TARTU ÜLIKOOL
Arvutiteaduse instituut
Informaatika õppekava

Karl-Mattias Tepp

Staatilise analüsaatori Goblint tulemuste visualiseerimine

Bakalaureusetöö (9 EAP)

Juhendaja: Vesal Vojdani, PhD

Tartu 2017

Staatilise analüsaatori Goblint tulemuste visualiseerimine

Lühikokkuvõte: Goblint on staatiline andmejooksude analüsaator mitmelõimelistestele C-keelsetele programmidele. Käesoleva töö eesmärk on analüüsida ja tõsta Goblinti kasutatavust. Analüüsi käigus leitakse parandatavad aspektid ja väljundi parema visualiseerimise loogika. Analüüsi põhjal on loodud programm, mis võtab sisendiks esialgse Goblinti väljundi ja lihtsustab seda, tuues välja olulisema ja kaotades vähem olulise. Väljundi lihtsustamiseks leitakse tõekspidamistele vastavad ja nendest hälbivad kohad, kusjuures tõekspidamine on siinkohal komplekt mäluaadressi poole pöördumisest ja selle juures võetud lukust. Tulemused saab järjestada tõekspidamise kindluse järgi. Seeläbi saab esile tuua kõige silmapaistvamad ja kõige kergemini parandatavad vead. Lisaks sellele antakse kogu väljundile kompaktsem kuju, et selle lugemisele kuluks vähem aega. Uus tulemus on esialgsega võrreldes rohkem kui kolmandiku võrra lühem ja annab edasi programmeerijale olulist informatsiooni, mida muidu oleks pidanud käsitsi eraldama.

Võtmesõnad: staatiline analüüs, Goblint, mitmelõimelisus, andmejooks

CERCS: P170 Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine (automaatjuhtimisteooria)

Visualising the Output of the Static Analyser Goblint

Abstract: Goblint is a static data race analyzer for multithreaded programs written in C. The goal of this thesis is to improve the usability of Goblint by improving the readability of its output. A strategy for better race warning representation was found and implemented in a separate tool that simplifies Goblint's output. Beliefs are formed based on observed pairings of memory accesses with associated locks. The results are then sorted according to the strength of the belief, prioritizing the most apparent mistakes that violate strong beliefs. The new result is less than a third of the original, and more importantly, beliefs are used to pinpoint the access that is most likely the cause of a potential race. This is a significant improvement over the current Goblint output, which reports all accesses and requires the user to find faulty locations manually.

Keywords: static analysis, Goblint, multithreading, data race

CERCS: P170 Computer science, numerical analysis, systems, control

Sisukord

Sissejuhatus

Tarkvara arendamise juures on pidevaks ohuks, et arendamise ja testimise ajal ei ole kõigele mõeldud. Ilmnevad olukorrad, kus arendatud tarkvara käitub teisiti kui planeeritud, sest programmeerijad teevad inimlikke vigu ja saavad teiste kirjutatud koodist valesti aru. Leidub vigu, mis võivad kirjutamise käigus kergelt tekkida, aga lugemise käigus ei pruugi olla lihtsalt avastatavad. Siin tulevad appi tööriistad, mis loevad kirjutatud koodi ja üritavad sellest leida vigu.

Staatiline analüsaator on tööriist, mis analüüsib programmi koodi seda jooksutamata. Staatilise analüsaatori eesmärgiks on anda hinnang kirjutatud programmile ja aidata programmist leida vigu. Eriti kasulikud on analüsaatorid, mis on usaldusväärsed (*sound*). Usaldusväärsed analüsaatorid määratlevad ära, mis tüüpi vigu nad otsivad, ja kui need analüsaatorid programmist vigu ei leia, siis võib olla kindel, et antud tüüpi vigu programmis ei leidu.

Goblint on staatiline analüsaator, mida arendatakse Tartu Ülikoolis koostöös Müncheni Tehnikaülikooliga. Selle eesmärgiks on leida vigu mitmelõimelistes C keeles kirjutatud programmides, avastades usaldusväärselt andmejookse [22]. Selliseid vigu on programmeerijal endal väga keeruline programmist avastada ja seetõttu võiks Goblintist palju kasu olla. Juba praegu võib seda tööriista pidada parimaks omalaadseks, kuid tema akadeemilise olemuse ja spetsiifilisi teadmisi nõudva kasutajakogemuse tõttu ei ole Goblint veel levinud. Antud töö eesmärgiks on lõppkasutajatele Goblinti kasutamise mugavamaks muutmine.

Töö autori panuseks on Goblinti kasutatavuse analüüs ja seejärel kasutatavuse tõstmine. Kasutatavuse tõstmise oluliseks sammuks on väljundist optimeeritavate omaduste leidmine. Nende põhjal toimub lõpuks Goblinti väljundi töötlemine. Praktilise osa tulemusena valmis Java programm, mis võtab sisendiks esialgse Goblinti väljundi ja annab tulemusena välja uue, optimeeritud väljundi.

Antud töö esimeses peatükis antakse lugejale taustteadmised staatilise analüüsi ja mitmelõimelisuse kohta, kirjeldatakse Goblinti poolt avastatavaid vigu ehk andmejookse, seletatakse ära, kuidas lukkude abil saab andmejookse vältida ning tuuakse näiteid Goblintile sarnaste tööriistade kohta. Teine peatükk tegeleb Goblinti kasutatavuse analüüsiga ja väljundi visualiseerimise

loogika leidmisega. Selleks tuuakse välja omadused, mis ühel hästi kasutataval analüsaatoril olema peaksid, võrreldakse Goblinti teise analüsaatoriga ja lõpuks kirjeldatakse ideed, kuidas Goblinti väljundit võiks nende teadmiste põhjal parandada. Kolmas peatükk kirjeldab valminud töö praktilist osa, andes ülevaate eelmise peatüki põhjal kirjutatud lahendusest ja tuues näite programmi tööst. Neljandas peatükis analüüsitakse valminud lahendust kvalitatiivselt ja kvantitatiivselt ning demonstreeritakse uue väljundi paremust. Lisaks leitakse viis, kuidas kasutatavust ja loetavust tulevikus veelgi arendada saaks. Lisas on välja toodud näide Goblinti väljundist XML formaadis, sest see on praktilise töö tulemusena valminud programmi sisendiks. Töö lõpus on lihtlitsents lõputöö reprodutseerimiseks ja lõputöö üldsusele kättesaadavaks tegemiseks.

