

Analiza projekta korišćenjem alata za verifikaciju softvera

Seminarski rad u okviru kursa
Verifikacija softvera
Matematički fakultet

Milica Karličić, 1002/2022
milicakarlicic1801@gmail.com

januar 2023.

Sažetak

U ovom radu biće opisana detaljna analiza projekta *Fire and Water*. U daljem tekstu biće ukratko opisani alati i naredbe koji su korišćeni prilikom analize. Svaka glava sadrži pronađene greške ili određene statističke podatke vezane za dati projekat. Na samom kraju opisa rada i analize pojedinih alata mogu se naći i načini na koji bi pronađene greške mogle da se otklone.

Sadržaj

1	Uvod	2
2	Valgrind	2
2.1	Memcheck	2
2.2	Cachegrind	5
3	Perf	7
4	Clang-Tidy i Clazy	9
5	Clangd	11
6	Zaključak	12
	Literatura	13

1 Uvod

Projekat *Fire and Water* predstavlja jednu implementaciju popularne istoimene igrice. Igrica prati dva glavna igrača *waterGirl* i *fireBoy* čiji je cilj da sakupe što više dijamanta i da, izbjegavajući prepreke, stignu do cilja. Za testiranje projekta *FireandWater* korišćeni su alati za dinamičku i statičku analizu. Za dinamičku analizu projekta korišćeni su *Valgrind* i *Perf*. Izvještaj statičke analize dobijen je na osnovu rada alata *Clang-Tidy*, *Clazy* i *Clangd*. Alati za dinamičku analizu su pozivani iz komandne linije, dok se za alate statičke analize koristila podrška *Qt* razvojnog okruženja.

2 Valgrind

Valgrind je profajler otvorenog koda koji nadgleda rad željenog programa i prijavljuje nepravilnosti u radu programa. *Valgrind* distribucija sadrži naredne alate:

- *Memcheck* - detektor memorijskih grešaka
- *Massif* - praćenje rada dinamičke memorije
- *Cachegrind* - profajler keš memorije
- *Callgrind* - profajler funkcija
- *Helgrind* i *DRD* - detektor grešaka niti

[2]

Alati *Valgrinda* koriste metodu bojenja vrijednosti. Oni svaki registar i memorijsku vrijednost boje (zamjenjuju) sa vrijednošću koja govori nešto dodatno o originalnoj vrijednosti. Proces rada svakog alata *Valgrinda* je u osnovi isti. U ovom radu biće prikazana upotreba alata *Memcheck* i *Cachegrind*. [1]

Zbog velikog usporenja prilikom izvršavanja programa upotrebom *Valgrind*-a napravljena je druga verzija projekta koja je jednostavnija za analizu. Naime, kako je cilj testirati studentski kod, *Qt* biblioteke bi mogle da se izbace što samim tim podrazumijeva uklanjanje određenih klasa i metoda. U modifikovanom projektu sadržaj *main*-a prikazan je na 1.

2.1 Memcheck

Prilikom analize *Qt* projekta, *Memcheck* se može pozivati iz *QtCreator*-a ili korišćenjem komandne linije. U ovom radu biće simulirana upotreba alata iz terminala.

Greške koje mogu biti detektovane upotrebom *Memcheck* alata su:

- Korišćenje nedefinisanih vrednosti
- Čitanje ili pisanje u nedopuštenu memoriju na hipu, steku
- Neispravno oslobađanje memorije na hipu
- Poklapanje argumenata *src* i *dest* funkcije *memcpy* i njoj sličnim
- Prosleđivanje loših vrednosti za veličinu memorijskog prostora funkcijama za alokaciju memorije, npr. negativnih.
- Curenje memorije, npr. gubitak pokazivača na alociran prostor.

Slika 1: *Main* modifikovanog projekta

```
#include "../Headers/level.h"
#include "../Headers/levelinfo.h"
#include "../Headers/settings.h"

int main(int argc, char *argv[])
{
    int i = 0;
    int levelId = 1;
    while(i++ != 1000) {
        levelId = levelId % 3;
        if(!levelId)
            levelId++;

        Settings *s = new Settings();
        LevelInfo *li = new LevelInfo(s);
        Level *level = new Level(levelId, li);

        level->start();

        delete s; delete li; delete level;
    }
}
```

Na osnovu prethodno pomenutog *main*-a *Memcheck* je detektovao curenje memorije na nekoliko mjesta. U okviru *makefile*-a veoma je važno proslijediti kompajleru g++ opcije -g -O0 kako bi se kod preveo sa debug simbolima i bez optimizacija tj. kako bismo imali uvid u tačan redni broj linije koda u kojoj je došlo do neke greške (u našem slučaju u pitanju je curenje memorije). Opcije koje ćemo koristiti prilikom poziva Memcheck alata su -leak-check=full i -show-leak-kinds=all koje nam prikazuju detaljan opis pronađenih curenja memorije. Pozivom programa sa valgrind -leak-check=full -show-leak-kinds=all ./a.out dobijamo izlaz prikazan na slikama 2 i 3.

