Analiza projekta korišćenjem alata za verifikaciju softvera

Seminarski rad u okviru kursa
Verifikacija softvera
Univerzitet u Beogradu
Matematički fakultet

Stošić, Emilija
emilijazstosic@gmail.com

28. avgust 2023.

Sažetak

U radu će biti prikazana analiza projekta FingerCoaster. Analiza projekta je dobijena primenom alata za verifikaciju softvera, i ukratko će biti opisani alati i naredbe koji su korišćene. Projekat nad kojem se vrši analiza se nalazi na adresi https://gitlab.com/matf-bg-ac-rs/course-rs/projects-2021-2022/18-FingerCoaster, a autori su Ana Bolović, Višeslav Đurić, Vladimir Mandić i Pavle Vlajković. Projekat je nastao za potrebe kursa Razvoj softvera.

Sadržaj

1 Uvod

Igrica FingerCoaster je takmičenje u brzom kucanju. Igranje igre je moguće i online i offline. U takmičenju mogu da učestvuju više korisnika. Svaki korisnik ima svoj profil sa informacijama o istoriji igranja. Ideja je uzeta od igara kao što su TypeRacer i Ten FIngers. Za testiranje projekta korišćeni su alati: Clang-Tidy, Clazy, Clang-Format, Memcheck, Massif, Callgrind.

2 Valgrind

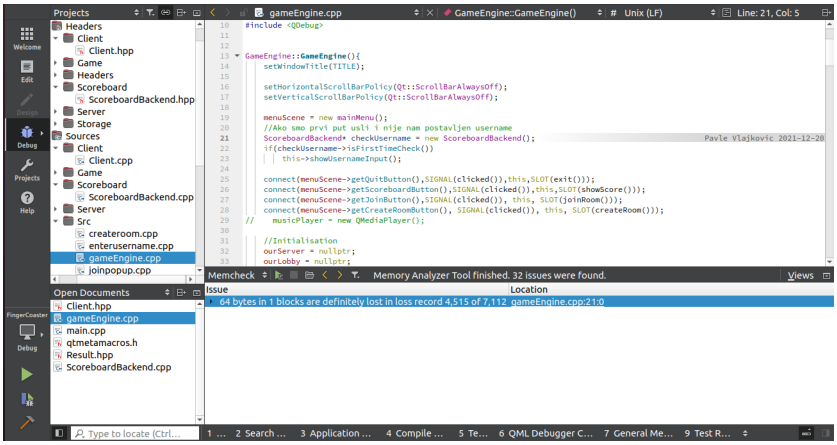
Valgrind alat je alat za dinamičku analizu koda. Postoje Valgrind alati koji mogu automatski otkriti mnoge greške u upravljanju memorijom i nitima, i detaljno profilisati programe.

2.1 Memcheck

Memcheck je najpoznatiji Valgrind alat. Koristi se za detektovanje memorijskih grešaka i radi analizu nad mašinskim kodom. Može da detektuje greške kao što su:

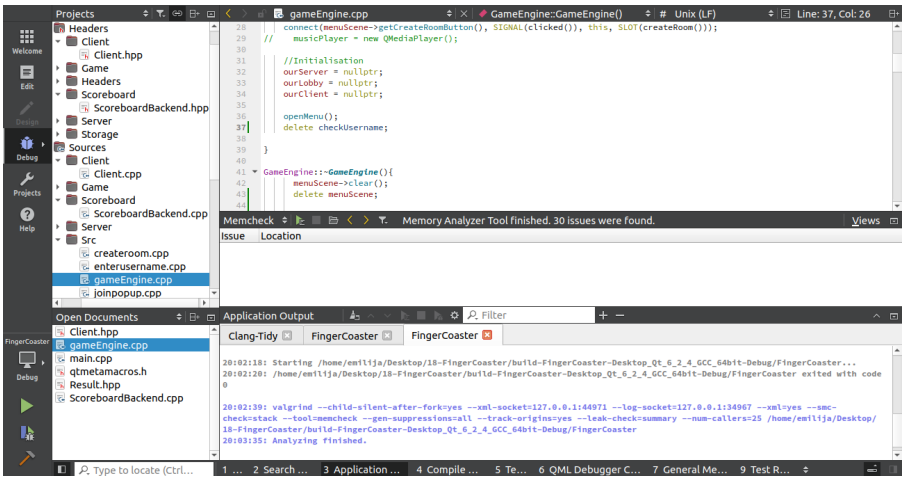
- Korišćenje neinicijalizovane memorije
- Pristup već oslobođenoj memoriji
- Curenje memorije

Pokrećemo alat iz QtCreator-a. Potrebno je da kliknemo na *Analyze*, zatim odaberemo opciju *Valgrind Memory Analyzer*. Rezultat rada alata vidimo na slici 1.