1 Taustainfo

Antud töö loetavuse huvides antakse selles peatükis vajaminev taustainfo ja terminite seletused. Esimeses jaotises kirjeldatakse staatilise analüüsi põhimõtteid. Teises jaotises räägitakse mitmelõimelisusest, ühest mitmelõimelisusega kaasnevast ohust ehk andmejooksust ja mitmelõimelisusega seonduvate probleemide ühest lahendusest ehk lukkudest. Kolmandas jaotises tuuakse näiteid Goblintiga sarnaste tööriistade kohta ning võrreldakse ühe tööriista ja Goblinti väljundeid.

1.1 Staatiline analüüs

Aastal 1953 tõestati peatumise probleemi üldistus — Rice'i teoreem. Sellest on teada, et programmi funktsionaalse käitumise kohta esitatavatele mitte-triviaalsetele küsimustele ei saa algoritmi abil vastata kindlalt „jah“ või kindlalt „ei“ (triviaalne probleem on siin kontekstis selline, mille vastus on iga programmi korral „jah“ või iga programmi korral „ei“) [17]. Samamoodi ei saa anda täpseid vastuseid ka paljudele programmi mitte-funktsionaalse käitumise kohta käivatele küsimustele, nagu näiteks dünaamiline mälu [10] ja mitmelõimelisus [16]. Staatilise analüüsi eesmärgiks on sellele vaatamata ennustada programmi käitumise kohta käivatele küsimustele vastuseid.

Näiteks kui meil on suvalise programmi kohta küsimus: „Kas ühe mälu aadressi poole võib samal ajal pöörduda kaks erinevat lõime?“, siis ei saa sellele kindlat vastust anda. Staatilise analüüsi abil vastatakse aga kas „Ei“ või „Ei tea“. Baasjuhul saab lihtsalt iga kord vastata „Ei tea“, aga mida tihemini saab vastata kindlalt „Ei“, seda parema analüüsiga on tegemist [12]. Programmeerija saab manuaalselt üle vaadata kõik kohad koodis, kus analüsaator ei ole kindel, kas seal kirjutatud kood on korrektne või vigane, ja võimalusel koodi parandada nii, et analüsaator saaks kindlat öelda, et antud analüüsi perspektiivist on kood korrektne.

1.2 Mitmelõimelisus ja andmejooks

Programmi lõim on iseseisev kergekaaluline protsess ja erinevad programmi lõimed jagavad ühist mälu. Mitmelõimeliste programmide juures jooksevad mi-

tu programmi osa samaaegselt erinevates lõimedes [21]. Samaaegselt jooksvad programmi osad võivad täita kas sama koodi või erinevat koodi. Näiteks veebilehitseja erinevad sakid täidavad sama eesmärki, kuid programmi täitmine saab toimuda mitmes sakis samaaegselt. Mitmelõimelisus on oluline programmide omadus, sest võimaldab väiksema kuluga luua palju protsesse ning lisaks saab igal ajahetkel jagada arvutusressurssi nendesse programmi osadesse, kus seda kõige rohkem vaja on. Mitmelõimelisusega kaasnevad aga erinevad ohud, näiteks võib valesti programmeeritud mitmelõimelise programmi korral programmi käitumine ja tulemus muutuda ettearvamatuks.

Andmejooks esineb programmi koodis, kui kaks samaaegselt töötavat lõime võivad pöörduda sama mäluaadressi poole samaaegselt nii, et vähemalt üks neist kirjutab antud aadressile [18]. Mälupöördusi saab mitmelõimelise programmi juures organiseerida lukkude (*lock*) abil. Kui osa koodist on kaitstud lukuga, siis selleks, et pääseda seda osa koodist täitma, peab lõim selle luku enda kasutusse võtma. Ühte kindlat lukku saab aga võtta korraga ainult üks lõim ja teised peavad ootama, kuni lukk vabastatud on. Kui leidub lukk, mida võtavad kõik lõimed enne kindla jagatud mäluaadressi poole pöördumist, siis selle mäluaadressi juures andmejooksu ei toimu, sest korraga pöördub selle mäluaadressi poole vaid üks lõim.

```
#include <pthread.h>
#include <stdio.h>
int myglobal;
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t mutex2 = PTHREAD_MUTEX_INITIALIZER;

void *t_fun(void *arg) {
  pthread_mutex_lock(&mutex1);
  myglobal=myglobal+1; // RACE!
  pthread_mutex_unlock(&mutex1);
  return NULL;
}
int main(void) {
  pthread_t id;
  pthread_create(&id, NULL, t_fun, NULL);
  pthread_mutex_lock(&mutex2);
  myglobal=myglobal+1; // RACE!
  pthread_mutex_unlock(&mutex2);
  pthread_join (id, NULL);
  return 0;
}
```

Joonis 1: Lihtne näidisprogramm, mis vea tõttu sisaldab andmejooksu

Andmejooksu näitena on joonisel 1 toodud programm, kus funktisoonis *main* luuakse uus lõim funktsioonist *t_fun*. Kuna mõlemas funktsioonis loetakse ja kirjutatakse üle muutuja *myglobal* ja võetud lukud on erinevad, siis järelikult saab muutuja poole pöördumine toimuda samaaegselt. See omakorda tähendab andmejooksu. Samaaegselt toimuv mälupöördus erinevate lõimede poolt muudab programmi käitumise ettearvamatuks. Selline ettearvamatus on väga harva programmi soovitud käitumine ja seetõttu on hea, kui saab programmist kõik andmejooksud üles leida.

