

UNIVERZITET U BEOGRADU
MATEMATIČKI FAKULTET



Lazar S. Mladenović

AUTOMATSKO ISPRAVLJANJE GREŠAKA DETEKTOVANIH POMOĆU ALATA MEMCHECK

master rad

Beograd, 2020.

Mentor:

dr Milena VUJOŠEVIĆ JANIČIĆ, vanredni profesor
Univerzitet u Beogradu, Matematički fakultet

Članovi komisije:

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

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 i platforme za dinamičku verifikaciju softvera, a jedna značajna platforma 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 koje nije lako otkriti. Važan alat platforme *Valgrind* je *Memcheck*. Ovaj alat analizira i ukazuje na greške u radu sa memorijom. Te greške mogu da izazovu pad programa, a uključuju, na primer, korišćenje neinicijalizovane vrednosti i pristup oslobođenoj, odnosno nedozvoljenoj memoriji. Cilj ovog rada je konstrukcija alata koji će koristeći izveštaj koji daje *Valgrind*, odnosno *Memcheck*, otkriti i ispraviti uzroke grešaka u programu. Osnovna ideja za ostvarivanje ovog cilja je da se definišu relevantni šabloni za ispravljanje grešaka i da se implementira automatska analiza izveštaja koji se dobija iz alata *Memcheck* i koja korišćenjem odgovarajućih šablona postiže željenu funkcionalnost.

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

Sadržaj

1	Uvod	1
2	Alat Valgrind	3
2.1	Arhitektura i princip funkcionisanja	3
2.2	Prijavljivanje grešaka	5
2.3	Memcheck	7
2.4	Ostali važni alati Valgrind distribucije	23
3	Regularni izrazi	29
3.1	Kratak pregled teorije jezika i regularnih izraza	29
3.2	Modul re	32
4	Alat Koronka	35
4.1	Korišćenje alata	35
4.2	Algoritam izvršavanja	36
4.3	Klasa greške	37
4.4	Mehanizam praćenja istorije	39
4.5	Šabloni za ispravljanje grešaka	40
4.6	Primeri rada alata	47
5	Zaključak	60
	Literatura	62

Glava 1

Uvod

Razvoj softvera predstavlja složen proces koji obuhvata razne aktivnosti. Ne-pisano je pravilo da se prilikom pisanja programa može desiti da se provuče neka greška kao npr. neinicijalizacija promenljive, ili neoslobađanje alocirane memorije, koja u tom trenutku možda biva bezazlena. Međutim, kako razvoj programa odmiče, dodatne greške koje ta početna, naizgled bezazlena greška izaziva, bivaju krucijalne i od vitalnog značaja. Na neke od gore pomenutih grešaka, 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 je teško naći.

Verifikacija softvera ima za cilj ispravljanje tih grešaka i zauzima bitno mesto u procesu razvoja softvera. Statička verifikacija softvera analizira ispravnost programa bez njegovog izvršavanja, tj. vrši analizu izvornog koda, dok dinamička verifikacija softvera predstavlja tehniku ispitivanja ispravnosti koda u toku njegovog izvršavanja [14]. Binarna analiza obuhvata analizu na nivou mašinskog koda, snimljenog kao objektni kôd (nepovezan) ili kao izvršni kôd (povezan) [13]. Primer platforme koji vrši dinamičku binarnu analizu koda jeste *Valgrind*.

Valgrind je platforma koja pruža brojne alate za debugovanje i profajliranje koji pomažu da se brže i pravilnije otklone greške u programima. Najpopularniji od ovih alata zove se *Memcheck*. Ovaj alat može otkriti mnoge greške u vezi sa memorijom koje su česte u programima C i C++, a koje mogu dovesti do pada ili nepredvidivog ponašanja programa.

Platforma *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 koristiti precizno definisane šablone za ispravljanje grešaka. 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 platforma *Valgrind* će biti detaljno opisana. Biće pobrojani i obrađeni svi alati *Valgrind*-a. Najveći akcenat će biti na alatu *Memcheck*, dok će o ostalima biti date osnovne relevantne informacije. U glavi 3 biće ukratko reči o procesu kompilacije programa i njenim fazama. Centralno mesto zauzeće teorija o jezicima i regularnim izrazima. U glavi 4 biće predstavljen alat *Koronka*, koji je osnovni doprinos ovog rada: biće 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. Kako tema to afirmiše, a i sam alat je napravljen tako da može da se stalno unapređuje, u smislu uopštavanja postojećih šablona, primene drugih vrsta analiza i alata *Valgrind*-a i sl., biće iznete i ideje i mogućnosti za dalji razvoj ovog projekta.

Glava 2

Alat Valgrind

*Valgrind*¹ je platforma za pravljenje alata za dinamičku analizu [11]. Dolazi sa skupom alata, od kojih svaki obavlja neku vrstu debugovanja, profajliranja ili slične zadatke koji pomažu programeru da poboljša program koji razvija. *Valgrind* se može koristiti i kao alat za pravljenje novih alata, bez narušavanja postojeće strukture. *Valgrind* distribucija trenutno broji sledeće alate:

1. *Memcheck* [1] — alat za detekciju memorijskih grešaka.
2. *Cachegrind* [3] — profajler keš memorije i skokova.
3. *Callgrind* [4] — profajler poziva funkcija.
4. *Helgrind* [7] — alat za detekciju grešaka niti.
5. *DRD* [6] — alat za detekciju grešaka niti.
6. *Massif* [8] — profajler korišćenja dinamičke memorije.
7. *DHAT* [5] — profajler korišćenja dinamičke memorije.
8. *BBV* [2] — eksperimentalni alat za podršku razvoja novih arhitektura računara.

2.1 Arhitektura i princip funkcionisanja

Svaki alat *Valgrind*-a koristi jezgro *Valgrind*-a. Jezgro *Valgrind*-a omogućava izvršavanje klijentskog programa, kao i snimanje izveštaja koji su nastali prilikom

¹Ime *Valgrind* potiče iz nordijske mitologije.

analize samog programa. Alat za dinamičku analizu koda se kreira kao dodatak na jezgro *Valgrind*-a, pisan u programskom jeziku C.

Svi *Valgrind* alati rade na istoj osnovi, s tim da informacije koje se emituju variraju. Na osnovu tih informacija, a koje direktno zavise od alata kojim je izvršena analiza, možemo na primer zaključiti kako da ispravimo greške ili dodatno optimizujemo deo koda na koji je određen alat ukazao.

Sledeća komanda ilustruje način pokretanja alata *Valgrind*:

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

Kao što je već rečeno, svaki *Valgrind*-ov alat sastoji se od koda jezgra *Valgrind*-a i koda alata, kojeg biramo koristeći opciju `--tool`. Pomenuta opcija ukazuje *Valgrind*-u na to koji alat treba da nasloni na jezgro, odnosno koji alat treba da pokrene. Sam alat je jedna statički povezana izvršna datoteka. Sem opcije `--tool` postoji mnoštvo opcija koje precizno definišu rad *Valgrind* alata, kao npr.:

- `--log-file` koja će izlaz alata *Valgrind* proslediti u navedenu datoteku (podrazumevano *Valgrind* svoj izlaz ispisuje na terminal),
- `--track-origins` koja daje bliže informacije i detaljniji izveštaj o greškama koje su izazvane,
- `--leack-check` koja svako curenje memorije prikazuje detaljnno,
- `--show-leack-kinds` koja prikazuje sve vrste curenja koje su definitivne, indirektne, moguće i dostupne u celom izveštaju.

Poslednje tri opcije odnose se na alat *Memcheck*, i biće detaljnije objašnjene u sekcijama koje slede. Naredni primer ilustruje pokretanje *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 *Memcheck*, tako da se u prethodnom primeru poziva alat *Memcheck* nad izvršnim programom *a.out*.

Bez obzira koji se alat koristi, *Valgrind* preuzima kontrolu nad programom pre nego što započne da se izvršava. Na taj način *Valgrind* je u stanju da generiše poruke o greškama i druge izlaze u smislu lokacija izvornog koda, kada je to

potrebno. Program se pokreće na sintetičkom procesoru koji obezbeđuje jezgro *Valgrind*-a. Kako se kôd izvršava prvi put, jezgro predaje kôd odabranom alatu. Alat na to dodaje svoj vlastiti kôd instrumentacije i vraća rezultat natrag u jezgro, koje koordinira dalje izvršavanje ovog instrumentiranog koda. Količina dodatog koda instrumentacije je različita za sve alate pojedinačno, i zavisi isključivo od izabranog alata. Na primer, alat *Memcheck* dodaje kôd za proveru svakog pristupa memoriji i svake izračunate vrednosti, čineći ga znatno sporijim od izvornog. *Valgrind* simulira svaku instrukciju koju izvršava program pojedinačno. Zbog toga aktivni alat proverava ili profajlira ne samo kôd programa, već i sve prateće dinamički povezane biblioteke, kao npr. C biblioteku i grafičke biblioteke [10].

Za korišćenje *Valgrind*-ovih alata, korisno je ponovo kompajlirati 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 profajliranje rezultata značajno manje korisnim. Korišćenjem opcije `-g` prilikom kompilacije, dobijaju se poruke koje vode direktno do odgovarajuće linije izvornog koda [10].