Na osnovu rezultata možemo vidjeti prisustvo curenja memorije na šest mjesta u programu. Detektovana curenja memorije su sledeća:

- u klasi player.cpp:
 - Linija 12: inicijalizovana promenljiva info se nikada ne oslobađa (greška se javlja dva puta, jednom prilikom inicijalizacije vatrenog igrača, a drugi put prilikom inicijalizacije vodenog igrača)
- u klasi input.cpp:
 - Linija 23: objekat koji odgovara vatrenom igraču se nikada ne oslobađa
 - Linija 24: objekat koji odgovara vodenom igraču se nikada ne oslobađa
- u klasi level.cpp:
 - Linija 36: inicijalizovana promenljiva info se nikada ne oslobađa
 - Linija 51: inicijalizovana promenljiva input se nikada ne oslobađa

Detektovana curenja memorije se mogu lako otkloniti dodavanjem naredbe delete u odgovarajućem destrukturu. Takođe, na osnovu izvještaja

Slika 2: *Memcheck*

```

==4498== Memcheck, a memory error detector
==4498== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==4498== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==4498== Command: ./a.out
==4498==
==4498== HEAP SUMMARY:
==4498==   in use at exit: 360 bytes in 6 blocks
==4498== total heap usage: 36 allocs, 30 frees, 74,192 bytes allocated
==4498==
==4498== 24 bytes in 1 blocks are indirectly lost in loss record 1 of 6
==4498==   at 0x4845013: operator new(unsigned long) (in /usr/libexec/valgrind/vgpreload_memcheck
==4498==   by 0x10D079: Input::Input() (in /home/milica/Desktop/VS_projekat/v1/Sources/a.out)
==4498==   by 0x10D554: Level::start() (in /home/milica/Desktop/VS_projekat/v1/Sources/a.out)
==4498==   by 0x10A64C: main (in /home/milica/Desktop/VS_projekat/v1/Sources/a.out)
==4498==
==4498== 24 bytes in 1 blocks are indirectly lost in loss record 2 of 6
==4498==   at 0x4845013: operator new(unsigned long) (in /usr/libexec/valgrind/vgpreload_memcheck
==4498==   by 0x10D0B2: Input::Input() (in /home/milica/Desktop/VS_projekat/v1/Sources/a.out)
==4498==   by 0x10D554: Level::start() (in /home/milica/Desktop/VS_projekat/v1/Sources/a.out)
==4498==   by 0x10A64C: main (in /home/milica/Desktop/VS_projekat/v1/Sources/a.out)
==4498==
==4498== 96 bytes in 1 blocks are indirectly lost in loss record 3 of 6
==4498==   at 0x4845013: operator new(unsigned long) (in /usr/libexec/valgrind/vgpreload_memcheck
==4498==   by 0x10CCC5: Player::Player(PlayerColor, float, float) (in /home/milica/Desktop/VS_pro
==4498==   by 0x10D0A1: Input::Input() (in /home/milica/Desktop/VS_projekat/v1/Sources/a.out)
==4498==   by 0x10D554: Level::start() (in /home/milica/Desktop/VS_projekat/v1/Sources/a.out)
==4498==   by 0x10A64C: main (in /home/milica/Desktop/VS_projekat/v1/Sources/a.out)

```

Slika 3: *Memcheck*

```

==4498== 96 bytes in 1 blocks are indirectly lost in loss record 4 of 6
==4498==   at 0x4845013: operator new(unsigned long) (in /usr/libexec/valgrind/vgpreload_memcheck-
==4498==   by 0x10CCC5: Player::Player(PlayerColor, float, float) (in /home/milica/Desktop/VS_pro
==4498==   by 0x10D0DA: Input::Input() (in /home/milica/Desktop/VS_projekat/v1/Sources/a.out)
==4498==   by 0x10D554: Level::start() (in /home/milica/Desktop/VS_projekat/v1/Sources/a.out)
==4498==   by 0x10A64C: main (in /home/milica/Desktop/VS_projekat/v1/Sources/a.out)
==4498==
==4498== 96 bytes in 1 blocks are definitely lost in loss record 5 of 6
==4498==   at 0x4845013: operator new(unsigned long) (in /usr/libexec/valgrind/vgpreload_memcheck-
==4498==   by 0x10D462: Level::Level(int, LevelInfo*) (in /home/milica/Desktop/VS_projekat/v1/Sour
==4498==   by 0x10A63C: main (in /home/milica/Desktop/VS_projekat/v1/Sources/a.out)
==4498==
==4498== 264 (24 direct, 240 indirect) bytes in 1 blocks are definitely lost in loss record 6 of 6
==4498==   at 0x4845013: operator new(unsigned long) (in /usr/libexec/valgrind/vgpreload_memcheck-
==4498==   by 0x10D543: Level::start() (in /home/milica/Desktop/VS_projekat/v1/Sources/a.out)
==4498==   by 0x10A64C: main (in /home/milica/Desktop/VS_projekat/v1/Sources/a.out)
==4498==
==4498== LEAK SUMMARY:
==4498==   definitely lost: 120 bytes in 2 blocks
==4498==   indirectly lost: 240 bytes in 4 blocks
==4498==   possibly lost: 0 bytes in 0 blocks
==4498==   still reachable: 0 bytes in 0 blocks
==4498==   suppressed: 0 bytes in 0 blocks
==4498==
==4498== For lists of detected and suppressed errors, rerun with: -s
==4498== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)

