Analiza projekta korišćenjem alata za verifikaciju softvera

Samostalni seminarski rad u okviru kursa
Verifikacija softvera
Matematički fakultet

Dunja Čitlučanin, 1024/2022
dunja.citlucanin1@gmail.com

28. avgust 2023.

Sažetak

U ovom radu biće prikazana analiza projekta `TankAttack`, dobijena primenom alata za verifikaciju softvera, i ukratko će biti opisani alati i naredbe koji su korišćeni u tu svrhu. Analizirani projekat nalazi se na sledećoj adresi:https://gitlab.com/matf-bg-ac-rs/course-rs/projects-2020-2021/17-tankattack, a autori su Nikola Mićić, Luka Miletić, Nikola Lazarević, Slobodan Jovanović i Mihailo Trišović. Projekat je nastao u okviru kursa Razvoj softvera.

Sadržaj

1 Uvod

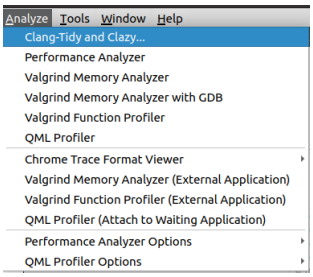
`TankAttack` je igrica,namenjena za dva igrača, koja simulira bitku između tenkova koji se nalaze u lavirintu. Cilj je eliminsati protivnika-tenkovi inicijalno ispaljuju tenkovsko đule, a tokom bitke moguće je i skupljanje supermoćnog oružja koje olakšava eliminaciju protivnika. Takođe, tokom igrice, moguće je i skupljanje srca koja omogućavaju punjenje helti na maksimum. Tenk koji prvi osvoji 3 pobede je pobednik. Za testiranje projekta korišćeni su alati: `Clang-Tidy i Clazy, Lcov, Cppcheck i Clangd`.

2 Clang-Tidy i Clazy

Prvi alat koji ćemo primeniti je alat za statičku analizu. U pitanju je `Clang-Tidy`. On se koristi za detekciju grešaka i upotorenja, kao i stilskih problema u `C++` kodu. Veoma je koristan zato što omogućava otkrivanje potencijalnih problema u kodu pre kompilacije i njihovo ispravljanje u fazi razvoja.

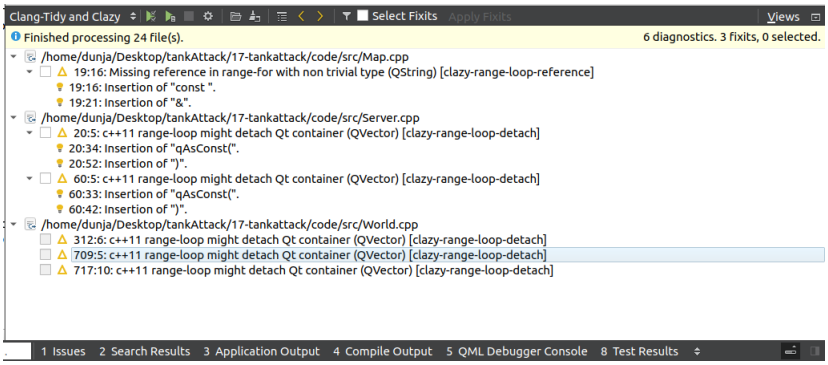
Dodatak ovom alatu je alat `Clazy`, koji je razvijen posebno za Qt aplikacije. Ovaj alat može da detektuje potencijalno curenje memorije, nepravilne konverzije tipova podataka, neefikasne upotrebe Qt API-ja, često se koristi u Qt projektima i omogućava otkrivanje problema specifičnih za ovu tehnologiju. Kada se koriste zajedno, `Clang-Tidy` i `Clazy`, pružaju dublju analizu Qt koda, poboljšanje kvaliteta i bezbednosti.

Pokazaćemo upotrebu ovih alata pomoću QtCreator-a. Potrebno je da klikom na *Analyze* izaberemo opciju *Clang-Tidy and Clazy*, a potom treba da izaberemo fajlove na koje želimo da primenimo analizu i pokrenemo je klikom na dugme *Analyze*.



Slika 1: Clang

Rezultat ovih alata prikazan je na slici 2.

