Извештај примене алата за верификацију над библиотеком за детекцију лица

Марија Ерић
marija.eric@matf.bg.ac.rs

Јануар 2023

Сажетак

Пројекат на коме је извршена анализа је библиотека за детекцију лица, написана у $C++$. Библиотека је отвореног кода и може се пронаћи на наредном линку. Примена алата је извршена на главној грани, над комитом чији је хеш код: $ec528ce43af9de94bf2fab308ce2d6270584881c$. У коду нису пронађени већи пропусти, поред некоришћених променљивих и цурења меморије.

Садржај

1 Верификација софтвера **2**
1.1 Динамичка анализа . . . 2
1.1.1 Измена програма . . . 2
1.1.2 *Gcov* . . . 3
1.2 Профајлирање . . . 5
1.2.1 *Memcheck* . . . 5
1.2.2 *Massif* . . . 7
1.2.3 *Callgrind* . . . 10
1.3 Статичка анализа . . . 11
1.3.1 *CppCheck* . . . 11

2 Закључак **14**

3 Покретање скрипти за репродуковање резултата **14**
3.1 Покретање алата . . . 15

1 Верификација софтвера

1.1 Динамичка анализа

1.1.1 Измена програма

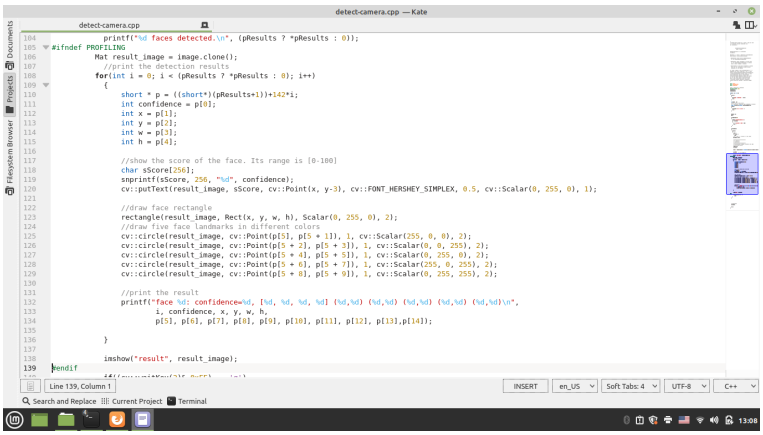
Да бисмо могли да репордукујемо резултате добијене приликом извршавања програма, неопходна је измена у оквиру $detect - camera$. На слици 1 је приказана измена. На левој слици се налази оригиналан код, а на десној измењени код.

```
67      VideoCapture cap;
68      Mat im;
69
70      if( isdigit(argv[1][0]))
71      {
72          cap.open(argv[1][0]-'0');
73          if(! cap.isOpened())
74          {
75              cerr << "Cannot open the camera." << endl;
76              return 0;
77          }
78      }
```

```
67      VideoCapture cap;
68      Mat im;
69
70      if( isdigit(argv[1][0]))
71      {
72          cap.open("../TestSamples/camera-test.mp4");
73          if(! cap.isOpened())
74          {
75              cerr << "Cannot open the camera." << endl;
76              return 0;
77          }
78      }
79
```

Слика 1: Измене у оквиру $detect - camera$

Такође, у оквиру програма $detect - image$ и $detect - camera$, поред библиотеке за детекцију лица, екстензивно се користи и билиотека $openCv$, за учитавање слика и приказивање резултата. Приликом покретања алата $Memcheck$ и $Massif$ желимо да тестирамо да ли у оквиру библиотеке за детекцију постоји цурење меморије, као и да ли се $heap$ одговорно користи. Из тог разлога уводимо још једну промену у $detect - camera$ и $detec - image$, где одређене делове кода не извршавамо приликом покретања горенаведених алата. На слици 2 је приказана промена у $detect - camera$. Аналогна промена је и у $detect - image$.



Слика 2: Измена у оквиру $detect - camera$

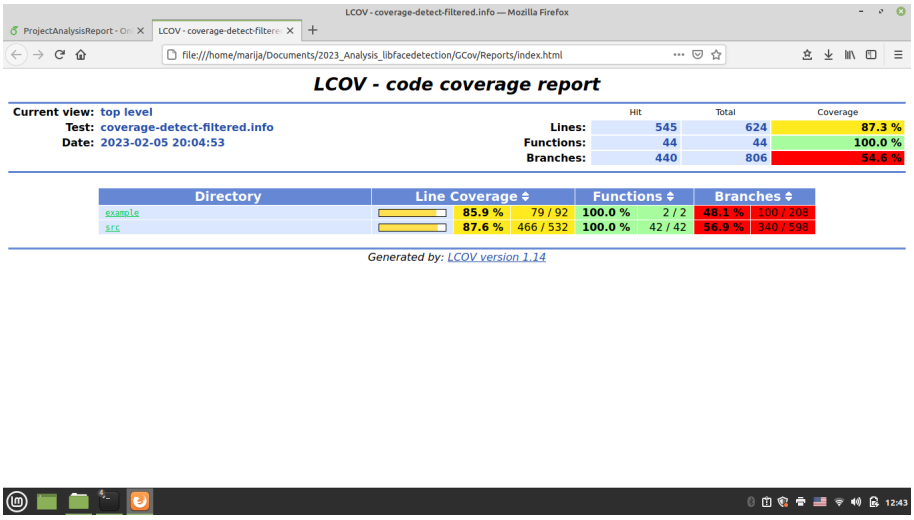
1.1.2 *Gcov*

Gcov алат за одредивање покривености кода приликом извршавања програма (*engl.codecoverage*). Користи се заједно са *gcc* компајлером да би се анализирао програм и утврдило како се може креирати ефикаснији, бржи код и да би се тестовима покрили делови програма. Зарад лепше репрезентације резултата детекције покривености кода извршавањем тест примера, користи се алат *lcov* [1].