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. Ukoliko je potrebno dobiti detaljnije informacije o tome šta se događa, treba pokrenuti *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 listingu 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 0xBFFFF74C is not stack'd, malloc'd or free'd
```

Listing 2.1: Primer izlaza alta *Memcheck* [10]

Ova poruka kaže da je program izvršio ilegalno 4-bajtno čitanje adrese 0xBFFFF74C, koja 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. Za greške povezane sa identifikovanim (trenutnim ili oslobođenim) blokom hipa, npr. čitanje oslobođene memorije, *Valgrind* izveštava ne samo o lokaciji na kojoj se dogodila 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. Na ovaj način, alat izbegava zatrpavanje velikim brojem duplikata izveštaja o greškama.

Zanimljivo je da se greške prijavljuju pre nego što se pridružena operacija zaista dogodi. Na primer, ako se koristi *Memcheck* i program pokušava da čita sa adrese nula, *Memcheck* će poslati poruku u tom smislu, a program će onda verovatno zaustaviti rad sa greškom *Segmentation fault*. Generalno, dobra je praksa ispravljati greške redosledom u kojem su prijavljene. Ukoliko se ne primeni pomenuta praksa, 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 može prouzrokovati značajno usporenje ako program generiše ogromne količine grešaka. Da bi izbegao ozbiljnije probleme, *Valgrind* će jednostavno zaustaviti prikupljanje grešaka nakon što uoči 1.000 različitih grešaka ili ukupno 10.000.000 grešaka. U ovoj situaciji potrebno je zaustaviti program i popraviti ga, jer *Valgrind* neće prijaviti ništa drugo nakon ovoga. Ograničenja od 1.000 / 10.000.000 primenjuju se nakon uklanjanja potisnutih grešaka, odnosno onih grešaka za koje *Memcheck* utvrdi da su duplikati. Ova ograničenja su definisana u *m_errormgr.c* i mogu se povećati ako je potrebno [10].

2.3 Memcheck

Memcheck je alat koji vrši detekciju grešaka u radu sa memorijom [9]. Može da otkrije sledeće probleme koji su česti u programima C i C++:

- Pristup memoriji kojoj ne biste smeli, tj. prekoračenje i potkoračenje 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 memorije hipa, 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 padove, čiji je pravi razlog obično teško uočljiv. Upravo iz tih razloga memorijske greške spadaju u grupu najteže detektujućih grešaka pa je samim tim i njihovo otklanjanje težak zadatak.

Kao što je rečeno u prethodnoj sekciji, da bi se dobile detaljne, a pritom i neophodne informacije o programu, koristeći *Memcheck*, program treba kompajlirati sa uključenom opcijom `-g`. Što se optimizacije tiče, prilikom korišćenja alata *Memcheck*, u retkim prilikama, primećeno je da su optimizacije kompajlera (optimizacija `-O2` i više optimizacije, a ponekad čak i optimizacija `-O1`) generisale kôd koji zavarava *Memcheck* da pogrešno prijavljuje greške vezane za neinicijalizovane vrednosti [10]. Razvojni tim *Valgrind*-a je detaljno razmotrio kako to popraviti i nažalost došli su do rezultata da bi popravljavanje te greške dovelo do značajnog usporavanja alata. Dakle, najbolje rešenje je potpuno isključiti optimizaciju. Budući da ovo često stvari čini nekontrolisano sporima, razuman kompromis je korišćenje opcije `-O0`. Ovo donosi većinu prednosti viših nivoa optimizacije, dok istovremeno ima

relativno male šanse za lažno pozitivne ili lažno negativne podatke od *Memcheck*-a. Takođe, poželjno je kompajlirati kôd sa `-Wall` jer on može identifikovati neke ili sve probleme koje *Valgrind* može propustiti na višim nivoima optimizacije. Nivo optimizacije ne utiče na sve ostale alate, a za alate za profajljanje kao što je *Cachegrind* bolje je da program kompajlirate na normalnom nivou optimizacije [10].

Da biste koristili ovaj alat, možete da navedete opciju `--tool = memcheck` u komandnoj liniji prilikom pokretanja *Valgrind*-a, ali niste u obavezi s obzirom da je *Memcheck* podrazumevani alat.

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

Program koji radi pod kontrolom *Memcheck*-a obično je deset do pedeset puta sporiji nego kada se izvršava samostalno, što je posledica translacije koda [10]. Više o tome biće rečeno u sekciji koja sledi. Izlaz iz programa biće povećan za izlaz koji daje *valgrind* i sam alat *Memcheck*, koji se ispisuje na standardnom izlazu za greške ukoliko se ne definiše drugačije [1].

Princip funkcionisanja alata Memcheck

U ovoj sekciji biće opisani mehanizmi koje *Memcheck* koristi u detektovanju grešaka, nakon čega će biti obrađene greške koje može detektovati.

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 (valjane vrednosti).

Svaki bajt u sistemu iz tog razloga ima i 8V bitova koji ga prate. Na primer, kada CPU učita 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 za brojeve u pokretnom zarezu, vektori i uslovi) imaju svoje V bitne vektore. Da bi ovo moglo da funkcioniše dovoljno efikasno, *Memcheck* koristi veliku količinu kompresije da bi kompaktno prikazao V bitove.

Kopiranje vrednosti 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 programa, pridruženi V bitovi se tada proveravaju. Ako bilo koji od njih ukazuje da je vrednost nedefinisana (odnosno promenljiva definisana ali neinicijalizovana), čak i delimično, prijaviće 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, alat ne prijavljuje grešku. Provere definisanosti dešavaju se samo na tri mesta:

- Kada se vrednost koristi za generisanje memorijske adrese.
- Kada treba doneti odluku o kontroli toka izvršavanja programa.
- Kada se detektuje sistemski poziv, u kom slučaju će *Memcheck* proveravati definisanost parametara prema potrebi.

Kada provera detektuje nedefinisanost, izdaje se poruka o grešci. Dobijena vrednost se na dalje smatra dobro definisanom, kako bi se izbegli dugi lanci poruka o greškama. Drugim rečima, kada *Memcheck* prijavi grešku nedefinisanosti, pokušava da izbegne prijavljivanje daljih grešaka izvedenih iz te iste nedefinisanosti. Ideja o proveravanju svih čitanja iz memorije i žalbi ukoliko je nedefinisana vrednost učitana u registar procesora ne funkcioniše dobro, jer savršeno legitimni C programi rutinski kopiraju neinicijalizovane vrednosti u memoriji, što bi u slučaju primene pomenute ideje dovelo do enormno velikog broja prijavljenih grešaka. Primer sa listinga 2.2 ilustruje tu problematiku.

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

```
8 s1.c = 'a';  
9 s2=s1;
```

Listing 2.2: Primer kratkog segmenta koda

Nakon analize koda sa listinga 2.2, može da deluje da struktura `S`, kako tip podataka `int` zauzima 4 bajta, a tip podataka `char` jedan bajt, zauzima 5 bajtova. Međutim, svi kompajleri, u normalnom režimu rada, zaokružiće veličinu strukture `S` na ceo broj reči, tj. na 8 bajtova. Dakle, `s1` zauzima 8 bajtova, ali će samo njih 5 biti inicijalizovano. Za zadatak `s2=21`, GCC generiše kôd za kopiranje svih 8 bajtova u `s2` bez obzira na njihovo značenje. Ako bi *Memcheck* jednostavno proveravao vrednosti redom kako su izašle iz memorije, oglasio 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 [9].

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.

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 program čita ili upisuje u memoriju, *Memcheck* proverava A bitove povezane sa adresom. Ako bilo koji od njih označi neispravnu adresu, prijavljuje se greška. Sama čitanja i pisanja ne menjaju A bitove, već ih samo konsultuju. 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 nevažeće vrednosti, a kao rezultat toga dobila bi se ne samo greška sa nevažećom adresom (čitanje / pisanje), već i potencijalno veliki skup greškaka neinicijalizovanih vrednosti — jedna za svaki put kada se vrednost koristi.

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 označeni su kao pristupačni. 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 označeno kao pristupačno, a ispod SP nepristupačno.
- Pri obavljanju sistemskih poziva, A bitovi se odgovarajuće menjaju. Na primer, *mmap* čini da se datoteke pojavljuju u adresnom prostoru procesa, pa se A bitovi moraju ažurirati ako *mmap* uspe [9].

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 omogućava da troškovi obično iznose oko 25% vremena i prostora.
- 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.
- *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 može da se prosledi bilo šta, ali ispravno je samo ako se prosledi pokazivač koji je prethodno izdao *malloc/calloc/realloc*.
 - *free/delete/delete[]*: ovim funkcijama može se proslediti samo pokazivač koji je prethodno izdala odgovarajuća funkcija dodeljivanja. Inače, *Memcheck* prijavljuje nepravilnost. 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 [1].

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

```
Invalid read of size 4
at 0x40F6BBCC: (within /usr/lib/libpng.so.2.1.0.9)
by 0x40F6B804: (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 0xBFFFF0E0 is not stack'd, malloc'd or free'd
```

Listing 2.3: Primer ispisa greške nevalidnog čitanja [9]

Ove greške se javljaju kada program čita ili piše u memoriju na mestu za koje *Memcheck* smatra da ne bi trebalo. U ovom primeru, program je izvršio četvorobajtno čitanje na adresi 0xBFFFF0E0, negde unutar sistemske biblioteke *libpng.so.2.1.0.9*, koja je pozvana negde drugde u istoj biblioteci, a koje je izvršeno iz linije 326 u okviru fajla *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, alat će emitovati tu informaciju, kao i podatak 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, alat će i to registrovati i prijaviti, kao i podatak 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 greška je verovatno uzrokovana GCC-om koji generiše nevažeći kôd, što je poznata greška u nekim drevnim verzijama GCC-a.

Treba imati na umu da *Memcheck* samo govori da će program uskoro pristupiti memoriji na nedozvoljenoj adresi i da pritom taj pristup ne može sprečiti. Dakle, kada program pristupi, što bi obično rezultiralo greškom (eng. *Segmentation fault*), program će i dalje imati isto ponašanje - ali *Memcheck*-a će izdati poruku neposredno pre toga. U ovom konkretnom primeru čitanje loših (nevalidnih) vrednosti sa steku nije fatalno, i program nastavlja sa radom [9].

Korišćenje neinicijalizovanih vrednosti

Greška pri korišćenju neinicijalizovane vrednosti se prijavljuje kada program koristi vrednost koja nije inicijalizovana - drugim rečima, nije definisana. Ovde se nedefinisana vrednost koristi negde unutar funkcije `printf` iz standardne C biblioteke. Ova greška je prijavljena prilikom pokretanja programa sa listinga 2.4, dok je njen ispis dat na listingu 2.5.

```
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 }
```

Listing 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
```

Listing 2.5: Primer ispisa greške korišćenja neinicijalizovanih vrednosti

Poruka o grešci se izdaje samo kada program pokušava da koristi neinicijalizovane podatke na način koji može uticati na spolja vidljivo ponašanje programa, iako *Memcheck* prati i evidentira svako korišćenje neinicijalizovanih podataka. U ovom primeru, *x* je neinicijalizovano. *Memcheck* primećuje vrednost koja se prosleđuje *_IO_printf* i odatle *_IO_vfprintf*, ali ne daje komentar. Međutim, *_IO_vfprintf* mora ispitati vrednost kako bi je mogao pretvoriti u odgovarajući ASCII niz i *Memcheck* u ovom trenutku prijavljuje grešku.

Izvori grešaka neinicijalizovanih vrednosti su sledeći:

- Lokalne promenljive u procedurama koje nisu inicijalizovane, kao u primeru sa listinga 2.4.
- Sadržaj blokova hipa (dodeljen *malloc*-om, *new* ili sličnom funkcijom) pre nego što se nešto u tom bloku ispiše, odnosno pre inicijalizacije.

Za dobijanje informacija o izvorima neinicijalizovanih podataka u programu, neophodno je koristiti opciju `--track-origins=yes`. Ovo čini *Memcheck* sporijim, ali može znatno olakšati pronalaženje osnovnih uzroka neinicijalizovanih grešaka [1].