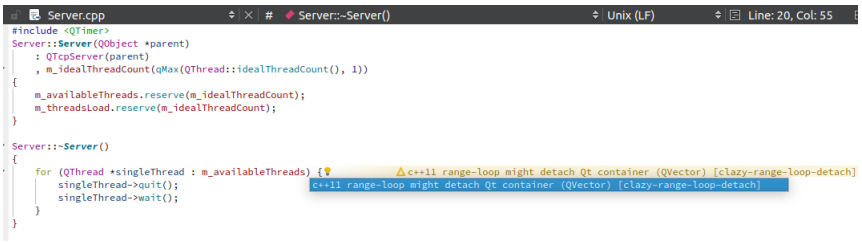


Slika 2: Rezultat primene alata Clang-Tidy i Clazy

Prva poruka, odnosno poruka prikazana na slici 3, koja se odnosi na fajl `Map.cpp`, ukazuje na to da nije korišćena referenca u `for` petlji, što znači da će se kopirati elementi iz opsega. Takođe, poruka ukazuje na problem zbog toga što se for petlja koristi za prolazak kroz ne-trivijalni tip podataka koji obično imaju složene operacije prilikom kopiranja (konstruktore, destruktore...), u ovom slučaju QString. Ovu poruku možemo rešiti tako što dodamo referencu u for pelji čime bismo izbegli kopiranje elemenata.

Slika 3: Poruka koja se odnosi na fajl Map.cpp

Sledeća poruka koju ćemo komenarisati, pojavljuje u analizi fajlova `Server.cpp` i `World.cpp`, prikazana je na slici 4. Ovo upozorenje ukazuje na potencijalni problem prilikom korišćenja `for` petlje sa opsegom za iteriranje kroz Qt kontejnere, kao što je QVector. Postoji opasnost od odvajanja kontejnera, tj. situacije u kojoj se napravi kopija kontejnera kako bi se izvršila iteracija, što može da dovede do promena u originalnom kontejneru ako se on menja u telu petlje. Jedan od načina da se reši ovaj problem je upotrebom const reference u `for` petlji.



Slika 4: Poruka koja se pojavljuje za fajl Server.cpp (kao i za World.cpp

3 LCOV

Sledeći alat koji ćemo primeniti jeste alat za određivanje pokvrivenosti koda prilikom izvršavanja programa. Alat `gcov` dolazi uz gcc kompajler i koristi se zajedno sa njim prilikom analize programa. Korišćenjem ovog alata dobijamo evidenciju toga koje i koliko puta su izvršene grane ,funkcije, linije, itd. Međutim, dobijeni izveštaj nije baš čitljiv, pa iz tog razloga, koristimo alat `lcov`.
Alat `lcov` možemo jednostavno instalirati pomoću sledeće naredbe:

```
sudo apt install lcov
```

Prilikom kompilacije potrebno je koristiti dodatne opcije:

```
-fprofile-args
-ftest-coverage
-O0
```

Prve dve opcije mogu se zameniti jednom:

```
--coverage
```

Dakle, prvi korak je dodati sledeće flagove u fajl tank_attack.pro:

```
QMAKE_CXXFLAGS_RELEASE_WITH_DEBUGINFO -= -O0
QMAKE_CXXFLAGS += --coverage
QMAKE_LFLAGS += --coverage
```

Za prevođenje projekta korišćen je qmake. Dodatno generisani fajlovi imaju ekstenziju .gcno, a nakon izvršavanja programa biće generisani fajlovi sa ekstenzijom .gcda. Sada je vreme da pokrenemo alat lcov. To radimo sledećom naredbom:

```
lcov --rc lcov_branch_coverage=1 -c -d . -o coverage_test.info
```

Ovom linijom menjamo konfiguracioni parametar koji testira pokrivenost grana, kreiramo izveštaj o pokrivenosti, za direktorijum biramo tekući direktorijum i preusmeravamo izlaz u navedeni fajl.
Ovom naredbom dobili smo izveštaj i o datotekama čija nas pokrivenost ne zanima, pa njih uklanjamo sledećom naredbom:

```
lcov --rc lcov_branch_coverage=1 -r coverage_test.info '/usr/*' '/opt/*' '*.moc'
-o coverage_filtered.info
```