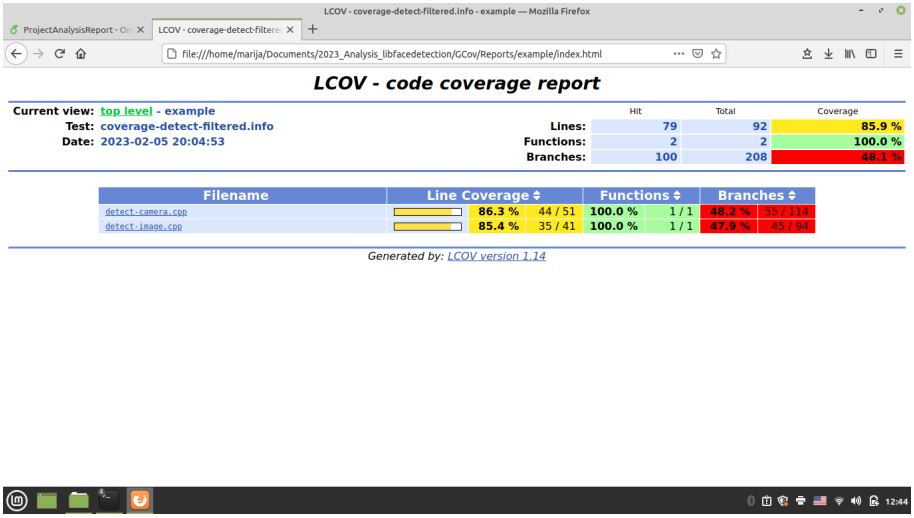
Покретања алата: Приликом компилације неопходно је користити додатне опције компајлера које омогућавају снимање колико је пута која линија, грана и функција извршена. Након извршавања програма може се генерисати извештај. У наставку се налазе наредбе за покретање алата *lcov*, филтрирање резултата и генерисање *html* извештаја.