```

možemo vidjeti da se poslednja dva curenja odnose na direktno izgubljene blokove tj. na promenljive info i input iz start() metode klase level.cpp. Direktno izgubljeni blokovi mogu povlačiti gubitak i nekih drugih bloko-

va koji u tom slučaju postaju indirektno izgubljeni. Indirektno izgubljeni objekti u ovom primjeru su: promenljive info iz klase player.cpp i promenljive koja odgovara vatrenom odnosno vodenom igraču iz klase input.cpp.

Kako je analiza rađena na modifikovanoj verziji projekta neophodno je provjeriti da li su pronađena curenja prisutna i u originalnoj verziji. Jednostavnim pregledom klasa se potvrđuje prisustvo svih navedenih grešaka i u originalnom kodu, što je i očekivano. U originalnom projektu moguće je curenje memorije i na nekim drugim mjestima koja su vezana za GUI aplikacije.

2.2 Cachegrind

Prilikom analize *Qt* projekta, *Memcheck* se može pozivati iz *Qt Creator*-a ili korišćenjem komandne linije. U ovom radu biće simulirana upotreba alata iz terminala. Cachegrind je alat Valgrind-a koji omogućava softversko profajliranje keš memorije tako što simulira i prati pristup keš memoriji mašine na kojoj se program, koji se analizira, izvršava. Takođe, može se koristiti i za profajliranje izvršavanja grana. [2] Cachegrind simulira memoriju mašine, koja ima prvi nivo keš L1 memorije podeljene u dve odvojene nezavisne sekcije: I1 - sekcija keš memorije u koju se smeštaju instrukcije D1 - sekcija keš memorije u koju se smeštaju podaci. Drugi nivo keš memorije koju Cachegrind simulira je objedinjen - LL, skraćeno od eng. last level. Ovaj način konfiguracije odgovara mnogim modernim mašinama. [1] U okviru *makefile*-a veoma je važno da kompajleru g++ ne prosljedimo opciju -O0 tj. sasvim je u redu da vršimo analizu optimizovanog koda jer želimo da testiramo program u njegovom normalnom izvršavanju. Pozivom programa sa valgrind -tool=cachegrind ./a.out dobijamo izlaz prikazan na slici 4.

Slika 4: *Cachegrind* - izlaz prilikom jednog izvršavanja petlje

```
==23000== Cachegrind, a cache and branch-prediction profiler
==23000== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.
==23000== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==23000== Command: ./a.out
==23000==
--23000-- warning: L3 cache found, using its data for the LL simulation.
==23000==
==23000== I  refs:      1,887,922
==23000== I1 misses:    2,179
==23000== LLi misses:   2,068
==23000== I1 miss rate:  0.12%
==23000== LLi miss rate: 0.11%
==23000==
==23000== D  refs:      652,423 (487,808 rd + 164,615 wr)
==23000== D1 misses:    15,737 ( 13,521 rd +   2,216 wr)
==23000== LLd misses:    8,799 (  7,442 rd +   1,357 wr)
==23000== D1 miss rate:  2.4% (  2.8% +   1.3% )
==23000== LLd miss rate: 1.3% (  1.5% +   0.8% )
==23000==
==23000== LL refs:      17,916 ( 15,700 rd +   2,216 wr)
==23000== LL misses:    10,867 (  9,510 rd +   1,357 wr)
==23000== LL miss rate:  0.4% (  0.4% +   0.8% )
```

Dobijeno upozorenje je u vezi sa radom *Cachegrind*-a. Naime, kao što je već rečeno *Cachegrind* radi sa dva nivoa cache memorije dok su na arhitekturi koja se koristi za testiranje prisutna tri nivoa. To nije toliko bitno, samo označava da je moguće da će se program brže izvršavati na

datoj arhitekturi. Ukoliko u *main.cpp* broj iteracija povećamo na 1000, dobijamo izlaz prikazan na slici 5.