Slika 1: Rezultat rada alata Memcheck

Sa slike vidimo da u fajlu *gameEngine.cpp* imamo 64bytes izgubljena. Problem rešavamo tako što dodajemo komandu *delete checkUsername*. Nakon ponovnog pokretanja alata, na slici 2 vidimo da se ne prijavljuje ova greška, tako da smo je uspešno rešili.



Slika 2: Ispravka greške

2.2 Massif

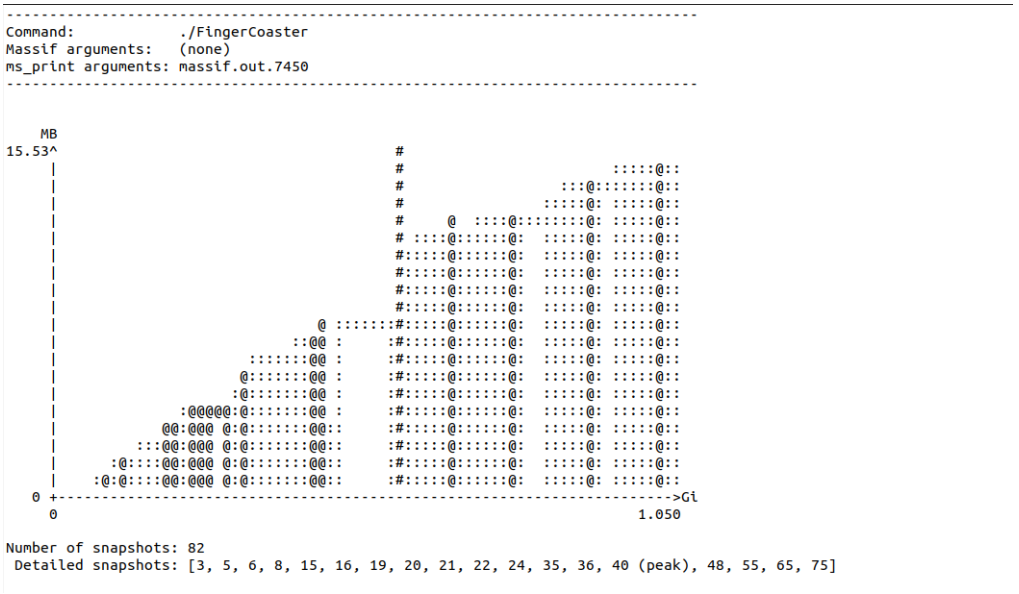
Massif je alatk koji meri koliko heap memorije naš program koristi. Postoje određena curenja memorije koja se ne otkrivaju alatom Memcheck. To je zato što se memorija zapravo nikada ne gubi, ostaje pokazivač na nju, ali se ne koristi. Programi koji imaju ovakvo curenje mogu nepotrebno povećati količinu memorije koju koriste tokom vremena. Massif može pomoći u identifikaciji ovih curenja. Massif govori ne samo koliko heap memorije naš program koristi, već daje i veoma detaljne informacije koje ukazuju na to koji delovi programa su odgovorni za alociranje memorije.

Da bismo pokrenuli Massif koristimo sledeću komandu:

valgrind –tool=massif ./FingerCoaster

Dobijemo fajl *massif.out.PID*, gde PID predstavlja identifikator procesa, koji je veoma nečitljiv, pa pokrećemo narednu komandu:

ms_print massif.out.PID > massif.txt



Slika 3: Grafik

Na slici 3 vidimo sadržaj massif.txt fajla i vidimo da je massif napravio 92 preseka, pik se dostiže u četrdesetom preseku i iznosi 15.53MB. Sa slike 4 se opaža da je potrošnja hipa u četrdesetom preseku mnogo veća nego u ostalim presecima.

n	time(i)	total(B)	useful-heap(B)	extra-heap(B)	stacks(B)
37	497,775,858	3,639,328	3,209,894	429,434	0
38	511,870,165	8,153,568	7,713,837	439,731	0
39	603,170,045	8,153,568	7,713,837	439,731	0
40	622,575,203	16,288,816	15,848,869	439,947	0