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

lcov --rc lcov_branch_coverage=1 --r coverage-detect.info '/usr/*' '9*'\
-o coverage-detect-filtered.info
```

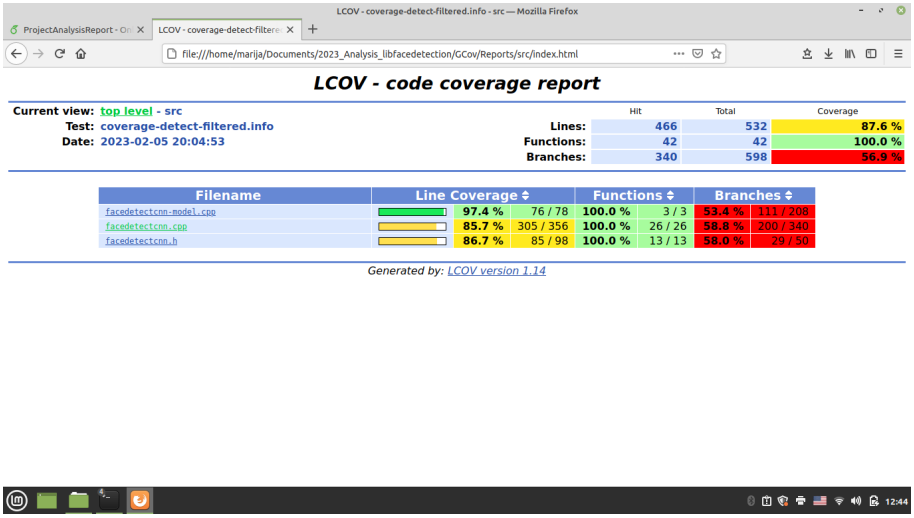
На слици 3 се налази извештај за оба примера употребе библиотеке: $detect - image$ и $detect - camera$. Анализирано је исправно покретање програма. Као што видимо покривеност кода је велика. Све функције дефинисане у оквиру библиотеке су искоришћене. Покривеност грана је мања (око 50%). У оквиру $detect - image$ и $detect - camera$ је велика покривеност наредби. Једине неизвршене су наредбе обраде погрешног улаза. Детаљан извештај се може видети на 4 и 5.



Слика 3: Извештај о покривеност кода



Слика 4: Покривеност кода $detect - image$ и $detect - cameta$



Слика 5: Покривеност кода $libfacedetection$

Резултати: Извештај је филтриран, и избачена је покривеност за *opencv* и *math*, са обзиром на то да није од значаја за нашу анализу.

Дакле, након покретања алата *lcov* можемо да закључимо да имамо велику покривеност кода, као и да немамо некоришћене фунцкије у оквиру библиотеке за детекцију лица.

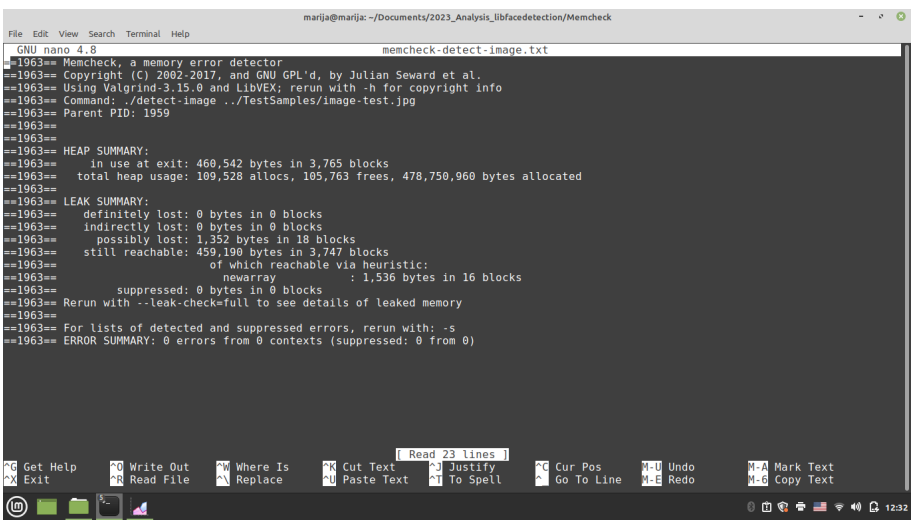
1.2 Профајлирање

Са обзиром на то да се у оквиру библиотеке користи динамичка алокација меморије, битно је да испитамо да ли је дошло до цурења меморије. Из тог разлога вршимо профајлирање програма $detect - image$ и $detect - camera$. У оквиру анализе су коришћени алати $Memcheck$, $Massif$ и $Callgrind$.

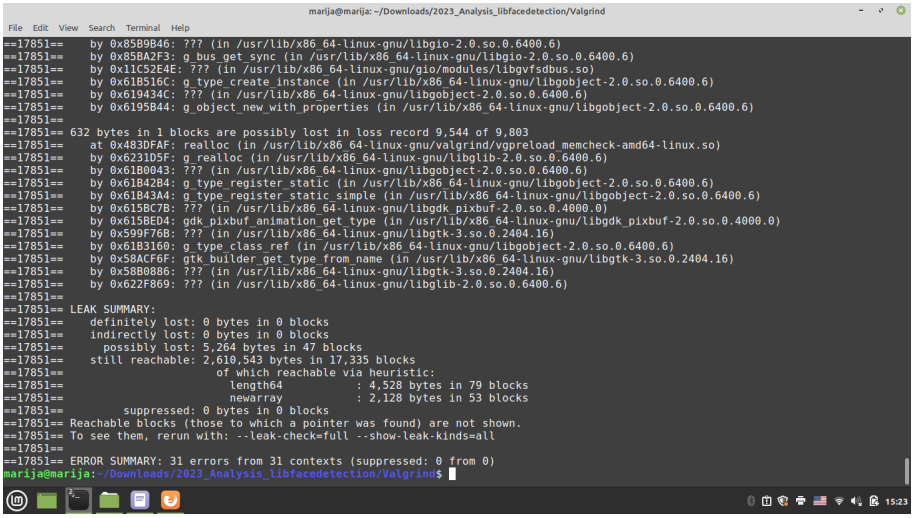
1.2.1 *Memcheck*

Покретање алата: Битно је да превођење извршимо у *debug* моду, као и да тестирамо само библиотеку за детекцију лица (додавањем $-DPROFILING$). Након тога, покрећемо $Memcheck$. Део извештаја је приказан на сликама 6 и 7, док се целокупан излаз може наћи у оквиру фолдера $Memcheck$ на *github* репозиторијуму. У наставку се налазе наредбе за покретање алата.

```
./../TestSamples/build CFLAGS="-DPROFILING"

valgrind  --log-file="memcheck-detect-image.txt"
./detect-image ../TestSamples/image-test.jpg

valgrind  --log-file="memcheck-detect-camera.txt" ./detect-camera 0
```



Слика 6: Део излаза алата $Memcheck$ над $detect - image$.

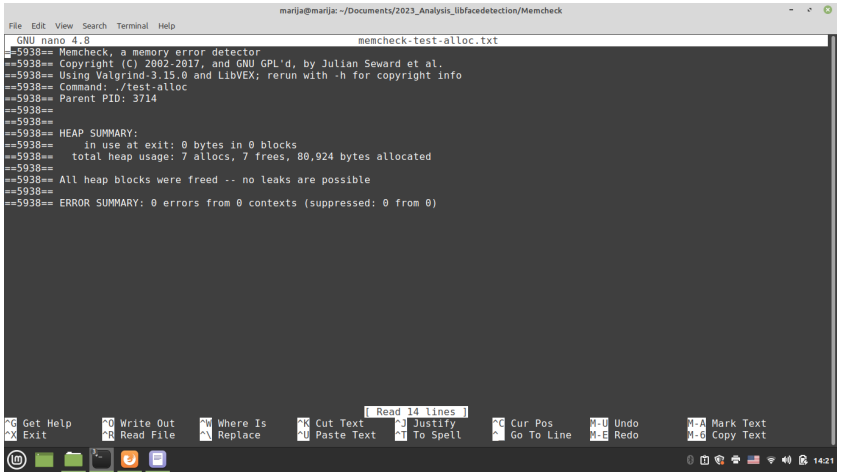