1.3 Goblintiga sarnased tööriistad

Goblinti väljund ülal toodud näidisprogrammi korral (vaata joonis 1) on toodud joonisel 2. Nagu jooniselt näha, on väljundis ära toodud kõik kirjutamised ja lugemised, mis andmejooksu põhjustava muutuja peal on tehtud, ning välja on kirjutatud kirjutamise ja lugemise ajal võetud lukud.

```
Memory location myglobal (race with conf. 110)
  read@01-simple_rc.c:10 with {lock:mutex1} (conf. 110)
  read@01-simple_rc.c:19 with {lock:mutex2, thread:main!} (conf. 110)
  write@01-simple_rc.c:10 with {lock:mutex1} (conf. 110)
  write@01-simple_rc.c:19 with {lock:mutex2, thread:main!} (conf. 110)

Summary for all memory locations:
        safe:            0
        vulnerable:      0
        unsafe:          1
        ------------------
        total:           1
```

Joonis 2: Näidisväljund

```
generic_nvram.c: error: potential read-write race:
        read by entry point nvram_llseek, generic_nvram.c:54:2
                return file->f_pos;
        write by entry point nvram_llseek, generic_nvram.c:53:2
                file->f_pos = offset;
```

Joonis 3: Analüsaatori Whoop väljund

Üks hiljuti avaldatud andmejookse avastav staatiline analüsaator Whoop analüüsib usaldusväärselt paarikaupa lukuhulkasid [5]. Selle väljund lihtsa

programmi korral on toodud joonisel 3. Goblinti väljund on analüsaatori Whoop väljundiga küllaltki sarnane. Mõlemas tuuakse välja kõik hoiatuse põhjustanud mälukohale tehtud kirjutamised ja lugemised. Analüsaatori Whoop väljundis on lisaks kirja pandud ka koodiread, mis andmejooksu põhjustavad. See võib aidata rida kiiremini üles leida, kuid peamine koodiridade otsimine toimub ikkagi rea numbri järgi, mis on toodud mõlemas väljundis. Samal ajal ei ole analüsaatori Whoop väljundis kirjas, mis muutuja peal andmejooks toimub, see võib pikemate koodiridade korral muuta hoiatuse mõistmise aeganõudvamaks.

Analüsaatori Whoop väljundis ei ole toodud lukuhulkasid ja hoiatuse kindlust. Ühest küljest muudab see väljundi lühemaks ja mugavamaks lugeda, kuid teisest küljest annab Goblinti väljund seega olulist lisainformatsiooni, mis võib vigade parandamise juures kasuks olla. Näiteks käesoleva töö juures on väljastatud lukuhulgad suure tähtsusega ja aitavad informatsiooni oluliselt organiseerida.

Leidub teisigi andmejooksu analüsaatoreid nagu näiteks CoBE [9], Locksmith [15] ja Relay [23]. Neil on sarnasusi analüsaatori Goblint lähenemisega, kuid paraku ei olnud nende analüsaatorite väljundid võrdluste tegemiseks kätte saadavad.

2 Lahenduse leidmine

Tööriista Goblint tulemuste visualiseerimiseks on tarvis teada, mis on analüsaatori Goblint kasutatavuse puudujäägid ja kuidas saaks kasutatavust tõsta. Selleks uuritakse varasemates artiklites kogutud arendajate tagasisidet staatilistele analüsaatoritele ja võrreldakse tööriista Goblint väljundit praegu levinud analüsaatori väljundiga. Nii saab kokku panna tervikpildi hea kasutatavusega analüsaatorist. Lõpuks toob antud peatükk varasemalt kirjutatud teadusartiklite ja eelneva analüüsi põhjal välja ideid, kuidas olemasolevat väljundit parandada saaks.

2.1 Analüsaatori vajalikud omadused

Staatiliste analüsaatorite arendajate jaoks on olulised küsimused analüsaatori usaldusväärsus ja valepositiivsete tulemusete hulk [4]. Need aspektid on olulised ka tarkvara arendaja jaoks, kes analüsaatorit kasutab, kuid lisaks nendele huvitavad praktiseerijat ka vähem formaalsed küsimused. Oluline on lisaks eelmainitule ka analüsaatori tulemuste loetavus, hoiatuste modifitseeritavus, tööriista sobivus arendusprotsessi, koht arendusprotsessis jms [13]. Järelikult on analüsaatori väljund selle sobivuse hindamisel suure kaaluga.

Microsofti töötajate seas tehtud staatiliste analüsaatorite empiirilisest uuringust tuleb välja, et tarkvara arendajaid häirib oluliselt see, kui hoiatavad sõnumid ei ole selged ja kui valepositiivseid tulemusi on liiga palju. Lisaks on oluline analüsaatori modifitseeritavus ja hoiatuste vaigistamise võimalus. Huvitaval kombel peetakse üksikute vigade avastamata jätmist vähem oluliseks [4]. Siit võib järeldada, et akadeemiliste staatiliste analüsaatorite arendajate eesmärgid ei lange kokku analüsaatori kasutajate eesmärkidega. Kui esimesed püüdlevad täielikult usaldusväärse analüsaatori poole, siis teiste jaoks on usaldusväärsus teisejärguline.

Analüsaatori väljundi kokku pakkimine ja vähem olulise info peitmine aitab tõsta tulemuste loetavust ja modifitseeritavust ning vähendab valepositiivsete hoiatuste lugemisele kuluvat aega. Järelikult on kokku pakkimiseks vajalike parameetrite leidmine ja analüsaatori väljundi analüüs olulise tähtsusega.

Veel selgub eelmainitud uuringust, et lõimede vahelise suhtluse vigade leidmist peetakse olulisuselt kolmandaks analüsaatori ülesandeks ja keerukate

probleemide avastamiseks mõeldud analüsaatoreid ollakse valmis pikemaks ajaks (näiteks ööseks) käima jätma [4]. See näitab, et tööriista Goblint vastu võiks arendajatel olla huvi, sellest oleks koodi arendamisel kasu ja see sobituks ka tarkvaraarendamise protsessi öisesse järku.

2.2 Laialt kasutatava analüsaatori väljundi võrdlus Goblintiga

Võrdluseks Goblinti väljundile (vaata joonis 2) saab vaadata ühte viimasel ajal tähelepanu alla sattunud staatilist analüsaatorit, mis on firma Facebook poolt arendatav tööriist Infer [1]. Selle väljund annab vea pealkirja, asukoha, kirjelduse ja lisaks veel kuus nummerdatud rida koodi, et kirjeldatud koodi koht oleks kergelt leitav (vaata joonis 4). Facebookis kasutatakse seda tööriista automaatselt iga kord, kui kood versioonihaldussüsteemi lisatakse ja arendajad lasevad oma koodi Inferil kontrollida enne, kui selle koodibaasi lisavad [3].

```
./Root/Hello.java:27: error: NULL_DEREFERENCE
  object a last assigned on line 25 could be null and is dereferenced at line 27
  25.       Pointers.A a = Pointers.mayReturnNull(rng.nextInt());
  26.       // FIXME: should check for null before calling method()
  27. >     a.method();
  28.     }
  29.
  30.
```