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 <unistd.h>
2 #include <stdlib.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 }
```

Listing 2.6: Program koji izaziva grešku korišćenja neinicijalizovane ili neadresirane vrednosti u sistemskom pozivu [9]

```
Syscall param write(buf) points to uninitialised byte(s)
  at 0x25A48723: __write_nocancel (in /lib/tls/libc-2.3.3.so)
  by 0x259AFAD3: __libc_start_main (in /lib/tls/libc-2.3.3.so)
  by 0x8048348: (within /auto/homes/njn25/grind/head4/a.out)
Address 0x25AB8028 is 0 bytes inside a block of size 10 alloc'd
  at 0x259852B0: malloc (vg_replace_malloc.c:130)
  by 0x80483F1: main (a.c:5)

Syscall param exit(error_code) contains uninitialised byte(s)
  at 0x25A21B44: __GI__exit (in /lib/tls/libc-2.3.3.so)
  by 0x8048426: main (a.c:8)
```

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

Na listinzima 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. Sa listinga 2.7 vidimo da je *Memcheck* prikazao dve greške sa informacijama o korišćenju neinicijalizovanih vrednosti u sistemskim pozivima, kao i linije izvornog koda gde se ove vrednosti koriste. Prva greška posledica je korišćenja neinicijalizovane vrednosti *arr* kao parametra sistemskog poziva *write()*, dok je druga greška posledica prosleđenog nedefinisanog argumenta sistemskom pozivu *exit* [9].

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. Listing 2.8 ilustruje program koji je oslobodio isti blok dva puta.

```
1 #include <stdio.h>
2 #include <stdlib.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 }
```

Listing 2.8: Program koji oslobađa 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. Ova poruka će biti emitovana i ako program pokuša da oslobodi pokazivač koji ne pokazuje na početak hip bloka. Listing 2.9 ilustruje izlaz dobijen puštanjem programa sa listinga 2.8 na analizu alatom *Memcheck* [1].

```
==6215== Memcheck, a memory error detector
==6215== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et
        al. 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 0xffffffffffffff90 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 0xffffffffffffff90 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==    For counts of detected and suppressed errors, rerun with: -v
==6215== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from
0)
```

Listing 2.9: Primer ispisa greške nedopuštenog oslobađanja memorije

Preklapanje izvornog i odredišnog bloka

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

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

Listing 2.10: Primer ispisa greške preklapanja izvornog i odredišnog bloka [9]

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 neoznačeni 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 listinga 2.11.

```
==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)
```

Listing 2.11: Primer ispisa greške sumnjive vrednosti argumenata [9]

Detekcija curenja memorije

Na kraju izvršavanja programa *Memcheck* ima informaciju o svim blokovima hip memorije koji nisu oslobođeni, jer tokom izvršavanja programa *Memcheck* vodi računa o svim blokovima hipa koji su alocirani. *Memcheck* takođe može odrediti da li se svakom od tih blokova može pristupiti preko dostupnog pokazivača, ukoliko se na odgovarajući način podesi opcija `--leak-check` [1].

Memorijskim blokovima se može pristupiti pomoću pokazivača. Postoje dve vrste pokazivača koje u tu svrhu koristimo:

- Pokazivač na početak memorijskog bloka (eng. *start-pointer*).
- Pokazivač na sadržaj unutar memorijskog bloka (eng. *interior-pointer*).

Ukoliko imamo pokazivač koji je pokazivao na početak bloka, koji je u međuvremenu pomeren da pokazuje na unutrašnjost bloka, ili postoji slučajna i nepovezana vrednost u memoriji koja pokazuje na unutrašnjost nekog memorijskog bloka, automatski znamo da imamo pokazivač na unutrašnjost memorijskog bloka.

Imajući to na umu, na slici 2.1 navedeno je devet mogućih slučajeva kada pokazivači pokazuju na neke memorijske blokove.

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 2.1. Pronađen je pokazivač na početak bloka ili više

	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
- (_,n)XY: XY u svakom slučaju

Slika 2.1: Primer pokazivača na memorijski blok

takvih pokazivača. Kako ovi pokazivači pokazuju tačno na neoslobodeni segment memorije, isti se može osloboditi pre završetka rada programa. Blokovi iz ove kategorije su vrlo česti i verovatno nisu problem. Dakle, *Memcheck* podrazumevano neće pojedinačno prijavljivati takve blokove [1].

- **Definitivno izgubljen (eng. *definitely lost*)** – Ova klasa pokriva slučaj 3 naveden na slici 2.1. 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 [1].
- **Indirektno izgubljen (eng. *indirectly lost*)** – Ova klasa pokriva slučajeve 4 i 9 navedene na slici 2.1. To znači da je blok izgubljen, ne zato što na njega ne pokazuje ni jedan pokazivač, već zato što su svi blokovi koji na njega upućuju sami izgubljeni. Ukoliko imamo jednostruko povezanu listu, a u nekom trenutku, iz nekog razloga, pokazivač koji je pokazivao na početak liste prestane da pokazuje na početak, već pokazuje na neki element unutar liste, svi elementi od početka liste do elementa na koji pokazivač trenut-

no pokazuje nisu više dostupni, odnosno indirektno su izgubljeni. Ovo će *Memcheck* prijaviti kao definitivno izgubljene blokove, sem ako se ne podesi opcija `-show-reachable=yes` [1].

- **Moguće izgubljen (eng. *possibly lost*)** – Ova klasa obuhvata slučajeve 5-8 navedene na slici 2.1. To znači da je pronađen jedan ili više pokazivača na memorijski blok, ali bar jedan od pokazivača 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 [1].

Treba napomenuti da ovo mapiranje devet mogućih slučajeva na četiri vrste curenja nije nužno najbolji način za prijavljivanje curenja memorije. Naročito se prema unutrašnjim pokazivačima postupa nedosledno. Moguće je da će se kategorizacija u budućnosti poboljšati. Na listingu 2.12 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
```