```
marija@marija: ~/Downloads/2023_Analysis_libfacedetection/Valgrind
File  Edit  View  Search  Terminal  Help
==17851==    by 0x85B9B46: ??? (in /usr/lib/x86_64-linux-gnu/libgio-2.0.so.0.6400.6)
==17851==    by 0x85BA2F3: g_bus_get_sync (in /usr/lib/x86_64-linux-gnu/libgio-2.0.so.0.6400.6)
==17851==    by 0x11C52E4E: ??? (in /usr/lib/x86_64-linux-gnu/gio/modules/libgvfsdbus.so)
==17851==    by 0x61B516C: g_type_create_instance (in /usr/lib/x86_64-linux-gnu/libgobject-2.0.so.0.6400.6)
==17851==    by 0x619434C: ??? (in /usr/lib/x86_64-linux-gnu/libgobject-2.0.so.0.6400.6)
==17851==    by 0x6195B44: g_object_new_with_properties (in /usr/lib/x86_64-linux-gnu/libgobject-2.0.so.0.6400.6)
==17851==
==17851== 632 bytes in 1 blocks are possibly lost in loss record 9,544 of 9,803
==17851==    at 0x483DFAF: realloc (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==17851==    by 0x6231D5F: g_realloc (in /usr/lib/x86_64-linux-gnu/libglib-2.0.so.0.6400.6)
==17851==    by 0x61B0043: ??? (in /usr/lib/x86_64-linux-gnu/libgobject-2.0.so.0.6400.6)
==17851==    by 0x61B42B4: g_type_register_static (in /usr/lib/x86_64-linux-gnu/libgobject-2.0.so.0.6400.6)
==17851==    by 0x61B43A4: g_type_register_static_simple (in /usr/lib/x86_64-linux-gnu/libgobject-2.0.so.0.6400.6)
==17851==    by 0x615BC7B: ??? (in /usr/lib/x86_64-linux-gnu/libgdk_pixbuf-2.0.so.0.4000.0)
==17851==    by 0x615BED4: gdk_pixbuf_animation_get_type (in /usr/lib/x86_64-linux-gnu/libgdk_pixbuf-2.0.so.0.4000.0)
==17851==    by 0x599F76B: ??? (in /usr/lib/x86_64-linux-gnu/libgtk-3.so.0.2404.16)
==17851==    by 0x61B3160: g_type_class_ref (in /usr/lib/x86_64-linux-gnu/libgobject-2.0.so.0.6400.6)
==17851==    by 0x58ACF6F: gtk_builder_get_type_from_name (in /usr/lib/x86_64-linux-gnu/libgtk-3.so.0.2404.16)
==17851==    by 0x58B0086: ??? (in /usr/lib/x86_64-linux-gnu/libgtk-3.so.0.2404.16)
==17851==    by 0x622F869: ??? (in /usr/lib/x86_64-linux-gnu/libglib-2.0.so.0.6400.6)
==17851==
==17851== LEAK SUMMARY:
==17851==    definitely lost: 0 bytes in 0 blocks
==17851==    indirectly lost: 0 bytes in 0 blocks