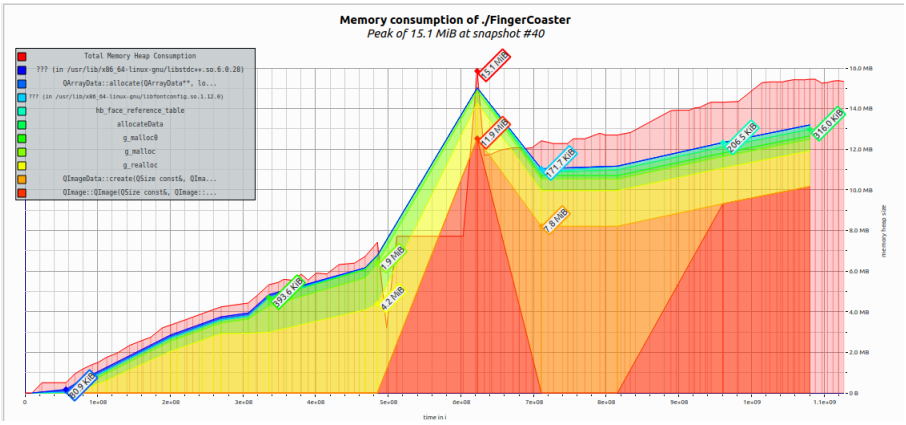
Slika 4: Tabela presek

Možemo da koristimo alat *Massif Visualizer* koji vizualizuje podatke izgenerisane od strane massif alata. Rezultat pokretanja *massif.out.PID* fajla pomoću *Massif Visualizer* alata se vidi na slici 5.

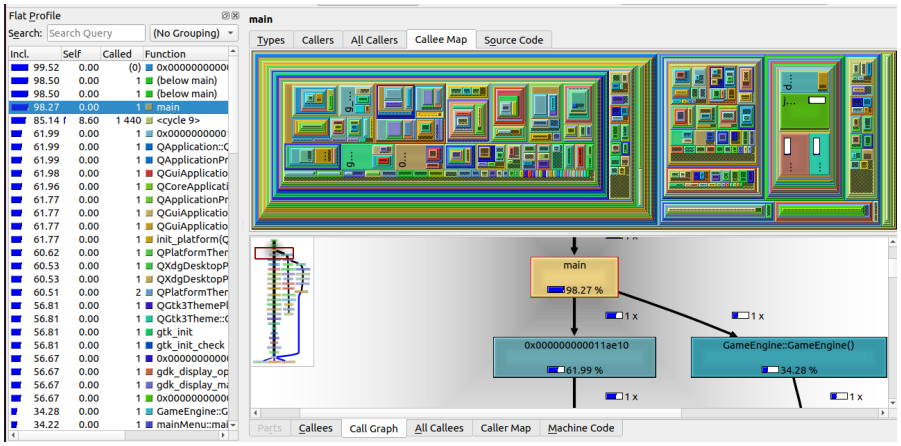
2.3 Callgrind

Callgrind je alat koji u vidu grafa generiše listu poziva funkcija korisničkog programa. Podrazumevano, prikupljeni podaci se sastoje od broja izvršenih instrukcija, njihovog odnosa prema izvornim linijama, odnosa između pozivajućih i pozvanih funkcija, i broja takvih poziva.

Kada se završi rad programa i Callgrind alata podaci koji se analiziraju su zapisani u fajlu callgrind.out.PID , gde PID predstavlja identifikator procesa. Otvaramo fajl pomoću alata *KCachegrind*. Rezultat rada alata se vidi na slici 6.