Joonis 4: Analüsaatori Infer väljund

Analüsaatori Infer kasutatavuse eelis tuleneb sellest, et arendajal on koheselt näha, kus kohas analüsaator arvab viga olevat. See annab programmeerijale palju lisainformatsiooni ja kiirendab koodi parandamise protsessi. Peamine erinevus analüsaatorite Goblint ja Infer vahel tuleb tööriistade erinevast tööpõhimõttest. Kuna analüsaator Goblint otsib mitme lõime vahelist andmejooksu, mis on tingitud erinevates kohtades paiknevate koodiridade koosmõjust, siis ei piisa ühele koodireale viitamisest, kuid mida rohkem suudab analüsaator teha arendajale selgeks, kust viga võib pärineda, seda parem. Siit saab järeldada, et Goblinti kasutatavus oleks oluliselt parem, kui väljundis oleks selgemaid vihjeid, kust kohast hakata viga parandama. Näiteks võiks väljund

sisaldada teavet, milline lukk puudu on. Selleks tuleb leida viis, kuidas seda lisainformatsiooni automaatselt luua.

2.3 Lahenduse loogika

Suuremate rakenduste korral võib Goblinti väljund olla väga mahukas ja seetõttu raskesti hallatav. Näiteks kui ühe globaalse muutuja poole pöördutakse mitmetes kohtades ja Goblint avastab, et antud muutuja juures on andmejooks tõenäoline, siis väljastab Goblint kõik pöördumise kohad ja nende pöördumiste juures võetud lukud. Selline suur kogus infot muudab väljundi interpreteerimise keeruliseks ja aeganõudvaks. Üldiselt soovib programmeerija kirjutada korrektset koodi, mis tähendab, et suurem osa sellest infost hoiatuse juures on liigne. Kuidagi peaks proovima selles mahukas väljundis rõhutada seda osa, mis viitab programmi piirkonnale, mis suurema tõenäosusega sisaldab viga.

Kui võrrelda omavahel hoiatuse poolt kuvatud lukkude hulkasid, siis võib vahel näha olla, et üks hulk erineb selgelt teistest. Kuna suurem osa koodist töötab eelduste kohaselt õigesti, siis teistest erinev lukkude hulk mingi mälupöörduse juures annab juba küllaltki selge vihje, et tõenäoliselt on andmejooks põhjustatud just selle mälupöörduse poolt. Väljundi lihtsustamiseks saaks seetõttu programmeerijale kuvada esmalt just tavalisest hälbivaid kohtasid ja alles hiljem soovi korral kuvada kogu tagasiside. Varasemaltki on artiklites kirjeldatud, kuidas programmi koodi põhjal saab luua programmi kohta tõekspidamisi ja seeläbi tuua välja tõekspidamistest hälbivaid kohti kui vigu [7, 11]. Goblinti väljundi juures sobib tõekspidamiseks kombinatsioon mäluaadressi poole pöördumisest ja selle juures võetud lukust. Kuna selline lähenemine on varem osutunud edukaks ja seda saab eelmainitud viisil kasutada ka Goblinti väljundi analüüsi juures, siis on see oluline osa käesoleva töö lahenduse juures.

Sarnaselt artiklis *Bugs as Deviant Behavior* [7] kirjeldatud lähenemisele, saab ka Goblinti väljundi juures järjestada leitud andmejookse. Järjestuses ettepoole tuuakse mälupöördused, mille juures on leitud tõekspidamine ehk luku ja mälupöörduse kombinatsioon. Need andmejooksud saab omakorda järjestada vastavalt mäluaadressi poole pöördumiste ja nendest pöördumistest tõekspidamisele vastavate pöördumiste arvule. Mida rohkem on selliseid mälupöördusi,

kus tõekspidamine kehtib ja mida vähem on selliseid, kus tõekspidamine ei kehti, seda kõrgemal on antud tõekspidamine järjestuses.

Goblinti kvalitatiivse analüüsi käigus selgus, et suure osa hoiatuste korral ei ole ühegi mälupöörduse juures võetud ühtegi lukku. Sellest lähtuvalt saab Goblinti väljundit muuta lühemaks ja selgemaks, kui anda kompaktsem kuju sellisele hoiatusele. Praegu esitatakse lukkudeta hoiatuse korral eraldi ridadel iga mälupöördus eraldi ja koos sellega kuvatakse tühi lukuhulk. Selle asemel saaks kompaktsemas hoiatuses kuvada kõik mälupöördused ja öelda, et nende mälupöörduste juures lukke võetud ei ole.

3 Implementatsioon

See peatükk kirjeldab leitud lahenduse põhjal valminud tööriista, mis modifitseerib Goblinti väljundit. Esimene jaotis selgitab ja põhjendab tehnoloogilisi valikuid. Teine jaotis selgitab näite abil, mida on väljundis muudetud. Kolmas jaotis kirjeldab koodi arhitektuuri ja lahenduse põhjal loodud konkreetset algoritmi, mida on väljundi parandamiseks kasutatud. Neljas jaotis kirjeldab, kust on kood kättesaadav ja kuidas seda jooksutada.

3.1 Tehnoloogilised valikud

Tööriista arendamiseks on kasutatud programmeerimiskeelt Java [8], sest selle objektorienteeritud paradigma võimaldab esitada Goblinti väljundit loomulikul kujul objektidena ja Java virtuaalmasina tõttu saab sama koodi jooksutada erineval riistvaral ja operatsioonisüsteemidel hõlpsalt. Veel on kasutusel Gradle konstruktor tööriist [6], mis võimaldab hallata sõltuvusi ja lihtsustab programmi kasutamist. Lisaks neile on kasutatud Google Guava [14] teeki, mis võimaldab esitada andmeid programmi koodis lühemal ja selgemal kujul. Tööriista arendamise juures on kasutusel programmeerimiskeskkond IntelliJ IDEA [2], sest see annab häid stiili soovitusi, võimaldab mugavat dokumentatsiooni lugemist ja tõstab arendamise efektiivsust, lõpetades alustatud koodiread kiire klahvikombinatsiooni abil.