==17851==      possibly lost: 5,264 bytes in 47 blocks
==17851==    still reachable: 2,610,543 bytes in 17,335 blocks
==17851==                       of which reachable via heuristic:
==17851==                         length64           : 4,528 bytes in 79 blocks
==17851==                         newarray           : 2,128 bytes in 53 blocks
==17851==         suppressed: 0 bytes in 0 blocks
==17851== Reachable blocks (those to which a pointer was found) are not shown.
==17851== To see them, rerun with: --leak-check=full --show-leak-kinds=all
==17851==
==17851== ERROR SUMMARY: 31 errors from 31 contexts (suppressed: 0 from 0)
marija@marija:~/Downloads/2023_Analysis_libfacedetection/Valgrind$
```

Слика 7: Део излаза алата *Memcheck* над *detect − camera*.

Резултати: Добијамо јако сличне излазе за оба програма. На основу сазхетка алата видимо да нема цурења меморије. Такође видимо да имамо могући губитак. Када испитамо стек позива видимо да је могући губитак изазвао позив функције *calloc*, али закључујемо да не постоји губитак меморије који је изазван од стране програмера.

Додатно, након анализе изворног кода библиотеке, можемо закључити да се приликом динамичке алокације користе *MyAlloc* и *MyFree* функције. Да бисмо били сигурни да се алокације и деалокације врши исправно, написан је додатни *test − alloc.cpp*, у ком се врши позивање ових функција. Извештај алата *Memcheck* се налази на слици 8. Видимо да нема цурења меморије, и да се алокација и деалокација врши исправно.



Слика 8: Извештај алата *Memcheck* за програм $test - alloc$

1.2.2 *Massif*

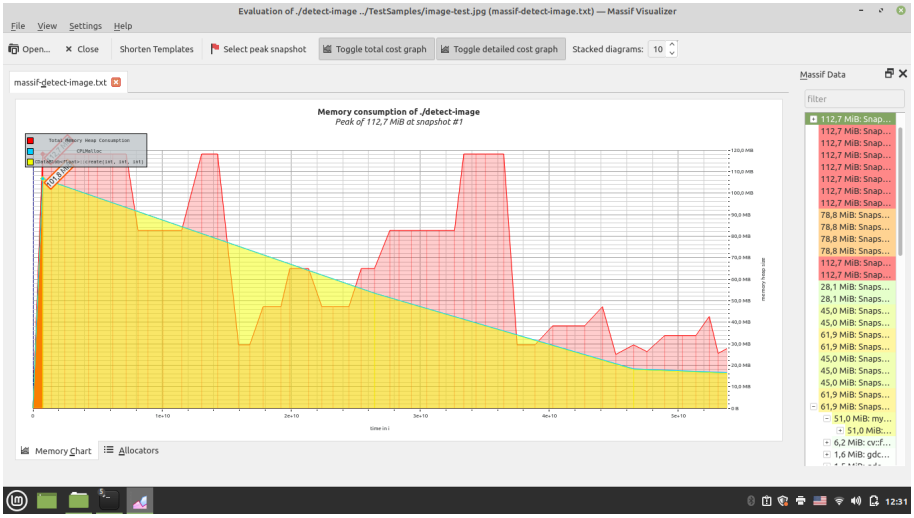
Massif представља профајлер *heap* меморије [4]. Овај алат мери колико *heap* меморије програм користи (корисна меморије + додатна меморија за администрацију). У оквиру ове анализе је мерена и величина стека коришћена у оквиру програма.

Покретање алата: Мотивација иза коришћења и овог алата лежи у томе да Мемчецк не може да детектује сва потенцијална цурења меморије.

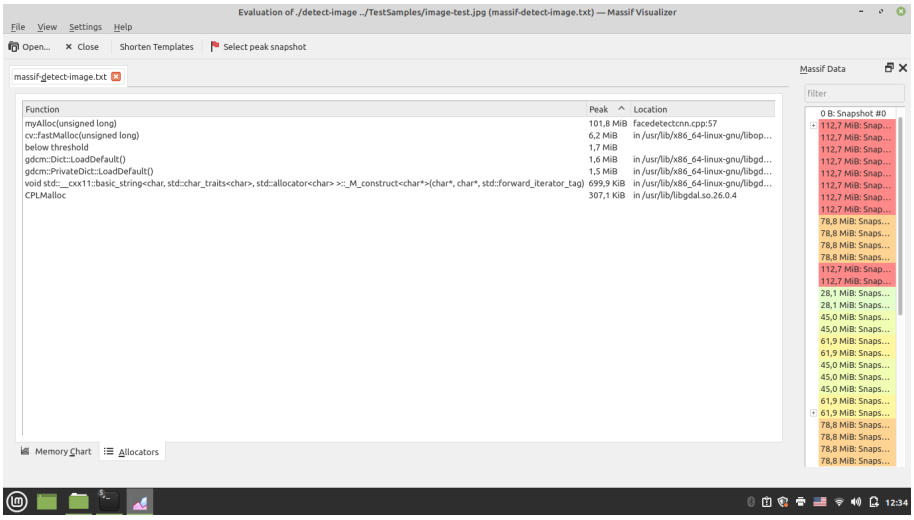
```
./../TestSamples/build CFLAGS="-DPROFILING"

