tu da pomogne u prevazilaženju tog problema. Alat *Memcheck* otkriva greške u radu sa memorijom koje mogu da dovedu do pada programa, a koje kompilator nije u mogućnosti da otkrije, a koji zauzima centralno mesto ovog rada. Ostali alati *Valgrind* distribucije mogu da otkriju druge probleme, poput grešaka u radu sa nitima i sl., čijom detekcijom i ispravljanjem možete doprineti performansama i boljem radu softvera koji razvijate. Princip funkcionisanja ovih alata, a pre svega alata *Memcheck*, obrađeno je u ovom radu.

Nakon što analizom programa nekim od alata distribucije *Valgrind* detektujemo greške, ukoliko postoje, neophodno je da razumemo izlaz koji nam je alat dao, i da na osnovu njega pokušamo da popravimo greške. Alat koji je iznet u ovom radu pokušava da taj proces automatizuje, odnosno da razume izlaz koji je alat *Memcheck* dao, i da na osnovu njega korišćenjem adekvatnih šablona ispravi otkrivene greške. Pokriven je širok dijapazon grešaka, što ne isključuje da postoje greške koje mogu biti otkrivene daljim razvojem, a koje alat *Koronka* ne pokriva.

Što se daljeg razvoja alata tiče, mogu se dodatno specijalizovati i unaprediti postojeći mehanizmi i šabloni. Kako struktura alata liči na mikro servis, alat se može iskoristiti kao deo nekog novog alata, ili kao odvojeni deo skupa alata. Takođe, može biti proširen dijapazon grešaka koji alat ispravlja korišćenjem ostalih alata distribucije *Valgrind* za otkrivanje istih.

# Literatura

- [1] BBV: an experimental basic block vector generation tool, 2000-2020. URL: <https://www.valgrind.org/docs/manual/bbv-manual.html>.
- [2] Cachegrind: a cache and branch-prediction profiler, 2000-2020. URL: <https://www.valgrind.org/docs/manual/cg-manual.html>.
- [3] Callgrind: a call-graph generating cache and branch prediction profiler, 2000-2020. URL: <https://www.valgrind.org/docs/manual/cl-manual.html>.
- [4] DHAT: a dynamic heap analysis tool, 2000-2020. URL: <https://www.valgrind.org/docs/manual/dh-manual.html>.
- [5] DRD: a thread error detector, 2000-2020. URL: <https://www.valgrind.org/docs/manual/drd-manual.html>.
- [6] Helgrind: a thread error detector, 2000-2020. URL: <https://www.valgrind.org/docs/manual/hg-manual.html>.
- [7] Massif: a heap profiler, 2000-2020. URL: <https://www.valgrind.org/docs/manual/ms-manual.html>.
- [8] Memcheck : a memory error detector, 2000-2020. URL: <https://www.valgrind.org/docs/manual/mc-manual.html>.
- [9] Using and understanding the Valgrind core, 2000-2020. URL: <https://www.valgrind.org/docs/manual/manual-core.html>.
- [10] Valgrind, 2000-2020. URL: <https://valgrind.org>.
- [11] re — Regular expression operations, 2001-2020. URL: <https://docs.python.org/3/library/re.html>.

- [12] doc. dr Milena Vujošević Jančić. Dinamička analiza, 2019. URL: [http://www.verifikacijasoftera.matf.bg.ac.rs/vs/predavanja/03\\_dinamicka\\_analiza/03\\_dinamicka\\_analiza.pdf](http://www.verifikacijasoftera.matf.bg.ac.rs/vs/predavanja/03_dinamicka_analiza/03_dinamicka_analiza.pdf).
- [13] doc. dr Milena Vujošević Jančić. Verifikacija softvera - Motivacija, 2019. URL: [http://www.verifikacijasoftera.matf.bg.ac.rs/vs/predavanja/01\\_uvod/02\\_motivacija.pdf](http://www.verifikacijasoftera.matf.bg.ac.rs/vs/predavanja/01_uvod/02_motivacija.pdf).
- [14] Aleksandra Karadžić. Alat Valgrind - implementacija konvencije FPXX za arhitekturu MIPS, Master rad, 2018.
- [15] Miroslav Marić. *Operativni sistemi*. Matematički Fakultet, Univerzitet u Beogradu, 2016.
- [16] Dusko M. Vitas. *Prevodioci i interpretatori*. Matematički Fakultet, Beograd, 2006.

# Biografija autora

**Lazar Mladenović** rođen je 24.02.1996. u Leskovcu. Osnovnu školu završio je 2010. u Leskovcu, kao đak generacije i nosilac Vukove diplome. U tom periodu biva zainteresovan za programiranje, pa se može reći da mu je to odredilo dalji tok obrazovanja. Prirodno-matematički smer leskovačke Gimnazije završava 2014., takođe kao nosilac Vukove diplome.

2014. upisuje Matematički fakultet u Beogradu, smer Primenjena matematika, da bi se dve godine kasnije prebacio na smer Računarstvo i informatika. Isti završava 2019. godine. Nakon diplomiranja upisuje master studije na istom smeru i fakultetu.

Oblasti interesovanja uključuju pre svega razvoj i verifikaciju softvera, kao i primenu programiranja u auto industriji.