Slika 5: *Cachegrind* - izlaz prilikom povećavanja iteracija petlje

```
==25591== Cachegrind, a cache and branch-prediction profiler
==25591== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.
==25591== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==25591== Command: ./a.out
==25591==
--25591-- warning: L3 cache found, using its data for the LL simulation.
==25591==
==25591== I   refs:      19,043,683
==25591== I1 misses:      2,212
==25591== L1i misses:      2,079
==25591== I1 miss rate:    0.01%
==25591== L1i miss rate:  0.01%
==25591==
==25591== D   refs:      9,317,110 (5,619,932 rd + 3,697,178 wr)
==25591== D1 misses:      22,608 ( 13,635 rd +   8,973 wr)
==25591== L1d misses:      15,542 (   7,442 rd +   8,100 wr)
==25591== D1 miss rate:    0.2% (   0.2% +   0.2% )
==25591== L1d miss rate:  0.2% (   0.1% +   0.2% )
==25591==
==25591== LL refs:      24,820 ( 15,847 rd +   8,973 wr)
==25591== LL misses:      17,621 (   9,521 rd +   8,100 wr)
==25591== LL miss rate:    0.1% (   0.0% +   0.2% )
```

Na osnovu ovog i prethodnog primjera možemo vidjeti da se broj pro-mašaja keša za instrukcije i podatke značajno smanjio. Takođe, možemo vidjeti da se u prvom slučaju skoro tri puta više pristupalo kešu sa in-strukcijama nego kešu sa podacima, dok se u drugom slučaju taj broj smanjio na dva.

Detaljniji izvještaj *Cachegrind*-a nalazi se u fajlovima *cachegrind.out.<PID>* čiji sadržaj možemo ispisati komandom *cg_annotate*. Na samom početku datog fajla možemo pročitati neke karakteristike keš memorija koje se koriste što se može vidjeti na slici 6.

Slika 6: *Cachegrind* - osnovne karakteristike keša

```
I1 cache:      32768 B, 64 B, 8-way associative
D1 cache:      32768 B, 64 B, 8-way associative
LL cache:      16777216 B, 64 B, 8-way associative
Command:       ./a.out
Data file:     cachegrind.out.27208
Events recorded: Ir I1mr I1Lmr Dr D1mr D1Lmr Dw D1mw D1Lmw
Events shown:  Ir I1mr I1Lmr Dr D1mr D1Lmr Dw D1mw D1Lmw
Event sort order: Ir I1mr I1Lmr Dr D1mr D1Lmr Dw D1mw D1Lmw
Thresholds:    0.1 100 100 100 100 100 100 100 100
Include dirs:
User annotated:
Auto-annotation: on
```

Prvi broj predstavlja veličinu keša, drugi broj predstavlja veličinu linije (u keš pišemo liniju po liniju) dok treći predstavlja asocijativnost keša. Prilikom pozivanja alata *Cachegrind* moguće je da promijenimo veličinu *LL* keša. Ukoliko povećamo veličinu linije na 128 dobijamo izlaz koji se može vidjeti na slici 7.

Slika 7: *Cachegrind* - promjena veličine linije *LL* keša

```
milica@milica: ~/Desktop/VS_projekat/v1/Sources$ valgrind --tool=cachegrind --LL=16777216,8,128 ./a.out
==27325== Cachegrind, a cache and branch-prediction profiler
==27325== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.
==27325== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==27325== Command: ./a.out
==27325==
--27325-- warning: L3 cache found, using its data for the LL simulation.
==27325==
==27325== I   refs:      19,043,683
==27325== I1 misses:      2,212
==27325== LLi misses:      1,294
==27325== I1 miss rate:      0.01%
==27325== LLi miss rate:      0.01%
==27325==
==27325== D   refs:      9,317,110 (5,619,932 rd + 3,697,178 wr)
==27325== D1 misses:      22,608 ( 13,635 rd + 8,973 wr)
==27325== L1d misses:      8,310 ( 4,245 rd + 4,065 wr)
==27325== D1 miss rate:      0.2% ( 0.2% + 0.2% )
==27325== L1d miss rate:      0.1% ( 0.1% + 0.1% )
==27325==
==27325== LL refs:      24,820 ( 15,847 rd + 8,973 wr)
==27325== LL misses:      9,604 ( 5,539 rd + 4,065 wr)
==27325== LL miss rate:      0.0% ( 0.0% + 0.1% )
```

Dakle, vidimo da se broj promašaja *LL* keša smanjio skoro duplo, što je i očekivano s obzirom na to da je veličina linije duplo povećana. Sa druge strane, ako povećamo veličinu *LL* keša na *32MB* izlaz *Cachegrind*-a će ostati potpuno isti, tj. u ovom slučaju veličina keša ne utiče na učestalost promašaja 8.

Slika 8: *Cachegrind* - promjena veličine *LL* keša

```
milica@milica: ~/Desktop/VS_projekat/v1/Sources$ valgrind --tool=cachegrind --LL=33554432,8,64 ./a.out
==29850== Cachegrind, a cache and branch-prediction profiler
==29850== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.
==29850== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==29850== Command: ./a.out
==29850==
--29850-- warning: L3 cache found, using its data for the LL simulation.
==29850==
==29850== I   refs:      19,043,683
==29850== I1 misses:      2,212
==29850== LLi misses:      2,079
==29850== I1 miss rate:      0.01%
==29850== LLi miss rate:      0.01%
==29850==
==29850== D   refs:      9,317,110 (5,619,932 rd + 3,697,178 wr)
==29850== D1 misses:      22,608 ( 13,635 rd + 8,973 wr)
==29850== L1d misses:      15,542 ( 7,442 rd + 8,100 wr)
==29850== D1 miss rate:      0.2% ( 0.2% + 0.2% )
==29850== L1d miss rate:      0.2% ( 0.1% + 0.2% )
==29850==
==29850== LL refs:      24,820 ( 15,847 rd + 8,973 wr)
==29850== LL misses:      17,621 ( 9,521 rd + 8,100 wr)
==29850== LL miss rate:      0.1% ( 0.0% + 0.2% )
```

Analiza *cachegrind.out.<PID>* fajla iz terminala dosta je nepregledna pa se u tu svrhu preporučuje upotreba programa *KCachegrind*.