# Pokretanje Massifa za detect-camera
valgrind --tool=massif --stacks=yes --massif-out-file="massif-detect-image.txt" \
./detect-image ../TestSamples/image-test.jpg

valgrind --tool=massif --stacks=yes --massif-out-file="massif-detect-camera.txt" \
./detect-camera 0
```

Резултати: За визуелизацију резултата коришћен KDE massif-visualizer.

На сликама 9 и 10 се налазе извештаји везани за програм $detect - image$, док се на сликама 11 и 12 налазе извештаји везани за програм $detect - camera$.



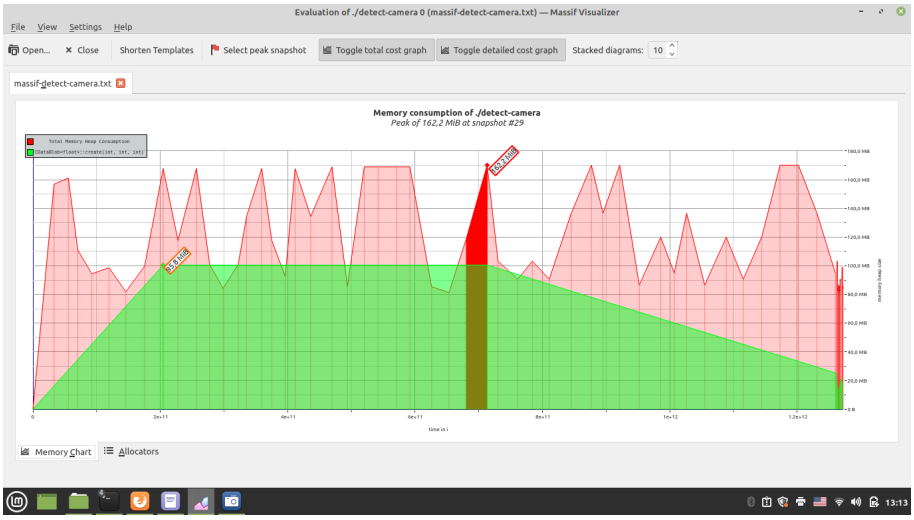
Слика 9: Коришћење меморије у програму $detect - image$



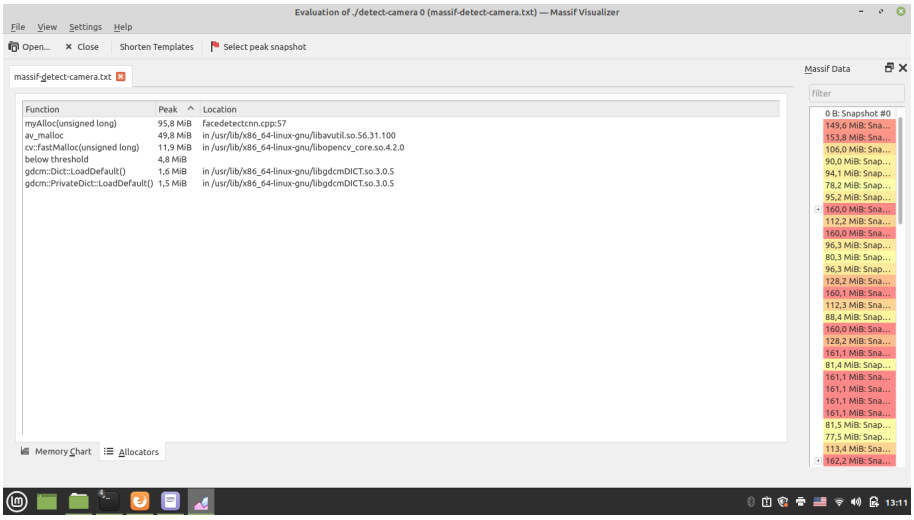
Слика 10: Алокатори у програму $detect - image$

На слици 10 можемо видети главне алокаторе током извршавања програма $detect - image$. Може се приметити да од свих алокатора, једино је функција *create* из библиотеке за детекцију лица. Фунцкија *create* позива функцију $MyAlloc$ коју смо већ тестирали. На слици 9 се налази детаљан приказ $heap - a$ током извршавања. Црвеном бојом је приказана укупна меморија $heap - a$, а наранџастом колико меморије заузима *create*. Видимо да се на почетку извршавања врши алоцирање меморије која ће се кори-

стити у наставку, и *peak* се дешава у неколико првих пресека. Након тога се само вршу деалокација меморије. На основу графика закључујемо да се *heap* одговорно користи.



Слика 11: Коришћење меморије у програму *detect – camera*



Слика 12: Алокатори у програму *detect – camera*

На слици 11 је приказао стање *heapa* током извршавања програма *detect–camera*. Промене на стеку су још чешће, пошто се детекција врши на већем

броју слика, али нема скокова приликом позива фунцкија из библиотеке за детекцију.

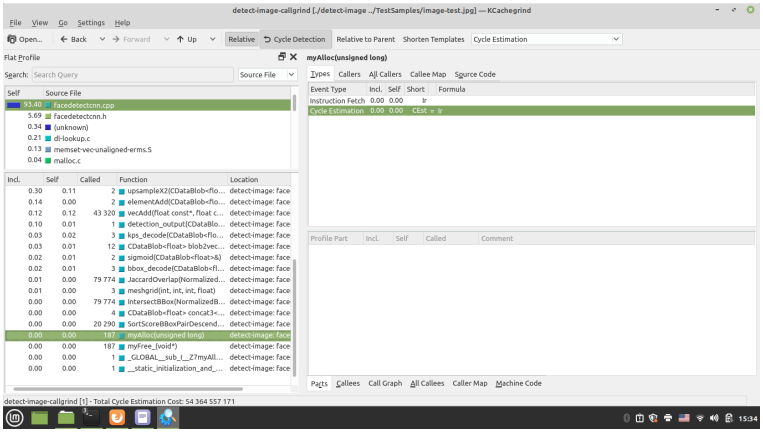
1.2.3 *Callgrind*

Callgrind је алат који генерише листу позива функција корисничког програма у виду графа. [3]. Покретањем овог алата добијамо информацију о броју извршених инструкција, позиваоца одређене функције, као и број позива. Као додатна потврда резултата добијених позивом *Memcheck* и *Massif*, покренут је и алат *Callgrind*.

Покретање алата:

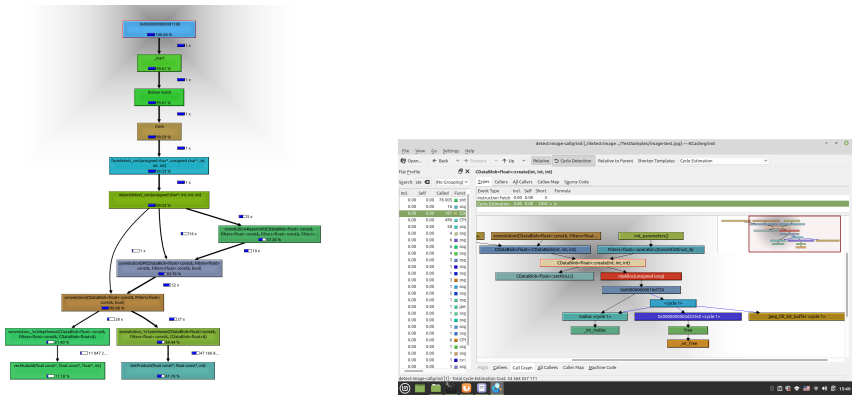
```
./../TestSamples/build CFLAGS="-DPROFILING"