Listing 2.12: Rezime 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* ne može reći kada, kako ili zašto se izgubio pokazivač na isureli blok. Generalno, treba težiti tome da 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 listingu 2.13 prikazan je izveštaj koji 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-amd64-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-amd64-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-amd64-linux.so)
    by 0x109193: main (4.c:10)
```

Listing 2.13: Izveštaj o curenju memorije

Memcheck koristi odgovarajući kriterijum za prijavljivanje i tretman grešaka vezanih za curenje memorije, budući da postoje različite vrste curenja memorije različitih težina. Ukoliko je uključena opcija `--leak-check=full`, *Memcheck* će smatrati da je curenje memorije greška, odnosno, ukoliko podaci o curenju memorije nisu prikazani, iako oni postoje, *Memcheck* smatra da to curenje nije greška. Takođe, definitivno i moguće izgubljene blokove *Memcheck* smatra greškom, dok indirektno izgubljene i još uvek dostupne blokove ne smatra greškom [9].

- 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 [1].

2.4 Ostali važni alati Valgrind distribucije

U ovoj sekciji biće obrađeni ostali važni alati *Valgrind* distribucije. Biće reči o njihovoj osnovnoj nameni i načinu korišćenja.

Da bi uspešno izvršili analizu alatima *Cashegrind*, *Callgrind* i *DHAT* neophodno je kompajlirati program sa opcijom za debug mod (opcija `-g`) i uključenom optimizacijom, jer nema smisla profajlirati kôd koji je drugačiji od onoga koji će se normalno izvršavati. Za analizu programa alatom *Massif* neophodno je kompajlirati program sa opcijom `-g`, dok nivo optimizacije nije od presudnog značaja. Što se alata *Helgrind* i *DRD* tiče, oni nisu zavisni od uslova kompilacije, tj. opcije `-g` i uključene optimizacije.

Cachegrind

Cachegrind je alat koji simulira i prati kako program pristupa keš memoriji [3]. Takođe, može se koristiti i za profajliranje izvršavanja grana. Simulira mašinu sa nezavisnim kešom instrukcija prvog nivoa i kešom podataka (I1 i D1), potpomognutim objedinjenom keš memorijom drugog nivoa (L2).

Što se modernih mašina tiče, tj. onih mašina sa tri ili četiri nivoa keš memorije, 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*. Ovakav pristup posledica je toga 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 kôd loše interaguje sa ovom keš memorijom [3].

Da bi se koristio alat *Cachegrind* neophodno je podesiti opciju `-tool=cachegrind`. Izvršavanje programa će trajati dugo, a po završetku štampaće se sažeti statistički podaci. Pored štampanja sažetka na standardni izlaz, *Cachegrind* u odgovarajuću datoteku upisuje i detaljnije informacije o profajliranju. Za dobijanje detaljnijih informacija koristimo pomenutu datoteku kao ulaz za `cg_annotate`, koji je deo pa-

keta *Valgrind*. Alternativno, možete koristiti `cg_diff` za razlikovanje više izlaza iz alata *Cachegrind*, koje kasnije koristimo kao ulaz za `cg_annotate` [3].

Callgrind

Callgrind je alat za profajliranje koji generiše istoriju poziva funkcija korisničkog programa u vidu grafa. Prikupljeni podaci podrazumevano se sastoje od broja izvršenih instrukcija, njihovog odnosa sa linijama izvršnog koda, odnosa pozivaoc/pozvan između funkcija, kao i broja takvih poziva. Opciono, simulacija keš memorije i/ili profajliranje grana (slično *Cachegrind*-u) mogu dati dodatne informacije o ponašanju aplikacije u toku izvršavanja [4]. Alat sadrži neka preklapanja sa *Cachegrind* - om, ali takođe pruža neke informacije koje *Cachegrind* ne pruža.

Da bi se koristio *Callgrind*, potrebno je navesti opciju komandne linije `--tool = callgrind` u komandnoj liniji *Valgrind*-a. Nakon završetka programa i rada alata, generiše se odgovarajuća datoteka sa podacima o izvršenom profajliranju. Za prezentaciju podataka i interaktivnu kontrolu profajliranja na osnovu date datoteke, obezbeđena su dva alata komandne linije. Jedan je `callgrind_annotate`, koji čita generisani fajl i ispisuje sortirane liste funkcija, opciono sa anotacijom izvora. Drugi je `callgrind_control` koji omogućava interaktivno posmatranje i kontrolisanje statusa programa koji se trenutno izvršava pod kontrolom *Callgrind*-a, bez zaustavljanja programa. Mogu se dobiti statistički podaci kao npr. stanje steka, a u svakom trenutku se takođe može generisati i profil [4].

Helgrind

Helgrind je *Valgrind*-ov alat za otkrivanje grešaka sinhronizacije u programima C, C++ i Fortran prilikom upotreba modela niti *POSIX* [7]. Model niti *POSIX* prate određene apstrakcije — 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, spinlokovi, semafori i barijere [7].

Da bi se koristio ovaj alat, potrebno je navesti opciju komandne linije `--tool = helgrind` u komandnoj liniji *Valgrind*-a.

Helgrind može otkriti sledeća tri tipa grešaka:

1. Pogrešna upotreba interfejsa za programiranje *POSIX* niti.

2. Potencijalno blokiranje prouzrokovano redosledom zaključavanja.
3. Trka sa podacima (eng. *data race*) - 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.

Što se pogrešne upotrebe interfejsa za programiranje *POSIX* niti tiče, *Helgrind* presreće pozive ka funkcijama biblioteke *pthread*, i zbog toga je u mogućnosti da otkrije veliki broj grešaka. Neke od njih 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 [7].

Što se potencijalnog blokiranja prouzrokovanog redosledom zaključavanja tiče, *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 mogu javiti prilikom nekog drugog pokretanja programa [7].

Što se trke sa podacima tiče, *Helgrind*-ov algoritam je (konceptualno) vrlo jednostavan - nadgleda sve pristupe memorijskim lokacijama. Ako lokaciji pristupaju dve različite niti, *Helgrind* proverava da li su dva pristupa uređena relacijom „dogodilo se pre” (eng. *happens-before relation*). Ako je tako, to je u redu, ali ukoliko nije, *Helgrind* prijavljuje trku sa podacima.

DRD

DRD je *Valgrind*-ov alat za otkrivanje grešaka u višenitnim programima napisanim u programskim jezicima C i C++ [6]. Alat radi za bilo koji program koji koristi niti bazirane na modelu *POSIX* [6]. Sličan je *Helgrind*-u, ali koristi različite tehnike analize i tako može naći različite probleme.

Da bi se koristio ovaj alat, potrebno je navesti opciju komandne linije `--tool = drd` u komandnoj liniji *Valgrind*-a.

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

- Trka sa podacima.
- Zadržavanje katanaca - jedna nit blokira napredovanje jedne ili više drugih niti držeći predugo katanac zaključanim.
- Pogrešna upotreba interfejsa za programiranje *POSIX* niti.
- 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.

Što se tiče grešaka vezanih za pogrešnu upotrebu interfejsa za programiranje *POSIX* niti, kao i trke sa podacima, greške koje *DRD* otkriva su veoma slične onim koje otkriva *Helgrind*, a koje su opisane u sekciji o *Helgrind*-u, 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 [6]. Alat *DRD* svakoj niti dodeljuje jedinstveni ID broj. Brojevi koji se dodeljuju nitima u svojstvu identifikatora počinju od jedinice i dodeljuju se jednokratno, odnosno jednom iskorišćeni broj ne može se više koristiti. Segment uvek započinje i završava se operacijom sinhronizacije, a sam termin se odnosi na uzastopni niz operacija učitavanja, skladištenja i sinhronizacije - sve izvršene u istoj niti. Da bi detektovao grešku trke sa podacima, *DRD* analizu vrši između segmenata, što kao rezultat daje bolje performanse [6].

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 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*) [6]. Ovakva pojava je nepoželjna u višenitnim sistemima, a u njenom otklanjanju pomaže alat *DRD* koji otkriva ovaj tip problema.

Massif

Massif je alat za profajliranje hipa. Meri koliko memorije hipa program koristi, i na koji način. To uključuje memoriju kojoj korisnik može pristupiti, kao i memoriju koja se koristi za dodatne koncepte, poput *book-keeping* bajtova i prostora za poravnanje [8]. Takođe može izmeriti veličinu steka programa, s tim da to ne radi podrazumevano. Profajliranje hipa može pomoći u smanjivanju količine memorije koju program koristi. Na modernim mašinama ovo pruža prednosti u vidu ubrzanja programa i smanjenja šansi za izgladnjivanje prostora za razmenu mašine (eng. *swap space*) [8].

Može se desiti da postoji pokazivač na memoriju, ali da se ta memorija ne koristi, što je jedan vid curenja memorije. Programi koji imaju ovakvo curenje mogu s vremenom nepotrebno povećati količinu memorije koju koriste. *Massif* može pomoći u identifikovanju ovakvog tipa curenja memorije [8].

Da bi se koristio ovaj alat, potrebno je navesti opciju komandne linije `--tool = massif` u komandnoj liniji *Valgrind*-a. Nakon završetka procesa profajliranja, podaci se smeštaju u odgovarajuću datoteku. Nakon toga treba pokrenuti `ms_print` da bi podatke iz datoteke predstavili na čitljiv način.

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 [5]. 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.

Da bi se koristio alat *DHAT* potrebno je navesti opciju komandne linije `-tool=dhat` u komandnoj liniji *Valgrind*-a. Analiza programa alatom *DHAT* će se izvršavati prilično sporo, a po završetku, štampaće se sažeti statistički podaci. Korisnije informacije mogu se videti sa *DHAT*-ovim prikazivačem. To omogućava činjenica da pored štampanja rezimea, *DHAT* u datoteku upisuje i detaljnije informacije o profajliranju. Ova datoteka je u JSON formatu i namenjena je pregledu *DHAT*-ovog pregledača [5].

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 [2].

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 kôd 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 [2].

Da bi se kreirala datoteka osnovnog vektorskog bloka, neophodno je u komandnoj liniji *Valgrind*-a navesti opciju `-tool=exp-bbv`. Nakon izvršene analize biće kreirana odgovarajuća datoteka koja sadrži vektor osnovnog bloka. Za kreiranje stvarnih *SimPoint* rezultata na osnovu pređašnje generisane datoteke, potreban je uslužni program *SimPoint*, dostupan na *SimPoint* veb stranici. Uslužni program *SimPoint* vrši slučajnu linearnu projekciju pomoću 15 dimenzija, a zatim k-dimenziono grupisanje kako bi izračunao koji intervali su od interesa [2].

Glava 3

Regularni izrazi

Jezici, a samim tim i regularni izrazi koji predstavljaju jednu klasu jezika, vrlo su važni za proces kompilacije. Kompilacija se sastoji iz nekoliko etapa, koje se kasnije dele u nekoliko faza. Jezici imaju važnu ulogu u etapi analize izvornog koda. Preduslov za ispitivanje izvornog koda jeste precizna definicija jezika [15].

U današnje vreme, programski jezici imaju veoma dobru podršku za rad sa regularnim izrazima i koji imaju širok dijapazon upotrebe. Da bi u potpunosti iskoristili podršku za rad sa regularnim izrazima koju nam programski jezici pružaju, neophodno je pre svega razumeti šta su regularni izrazi i koje operacije je moguće vršiti nad njima. U tu svrhu biće definisani jezik, regularni izraz kao posebna klasa jezika, kao i operacije nad regularnim izrazima.

3.1 Kratak pregled teorije jezika i regularnih izraza

Definicija izvornog jezika sadrži konačan skup simbola koje možemo koristiti u jeziku, tj. azbuku jezika. Takođe, sadrži i sintaksička i semantička pravila jezika, tj. opis svih reči koje pripadaju datom jeziku kao i njihovo značenje [15].

Neka je Σ konačan skup. Elemente skupa Σ nazivamo slova ili simboli, dok sam skup nazivamo alfabet ili azbuka [15]. Reč (eng. *word*) nad Σ 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 $|x|$. Prazna reč obeležava se simbolom ε , odnosno važi $|\varepsilon| = 0$. Ako je

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

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

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

Klinijevo zatvorenje azbuke Σ , u oznaci Σ^* , je skup svih niski nad azbukom Σ . Na primer, ukoliko je $\Sigma = \{a, b\}$, onda je

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

Neutral u odnosu na dopisivanje je prazna reč, tj. za $\forall x \in \Sigma, x\varepsilon = \varepsilon x = x$. Skup svih reči bez prazne reči označava se kao Σ^+ , tj. $\Sigma^+ = \Sigma^* - \varepsilon$ [15].

Definicija 1. Bilo koji podskup skupa Σ^* : $L \subset \Sigma^*$ predstavlja jedan (formalni) jezik nad azbukom Σ . Ako $x \in L$, onda x predstavlja rečenicu jezika L .

Neka je data azbuka $\Sigma = \{a, b, c\}$. Neki od jezika nad ovom azbukom 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 .

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

$L_3 = \{ca^n ccc b^m | m, n > 0\}$ – jezik koji se sastoji od niski u kojima se nakon simbola c proizvoljan broj puta pojavljuje simbol a , iza čega sledi tri pojavljivanja simbola c i pojavljivanje simbola b proizvoljan broj puta.

Neka su L i M dva proizvoljna jezika. Kako su i sami jezici skupovi, nad jezicima je moguće izvršiti sledeće operacije:

- Unija dva jezika: $L \cup M = \{w | w \in L \vee w \in M\}$.
- Presek dva jezika: $L \cap M = \{w | w \in L \wedge w \in M\}$.
- Razlika dva jezika: $L \setminus M = \{w | w \in L \wedge w \notin M\}$.

Pored pomenutih, nad jezicima definišemo još i operaciju proizvoda, n -tog stepena i zatvorenja jezika L .

Definicija 2. Proizvod jezika L_1 i L_2 nad azbukom Σ 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 ε je njen neutralni element.

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

$$L_0 = \{\varepsilon\}, \quad L_1 = L, \quad L_n = LL_{n-1}, \text{ za } n > 1.$$

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:

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

Za opisivanje jezika koristimo regularne izraze. Međutim, ne mogu se svi jezici opisati regularnim izrazima.

Definicija 5. Ukoliko se jezik može predstaviti regularnim izrazom, takav jezik se smatra regularnim.

Važnost ove notacije ogleda se u tome što se na isti način zapisuju simbol azbuke, reč koja se sastoji od tog simbola, regularni izraz koji predstavlja taj simbol, i na kraju jezik predstavljen tim regularnim izrazom [15]. Kažemo da je za neki regularni izraz p , jezik $L(p)$ jezik opisan regularnim izrazom p .

Definicija 6. Regularni izrazi nad azbukom Σ opisuju se rekurzivno na sledeći način:

1. Prazan skup je regularni izraz koji se prikazuje simbolom \emptyset .
2. Regularni izraz ε 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 $L(p)$.

Opisani jezik se neće promeniti ako se zagrade izostave iz regularnog izraza [15]. Kada prioritet operacija nije eksplicitno naveden, primenjuju se podrazumevani prioriteti nad operatorima regularnih izraza, tj. jezika, u sledećem redosledu:

1. Operator $*$
2. Konkatenacija
3. Operator $+$

Zapisivanje regularnih izraza može se znatno pojednostaviti uvođenjem sledećih konvencija:

- Neka je r regularni izraz koji opisuje jezik $L(r)$. Jezik $(L(r))(L(r))^*$ se opisuje regularnim izrazom $(r)^+$, dok se jezik $L(r) \cup \{\varepsilon\}$ opisuje regularnim izrazom $(r)?$.
- 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 tako da važi $c_1 \leq c \leq c_2$.

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

3.2 Modul re

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. Većina operacija sa regularnim izrazima je dostupna u vidu funkcija i metoda na nivou modula sa kompajliranim regularnim izrazima. Razlika između funkcija i metoda je ta da funkcije ne zahtevaju da prvo kompajlirate objekat regularnog izraza, ali je mana što im nedostaju neki parametri za fino podešavanje.

Regularni izraz predstavljen je odgovarajućom niskom, dok funkcije u ovom modulu omogućavaju da proverite da li se određeni niz karaktera poklapa sa datim regularnim izrazom (ili da li se regularni izraz podudara sa određenim nizom karaktera). Neke od funkcija koje ovaj modul pruža, a koje su iskorišćene u alatu koji ovaj rad prati su sledeće:

re.search(string pattern, string string, int flags=0) — Skenira kroz *string* tražeći prvu lokaciju na kojoj obrazac regularnog izraza daje podudaranje i vraća odgovarajući objekt podudaranja. Vraće *None*¹, ako nijedna pozicija u nizu karaktera ne odgovara obrascu. U ovoj, kao i u ostalim funkcijama, *pattern* predstavlja regularni izraz zadat odgovarajućim nizom karaktera, dok se osobine regularnog izraza mogu izmeniti navođenjem vrednosti argumenta *flag*, npr. `flags=re.IGNORECASE`.

re.match(string pattern, string string, int flags=0) — Ako se nula ili više znakova na početku niza karaktera *string* podudaraju sa regularnim izrazom, vraće odgovarajući objekt podudaranja. Vraće *None* ako se niz ne podudara sa obrascem.

re.split(string pattern, string string, int maxsplit=0, int flags=0) — Podeliće nisku *string* koristeći kao separator za deljenje sva poklapanja sa regularnim izrazom *pattern*. Ako argument *maxsplit* nije 0, u rezultatu će se javiti maksimum *maxsplit* podeljenih niski.

re.findall(string pattern, string string, int 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(string pattern, string/function repl, string string, int count=0, int flags=0) — Vraća string dobijen zamenom krajnjeg levog poklapanja sa *pattern*-om u *string*-u sa *repl*. Ako poklapanje nije pronađeno, *string* se vraća nepromenjen. Argument *repl* može biti niz ili funkcija.

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

¹U *Python*-u ključna reč *None* je objekat klase *NoneType*. Može se dodeliti bilo kojoj promenljivoj, ali ne može se kreirati novi *NoneType* objekat.

`pattern.funkcija(argumenti).`

Objekti podudaranja sadrže ono što se poklopilo sa navedenim regularnim izrazom, što će u okviru *if* izraza kao takvo biti tretirano kao logička vrednost *True*. Ukoliko nije bilo podudaranja, funkcije *match()* i *search()* vraćaju *None*. Tako proveru da li je postojalo podudaranje možemo ostvariti jednostavnim *if* izrazom — `if(matchObject)`, koji će vratiti *True* ukoliko je bilo poklapanja, dok će u suprotnom vratiti *False*.

Objekti podudaranja podržavaju mnoge metode i attribute. Najiskorišćenija u alatu koji je razvijen, a u cilju izvlačenja potrebnih informacija iz poklapanja sa regularnim izrazom je:

Match.group(groupNum [, groupNum2, ...]) — Vraća jednu ili više podgrupa podudaranja. Grupa je definisana delom regularnog izraza u okviru zagrada (). Indeksiranje grupa vrši se s leva na desno, počevši od 1. Ako postoji jedan argument, rezultat je jedan string, koji odgovara grupi čiji je indeks dati argument, dok ako postoji više argumenata, rezultat je t-orka sa jednom stavkom po argumentu. Bez argumenata, *groupNum* je podrazumevano nula, tj. vraća se ceo *Match* objekat. Ako je indeks *groupNum* u opsegu [1,99], to je string koji odgovara odgovarajućoj grupi u zgradama 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 podudarao više puta, vraća se poslednje podudaranje [12].

Primer korišćenja funkcije *group* dat je na listingu 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')
>>>
```