Slika 5: Grafik



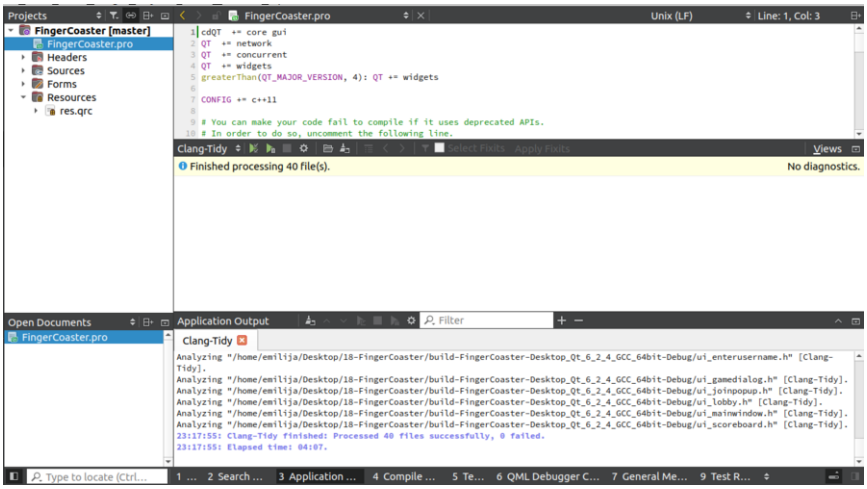
Slika 6: Callgrind

3 Clang-Tidy, Clazy i Clang-Format

Clang-Tidy je alat za statičku analizu koda. Omogućava analiziranje koda bez izvršavanja programa sa ciljem pronalaženja grešaka i poboljšanja kvaliteta koda.

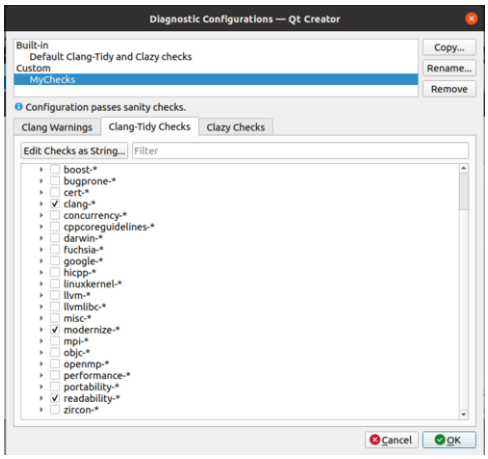
Upotrebu alata Clang-Tidy pokazaćemo pomoću QtCreator-a. U okviru QtCreator-a potrebno je da kliknemo na *Analyze*, zatim odaberemo opciju *Clang-Tidy*, odaberemo fajlove na koje želimo da primenimo alat i pritiskom na dugme *Analyze* pokrećemo alat. Na ovaj način pokrećemo alat bez dodatnih opcija i parametara. Rezultata rada se vidi na slici 7. Možemo da vidimo da alat nije pronašao grešku i da nije izdao nikakvo upozorenje.

Da bismo dodali nove opcije alatu, kliknemo na *Edit*, zatim odaberemo opciju *Preferences*, i opciju *Analyzer*. Pritiskom na dugme *Default Clang-Tidy and Clazy checks* pravimo novu konfiguraciju i biramo željene opcije, to se vidi na slici 8. Opcija **clang-*** je uvek uključena. Dodali



Slika 7: Rezultat rada alata Clang-Tidy bez dodatnih opcija

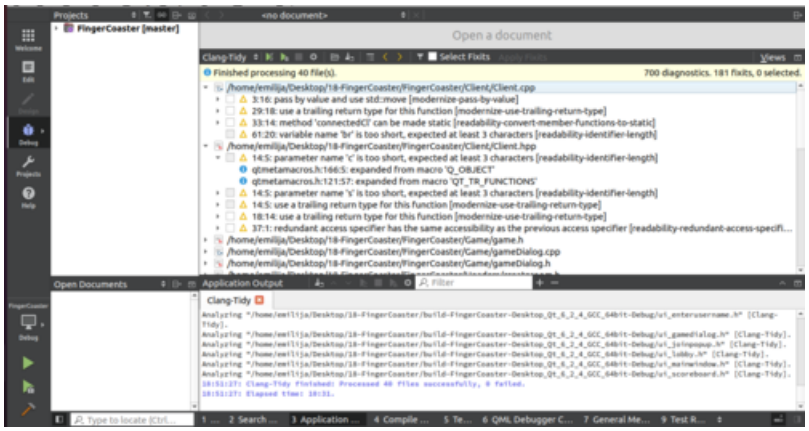
smo opcije **modernize-*** i **readability-***. Klikom na dugme *Ok* opcije su postavljene i možemo da pokrenemo alat kao u prethodnom primeru.