valgrind --tool=callgrind --callgrind-out-file='detect-image-callgrind' \
./detect-image ../TestSamples/image-test.jpg

valgrind --tool=callgrind --callgrind-out-file='detect-camera-callgrind' \
./detect-camera 0
```

Резултати: За визуелизацију резултата коришћен Kcachegrind. Пре свега, на слици 13 се може видети да имамо једнак број позива фунцкије *MyAlloc* и *MyFree*, чиме одбијамо још једну потврду да је рад са динамичком алокацијом исправан.



Слика 13: Извештај за позивање алата *callgrind* над $detect - image$.

На слици 14 са леве стране се налази граф позива, док је на десној слици приказан граф позива функције *create* који смо анализирали алатима



Слика 14: Граф позива функција у $detect - image$

$Memcheck$ и $Massif$. Можемо видети да се она састоји од већ анализиране функције $MyAlloc$. Овим завршавамо анализу динамичке алокације, и сматрамо да нема пропуста у раду са динамичком меморијом.

Са обзиром на то да програм $detect - camera$ има сличну функционалност као и $detect - image$, само детекцију врши над већим бројем слика, не постоји разлика у самој употреби анализиране библиотеке, па су и резултати позива алата $Callgrind$ слични. Једина разлика је у броју позива функција, не и у графу позива. Детаљан извештај се може наћи у оквиру $github$ репозиторијума.

1.3 Статичка анализа

1.3.1 *CppCheck*

$CppCheck$ је алат за статичку анализу $C/C++$ кода, који пружа јединствену анализу за детекцију грешака, са фокусом на недефинисано понашање и са опасним конструктима кода. Анализа добијена $CppCheck$ није ни сагласна, нити је потпуна. Дакле, може имати и лажно позитивне резултате, као и лажно негативне. Могуће поруке:

- Грешка - недефинисано понашање (цурење меморије или цурење ресурса)
- Стил - редундантност, некоришћене функције/променљиве, потенцијалне грешке
- Перформансе - поправка ових порука не гарантује убрзање (јер је статичка анализа у питању)
- Преносивост
- Конфигурационе информације

Покретање алата: Детаљна инсталација званичној страници. Детаљан опис начина коришћења алата се може наћи у [2].

Приликом инсталације на Дебиан дистрибуцији:

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

Покретање алата *cppcheck* над пројектом $libfacedetection$:

```
cppcheck --enable=all --output-file="cppCheckOut.xml" --xml
--inconclusive libfacedetection/
```

Додатни флагови:
$--enable = all$ - проналазак сви грешака
$--output - file$ - дефинисање фајла у који се уписује
$--xml$ - поруке у xml формату
$--inconclusive$ - неуверљиве грешке (потенцијални $false\ positive$)
У оквиру алата је могуће направити $HTMLreport$ од излазне поруке сачуване у xml формату.