3.2 Näide paranenud väljundist

All on toodud näide tavalisest Goblinti väljundist ühe lühikese 40-realise programmi korral (vaata joonis 5). Siin on kirjas muutuja asukoht, kus potentsiaalne andmejooks toimub, ja selle all kuvatud kõik kirjutamised ja lugemised, mis antud muutujale on tehtud. Iga kirjutamise ja lugemise juures on kirjas võetud lukud ning praegusel juhul ka regioon [19].

Goblinti väljundis märgitud regioonide vahel on mälu täielikult eraldatud. See tähendab, et kui mingis regioonis on kõikidel sama mäluaadressi poole pöördustel võetud kindel lukk, siis selle regiooni mälupöördused on turvalised ja ei sisalda andmejooksu. See on esimene parandatav aspekt, mis paistab muutmata Goblinti väljundist välja (vaata joonis 5). Nimelt on regioonis

B nii kirjutamisel kui ka lugemisel võetud lukk B. Järelikult ei ole need mälupöördused ohtlikud ja need võib julgelt hoiatusest välja jätta.

```
Memory location (alloc@tests/regression/09-regions/28-list2alloc.c:12).datum
(race with conf. 110)
        read@28-list2alloc.c:27 in {region:B} with {lock:B_mutex} (conf. 110)
        write@28-list2alloc.c:27 in {region:B} with {lock:B_mutex} (conf. 110)
        read@28-list2alloc.c:23 in {region:A} with {lock:A_mutex} (conf. 110)
        write@28-list2alloc.c:23 in {region:A} with {lock:A_mutex} (conf. 110)
        write@28-list2alloc.c:44 in {region:A} with {thread:main!} (conf. 110)
```

Joonis 5: Muutmata Goblinti väljund

Lisaks sellele oleks programmeerijal hoiatus palju loetavam, kui tal oleks varakult näha, mis koodirea tõttu potentsiaalne andmejooks võiks tekkida. Muutmata Goblinti väljundis on näha, et regioonis A on kahel juhul võetud lukk A ja ühel juhul seda võetud ei ole. Nagu kirjeldatud jaotises 2.3, siis eeldades, et programmeerija kirjutab suuremalt jaolt korrektset koodi, saab väita, et suurema tõenäosusega on viga just selles kohas, kus lukku A võetud ei ole.

```
Possible datarace at tests/regression/09-regions/28-list2alloc.c:12
        Probably caused by write@28-list2alloc.c:44 not having lock:A_mutex
        like in read&write@28-list2alloc.c:23
```

Joonis 6: Uus Goblinti väljund

Lisaks eelmainitule saab loetavuse parandamiseks ja lühema väljundi jaoks võtta kokku lugemise ja kirjutamise, mis toimuvad samal real, samas regioonis ja samade lukkudega. Tulemuseks on varasema kuue rea asemel kaks (vaata joonis 6; käesoleva töö vormistuse huvides on teine rida poolitatud ja seetõttu on väljundis näiliselt kolm rida).

3.3 Arhitektuur ja algoritm

Valminud lahendus vajab parameetrina Goblinti väljundit XML kujul (joonisele 5 vastavat XML kuju vaata Lisa A). XML fail loetakse sisse ja selle järgi moodustatakse iga hoiatuse kohta objekt. Igast sellisest objektist on kättesaadav hoiatuse identifikaator, mis on tavaliselt hoiatuse põhjustanud

muutuja asukoht koodis, ja iga mälupöörduse kohta loodud objektid. Igast mälupöörduse objektist on kättesaadavad kasutatud lukud ja võimalusel ka regioon.

Esimeseks väljundi töötlemise sammuks on hoiatuse kohta käivate mälupöörduste regioonide analüüs. Kuna erinevates regioonides paiknevad mälupöördused kasutavad ka eraldiseisvaid mälu osasid, siis saab iga regiooni kohta mälupöörduste turvalisust eraldi hinnata. Seetõttu jagatakse esimese sammuna kõik mälupöördused hoiatuses regiooni järgi gruppideks. Igast grupist tehakse uus hoiatus ja vana hoiatus kustutatakse. Nüüd on ühes hoiatuses ainult ühe regiooni mälupöördused.

Järgmiseks võetakse ühe hoiatuse kõikidest lukuhulkadest ühisosa. Kui ühisosa on tühi, siis peavad tõesti olema antud regiooni kõik mälupöördused hoiatuses kajastatud. Kui ühisosa ei ole tühi, siis järelikult leidub lukk, mis kaitseb antud regiooni kõiki mälupöördusi. See tähendab, et selles regioonis olevad mälupöördused on tegelikult turvalised ja selle hoiatuse saab välja jätta.

Kolmandaks sammuks on esitada kompaktsemal kujul hoiatused, mille mälupöörduste juures ei ole võetud ühtegi lukku. Selle asemel, et tuua eraldi ridadel välja mälupöördus ja tema tühi lukuhulk, öeldakse kompaktsemas väljundis, et hoiatus on põhjustatud mälupöördustest, mille juures pole lukku võetud. Vana ja uue väljundi võrdlus on kujutatud joonistel 7 ja 8.

```
Memory location apm_suspend_waitqueue.lock.__annonCompField18.rlock.owner (race
with conf. 80)
  write@apm-emulation.c:294 in {region:apm_suspend_waitqueue} with {} (conf. 80)
  write@apm-emulation.c:303 in {region:apm_suspend_waitqueue} with {} (conf. 80)
  write@apm-emulation.c:303 in {region:apm_suspend_waitqueue} with {} (conf. 80)
  write@apm-emulation.c:353 in {region:apm_suspend_waitqueue} with {} (conf. 80)
  write@apm-emulation.c:637 in {region:apm_suspend_waitqueue} with {} (conf. 80)
```

Joonis 7: Vana lukkudeta Goblinti hoiatus

```
Possible datarace at apm_suspend_waitqueue.lock.__annonCompField18.rlock.owner
   Detected because the following accesses take no locks: [write@apm-emulation.c:303,
   write@apm-emulation.c:294, write@apm-emulation.c:637, write@apm-emulation.c:353]
```

Joonis 8: Uus lukkudeta Goblinti hoiatus

Seejärel töödeldakse neid hoiatusi, mille juures on lukke võetud. Selleks tuuakse piisava kindluse korral välja tõekspidamised ja rõhutatakse neist hälbivaid

kohti koodis, nagu on kirjeldatud jaotises 2.3. Tõekspidamistest tuuakse välja ainult need, mille tugevus on suurem tõekspidamise lävendist või sellega võrdne. Tõekspidamise tugevuseks on suhe $\frac{t}{n}$, kus t on mälupöörduste arv, kus tõekspidamine kehtib, ja n on kõigi mälupöörduste arv antud hoiatuse juures. Vaikimisi on tõekspidamise lävendiks 0,5. See tähendab, et kui vähemalt pooltel mälupöördustel antud hoiatuse juures kehtib tõekspidamine, siis see tõekspidamine tuuakse ka välja.