3 Perf

Perf je alat za profajliranje koji pruža jednostavan interfejs preko komandne linije. Zbog brzine rada alata analizu ćemo prikazati nad originalnim projektom (za razliku od *Valgrind*-a za koji smo obrađivali modifikovanu verziju). *Perf* prikazuje statistike, npr. koliko vremena je provedeno u određenoj funkciji. Profil programa se pravi na osnovu uzorka. Navedeni alat se može koristiti na dva načina:

- kao *Performance Analyzer* iz *Qt Creator* okruženja
- iz komandne linije

Upotreba *Perf*-a iz *Qt Creator*-a je jednostavna: pokretanjem *Performance Analyzer*-a u okviru debug prozora biće prikazane statistike vezane za sam projekat. Ukoliko ipak želimo da koristimo *Perf* iz komandne linije koristimo sledeće naredbe:

- izgraditi *Qt* projekat
- pozicionirati se u *build* folder
- `perf record -call-graph dwarf ./FireAndWater`
- `perf report`

Slika 9: *Perf* - izvještaj

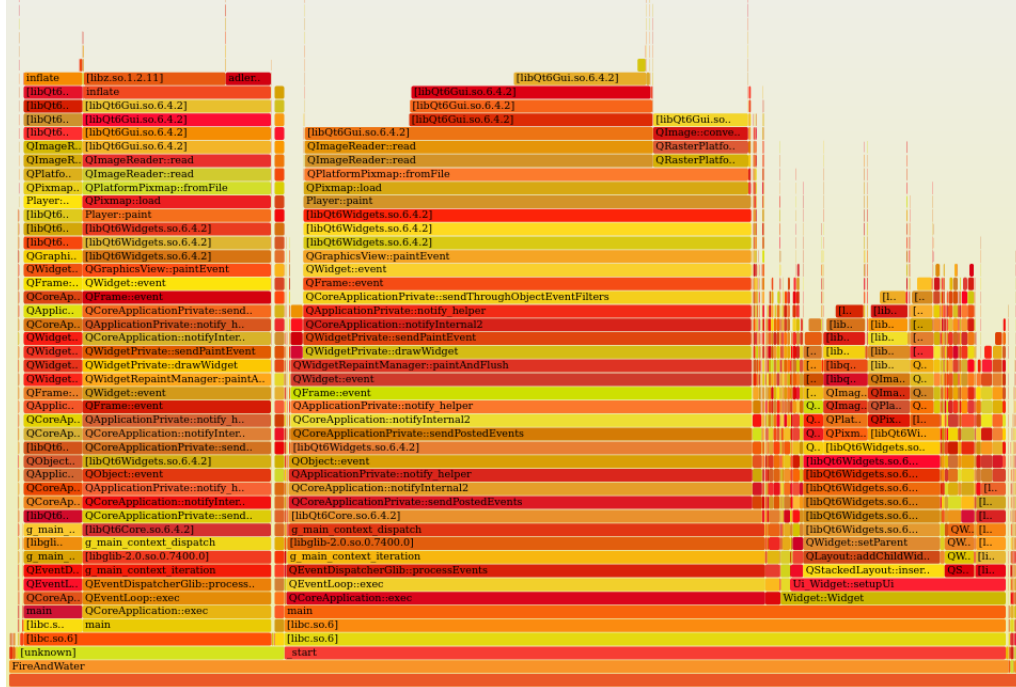
Samples: 193K of event 'cycles', Event count (approx.): 123170382047					
Children	Self	Command	Shared Object	Symbol	
+ 97,02%	0,00%	FireAndWater	FireAndWater	[.]	main
+ 96,06%	0,00%	FireAndWater	libc.so.6	[.]	0x00007f08ec42350f
+ 93,87%	0,00%	FireAndWater	libQt6Gui.so.6.4.2	[.]	QPixmap::load
+ 93,74%	0,00%	FireAndWater	libQt6Gui.so.6.4.2	[.]	QPlatformPixmap::fromFile
+ 84,22%	0,00%	FireAndWater	libQt6Gui.so.6.4.2	[.]	QImageReader::read
+ 84,22%	0,00%	FireAndWater	libQt6Gui.so.6.4.2	[.]	QImageReader::read
+ 79,35%	0,01%	FireAndWater	libQt6Core.so.6.4.2	[.]	QCoreApplication::notifyInt
+ 79,32%	0,00%	FireAndWater	libQt6Widgets.so.6.4.2	[.]	QApplicationPrivate::notify
+ 78,21%	0,00%	FireAndWater	libQt6Widgets.so.6.4.2	[.]	QWidget::event
+ 77,36%	0,00%	FireAndWater	libc.so.6	[.]	0x00007f08ec4235c8
+ 73,15%	0,00%	FireAndWater	libQt6Widgets.so.6.4.2	[.]	QFrame::event
+ 73,05%	0,00%	FireAndWater	libQt6Core.so.6.4.2	[.]	QEventLoop::exec
+ 73,05%	0,00%	FireAndWater	libQt6Core.so.6.4.2	[.]	QEventDispatcherGlib::proc
+ 73,00%	0,00%	FireAndWater	libQt6Core.so.6.4.2	[.]	QCoreApplication::exec
+ 72,96%	0,00%	FireAndWater	libglib-2.0.so.0.7400.0	[.]	g_main_context_iteration
+ 72,75%	0,00%	FireAndWater	libglib-2.0.so.0.7400.0	[.]	g_main_context_dispatch
+ 72,74%	0,00%	FireAndWater	libglib-2.0.so.0.7400.0	[.]	0x00007f08ec234227
+ 72,40%	0,00%	FireAndWater	libQt6Core.so.6.4.2	[.]	QObject::event
+ 71,53%	0,00%	FireAndWater	FireAndWater	[.]	_start
+ 71,52%	0,00%	FireAndWater	libQt6Core.so.6.4.2	[.]	QCoreApplicationPrivate::se
+ 71,52%	0,00%	FireAndWater	libQt6Core.so.6.4.2	[.]	0x00007f08ecfe1032
+ 71,46%	0,00%	FireAndWater	libQt6Widgets.so.6.4.2	[.]	QWidgetRepaintManager::pair
+ 71,30%	0,00%	FireAndWater	libQt6Widgets.so.6.4.2	[.]	0x00007f08ee317d4b
+ 70,15%	0,00%	FireAndWater	libQt6Widgets.so.6.4.2	[.]	QWidgetPrivate::drawWidget
+ 70,11%	0,00%	FireAndWater	libQt6Widgets.so.6.4.2	[.]	QWidgetPrivate::sendPaintEv
+ 70,11%	0,00%	FireAndWater	libQt6Core.so.6.4.2	[.]	QCoreApplicationPrivate::se
+ 70,10%	0,01%	FireAndWater	libQt6Widgets.so.6.4.2	[.]	QGraphicsView::paintEvent
+ 70,07%	0,00%	FireAndWater	libQt6Widgets.so.6.4.2	[.]	0x00007f08ee31bfc4
+ 70,06%	0,00%	FireAndWater	libQt6Widgets.so.6.4.2	[.]	0x00007f08ee31908b
+ 70,04%	0,00%	FireAndWater	libQt6Widgets.so.6.4.2	[.]	0x00007f08ee31824c
+ 70,04%	0,00%	FireAndWater	FireAndWater	[.]	Player::paint