```
cppcheck-htmlreport --report-dir=CppCheckReport --output-file="cppCheckOut.xml"
```

Резултати: На слици 15 се налази статистика анализе.

Слика 15: Статистика анализе алата *cppcheck*

На основу статистике видимо да је пронађена једна порука типа *грешка*, и 5 порука типа *стил*.

Детаљније о порукама се може видети на слици 16.

Cppcheck report - [project name]:

Defect summary:

- [x] Toggle all

Show # Defect ID

- [x] 2 unreadVariable
- [x] 1 constArgument
- [x] 1 constParameter
- [x] 1 funcArgNamesDifferent
- [x] 1 memleak
- [x] 1 missingInclude
- [x] 1 unusedFunction

8 total

Line	Id	CWE	Severity	Message
	missingInclude		information	Cppcheck cannot find all the include files (use --check-config for details)
libfacedetection/example/benchmark.cpp				
45	constArgument	570	style	Argument '131072*num_thread' to function malloc is always 131072
57	unreadVariable	563	style	Variable 'pResults' is assigned a value that is never used.
71	unreadVariable	563	style	Variable 'pResults' is assigned a value that is never used.
libfacedetection/example/detect-camera.cpp				
76	memleak	401	error	Memory leak: pBuffer
libfacedetection/mobile/Android/FaceDetection/app/src/main/cpp/facedetectcnn-jni.cpp				
15	unusedFunction	561	style	The function 'Java_org_dp_facedetection_MainActivity_facedetect' is never used.
libfacedetection/src/facedetectcnn.cpp				
353	constParameter	398	style	Parameter 'inputoutputData' can be declared with const
661	funcArgNamesDifferent	628	style, inconcl	Function 'kps_decode' argument 1 names different: declaration 'bbox_pred' definition 'kps_pred'

Cppcheck 1.90 - a tool for static C/C++ code analysis
Statistics

Internet: http://cppcheck.net
IRC: irc://irc.freenode.net/cppcheck

Слика 16: Извештај алата *cppcheck*

Грешка се налази у 76 линији у фајлу $detect-camera.cpp$. Део кода који изазива грешку је дат у наставку:

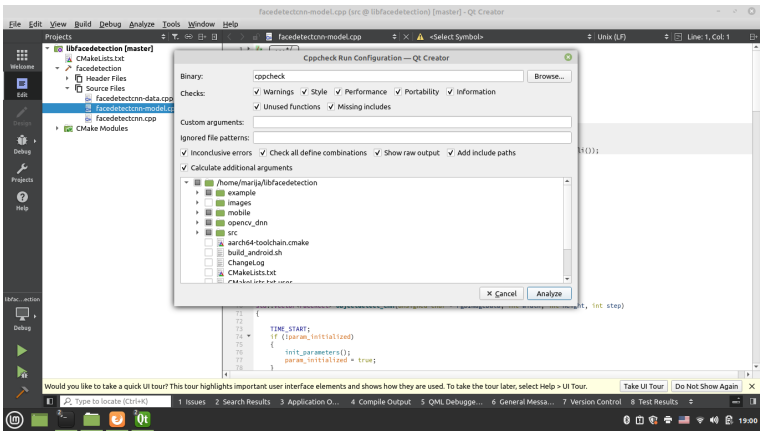
```
55
56          int * pResults = NULL;
57      //pBuffer is used in the detection functions.
58      //If you call functions in multiple threads, please create one buffer for each thread!
59      unsigned char * pBuffer = (unsigned char *)malloc(DETECT_BUFFER_SIZE);
60      if(!pBuffer)
61      {
62          fprintf(stderr, "Can not alloc buffer.\n");
63          return -1;
64      }
65
66
67      VideoCapture cap;
68      Mat im;
69
70      if( isdigit(argv[1][0]))
71      {
72          cap.open(argv[1][0]-'0');
73          if(! cap.isOpened())
74          {
75              cerr << "Cannot open the camera." << endl;
76              return 0;
77          }
```

Дакле, меморија које је алоцирана у линији 59 се не ослобађа пре прекида извршавања програма, уколико је $VideoCapture$ неуспешно отворен.

Поред тога су пријављена упозорења за некоришћене променљиве и функције. Детаљан извештај се може пронаћи у оквиру репозиторијума за анализу пројекта, у оквиру фолдера $CppCheckReport$. У наведеном репозиторијуму је за сваки фајл у коме је алат пронашао неки проблем направљен *html* извештај са обележеним местима са пронађеним проблемима.

Такође, могуће је покретање алата у оквиру развојног окрижења *QtCreator*. Прво је потребно додати алат: **Help->About Plugins -> Code Analyzer**. Изабрати *Cppcheck*. Након тога је неопходно рестартовати окружење.

Покретање алата: **Analyze -> Cppcheck** након чега се отвара прозор приказан на слици 17.



Слика 17: Покретање алата у оквиру *QtCreatora*

Селектовањем поља *inconclusive errors* се пријављују и лажна упозорења (*false positive*).

2 Закључак

У оквиру пројекта је анализирана библиотека за детекцију лица и два њена примера употребе - детекција лица на прослеђеној слици и детекција лица са камере уређаја. Коришћена су четири алата за анализу. На основу њих можемо закључити да библиотека нема већих пропуста. Динамичком анализом нису пронађени пропусти, док је статичка анализа пронашла цурење меморије и неколико некоришћених променљивих.

3 Покретање скрипти за репродуковање резултата

```
git clone https://github.com/MATF-Software-Verification/2023_Analysis_libfacedetection
cd 2023_Analysis_libfacedetection/libfacedetection
git submodule init
git submodule update
```

Превођење библиотеке

```
cd TestSamples
./buildLib
```

3.1 Покретање алата

Пре покретања алата потребно је преузимање пројекта и превођење библиотеке.
У наставку су наведени кораци за покретање скрипти за превођење програма $detect - image$ и $detect - camera$, као и покретање алата коришћених за анализу. Детаљан садржај скрипти се може пронаћи у оквиру репозиторијума.

```
cd ../TestSamples
./build
```

CppCheck

```
cd ../CppCheckReport
./run_cppcheck
```

Lcov

```
cd ../GCov
./run_gcov
```

Memcheck

```
cd ../Memcheck
./run_memcheck
```

Massif

```
cd ../Massif
./run_massif
```

Callgrind

```
cd ../Callgrind
./run_callgrind
```

Литература

[1] Ana Vulović Ivan Ristović. Verifikacija softvera skripta sa vežbi.

[2] Cppcheck team. Cppcheck manual.

[3] Josef Weidendorfer. Kcachegrind. `https://kcachegrind.github.io/html/Home.html`.

[4] Milian Wolff. KDE/massif-visualizer. `https://github.com/KDE/massif-visualizer`, 2011.