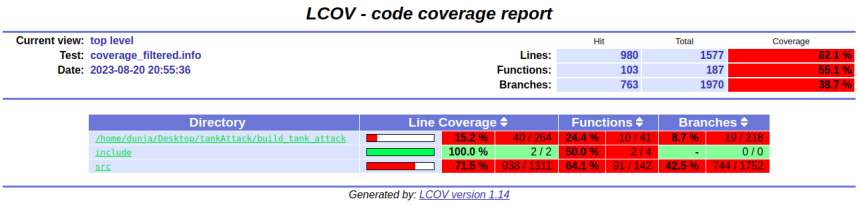
Kako bismo dobili grafički prikaz pokrećemo sledeću komandu:

```
genhtml --rc lcov_branch_coverage=1 -o Reports coverage_filtered.info
```

```
dunja@dunja-HP-Notebook:~/Desktop/tankAttack/build_tank_attack$ genhtml --rc lcov_branch_coverage=1 -o Reports coverage_filtered.info
Reading data file coverage_filtered.info
Found 22 entries.
Found common filename prefix "/home/dunja/Desktop/tankAttack/17-tankattack/code"
Writing .css and .png files.
Generating output.
Processing file include/Wall.hpp
Processing file include/SuperPower.hpp
Processing file src/World.cpp
Processing file src/Map.cpp
Processing file src/Input.cpp
Processing file src/Tank.cpp
Processing file src/main.cpp
Processing file src/Client.cpp
Processing file src/ServerWorker.cpp
Processing file src/Server.cpp
Processing file src/SuperPower.cpp
Processing file src/HealthBar.cpp
Processing file src/Rocket.cpp
Processing file src/Wall.cpp
Processing file /home/dunja/Desktop/tankAttack/build_tank_attack/moc_Server.cpp
Processing file /home/dunja/Desktop/tankAttack/build_tank_attack/qrc_resources.cpp
Processing file /home/dunja/Desktop/tankAttack/build_tank_attack/moc_Client.cpp
Processing file /home/dunja/Desktop/tankAttack/build_tank_attack/moc_World.cpp
Processing file /home/dunja/Desktop/tankAttack/build_tank_attack/moc_ServerWorker.cpp
Processing file /home/dunja/Desktop/tankAttack/build_tank_attack/moc_Rocket.cpp
Processing file /home/dunja/Desktop/tankAttack/build_tank_attack/moc_Tank.cpp
Processing file /home/dunja/Desktop/tankAttack/build_tank_attack/moc_HealthBar.cpp
Writing directory view page.
Overall coverage rate:
  lines......: 62.1% (980 of 1577 lines)
  functions..: 55.1% (103 of 187 functions)
  branches...: 38.7% (763 of 1970 branches)
```

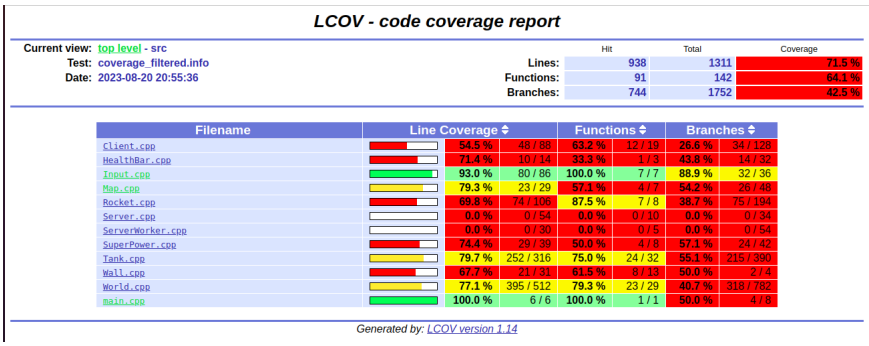
Slika 5: Terminal nakon pokretanja genhtml

U folderu `Reports` nalazi se fajl `index.html`, koji predstavlja generisan html izveštaj.