Tõekspidamiste välja toomiseks vaadatakse ühe hoiatuse kohta kõik mälupöördused läbi ja leitakse lukk, mis on võetud kõige rohkemate mälupöörduste juures. Antud luku võtmine saabki tõekspidamiseks. Seejärel jagatakse mälupöördused kahte hulka: mälupöördused, mis võtavad antud luku ja mälupöördused, mis seda ei võta. Siis arvutatakse välja tõekspidamise tugevus ja võrreldakse seda tõekspidamise lävendiga, et otsustada, kas antud hoiatus tuua välja või mitte.

Lõpuks järjestatakse tõekspidamise lävendit ületavad hoiatused nende tõekspidamise tugevuse järgi kahanevalt analoogiliselt joonisele 6. Nende hoiatuste järel esitatakse kompaktsemal kujul hoiatused, kus ei ole võetud ühtegi lukku. Kõige lõppu jäetakse hoiatused, mille tõekspidamise tugevus jääb alla tõekspidamise lävendit. Need esitatakse esialgsel kujul.

3.4 Käivitamise juhend

Valminud programmi kood on kättesaadaval Githubis[1].

Tööriist vajab sisendiks Goblinti väljundit XML kujul (joonisele 5 vastavat XML kuju vaata Lisa A). Goblinti käivitamise juhendit saab näha Goblinti Githubi lehel[2]. Selleks, et tööriist Goblint väljastaks tulemused XML kujul, on tarvis anda sellele ette käsurea argument *--sets warnstyle*. Näiteks kui fail, mida soovitakse analüüsida, asub kohalikus kataloogis ja selle nimi on *28-list2alloc.c*, siis käsk, mis analüüsib antud faili ja väljastab tulemuse XML kujul on

```
./goblint --sets warnstyle xml 28-list2alloc.c
```

[1] https://github.com/Karl-Mattias/goblint-output-analyser
[2] https://github.com/goblint/analyzer

Käesoleva töö käigus valminud tööriista käivitamiseks on lihtsaim viis alla laadida käivitatav JAR faili kokku pakitud programm[3]. JAR faili käivitamiseks peab olema installeeritud *Java Runtime Environment*. Kui see on olemas, siis saab programmi käivitada käsurealt käsuga

```
java -jar goblint-output-analyser-0.1.jar <argument>
```

kus <argument>asemele tuleb kirjutada tee Goblinti väljund XML failini.

Sama JAR faili saab luua ka lähtekoodist käsuga

```
gradlew build
```

Sellisel juhul luuakse JAR fail kausta *build/libs*.

[3] `https://s3.eu-west-2.amazonaws.com/bakalauerus/goblint-output-analyser-0.1.jar`

4 Analüüs

Uuenenud väljundi analüüsiks kasutatakse Goblinti jõudlustesti tööriista ja võrreldakse jõudlustesti uut väljundit vanaga. Goblinti jõudlustest on kätte saadav Githubist[4]. Esimeses jaotises on välja toodud kvantitatiivsed erinevused vana ja uue väljundi vahel. Teises jaotises on kirjeldatud kvalitatiivseid erinevusi väljundite vahel.

4.1 Kvantitatiivne analüüs

Kvantitatiivse analüüsi jaoks on kasutatud 25 Goblinti jõudlustesti jaoks välja valitud Linuxi draiverit. Siin on Goblint määratud sooritama ka regiooni ja muutuja analüüse (selleks kasutatakse Goblinti parameetreid *--set ana.activated[+] 'region' --set ana.activated[+] 'var_eq' --set ana.activated[+] 'symb_locks'*).

Esialgses Goblinti väljundis on draiverite analüüsi väljund ligikaudu 16 700 rida ja 1 200 000 tähemärki, parandatud väljundis on ridade arv ligikaudu 9100 ja tähemärke 707 000. Rohkem kui kolmandiku võrra vähenemine nii tähemärkide kui ka ridade arvus demonstreerib olulist edasiminekut loetavuses. Juba puhtalt loetava teksti hulga vähenemise arvelt muutub Goblinti kasutamine efektiivsemaks.

Analüüsitud 25 draiveris leiti 3315 esialgset hoiatust. Väljundi parandamise tulemusena saadi 803 korral esile tuua kindel lukk, mille võtmata jätmine võis suurima tõenäosusega hoiatuse põhjustada ehk selle luku võtmine ületas tõekspidamise lävendi (vaata jaotis 3.3). 2324 korral sai hoiatust muuta kompaktsemaks, sest hoiatust põhjustavad mälupöördused ei olnud võtnud ühtegi lukku. 592 korral jäi hoiatus endisele kujule. Väljundi parandamise tulemusena saadud hoiatuste summa ei kattu esialgsete hoiatuste arvuga, sest osad esialgsed hoiatused on jaotatud regiooni järgi mitmeks (vaata jaotis 3.3).

Kuna vähem kui viiendik hoiatusi jäi samale kujule, siis on väljundi muutus oluline. Ülejäänud 3127 hoiatust võimaldavad programmeerijal ühe lause lugemisega mõista hoiatuse põhjust. Peaaegu veerandil kordadest sai välja

[4] `https://github.com/goblint/bench`

tuua ühe selge luku, mille võtmine võiks hoida ära andmejooksu. Järelikult on tõekspidamisest hälbivate mustrite uurimisest selgelt kasu olnud.

4.2 Kvalitatiivne analüüs

Lisaks puhtalt väiksemale mahule, mida arendajal lugeda tuleb, on uus Goblinti väljund ka informatiivsem. Varem pidi programmeerija uurima Goblinti väljundi kuvatud lukuhulkasid ja leidma suuremahulistest hulkadest ise sobiva lahenduse hoiatusele. Selline tegevus on ühest küljest ajamahukas ja teisest küljest võib viia programmeerija vigadeni. Kui lukkudest valitakse välja mitte kõige levinum või otsustatakse täiesti uus lukk mälupöörduste kaitseks luua, siis see tekitab kehva koodi. Tõekspidamisest hälbiva mustri leidmise ja selle kuvamise abil saab programmeerija koheselt aru, kuidas kõige vähesema lisa koodiga vältida hoiatust.