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

Glava 4

Alat Koronka

Alat *Koronka* predstavlja alat koji automatski otkriva greške u kodu napisanom u programskom jeziku C koristeći alat *Memcheck*, a zatim ih ispravlja ukoliko je to u njegovoj moći. Alat je 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. 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*. U razvoju alata korišćen je *Python2.7*, pa su mogući neki neželjeni efekti prilikom pokretanja na novijim verzijama. Alat se pokreće sledećom komandom:

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

Navedeni arumenti podrazumevaju sledeće:

- **files** - sadrži dodatne fajlove glavnog programa (navedenog u okviru *path to c file*) koji se analizira (npr. *.h* fajlove). Ukoliko se program sastoji iz

više fajlova, neophodno je da se navedu kao argument. 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.
- **other args** - argumenti programa koji se analizira.

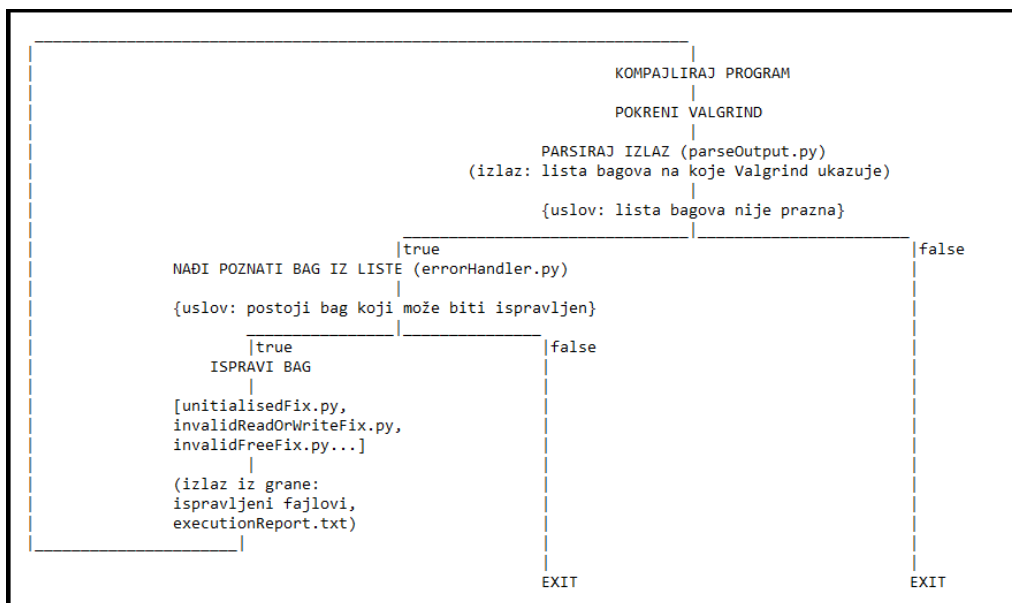
Takođe, alat *Koronka* sadrži i opciju `--help` čijim će navođenjem biti ispisano uputstvo za upotrebu alata.

Sve argumente koji se navedu prilikom pokretanja alata *Koronka* obradiće funkcija za obrađivanje argumenata komandne linije. 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. 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.txt*, predstavlja izlaz alata *Valgrind*, koji će se u procesu rada alata koristiti za parsiranje grešaka koje nađe alat *Memcheck*. Ukoliko se analizira ispravljeni program nakon završetka rada alata alatom *Memcheck*, biće dobijen isti izlaz koji možete videti u datoteci *ValgrindLOG.txt*. Način i logika rada alata biće opisani u sekcijama koje slede.

4.2 Algoritam izvršavanja

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 nađe na grešku, parsira je, nakon čega proba da je ispravi ukoliko je u mogućnosti. Ukoliko

nađe rešenje 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 4.1.



Slika 4.1: Algoritam izvršavanja alata *Koronka* sa pratećim fajlovima koji se koriste za njegovu realizaciju

4.3 Klasa greške

Prilikom ispravljanja grešaka, greška koja se trenutno obrađuje biće predstavljena kao objekat klase *ErrorInfo*. Na listingu 4.1 prikazana je pomenuta klasa.

Podaci koje klasa sadrži su sledeći:

- *errorType* - sadrži informaciju o tipu greške, npr. nevalidno oslobađanje memorije, nedozvoljeno čitanje/pisanje itd.
- *valgrindOutput* - sadrži kompletan segment teksta koji je *Valgrind* ispisao za grešku koja se trenutno obrađuje za potrebe dodatnog parsiranja informacija o grešci. Primer sadržaja za jednu grešku prikazan je na slici 4.2.
- *files* - sadrži niz fajlova koji su povezani sa greškom, a koji bi eventualno bili ispravljeni.

- *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 greške, tj. sadržaj kojim treba zameniti *changedLine* da bi greška bila ispravljena.

```
1 class ErrorInfo:
2
3     # Initialisation with errorType
4     def __init__(self, errorType, valgrindOutput, files):
5         self.errorType = errorType
6         self.valgrindOutput = valgrindOutput
7         self.files = files
8         self.changedFile = ''
9         self.changedLine = -1
10        self.problemLines = []
11        self.errorReason = []
12        self.bug = ''
13        self.bugFix = ''
```

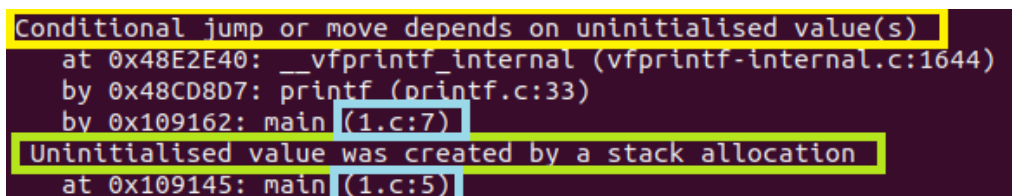
Listing 4.1: Potpis klase *ErrorInfo*

Za sve attribute klasa sadrži odgovarajuće metode za postavljanje i čitanje vrednosti, kao i funkciju *isKnownReason* koja proverava da li je razlog greške poslat kao argument, zaista razlog koji je izazvao grešku koja se obrađuje. Prikaz funkcije dat je na listingu 4.2.

```
1 def isKnownReason(self, newReason):
2     for reason in self.errorReason:
3         if reason.find(newReason) >= 0:
4             return True
5         break
6     return False
```

Listing 4.2: Prikaz funkcije koja proverava validnost razloga greške

Na slici 4.2 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 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 4.2: 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.