Slika 6: LCOV izveštaj

Alat pruža detaljne informacije o pokrivenosti koda na različitim nivoima, uključujući linije koda, funkcije i grane. Vidimo da u ovom projektu najveću pokrivenost imaju linije, zatim funkcije, pa onda grane. Procenat pokrivenosti nije baš dobar, vidimo i da su rezultati obojeni crvenom bojom što ukazuje na to (bolji procenti pokrivenosti obojeni su žutom i zelenom bojom)



LCOV - code coverage report

Current view: top level - src
Test: coverage_filtered.info
Date: 2023-08-20 20:55:36

	Hit	Total	Coverage
Lines:	938	1311	71.5 %
Functions:	91	142	64.1 %
Branches:	744	1752	42.5 %

Filename	Line Coverage		Functions		Branches	
Client.cpp	54.5 %	48 / 88	63.2 %	12 / 19	26.6 %	34 / 128
HealthBar.cpp	71.4 %	10 / 14	33.3 %	1 / 3	43.8 %	14 / 32
Input.cpp	93.0 %	80 / 86	100.0 %	7 / 7	88.9 %	32 / 36
Map.cpp	79.3 %	23 / 29	57.1 %	4 / 7	54.2 %	26 / 48
Rocket.cpp	69.8 %	74 / 106	87.5 %	7 / 8	38.7 %	75 / 194
Server.cpp	0.0 %	0 / 54	0.0 %	0 / 10	0.0 %	0 / 34
ServerWorker.cpp	0.0 %	0 / 30	0.0 %	0 / 5	0.0 %	0 / 54
SuperPower.cpp	74.4 %	29 / 39	50.0 %	4 / 8	57.1 %	24 / 42
Tank.cpp	79.7 %	252 / 316	75.0 %	24 / 32	55.1 %	215 / 390
Wall.cpp	67.7 %	21 / 31	61.5 %	8 / 13	50.0 %	2 / 4
World.cpp	77.1 %	395 / 512	79.3 %	23 / 29	40.7 %	318 / 782
main.cpp	100.0 %	6 / 6	100.0 %	1 / 1	50.0 %	4 / 8

Generated by: LCOV version 1.14

Slika 7: LCOV izveštaj

Za fajl `Input.cpp` vidimo odličnu pokrivenost linija i funkcija i nešto malo manju grana. `Map.cpp` ima dobru pokrivenost linija `Rocket.cpp` ima dobru pokrivenost funkcija, ali imaju loše ostale parametre. `Tank.cpp`, `World.cpp` imaju dobru, a `main.cpp` ima odličnu pokrivenost linija i funcija.
Ove informacije su jako važne zato što nam ukazuju na koje delove koda treba obratiti pažnju. Ukoliko imamo nisku pokrivenost linija ukazuje nam na to koje linije konkretno treba bolje testirati. Ako imamo nisku pokrivenost funkcija, pomoći će nam da identifikujemo funkcije koje nisu dovoljno testirane, a takođe, slično i za grane.

4 Cppcheck

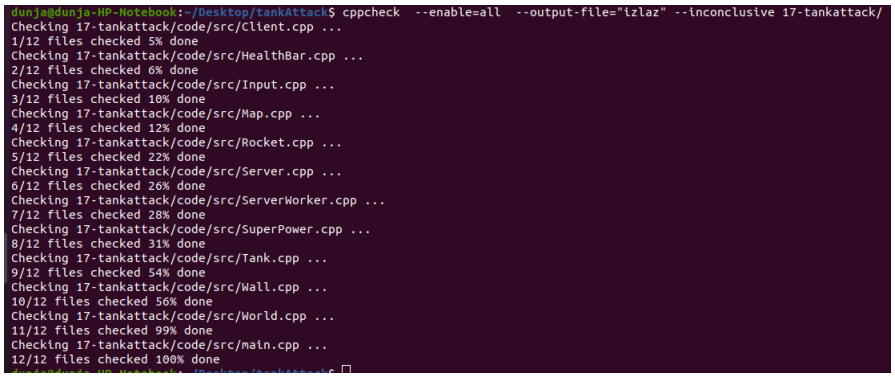
Analiziraćemo ovaj projekat još jednim moćnim alatom za statičku analizu `C` i `C++` koda. Sada je reč o alatu `Cppcheck`. I ovaj alat pomaže u otkrivanju potencijalnih grešaka, upozorenja i stilskih problema pre kompilacije i izvršavanja. Vrši veliki broj provera kao što su detektovanje neiskorišćenih funkcija, neinicijalizovanih promenljivih, loše alokacije memorije...Upotrebom ovog alata može se značajno poboljšati kvalitet koda i povećati pouzdanost softvera.