Slika 8: Odabir opcija

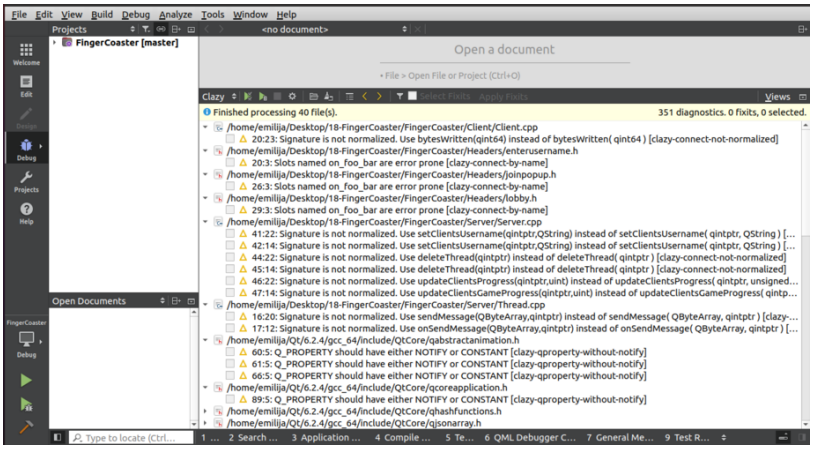
Rezultat rada alata sa dodatim opcijama se vidi na slici 9. Sada vidimo da se nakon pokretanja alata dobijaju razna upozorenja. Naziv promenljive i naziv parametara je kratak, očekivana dužina naziva je najmanje tri karaktera. Takođe, vidimo da je preporuka da metod *connectedCl* bude static.

Clazy alat je dodatak alatu Clang-Tidy i razvijen je za Qt aplikacije, može da detektuje potencijalno curenje memorije, nepravilne konverzije tipiva, neefikasne upotrebe Qt API-ja... Clazy provere su podeljene na nivoe. Nivo 0 je najnizi nivo, dok je nivo 3 najviši. Na nivou 0 provere su veoma stabilne i gotovo da ne pokazuju lažne pozitivne rezultate, dok su provere na nivou 3 više eksperimentalne.



Slika 9: Rezultat rada alata Clang-Tidy sa dodatnim opcijama

Da bismo pokrenuli Clazy okviru QtCreator-a potrebno je da kliknemo na *Analyze*, zatim odaberemo opciju *Clazy*, odaberemo fajlove na koje želimo da primenimo alat i pritiskom na dugme *Analyze* pokrećemo alat. Rezultat rada alata Clazy se mogu videti na slici 10.



Slika 10: Rezultat rada alata Clazy

Zaključak: Vidimo da je alat Clang-Tidy izdao dosta upozorenja, i da je alat Clazy našao mnogo grešaka, međutim kada pogledamo sva upozorenja i sve greške možemo zaključiti da su to uglavnom iste greške i ista upozorenja koja se odnose na različite fajlove.

Clang-Format je široko rasprostranjen C++ formater koda. Postoji vise razlicitih stilova npr. LLVM, Google, Chromium, Mozilla, WebKit, Microsoft, GNU, može se kreirati i sopstveni stil. Da bismo iskoristili ovaj alat potrebno je da napisemo python skriptu. Sadržaj skripte se vidi na slici 11.

Prvo koristimo sledeću komandu:

clang-format -style=llvm -dump-config > .clang-format

Na ovaj način smo odabrali llvm stil formatiranja. Za pokretanje python skripte koristimo komandu:

python3 run-clang-format.py . file

Na ovaj način je kod formatiran, python skripa je prošla kroz sve direktorijume i sve fajlove i primenila llvm stil formatiranja.

```python
import subprocess
import os
import sys

def isccppfile(filename):
    return filename.endswith('.h') \
        or filename.endswith('.c') \
        or filename.endswith('.hpp') \
        or filename.endswith('.cpp') \

if __name__ == "__main__":
    args = sys.argv[1:]
    if len(args) < 2:
        print('Usage: python3 run-clang-format.py <dir> <style>')
        exit(1)

    dir   = args[0]
    style = args[1]

    for dir, _, files in os.walk(dir):
        for filename in files:
            filepath = dir + '/' + filename
            if isccppfile(filename):
                subprocess.run('clang-format -i  -style={} {}'.format(style, filepath), shell=True)

```

Slika 11: Python skripta

4 Zaključak

Analizom projkta smo došli do zaključka da u programu postoje neke greške koje bi trebalo ispraviti. Dinamičkom analizom je utvrđeno da postoje određena curenja memorije, neka curenja memorije su razrešena. Statičkom analizom je uočeno da se u projektu dosta koriste magične konstante, imena promenljivih nisu jasna, što smanjuje čitljivost koda.