4.4 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 iste naredbe koje izazivaju grešku u dva različita fajla. Promena tih naredbi bez praćenja fajla nad kojim se vrši promena ne bi bila moguća u oba fajla, ukoliko je ispravka greške opet ista naredba 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 rešenje mogu biti dve potupuno iste naredbe, 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 4.2. 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 istoriji 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 novog 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. Navedena greška će ostati neispravljena, a korisnik će biti obavešten da postoji greška koja nažalost nije ispravljena, samim tim što će se izveštaj o njoj nalaziti u datoteci *ValgrindLOG.txt*. Prilikom nalaženja rešenja koje će zaista i biti implementirano, to rešenje biva dodato u istoriju, a alat nastavlja sa iteracijama ispravljanja ostalih grešaka.

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, a koje dobijamo iz izveštaja koji daje *Memcheck*. 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. listing 2.7). To je posledica principa rada alata *Memcheck*, koji 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. Nakon što se greška ispravi i program pusti na novu fazu kompilacije i analize alatom *Memcheck*, izveštaj koji će se generisati neće sadržati paragrafe koje je sadržao ranije, a za čije smo ispravljanje koristili samo jedan paragraf. Ovakva implementacija doprinosi efikasnosti alata, kako smo za otklanjanje određene greške koristili samo neophodnu količinu podataka, a ne sve podatke koje smo imali. 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 greške korišćenja neinicijalizovane vrednosti mapiran je na osnovu opisa koji daje alat *Memcheck*, tj. „*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 listingu 4.3.

```
==7863== Conditional jump or move depends on uninitialised value(s)
==7863==      at 0x48E2E40: _vfprintf_internal (vfprintf-internal.c
      :1644)
==7863==      by 0x48CD8D7: printf (printf.c:33)
==7863==      by 0x109172: main (1.c:7)
==7863== Uninitialised value was created by a stack allocation
==7863==      at 0x109155: main (1.c:5)
```

Listing 4.3: 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

$$([\ \backslash t]*)([a - zA - Z_+][\] + ([a - zA - Z0 - 9_+]).*);$$

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. Na primer, primenom navedenog regularnog izraza koristeći funkciju *search* modula *re* nad odgovarajućom linijom

koda dobijamo objekat poklapanja. Iz datog objekta, primenom funkcije *group* dobijamo segmente koje odgovaraju uparenim zagradama, i to:

- Tip promenljive na osnovu *matchObject.group(2)*.
- Ime promenljive na osnovu *matchObject.group(3)*.

Za inicijalizaciju promenljivih koristimo funkciju prikazanu na listingu 4.4, dok u slučaju nizova, odnosno višedimenzionih nizova, koristimo pomoćnu funkciju koja koristi petlje za inicijalizaciju, a koja se suštinski oslanja na funkciju sa listinga 4.4.

```
1 def initialise(varType):
2     initialisator = {
3         'int': '0',
4         'double': '0',
5         'float': '0',
6         'boolean': 'False',
7         'char': '\\\\0\\',
8         'short': '0',
9         'long': '0'
10    }
11
12    if varType in initialisator:
13        return initialisator[varType]
14    else:
15        return 'Invalid'
```

Listing 4.4: 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 listinga 4.4. Takođe, podržana je i inicijalizacija nizova, odnosno višedimenzionih nizova korisničkih struktura.

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 listingu 4.5.

```
==3756== Conditional jump or move depends on uninitialised value(s)
==3756==          at 0x48E2E40: __vfprintf_internal (vfprintf-internal.c
:1644)
==3756==          by 0x48CD8D7: printf (printf.c:33)
==3756==          by 0x1091BC: main (1.c:15)
==3756== Uninitialised value was created by a heap allocation
==3756==          at 0x483874F: malloc (in /usr/lib/x86_64-linux-gnu/
valgrind/vgpreload_memcheck-amd64-linux.so)
==3756==          by 0x10917C: main (1.c:9)
```

Listing 4.5: Ispis greške korišćenja dinamički alocirane neinicijalizovane promenljive (pokazivača)

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 kreirana. Način inicijalizacije je isti kao kod statički definisanih promenljivih, dok potrebne informacije dobijamo primenom odgovarajućih regularnih izraza. 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

$$(\textit{malloc}|\textit{calloc}|\textit{realloc})(.+);$$

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. Primenom funkcije *search* koristeći navedeni regularni izraz na osnovu *matchObject.group(1)* dobijamo informaciju koja funkcija alokacije je iskorišćena. Dalje, u zavisnosti od te funkcije primenjuju se pomenuti specijalizovani regularni izrazi i odgovarajuće transformacije nad *matchObject.group(2)* u cilju dobijanja svih preostalih neophodnih informacija. Takođe, i u ovom slučaju alat poseduje 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 kôd koji vrši inicijalizaciju, indirektno naslonjen na funkciju sa listinga 4.4 i inicijalizacione funkcije koje koriste petlje.

Nevalidno čitanje/pisanje

Šablon koji ispravlja greške nevalidnog čitanja/pisanja mapiran je na osnovu opisa koji daje alat *Memcheck*, tj. „*Invalid read of size x*” u slučaju nevalidnog čitanja, odnosno „*Invalid write of size x*” u slučaju nevalidnog pisanja, gde je *x* broj bajtova. Razlog greške kojim se ova greška mapira je „*Address adr is y bytes after a block of size z alloc'd*” u slučaju prekoračenja s desne strane, odnosno „*Address adr is y bytes before a block of size z alloc'd*” u slučaju prekoračenja s leve strane, gde *adr* predstavlja heksadekadni zapis adrese, a *y* i *z* broj bajtova. To se jasno može videti na listingu 4.6.

```
==4289== Invalid write of size 4
==4289==          at 0x10917A: main (4.c:8)
==4289== Address 0x4a59054 is 4 bytes after a block of size 16 alloc'
d
==4289==          at 0x483874F: malloc (in /usr/lib/x86_64-linux-gnu/
valgrind/vgpreload_memcheck-amd64-linux.so)
==4289==          by 0x10916D: main (4.c:7)
```

Listing 4.6: Ispis greške nevalidnog čitanja

Analizom segmenta sa listinga 4.6, 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. Moguća ispravka je adekvatno računanje izraza koji se koristi za indeksiranje, ali tu vrstu ispravke ovde nije moguće uraditi. Druga ispravka bi bila da memoriju koju program alocira, a sa kojom je pokušana interakcija, proširimo za vrednost koju dobijamo primenom pomenu-tih informacija. Takođe, na listingu 4.6 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 bajtova koji dobijemo analizom greške. Na ovaj način ispravljamo greške koje su povezane sa prekoračenjem s desne strane.

U slučaju prekoračenja s leve strane, pristupamo memoriji kojoj nam nije dozvoljeno, odnosno arument pristupa je vrednost manja od 0, kako znamo da indeksi pristupa alociranoj memoriji kreću od 0. Ovu grešku ispravljamo tako što pogrešan indeks pristupa zamenimo njegovom apsolutnom vrednošću i na taj na-

čin ispravni indeksi ostaju takvi, dok pogrešne stavljamo u interval ispravnih. U slučaju da apsolutna vrednost pomenutog indeksa izaziva grešku prekoračenja s desne strane, memorija će biti proširena, odnosno ispravljajući se greška nevalidnog čitanja/pisanja u slučaju prekoračenja s desne strane.

Nevalidno oslobađanje memorije

Šablon koji ispravlja greške nevalidnog oslobađanja memorije mapiran je na osnovu opisa koji daje alat *Memcheck*, tj. „*Invalid free() / delete / delete[] / realloc()*”. Za ovaj tip greške razlog nije od presudnog značaja, tako da se mapiranje vrši samo nad opisom greške. Na listingu 4.7 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.

```
==2638== Invalid free() / delete / delete[] / realloc()
==2638==      at 0x483997B: free (in /usr/lib/x86_64-linux-gnu/
        valgrind/vgpreload_memcheck-amd64-linux.so)
==2638==      by 0x10919F: main (2.c:11)
==2638== Address 0x4a590a0 is 0 bytes inside a block of size 12 free'
        d
==2638==      at 0x483997B: free (in /usr/lib/x86_64-linux-gnu/
        valgrind/vgpreload_memcheck-amd64-linux.so)
==2638==      by 0x109193: main (2.c:10)
==2638== Block was alloc'd at
==2638==      at 0x483874F: malloc (in /usr/lib/x86_64-linux-gnu/
        valgrind/vgpreload_memcheck-amd64-linux.so)
==2638==      by 0x109183: main (2.c:9)
```

Listing 4.7: Ispis greške nevalidnog oslobađanja memorije

Sumnjive vrednosti argumenata

Šablon koji ispravlja greške sumnjivih vrednosti argumenata mapiran je na osnovu opisa koji daje alat *Memcheck*, tj. „*Argument 'size' of function x has a fishy (possibly negative) value: y*”, gde je *x* naziv funkcije, a *y* vrednost argumenta. Za ovaj tip greške mapiranje se vrši samo nad opisom greške, kako je sam razlog nedozvoljena vrednost argumenta, što *Memcheck* ispisuje u okviru opisa

greške. Na listingu 4.8 prikazan je ispis greške koji koristimo da bismo je ispravili. Analizom ispisa sa slike dobijamo preciznu informaciju o liniji koda, kao i o fajlu gde gde je pozvana funkcija koja sadrži sumnjivi argument. U zavisnosti od funkcije, odgovarajućim transformacijama nad linijom koda u kojoj je poziv te funkcije, nalazimo mesto gde je sumnjivi argument. Nažalost, nije uvek moguće automatsko ispravljanje grešaka ovog tipa na uniforman način. Nedoželjena vrednost argumenta može biti na primer posledica pogrešno sračunatog izraza, što nije moguće ispraviti automatski. Ispravka koja je implementirana obezbeđuje da program nema *Runtime Error*, pa omogućava da se vrši debugovanje i lakše pronadu semantičke greške programa. Dakle, ovu grešku ispravljamo tako što izmenimo sumnjivi argument u pozivu funkcije u okviru problematične linije ispravnim, odnosno računamo apsolutnu vrednost sumnjivog argumenta.

```
==3565==  Argument 'size' of function realloc has a fishy (possibly
         negative) value: -12
==3565==          at 0x483AD4B: realloc (in /usr/lib/x86_64-linux-gnu/
         valgrind/vgpreload_memcheck-amd64-linux.so)
==3565==          by 0x109194: main (15.c:7)
```