Na slici 9 možemo vidjeti rezultat *Perf* izvještaja. Kolona *self* predstavlja procentualno izvršavanje funkcije na osnovu izabranog uzorka. Zbir kolone *self* treba da bude 100%. Za funkciju *g* kažemo da je dijete funkcije *f* ukoliko postoji konačan niz funkcija f_1, \dots, f_n $n \geq 0$ tako da $f \rightarrow f_1 \rightarrow \dots \rightarrow f_n \rightarrow g$ ($a \rightarrow b$ u značenju funkcija *a* poziva funkciju *b*). U koloni *children* prikazano je procentualno izvršavanje sve djece navedene funkcije. Kako pokrećemo igricu iz *main*-a to je očekivano da će broj izvršavanje njene djece biti najveći.

Za vizuelizaciju podataka dobijenih naredbom *perf report* koristićemo tzv. vatreni dijagram 10. Dijagram prikazuje populaciju uzoraka na *x* osi a dubinu steka na *y* osi. Svaka funkcija je jedan pravougaonik, širine relativne broju uzoraka. Vatreni dijagram se može dobiti na sledeći način:

- `git clone -depth 1 https://github.com/brendangregg/FlameGraph`
- `cp ../perf.data ./`
- `perf script | ./stackcollapse-perf.pl | ./flamegraph.pl > perf.svg`

Slika 10: *Perf* - vatreni dijagram



- Firefox perf.svg

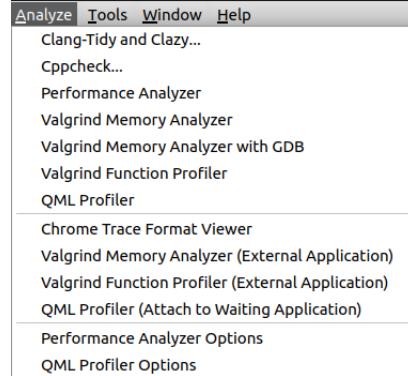
Na grafiku pored velikog broja *Qt* funkcija možemo uočiti i neke poznate funkcije i metode kao što su *main*, *Widget::widget*, *Player::paint*. Vodeći se datim statistikama, najviše napora za optimizaciju treba uložiti upravo u prethodne navedene funkcije, iz razloga što se najčešće izvršavaju. Nakon otvaranja vatrene grafika u *browser*-u možemo vidjeti iskorišćenje *CPU*-a na osnovu broja uzoraka koji su korišćeni.