Za instalaciju ovog alata potrebno je pokrenuti sledeću komandu u terminalu:

```
sudo apt-get install cppcheck
```

Pokrenućemo ovaj alat i uključiti neke dodatne opcije koje će doprineti našoj analizi.Prva opcija, '–enable=all', će uključiti sve dostupne provere, tj. alat će proveriti sve moguće vrste grešaka i upozorenja. Sa '–inonclusive' uključujemo i provere koje se svrstavaju u "neodlučne", tj. one provere koje cppcheck nije mogao da potvrdi kao greške ili upozorenja ali ih ipak prijavljuje. Sa '–output-file' preusmeravamo izlaz u dati fajl.

```
cppcheck  --enable=all  --output-file="izlaz" --inconclusive 17-tankattack/
```



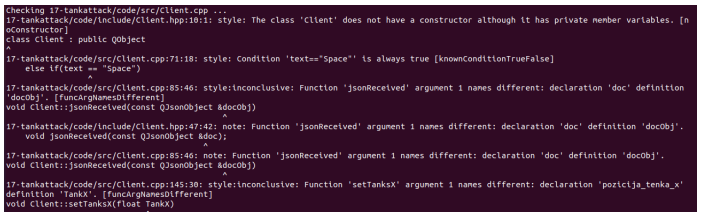
Slika 8: Pokretanje alata cppcheck

Ceo izlaz ovog alata može se naći u okviru foldera `Cppcheck`. Alat daje napomenu o velikom broju funkcija koje se nikada ne koriste, što može ukazivati na nepotreban kod i time narušavanje čitljivosti koda.

Slika 9: Deo izlaza alata cppcheck (neiskorišćene funkcije)

Zatim, pojavljuju se i različiti nazivi argumenata u deklaracijama i definicijama funkcija, što ne dovodi do greške u kompilaciji, ali takođe, narušava čitljivost i održivost koda. Još jedna infomacija koju alat daje je i postojanje promenljivih koje nisu inicijalizovane. Takođe, pojavljuje se i stilski problem hvatanja izuzetaka po vrednosti, što bi trebalo raditi po referenci ne bismo kopirali izuzetke.

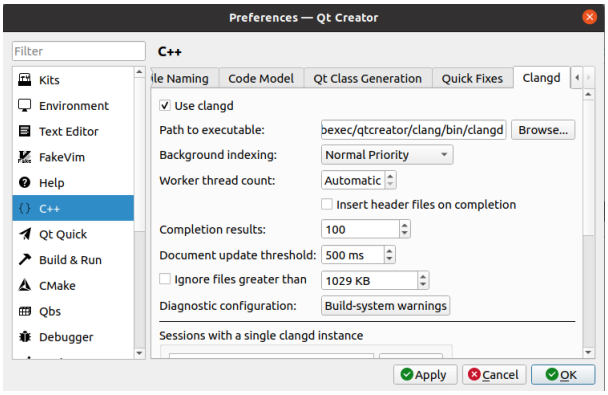
Poruke na slici 10 ukazuju na problem klase *Server* koja ima privatne promenljive a nema definisan konstruktor, koji bi bio potreban za njihovu inicijalizaciju. Zatim, za datoteku `Client.cpp` pojavljuje se poruka da je uslov u `else if` naredbi uvek tačan i treba proveriti da li postoji greška u logici, kao i mogućnost da je ovo potencijalno nepotreban kod.