Listing 4.8: Ispis greške sumnjive vrednosti argumenta

Korišćenje neadresirane ili neinicijalizovane vrednosti u sistemskom pozivu

Šablon koji ispravlja greške korišćenja neadresirane ili neinicijalizovane vrednosti u sistemskom pozivu mapiran je na osnovu opisa koji daje alat *Memcheck*, npr. „*Syscall param write(buf) points to uninitialised byte(s)*”, „*Syscall param exit_group(status) contains uninitialised byte(s)*”, ili nekim drugim opisom karakterističnim za određenu vrstu funkcije. Za ovaj tip greške mapiranje se vrši samo nad opisom greške, kako grešku izazivaju loši argumenti funkcija sistemskog poziva. Na listingu 4.9 prikazan je jedan od ispisa ove greške, u konkretnom slučaju korišćenja funkcije *write* koji koristimo da bismo je ispravili. Analizom ispisa sa slike dobijamo preciznu informaciju o liniji koda, kao i o fajlu gde gde je pozvana funkcija koja izaziva grešku, kao i linije koda gde je definisana neinicijalizovana ili neadresirana vrednost koja je u pozivu funkcije iskorišćena. Ispravka greške sastoji se u pronalaženju linije gde je definisana promenljiva i njena zamena ispravljenom linijom ili blokom koda koji sadrži tu liniju i dodatni kôd koji ispravlja grešku.

```
==5695== Syscall param write(buf) points to uninitialised byte(s)
==5695==      at 0x25A48723: __write_nocancel (in /lib/tls/libc-2.3.3.
      so)
==5695==      by 0x259AFAD3: __libc_start_main (in /lib/tls/libc-2.3.3.
      so)
==5695==      by 0x8048348: (within /auto/homes/njn25/grind/head4/a.out
      )
==5695== Address 0x25AB8028 is 0 bytes inside a block of size 10
      alloc'd
==5695==      at 0x259852B0: malloc (vg_replace_malloc.c:130)
==5695==      by 0x80483F1: main (a.c:5)
```

Listing 4.9: Primer ispisa greške korišćenja neinicijalizovane ili neadresirane vrednosti u sistemskom pozivu

4.6 Primeri rada alata

U ovoj sekciji biće predstavljeni primeri rada alata za svaku od grešaka koje alat ispravlja. Prvi primer će dati kompletan pregled onoga što alat *Koronka* radi sa programom koji analizira i šta ostavlja nakon svog rada. Ostali primeri biće izneti u formi listinga sa sledećom sadržinom:

- Program koji sadrži greške;
- Ispravljeni program alatom *Koronka*;
- Izveštaj koji je formirao alat *Koronka*.

Primeri koji su prikazani u ovoj sekciji deo su skupa test primera koji se nalaze u direktorijumu *Examples* u okviru glavnog direktorijuma alata. Podeljeni su po direktorijumima tako da svaki direktorijum sadrži program koji sadrži određene vrste grešaka. Takođe, mogu se naći i pokretanja i ispravke koje je *Koronka* implementirala nad tim test primerima. Ti fajlovi smešteni su u odgovarajući direktorijum formata *datum pokretanja-vreme pokretanja*, koje je alat *Koronka* kreirala prilikom izvršavanja. Skup test primera dostupan je na linku https://github.com/LMladenovic/Error_fixing_tool/tree/master/Examples.

Analiza i ispravka programa sa slike 4.3 predstavljena je u ovom primeru.

Dati program sadrži grešku upotrebe neinicijalizovane promenljive koja će biti ispravljena. Alat pokrećemo komandom u komandnoj liniji:

```
home > student > GIT > Error_fixing_tool > C program.c
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 4.3: Izgled programa koji sadrži grešku

python koronka.py program.c.

Izgled komandne linije nakon pokretanja alata prikazan je na slici 4.4.

```
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 #####

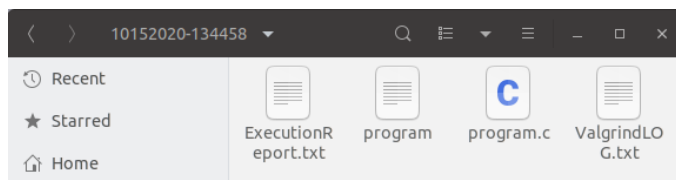
##### RUN STARTED #####
0

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

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

Slika 4.4: Izgled terminala nakon izvršavanja alata *Koronka*

Direktorijum koji je formiran i dodeljen ovom konkretnom pokretanju alata i njegovom izvršavanju prikazan je na slici 4.5, dok su izmenjen program nakon izvršavanja alata *Koronka*, kao i izveštaj, prikazani na slikama 4.6 i 4.7.



Slika 4.5: Izgled foldera u formatu *datum pokretanja - vreme pokretanja* koji formira alat *Koronka*

```
home > student > GIT > Error_fixing_tool > 10152020-134458 > C 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 4.6: Izgled ispravljenog programa alatom *Koronka*

```
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 4.7: Izgled izveštaja nakon izvršavanja alata *Koronka*

Primer programa koji sadrži greške korišćenja neinicijalizovanih ili neadresiranih vrednosti u sistemskom pozivu dat je na listingu 4.10, dok su ispravljeni program alatom *Koronka* i prateći izveštaj dati na listinzima 4.11 i 4.12.

```
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  }
```

Listing 4.10: Program koji sadrži greške korišćenja neinicijalizovanih ili neadresiranih vrednosti u sistemskom pozivu

```
1  #include <stdlib.h>
2  #include <unistd.h>
3
4  int main( void ){
5      char* arr  = calloc( sizeof(char), 10);
```

```

6  int*  arr2 = malloc(sizeof(int));
7  write( 1 /* stdout */, arr, 10 );
8  exit(0);
9  }

```

Listing 4.11: Ispravljeni program sa listinga 4.10 alatom *Koronka*

```

##### Based on Valgrind output: #####

Syscall param write(buf) points to uninitialised byte(s)
at 0x4978024: write (write.c:26)
by 0x10918E: main (16.c:7)
Address 0x4a59040 is 0 bytes inside a block of size 10 alloc'd
at 0x483874F: malloc (in /usr/lib/x86_64-linux-gnu/valgrind/
    vgppreload_memcheck-amd64-linux.so)
by 0x109166: main (16.c:5)
Uninitialised value was created by a heap allocation
at 0x483874F: malloc (in /usr/lib/x86_64-linux-gnu/valgrind/
    vgppreload_memcheck-amd64-linux.so)
by 0x109166: main (16.c:5)

##### Koronka made following changes in 16.c #####

Changed 5. line
    char* arr  = malloc(10);
with
    char* arr  = calloc( sizeof(char), 10);

##### Based on Valgrind output: #####

Syscall param exit_group(status) contains uninitialised byte(s)
at 0x494C926: _Exit (_exit.c:31)
by 0x48B23A9: __run_exit_handlers (exit.c:132)
by 0x48B23D9: exit (exit.c:139)
by 0x1091B0: main (16.c:8)
Uninitialised value was created by a heap allocation
at 0x483874F: malloc (in /usr/lib/x86_64-linux-gnu/valgrind/
    vgppreload_memcheck-amd64-linux.so)
by 0x109189: main (16.c:6)

##### Koronka made following changes in 16.c #####

Changed 8. line

```

```
    exit(arr2[0]);  
with  
    exit(0);
```

Listing 4.12: Izveštaj o radu alata *Koronka* za program sa listinga 4.10

Primer programa koji sadrži grešku nevalidnog oslobađanja memorije dat je na listingu 4.13, dok su ispravljeni program alatom *Koronka* i prateći izveštaj dati na listinzima 4.14 i 4.15.

```
1 #include <stdio.h>  
2 #include <stdlib.h>  
3  
4 int main()  
5 {  
6     char *p, q;  
7  
8     p = (char *) malloc(19);  
9     p = (char *) malloc(12);  
10    free(p);  
11    free(p);  
12  
13    p = &q;  
14    free(p);  
15  
16    return 0;  
17 }
```

Listing 4.13: Program koji sadrži grešku nevalidnog oslobađanja memorije

```
1 #include <stdio.h>  
2 #include <stdlib.h>  
3  
4 int main()  
5 {  
6     char *p, q;  
7  
8     p = (char *) malloc(19);  
9     p = (char *) malloc(12);  
10    free(p);  
11  
12  
13    p = &q;  
14
```



```

15
16     return 0;
17 }

```

Listing 4.14: Ispravljeni program sa listinga 4.13 alatom *Koronka*

```

##### Based on Valgrind output: #####

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)

##### Koronka made following changes in 2.c #####

Removed 11. line
    free(p);

##### Based on Valgrind output: #####

Invalid free() / delete / delete[] / realloc()
at 0x483997B: free (in /usr/lib/x86_64-linux-gnu/valgrind/
    vgpreload_memcheck-amd64-linux.so)
by 0x1091A7: main (2.c:14)
Address 0x1ffefffcef is on thread 1's stack
in frame #1, created by main (2.c:5)

##### Koronka made following changes in 2.c #####

Removed 14. line
    free(p);

```

Listing 4.15: Izveštaj o radu alata *Koronka* za program sa listinga 4.13

Primer programa koji sadrži grešku nevalidnog čitanja/pisanja dat je na listingu 4.16, dok su ispravljeni program alatom *Koronka* i prateći izveštaj dati na

listinzima 4.17 i 4.18.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(){
5
6     int *t = malloc(4*sizeof(int));
7
8     printf("Right side overdraft: %d\n", t[4]);
9     printf("Left side overdraft: %d\n", t[4-5]);
10    free(t);
11    return 0;
12 }
```

Listing 4.16: Program koji sadrži grešku nevalidnog čitanja/pisanja

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(){
5
6     int *t = malloc(4*sizeof(int) + 1*sizeof(int));
7     int __index__;
8     for( __index__ = 0; __index__ < 5; __index__ ++){
9         t[__index__] = 0;
10
11
12     printf("Right side overdraft: %d\n", t[4]);
13     printf("Left side overdraft: %d\n", t[abs(4-5)]);
14     free(t);
15     return 0;
16 }
```

Listing 4.17: Ispravljeni program sa listinga 4.16 alatom *Korona*

```
##### Based on Valgrind output: #####
```

```
Invalid read of size 4
at 0x109173: main (14.c:8)
Address 0x4a59050 is 0 bytes after a block of size 16 alloc'd
at 0x483874F: malloc (in /usr/lib/x86_64-linux-gnu/valgrind/
    vgppreload_memcheck-amd64-linux.so)
by 0x109166: main (14.c:6)
```

Koronka made following changes in 14.c

Changed 6. line

```
    int *t = malloc(4*sizeof(int));  
with  
    int *t = malloc(4*sizeof(int) + 1*sizeof(int));
```

Based on Valgrind output:

Conditional jump or move depends on uninitialised value(s)
at 0x48E2E40: __vfprintf_internal (vfprintf-internal.c:1644)
by 0x48CD8D7: printf (printf.c:33)
by 0x109187: main (14.c:8)
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 0x109166: main (14.c:6)

Koronka made following changes in 14.c

Changed 6. line

```
    int *t = malloc(4*sizeof(int) + 1*sizeof(int));  
with  
    int *t = malloc(4*sizeof(int) + 1*sizeof(int));  
        int __index__;  
    for( __index__ = 0; __index__ < 5; __index__ ++)  
        t[__index__] = 0;
```