4 Clang-Tidy i Clazy

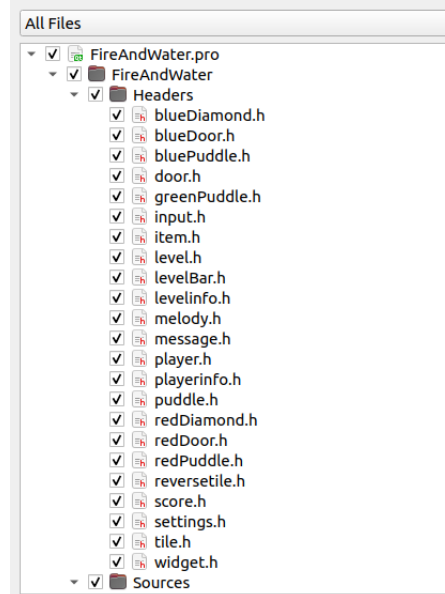
U okviru *Qt Creator*-a postoje ugrađeni alati *Clang-Tidy* i *Clazy* koji za detektovanje grešaka u programima pisanim u *C* i *C++* koriste statičku analizu. *Clang-Tidy* pronalazi standardne programerske greške kao što su nekonzistentnost u pisanju programa i greške u radu sa interfejsima. Statički analizator *Clang* je dio *Clang-Tidy*. *Clazy* omogućava *Clang*-u da razumije semantiku *Qt*-a. *Clazy* prikazuje upozorenja kompajlera vezana za *Qt*, u rasponu od nepotrebne alokacije memorije do zloupotrebe *API*-ja i pruža akcije za rešavanje nekih problema. Upotreba *Clang-Tidy* i *Clazy* iz *Qt Creator*-a (kao i ostalih analizatora koji su podržani od strane *Qt Creator*-a) se svodi na sledeće:

- U okviru kartice *Analyze* odabrati navedene alate 11.
- Označavanje fajlova koje želimo da testiramo. Zbog brzine analizatora i kompletnosti provjere preporučuje se analiza svih dostupnih fajlova u okviru projekta 12.

Slika 11: *Clang-Tidy i Clazy* - pokretanje



Slika 12: *Clang-Tidy i Clazy* - podešavanje



Pokretanjem alata *Clang-Tidy* i *Clazy* dobijamo izlaz prikazan na slici 13.

Slika 13: *Clang-Tidy i Clazy* - rezultat analize programa

```
public slots:
    void onPlayBtn();
    void onHighScoresBtn();
    void onSettingsBtn();
    void onQuitBtn();
    void onBackBtn();

private slots:
    void on_melodyBtn_stateChanged(int arg1);
    void updateScore();
    void createLevel();
```

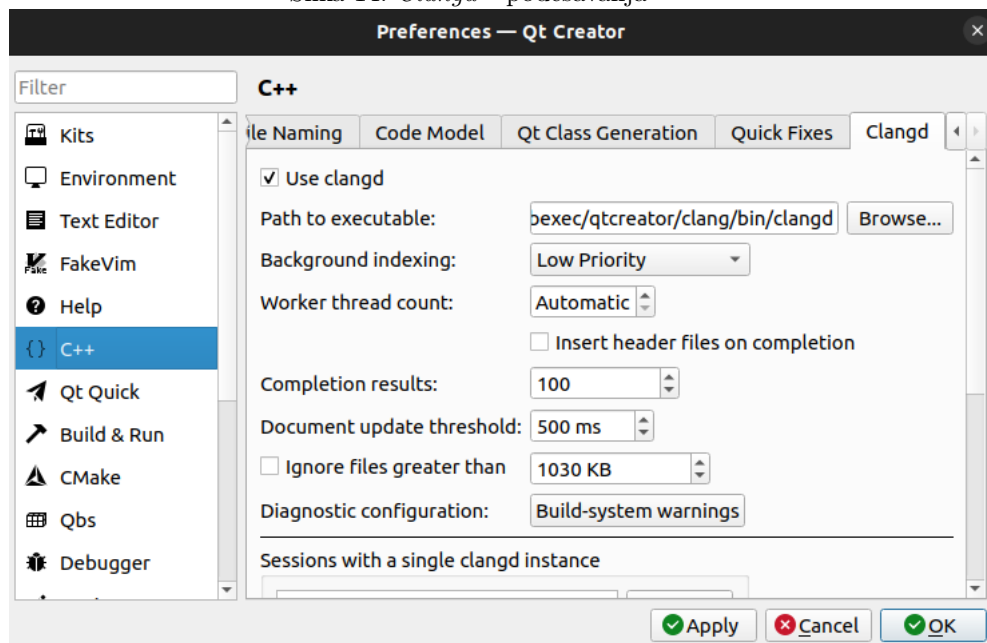
▲ Slots named on_foo_bar are error prone [clazy-connect-by-name]