Slika 10: Deo izlaza alata Cppcheck

5 Clangd

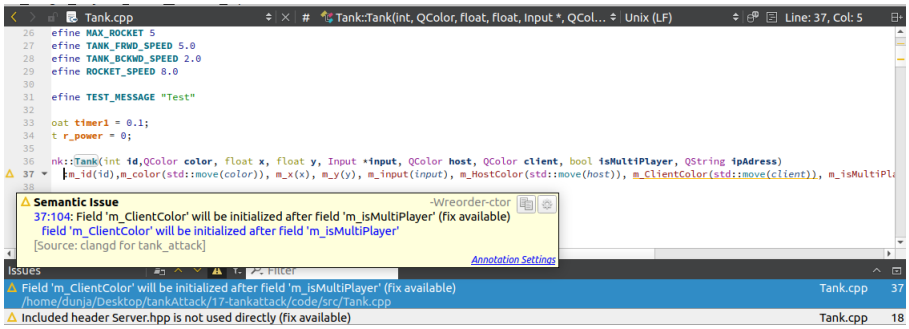
Sledeći alat koji ćemo primeniti je `Clangd`. Detektuje različite vrste grešaka i upozorenja u `C++` kodu, poput sintaksnih i semantičkih grešaka, upozorenja o kodu koji se ne koristi i slično. On omogućava brzu analizu te je zato pogodan za primenu i na velikim i kompleksnim projektima. Pruža podršku za refaktorisanje koda u cilju poboljšanja strukture i čitljivosti koda.
Pokazaćemo upotrebu ovog alata pomoću QtCreator-a. Potrebno je da klikom na *Edit* izaberemo opciju *Preferences* i odaberemo opciju *C++*.Potom, kao što je prikazano na sledećoj slici, treba da izaberemo karticu *Clangd*, čekiramo polje *"Use clangd"* i kliknemo na dugme *Apply*.



Slika 11: Podešavanja za clangd

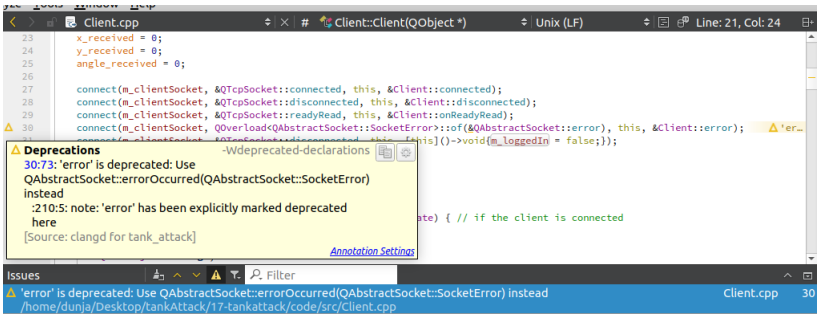
Alat nam daje poruke o više pronađenih grešaka u različitim fajlovima. Na slici 12 vidimo poruke za fajl `Tank.cpp`. Prva poruka je upozorenje o potencijanom problemu u inicijalizaciji članova klase Tank. Član klase 'm_ClientColor' će se inicijalizovati posle 'm_isMultiPlayer' što može biti problem ukoliko redosled inicijalizacije članova utiče na ispravno funkcionisanje klase. Podatak `'(fix aviable)'` nam govori da postoji automatsko popravljanje ovog problema, tj. zamenom redosleda inicijalizacije članova problem bi bio rešen. Druga poruka odnosi se na zaglavlje koje je uključeno, a ne koristi se direktno u kodu. Takođe, postoji automatsko popravljanje ovog problema, tj. clangd je prepoznao da se zaglavlje `Server.hpp` može isključiti iz fajla `Tank.cpp`.

Sve informacije dobijene ovim alatom date su u folderu `Clangd`. Prokomentarisaćemo još jedan tip poruke koju smo dobili za fajlove `Client.cpp`



Slika 12: Poruke za falj Tank.cpp

i `main.cpp`. Prikazana je na slici 13. Ukazuje na korišćenje zastarele funkcije `'error'` čije se dalje korišćenje ne preporučuje. Umesto nje preporučuje se korišćenje druge funkcije koja ima sličnu funkcionalnost, a koja neće generisati upozorenja.



Slika 13: Poruka za falj Client.cpp

6 Callgrind

Sledeći alat koji ćemo primeniti je Valgrind alat- `Callgrind`. Ovaj alat generiše listu poziva funkcija korisničkog programa u vidu grafa. Zahvaljujući grafu poziva, može da se odredi, počevši od main funkcije, koja funkcija ima najveću cenu poziva. Callgrind prikuplja podatke o tome koliko vremena program provodi u svakoj funkciji, što omogućava precizno merenje performansi, a takođe, broji i koliko se instrukcija izvršava u svakoj funkciji, što može pomoći u identifikaciji delova koda koji troše najviše procesorskog vremena. U kombinaciji sa `Kcachegrind` grafičkim korisničkim interfejsom, Callgrind omogućava programerima dublju analizu performansi njihovih programa i efikasno otkrivanje problema u kodu.