Vigade parandamine hoiatuste põhjal saaks olla iteratiivne protsess. Kui programmeerija saab esimeste hoiatuste põhjal esimesed vead ära parandada, siis saab uute lukkude põhjal Goblint uue väljundi näol alles jäänud hoiatustest juba täpsema ja kompaktsema tulemuse anda.

```
Possible datarace at apm_user_list[?].queue.event_head
   Probably caused by [read&write@apm-emulation.c:185, write@apm-emulation.c:185]
   not having LOCK:user_list_lock like in [read@apm-emulation.c:193,
   read@apm-emulation.c:185, read@apm-emulation.c:186]
Possible datarace at (struct apm_queue).event_head
   Probably caused by [read&write@apm-emulation.c:185, write@apm-emulation.c:185]
   not having LOCK:user_list_lock like in [read@apm-emulation.c:193,
   read@apm-emulation.c:185, read@apm-emulation.c:186]
```

Joonis 9: Kahel hoiatusel samad mälupöördused

```
Possible dataraces at [apm_user_list[?].queue.event_head,
(struct apm_queue).event_head]
   Probably caused by [read&write@apm-emulation.c:185, write@apm-emulation.c:185]
   not having LOCK:user_list_lock like in [read@apm-emulation.c:193,
   read@apm-emulation.c:185, read@apm-emulation.c:186]
```

Joonis 10: Pakitud, kuid mõlemad mäluaadressid on kuvatud

Kvalitatiivse analüüsi käigus selgus üks viis, kuidas saaks Goblinti väljundit arendada veelgi. Nimelt ilmneb uues väljundis kohti, kus Goblint annab kaks

hoiatust erinevate andmejooksu aadressidega, kuid kõik mälupöörduste aadressid on samad ja sama luku võtmine aitaks lahti saada mõlemast hoiatusest (vaata joonis 9).

Samade mälupöördustega hoiatused tekivad sellepärast, et Goblint jälgib ka draiveri parameetrina antud andmestruktuuride puhul, et kõik mälupöördused oleksid kirjas. Goblint ei pruugi alati teada, millise struktuuriga on tegemist, aga ta võib siiski teada, et selle struktuuri sees olevad andmed on kaitstud selle sama struktuuri sees oleva luku poolt. Selleks kasututakse viitade võrduste analüüsi [20]. Selliste mälupöörduste puhul seostab Goblint mälupöörduse andmestruktuuri tüübiga. Kuivõrd programmis võib esineda ka samade tüüpidega struktuure, siis ei saa välistada olukorda, kus need samad struktuurid on antud teatud meetodile argumendina. Sellisel juhul kuvatakse mainitud struktuur mitu korda.

Selliseid hoiatusi saaks kokku võtta näiteks öeldes, et andmejooks on kahel aadressil, aga põhjustatud samadest mälupöördumistest ja lukkudest (vaata joonis 10). Parem lahendus oleks aga, kui analüsaator väljastaks oma eelistusi mälukohtade täpsuste kohta – nii saaks kasutajale esmajoones anda tema programmis kasutatud muutuja kohta hoiatuse. Kuna need eelistused sõltuvad kasutatud analüüsist, siis Goblinti arendajad ei olnud töö kirjutamise ajal kindlad, kas seda pigem lahendada analüsaatori tasemel või oma eelistusi väljastada XML väljundis.

Kokkuvõte

Goblint on staatiline analüsaator, mis leiab C-keelsetest programmidest andmejookse. Kuigi see on tõenäoliselt parim seda tüüpi analüsaator, ei ole see väga levinud keeruka ja süvenemist nõudva väljundi tõttu. Antud töö eesmärgiks oli staatilise analüsaatori Goblint edasi arendamine, et see sobiks suuremale kasutajaskonnale. Selleks analüüsiti Goblinti kasutatavust ja leiti, et seal on arenguruumi. Staatilise analüsaatori väljundi lihtsus ja mõistetavus on väga olulised aspektid analüsaatori kasutamise juures. Antud töö tulemusena muutus väljund lühemaks ja eraldati juba valmis väljundist lisainformatsiooni. See lisainformatsioon aitab programmeerijal kiiremini aru saada vea olemusest ja annab vihje, kuidas neid vigu parandada.

Varasemates artiklites välja toodud staatilise analüsaatori kasutatavuse nõuete põhjal leiti puudujääke Goblinti kasutatavuse juures ja Goblinti väljundi võrdlemisel teise levinud staatilise analüsaatoriga leiti viise, kuidas tulemusi paremini visualiseerida võiks. Seejärel jõuti analüüsi ja varasemate artiklite tulemusena lahenduse loogika väljatöötamiseni. Et muuta Goblinti kasutamine kiiremaks, mugavamaks ja vähem veaohtlikuks lõppkasutajale, leitakse võimalusel väljundist lukk, mille võtmine aitab hoiatust vältida. Kui kuvada arendajale puuduv lukk ja kohad koodis, mis lukku vajaksid, siis saab programmeerija kiirelt aru, kuidas vigu vältida võiks. Lisaks sellele muudetakse kompaktsemaks hoiatused, mis on põhjustatud lukkudeta mälupöörduste poolt. Selliste hoiatuste korral ei ole otstarbekas kuvada kõikide mälupöörduste juures tühjasid lukuhulkasid, nagu see on esialgses väljundis.

Lahenduse loogika põhjal koostati programm, mis võtab sisendiks Goblinti väljundi XML kujul ja tagastab edasi arendatud väljundi. Lõpuks analüüsiti valminud programmi poolt optimeeritud väljundit võrreldes seda varasema väljundiga. Analüüsist selgus, et uus väljund on kasutaja jaoks nii märksa lühem ja lihtsam lugeda kui ka informatiivsem, andes parandamise ja arendamise vihjeid. Goblinti uuenenud väljund viitab selgelt parandamist vajavatele aspektidele, seega on Goblinti kasutamine mugav ka programmeerijale iseseisvalt.

Viited

[1] Infer | A static analyzer for mobile apps | Infer. fbinfer.com/ (11.05.2017).