Based on Valgrind output:

Invalid read of size 4
at 0x1091BD: main (14.c:13)
Address 0x4a5903c is 4 bytes before a block of size 20 alloc'd
at 0x483874F: malloc (in /usr/lib/x86_64-linux-gnu/valgrind/
vgpreload_memcheck-amd64-linux.so)
by 0x109166: main (14.c:6)

Koronka made following changes in 14.c

Changed 13. line

```
    printf("Left side overdraft: %d\n", t[4-5]);  
with
```

```
printf("Left side overdraft: %d\n", t[abs(4-5)]);
```

Listing 4.18: Izveštaj o radu alata *Koronka* za program sa listinga 4.16

Primer programa koji sadrži grešku sumnjive vrednosti arguenata dat je na listingu 4.19, dok su ispravljeni program alatom *Koronka* i prateći izveštaj dati na listinzima 4.20 i 4.21.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(int argv, char** argc){
5     char *r = malloc(-15);
6     int *t = malloc(3);
7     t = (int *)realloc(t, -3*sizeof(int));
8     free(r);
9     free(t);
10    return 0;
11 }
```

Listing 4.19: Program koji sadrži grešku sumnjive vrednosti arguenata

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(int argv, char** argc){
5     char *r = malloc(abs(-15));
6     int *t = malloc(3);
7     t = (int *)realloc(t,abs( -3*sizeof(int)));
8     free(r);
9     free(t);
10    return 0;
11 }
```

Listing 4.20: Ispravljeni program sa listinga 4.19 alatom *Koronka*

```
##### Based on Valgrind output: #####

Argument 'size' of function malloc has a fishy (possibly negative)
value: -15
at 0x483874F: malloc (in /usr/lib/x86_64-linux-gnu/valgrind/
vgpreload_memcheck-amd64-linux.so)
by 0x10916F: main (15.c:5)

##### Koronka made following changes in 15.c #####
```

```
Changed 5. line
    char *r = malloc(-15);
with
    char *r = malloc(abs(-15));

##### Based on Valgrind output: #####

Argument 'size' of function realloc has a fishy (possibly negative)
value: -12
at 0x483AD4B: realloc (in /usr/lib/x86_64-linux-gnu/valgrind/
vgpreload_memcheck-amd64-linux.so)
by 0x109192: main (15.c:7)

##### Koronka made following changes in 15.c #####

Changed 7. line
    t = (int *)realloc(t, -3*sizeof(int));
with
    t = (int *)realloc(t,abs( -3*sizeof(int)));
```

Listing 4.21: Izveštaj o radu alata *Koronka* za program sa listinga 4.19

Primer programa koji sadrži grešku za čije ispravljanje se koriste funkcije inicijalizacije korisnički definisanih struktura dat je na listingu 4.22, dok su ispravljani program alatom *Koronka* i prateći izveštaj dati na listinzima 4.23 i 4.24.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 typedef struct _point{
5     int x;
6     int y;
7     int consts[3];
8 }Point;
9
10 int main(){
11
12     Point A;
13
14     Point B[5];
15
16     printf("Point A(%d,%d)\n", A.x, A.y);
17
```

```
18  int i;
19  for (i=0;i<5;i++)
20      printf("B[%d] = (%d, %d)\n", i+1, B[i].x, B[i].y);
21
22  return 0;
23 }
```

Listing 4.22: Program koji sadrži grešku za čije ispravljanje se koriste funkcije inicijalizacije korisnički definisanih struktura

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  typedef struct _point{
5      int x;
6      int y;
7      int consts[3];
8  }Point;
9
10 int main(){
11
12     Point A;
13     A.x=0;
14     A.y=0;
15     int __index__;
16     for( __index__ = 0; __index__ < 3; __index__ ++){
17         A.consts[__index__] = 0;
18
19     Point B[5];
20     int __index2__;
21     for( __index2__ = 0; __index2__ < 5; __index2__ ++){
22         B[__index2__].x=0;
23         B[__index2__].y=0;
24         int __index3__;
25         for( __index3__ = 0; __index3__ < 3; __index3__ ++){
26             B[__index2__].consts[__index3__] = 0;
27
28         }
29
30     printf("Point A(%d,%d)\n", A.x, A.y);
31
32     int i;
33     for (i=0;i<5;i++)
```

```

34     printf("B[%d] = (%d, %d)\n", i+1, B[i].x, B[i].y);
35
36     return 0;
37 }

```

Listing 4.23: Ispravljeni program sa listinga 4.22 alatom *Koronka*

```

##### Based on Valgrind output: #####

Conditional jump or move depends on uninitialised value(s)
at 0x48E2E40: __vfprintf_internal (vfprintf-internal.c:1644)
by 0x48CD8D7: printf (printf.c:33)
by 0x10917D: main (12.c:16)
Uninitialised value was created by a stack allocation
at 0x109149: main (12.c:10)

##### Koronka made following changes in 12.c #####

Changed 12. line
    Point A;
with
    Point A;
    A.x=0;
    A.y=0;
    int __index__;
    for( __index__ = 0; __index__ < 3; __index__ ++ )
        A.consts[__index__] = 0;

##### Based on Valgrind output: #####

Conditional jump or move depends on uninitialised value(s)
at 0x48E2E40: __vfprintf_internal (vfprintf-internal.c:1644)
by 0x48CD8D7: printf (printf.c:33)
by 0x109228: main (12.c:25)
Uninitialised value was created by a stack allocation
at 0x109145: main (12.c:10)

##### Koronka made following changes in 12.c #####

Changed 19. line
    Point B[5];
with
    Point B[5];

```

```
int __index2__;  
for( __index2__ = 0; __index2__ < 5; __index2__ ++){  
    B[__index2__].x=0;  
    B[__index2__].y=0;  
    int __index3__;  
    for( __index3__ = 0; __index3__ < 3; __index3__ ++)  
        B[__index2__].consts[__index3__] = 0;  
}
```

Listing 4.24: Izveštaj o radu alata *Koronka* za program sa listinga 4.22

Glava 5

Zaključak

Alatima *Valgrind* distribucije mogu se otkriti greške i kritične tačke programa čijim ispravljanjem direktno poboljšavamo konzistentnost, funkcionalnost i performanse programa koji se razvija. Ručno otkrivanje tih grešaka obično je mukotrpan i najčešće neefikasan postupak. Srećom, alat *Valgrind* je tu da pomogne u prevazilaženju tog problema. Alat *Memcheck*, koji je deo *Valgrind* distribucije, otkriva greške u radu sa memorijom koje mogu da dovedu do pada programa, a koje kompilator nije u mogućnosti da otkrije. Ostali alati *Valgrind* distribucije mogu da otkriju druge probleme, poput grešaka u radu sa nitima i sl., čijom se detekcijom i ispravljanjem može doprineti performansama i boljem radu softvera koji se razvija.

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 razvijen 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. Alat pokriva širok dijapazon grešaka, što ne isključuje da postoje greške koje mogu biti otkrivene daljim razvojem, a koje alat *Koronka* u ovom trenutku ne pokriva. Greške koje alat *Koronka* uspešno otklanja su :

- Korišćenje neinicijalizovanih vrednosti kako primitivnog tipa, tako i korisnički definisanih struktura, bilo da su one statički definisane, ili dinamički alocirane;
- Nedozvoljeno čitanje i pisanje u memoriju, kako s leve, tako i s desne strane, a koja je dinamički alocirana;

- Sumnjive vrednosti argumenata;
- Nevalidna oslobađanja memorije;
- Nevalidni arugmenti sistemskih poziva.

Greške koje alat *Koronka* nije u stanju da ispravi, a koje *Memcheck* detektuje su specijalni slučajevi grešaka koje alat ispravlja, a koje šabloni ne pokrivaju, ili greške za koje šablon još uvek nije implementiran. To su na primer greške nedozvoljenog čitanja i pisanja iz statičke memorije i greške prilikom korišćenja funkcija poput *mmap* i *memcpy*.

Što se daljeg razvoja alata tiče, mogu se kreirati novi i 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*. Na primer, može se iskoristiti alat *Helgrind* koji detektuje greške u radu sa nitima. Nakon ispravljanja grešaka koje detektuje *Memcheck*, program bi se prepustio na dalju iterativnu analizu alatu *Helgrind*, i šablonima za ispravljanje grešaka u radu sa nitima, što je slično principu koji je već iskorišćen. Nažalost, alate poput *Cachegrind*-a i *Callgrind*-a nije moguće iskoristiti u ovakvoj arhitekturi, kako nije moguće iskoristiti automatsku analizu njihovih izveštaja za ispravljanje propusta na koje oni ukazuju.

Literatura

- [1] *Valgrind Documentation, Release 3.16.0 27 May 2020*. 2000-2019.
- [2] BBV: an experimental basic block vector generation tool, 2000-2020. URL: <https://www.valgrind.org/docs/manual/bbv-manual.html>.
- [3] Cachegrind: a cache and branch-prediction profiler, 2000-2020. URL: <https://www.valgrind.org/docs/manual/cg-manual.html>.
- [4] Callgrind: a call-graph generating cache and branch prediction profiler, 2000-2020. URL: <https://www.valgrind.org/docs/manual/cl-manual.html>.
- [5] DHAT: a dynamic heap analysis tool, 2000-2020. URL: <https://www.valgrind.org/docs/manual/dh-manual.html>.
- [6] DRD: a thread error detector, 2000-2020. URL: <https://www.valgrind.org/docs/manual/drd-manual.html>.
- [7] Helgrind: a thread error detector, 2000-2020. URL: <https://www.valgrind.org/docs/manual/hg-manual.html>.
- [8] Massif: a heap profiler, 2000-2020. URL: <https://www.valgrind.org/docs/manual/ms-manual.html>.
- [9] Memcheck : a memory error detector, 2000-2020. URL: <https://www.valgrind.org/docs/manual/mc-manual.html>.
- [10] Using and understanding the Valgrind core, 2000-2020. URL: <https://www.valgrind.org/docs/manual/manual-core.html>.
- [11] Valgrind, 2000-2020. URL: <https://valgrind.org>.
- [12] re — Regular expression operations, 2001-2020. URL: <https://docs.python.org/3/library/re.html>.

LITERATURA

- [13] doc. dr Milena Vujošević Jančić. Dinamička analiza, 2019. URL: http://www.verifikacijasoftware.matf.bg.ac.rs/vs/predavanja/03_dinamicka_analiza/03_dinamicka_analiza.pdf.
- [14] doc. dr Milena Vujošević Jančić. Verifikacija softvera - Motivacija, 2019. URL: http://www.verifikacijasoftware.matf.bg.ac.rs/vs/predavanja/01_uvod/02_motivacija.pdf.
- [15] 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.