Prvo je potrebno da prevedemo program u debug režimu, na sledeći način: pozicioniramo se u direktorijum projekta i prevedemo program sledećim komandama:

```
qmake CONFIG+=DEBUG
```

`make`

Ili u `.pro` fajlu dodamo:

`CONFIG+=DEBUG`

Pozicioniramo se u direktorijum gde se nalazi izvršni fajl i alat pokrećemo sledećom komandom:

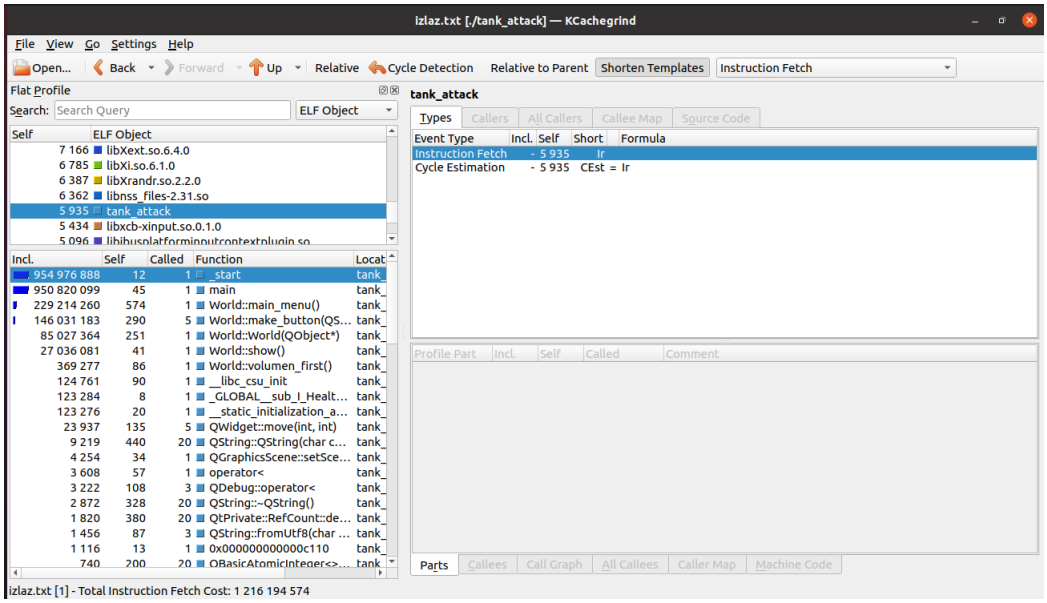
```
valgrind --tool=callgrind --callgrind-out-file="izlaz.txt" ./tank_attack
```

Kako bismo imali lep prikaz ovog izveštaja, otvorićemo ga uz pomoć Kcachegrind-a. Ukoliko je potrebno, prethodno ga treba instalirati:

```
sudo apt install kcachegrind
```