[2] IntelliJ IDEA. https://www.jetbrains.com/idea (11.05.2017).

[3] Cristiano Calcagno, Dino Distefano, Jeremy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. Moving Fast with Software Verification. In *NASA Formal Methods*, pages 3–11. Springer, April 2015.

[4] Maria Christakis and Christian Bird. What Developers Want and Need from Program Analysis: An Empirical Study.

[5] Pantazis Deligiannis, Alastair F Donaldson, and Zvonimir Rakamari. Fast and Precise Symbolic Analysis of Concurrency Bugs in Device Drivers.

[6] Hans Dockter and Adam Murdoch. Gradle User Guide Version 3.5, 2017. docs.gradle.org/3.5/userguide/userguide.html (11.05.2017).

[7] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, Benjamin Chelf, Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior. In *Proceedings of the eighteenth ACM symposium on Operating systems principles - SOSP '01*, volume 35, page 57, New York, New York, USA, 2001. ACM Press.

[8] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. The Java® Language Specification, 2015.

[9] Vineet Kahlon, Nishant Sinha, Erik Kruus, and Yun Zhang. Static data race detection for concurrent programs with asynchronous calls. In *ESEC/FSE'09*, pages 13–22. ACM Press, 2009.

[10] William Landi and William. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems*, 1(4):323–337, 12 1992.

[11] Benjamin Livshits, Thomas Zimmermann, Benjamin Livshits, and Thomas Zimmermann. DynaMine. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering - ESEC/FSE-13*, volume 30, page 296, New York, New York, USA, 2005. ACM Press.

[12] Matt Might. What is static analysis? matt.might.net/articles/intro-static-analysis/ (11.05.2017).

[13] David Notkin, Betty H.C. Cheng, Klaus Pohl, Robert IEEE Computer-Šociety., and Institute of Electrical and Electronics Engineers. *2013 35th International Conference on Software Engineering (ICSE) : proceedings : May 18-26, 2013, San Francisco, CA, USA*. IEEE Press, 2013.

[14] Joshua O'Madadhain. The Guava project User Guide, 2016. github.com/google/guava/wiki (11.05.2017).

[15] Polyvios Pratikakis, JeffreyŠ. Foster, and Michael Hicks. LOCKSMITH: Context-sensitive correlation analysis for detecting races. In *PLDI'06*, pages 320–331. ACM Press, 2006.

[16] G. Ramalingam and G. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Transactions on Programming Languages and Systems*, 22(2):416–430, 3 2000.

[17] H. G. Rice. Classes of Recursively Enumerable Sets and Their Decision Problems. *Transactions of the American Mathematical Society*, 74(2):358, 3 1953.

[18] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A Dynamic Data Race Detector for Multithreaded Programs.

[19] Helmut Seidl and Vesal Vojdani. Region analysis for race detection. In *SAS'09*, volume 5673 of *LNCS*, pages 171–187. Springer, 2009.

[20] Helmut Seidl, Vesal Vojdani, and Varmo Vene. A smooth combination of linear and Herbrand equalities for polynomial time must-alias analysis. In *FM'09*, volume 5850 of *LNCS*, pages 644–659. Springer, 2009.

[21] Theo Ungerer, Borut Rob, and Juri Silc. A Survey of Processors with Explicit Multithreading. *ACM Computing Surveys*, 35(1):29–63, 2003.

[22] Vesal Vojdani, Kalmer Apinis, Vootele Rõtov, Helmut Seidl, Varmo Vene, and Ralf Vogler. Static race detection for device drivers: the Goblint approach. *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering - ASE 2016*, pages 391–402, 2016.

[23] Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. RELAY: static race detection on millions of lines of code. In *ESEC/FSE'07*, pages 205–214. ACM Press, 2007.

A Goblinti väljund XML

```
<warnings>
<mem id="(alloc@tests/regression/09-regions/28-list2alloc.c:12).datum"
status="race" conf="110">
  <access type="read" loc="28-list2alloc.c:27" conf="110">
    <partitions>
      <part type="region" id="B" />
    </partitions>
    <protectors>
      <prot type="lock" id="B_mutex" />
      <prot type="lock" id="C_mutex" />
    </protectors>
  </access>
  <access type="write" loc="28-list2alloc.c:27" conf="110">
    <partitions>
      <part type="region" id="B" />
    </partitions>
    <protectors>
      <prot type="lock" id="B_mutex" />
      <prot type="lock" id="D_mutex" />
    </protectors>
  </access>
  <access type="read" loc="28-list2alloc.c:23" conf="110">
    <partitions>
      <part type="region" id="A" />
    </partitions>
    <protectors>
      <prot type="lock" id="A_mutex" />
    </protectors>
  </access>
  <access type="write" loc="28-list2alloc.c:23" conf="110">
    <partitions>
      <part type="region" id="A" />
    </partitions>
    <protectors>
      <prot type="lock" id="A_mutex" />
    </protectors>
  </access>
  <access type="write" loc="28-list2alloc.c:44" conf="110">
    <partitions>
      <part type="region" id="A" />
    </partitions>
    <protectors>
      <prot type="thread" id="main!" />
    </protectors>
  </access>
</mem>
</warnings>
```

Lihtlitsents lõputöö reprodutseerimiseks ja lõputöö üldsusele kättesaadavaks tegemiseks

Mina, Karl-Mattias Tepp,

1. annan Tartu Ülikoolile tasuta loa (lihtlitsentsi) enda loodud teose Staatilise analüsaatori Goblint tulemuste visualiseerimine
mille juhendaja on Vesal Vojdani,

1.1 reprodutseerimiseks säilitamise ja üldsusele kättesaadavaks tegemise eesmärgil, sealhulgas digitaalarhiivi DSpace-is lisamise eesmärgil kuni autoriõiguse kehtivuse tähtaja lõppemiseni;

1.2 üldsusele kättesaadavaks tegemiseks Tartu Ülikooli veebikeskkonna kaudu, sealhulgas digitaalarhiivi DSpace'i kaudu kuni autoriõiguse kehtivuse tähtaja lõppemiseni.

2. olen teadlik, et punktis 1 nimetatud õigused jäävad alles ka autorile.

3. kinnitan, et lihtlitsentsi andmisega ei rikuta teiste isikute intellektuaalomandi ega isikuandmete kaitse seadusest tulenevaid õigusi.

Tartus, 11.05.2017