Detektovana greška se odnosi na ime slota (funkcija koja reaguje na signale određenih objekata). Kao što možemo vidjeti na slici, svi slotovi izuzev jednog, koji koristi notaciju sa podvlakom, koriste kamilju notaciju. Zbog osobine konzistentnosti koja je veoma važna prilikom pisanja koda, kako bi se smanjila vjerovatnoća nastanka greške naziv `on_melodyBtn_stateChanged` bi trebalo popraviti u `onMelodyBtnStateChanged`.

5 Clangd

Clangd je jezički server koji postoji ugrađen u mnogim razvojnim okruženjima. Zasnovan je na *Clang C++* kompilatoru i dio je *LLVM* projekta. Koristi se za provjeru ispravnosti koda: ispituje kompletnost, nalazi kompilacione greške, ispituje definicije funkcija. . . U ovom izvještaju biće prikazana upotreba *Clangd*-a iz *Qt Creator*-a. Da bismo koristili *Clangd* neophodno je da podesimo odgovarajuću opciju. U gornjem meniju *Edit->Preferences->C++* i novom prozoru treba označiti opciju *Use clangd* što se može vidjeti na slici 14.

Slika 14: *Clangd* - podešavanja



Nakon toga klikom na *Apply* i *OK* biće započeta analiza projekta. Rezultati analize ispitivanog projekta mogu se vidjeti na slici 15.

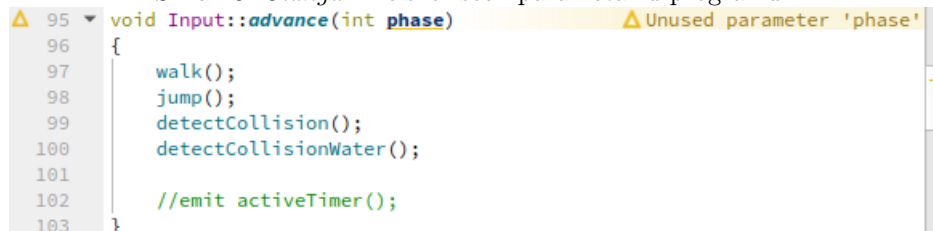
Slika 15: *Clangd* - rezultat analize

Issues			Filter
⚠	Unused parameter 'phase'	input.cpp	95
⚠	Non-void function does not return a value	input.cpp	333

Dakle, u klasi `input.cpp` postoji neiskorišćen parametar `phase` u liniji

95 što možemo vidjeti na slici 16. Druga pronađena semantička greška je

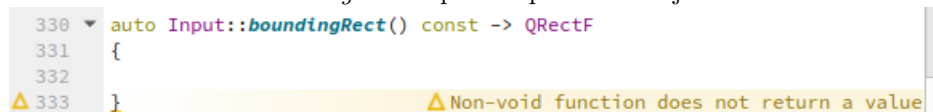
Slika 16: *Clangd* - neiskorišćen parametar u programu



```
95 void Input::advance(int phase) ⚠ Unused parameter 'phase'
96 {
97     walk();
98     jump();
99     detectCollision();
100    detectCollisionWater();
101
102    //emit activeTimer();
103 }
```

postojanje funkcija koja ne vraća vodi a nema povratnu vrijednost i može se vidjeti na slici 17.

Slika 17: *Clangd* - neispravna povratna vrijednost



```
330 auto Input::boundingRect() const -> QRectF
331 {
332
333 } ⚠ Non-void function does not return a value
```

U prvom slučaju dovoljno je izbrisati neiskorišćeni parametar a u drugom slučaju, s obzirom na to da je tijelo funkcije prazno, moguće je izbrisati kompletnu deklaraciju kao i definiciju u odgovarajućoj *.hpp* datoteci. Pronađene greške predstavljaju višak koda u projektu.

6 Zaključak

Na samom kraju možemo istaći koliko je primjena alata za verifikaciju važan dio razvoja softvera. Uz jednostavnu primjenu ovih alata u okviru projekta otkrivene greške su curenje memorije, nekonzistentnost i višak koda koje u nekim slučajevima mogu dovesti do fatalnih posledica. Veoma je važno da testiranje radimo što češće prilikom pisanja programa, kako bismo minimizirali vjerovatnoću nastanka grešaka.

Literatura

- [1] Ana Vulović Ivan Ristović. *Verifikacija softvera - vežbe*. Beograd, 2022.
- [2] Milena Vujošević Janičić. *Verifikacija softvera*. Matematički fakultet, Univerzitet u Beogradu, Beograd, 2022.