Pokrećemo ga komandom:

```
kcachegrind izlaz.txt
```

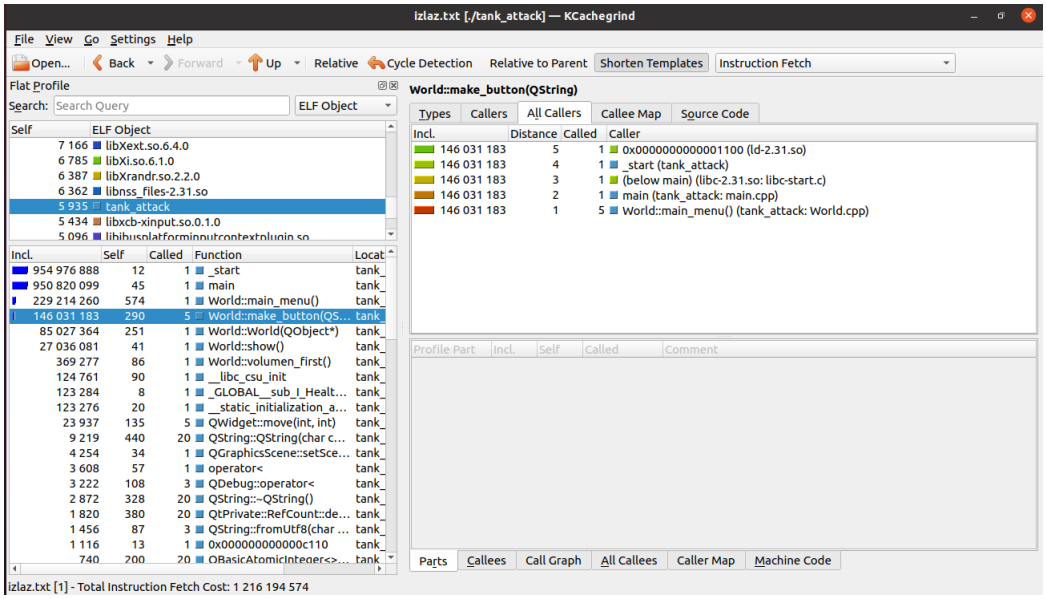


Slika 14: Vizuelan prikaz izvestaja preko Kcachegrind

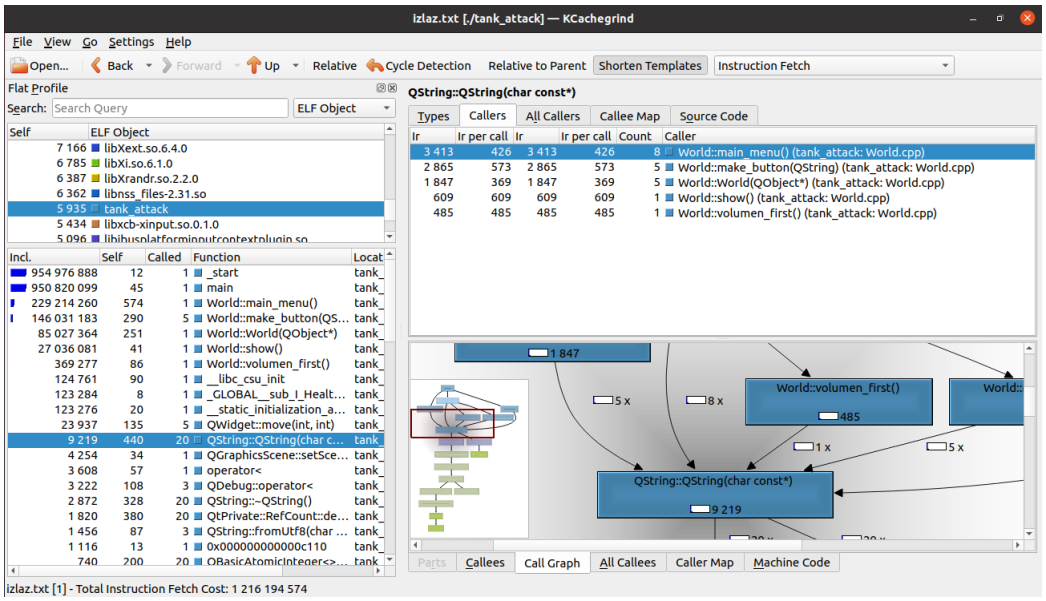
Zanimaju nas funkcije koje se najviše puta pozivaju. Na levoj strani se nalaze informacije o broju pozivanja svake funkcije i broju instrukcija koji je zahtevalo njeno izvršavanje, samostalno i uključu- jući izvršavanja drugih funkcija koje je pozivala. Na desnoj strani možemo izabrati opciju *All Callers* i videćemo koje sve funkcije su pozivale funkciju koja nas zanima.

Na slici 15 vidimo da je funkcija World::make_button pozivana 5 puta, a sa desne strane izborom opcije *All Callers* vidimo spisak funkcija koje su je pozivale.

Na slici 16 vidimo detalje o funkciji koja je pozivana 20 puta. Sada je generisan i prikaz grafa poziva, što je urađeno opcijom *Call Graph*. Zaključak je da nema velikih broja poziva funkcija u delu koji je implementiran od strane programera ovog projekta.



Slika 15: Vizuelan prikaz izvestaja preko Kcachegrind



Slika 16: Vizuelan prikaz izvestaja preko Kcachegrind

7 Zaključak

Analizom ovog projekta došli smo do zaključka da postoje greške koje bi trebalo ispraviti. Ukazano je na postojanje propusta koji narušavaju čitljivost i održivost koda. Za većinu poruka, koje smo dobili primenom

ovih alata, dato je detaljnije objašnjenje, kao i na koji način ispraviti greške i upozorenja na koje se poruke odnose.