Tartu Ülikool
Arvutiteaduse instituut
Informaatika õppekava

Epp Haavasalu

Töövahendi CoOpeRace loomine

Bakalaureusetöö (9 EAP)

Juhendaja: Vesal Vojdani, PhD

Tartu 2024

Töövahendi CoOpeRace loomine

Lühikokkuvõte:
Bakalaureusetöö eesmärk on luua töövahend CoOpeRace, mis ühendab verifitseerimiskoostöö põhimõtted andmejooksude tuvastamisega. Töös antakse ülevaade andmejooksu ja koostööverifitseerimise mõistetest, kirjeldatakse loodud töövahendis kasutatud kolme analüsaatorit Goblint, Locksmith ja Relay ning tutvustatakse CoOpeRace' kasutamist.

Võtmesõnad:
Staatiline analüüs, verifikatsioonikoostöö, andmejooks, Goblint, Locksmith, Relay

CESCS: P175, Informaatika, süsteemiteooria

CoOpeRace Tool Development

Abstract:
The objective of this Bachelor's thesis is to create CoOpeRace, a tool that combines the principles of cooperative verification with identifying data races. This work gives an overview of the concepts of data races and cooperative verification, describes the three analyzers used in the created tool - Goblin, Locksmith and Relay - and introduces the usage of CoOpeRace.

Keywords:
Static analysis, cooperative verification, data race, Goblint, Locksmith, Relay

CERCS: P175, Informatics, system theory

Sisukord

Sissejuhatus

Tarkvaraarendus jõuab aasta-aastalt kõrgemate ja kaugemate saavutusteni, maailm meie ümber toetub järjest enam infotehnoloogiale. Programmeerimise laialdase kasutuse ja koodi kirjutamise automatiseerimise tulemusena on tarkvara verifitseerimine saanud oluliseks uurimis- ja arenguvaldkonnaks ning verifitseerimiseks on loodud lugematuid töövahendeid. Need aga on enamasti ehitatud üles tiheda süsteemina, mis teeb uuenduste integreerimise ja komponentide vahetamise keeruliseks [1]. Erinevate tööriistade osad on võimalik ühendada üheks süsteemiks, aga sellest lihtsam lähenemine on kooperatiivne verifikatsioon - tööriistad töötavad koos samal eesmärgil, jäädes ise enamjaolt muutumatuteks [2].

Üks probleem, mida tarkvara verifitseerimisega on võimalik vältida, on andmejooks (ingl *data race*). See on olukord, kus kaks lõime proovivad samaaegselt saada ligi ühele mäluaadressile ja vähemalt üks ligipääsu toimingutest on kirjutamine [3]. Andmejooksude leidmiseks on loodud mitmeid vahendeid, millel on erinevad viisid nii koodi analüüsimiseks kui ka tulemuste esitamiseks. Kaks sellist vahendit on Goblint ja Locksmith.

Töö eesmärk on ühendada verifitseerimiskoostöö põhimõtted andmejooksude tuvastamisega ja kirjutada programm, mis paneb Goblinti ja Locksmithi koos töötama ning esitab kasutajale lihtsasti loetavas formaadis mõlema analüsaatori väljundi. Selle jaoks on vaja mõista valitud analüsaatorite tulemusi ja leida viis nendest olulise info leidmiseks ja selle info esitamiseks kasutajale. Lisaks on oluline teha tulevikus programmi uute analüsaatorite lisamine mugavaks, et analüsaatorite arendajad saaksid soovi korral võrrelda enda töövahendi väljundeid teistega ja kasutajad saaksid lisada teisi analüsaatoreid veel usaldusväärsemate tulemuste jaoks.

Bakalaureusetöö on jaotatud kolmeks osaks. Esmalt antakse ülevaade tarkvara verifitseerimisest ja staatilisest analüüsist ning tutvustatakse programmis kasutatavaid analüsaatoreid. Teises osas kirjeldatakse tööprotsessi ja valminud tööriista ülesehitust. Kolmas osa on ülevaade valminud tööst ja võimalused selle tulevikus edasi arendamiseks.

1. Mõisted ja terminid

Andmejooks (ingl *data race*) on olukord, kus kaks või enam lõime proovivad samaaegselt saada ligi ühele mäluaadressile ja vähemalt üks ligipääsu toimingutest on kirjutamine [3].

Tarkvara verifitseerimine (ingl *software verification*) on tarkvaraenduses protsess, kus kontrollitakse, kas tarkvara vastab antud nõuetele.

Verifitseerimiskoostöö (ingl *cooperative verification*) on lähenemine verifitseerimisele, mille eesmärk on panna verifitseerimistööriistad tööle ühe eesmärgi nimel, jättes need ise enamjaolt puutumata [2].

Lõim (ingl *thread*) on programmiosa, mida saab täita sõltumatult ja paralleelselt teiste programmiosadega, kuid jagab ressursse oma loojaga [4].

Muteks (ingl *mutex, mutually exclusive object*) või **lukk** (ingl *lock*) on vastastikku välistav objekt, lukustusvahend, mis reguleerib ressursi jagamist eri programmilõimede vahel ja ressurssi vabastada saab ainult omaniklõim [5].

Viit (ingl *pointer*) on andmeelement, mis näitab teise andmeelemendi asukohta [5].

Vootundetu (ingl *flow-insensitive*) analüüs akumuleerib üle terve programmi kehtivaid väiteid [6].

Vootundlik (ingl *flow-sensitive*) analüüs arvestab programmi juhtimisvooga [6].

Rajatundlik (ingl *path-sensitive*) analüüs arvestab kõiki programmi täitmisradasid eraldi [6].

Lõim-modulaarne analüüs (ingl *thread-modular analysis*) analüüsib lõimesid eraldi ülejäänud süsteemist arvestades nende omavahelist mõju vootundetult.

Täisarvu ületäitumine (ingl *integer overflow*) on vorminguga määratust pikema täisarvu teke tehte tulemusena [5]

Tüübi- ja efektisüsteem (ingl *type and effect system*) loogiline süsteem, mis lubab lisaks programmi terminite tulemusväärtuste tüüpideks liigitamisele kirjeldada nende täitmisel toimuva kõrvalmõju [7].

Surnud kood (ingl *deadcode*) on kompilaatori kontektis lähtekoodi mis tahes osa, mille tulemusi programm kunagi ei kasuta [8], aga verifitseerimise kontekstis on surnud koodi turvanõrkus (CWE-561) all mõeldud koodi, mida kunagi ei täideta [9].

Regulaaravaldis (ingl *regex, regular expression*) on tava- ja erisümbolitest koosnev avaldis , mis võimaldab luua mustreid, mis aitavad teksti sobitada, leida ja hallata [8].

2. Taust

Selles peatükis antakse ülevaade lõputöö teoreetilisest taustast. Täpsemalt selgitatakse tarkvara verifitseerimise ja programmide staatilise analüüsimise olemusi. Samuti kirjeldatakse töö loomisel kasutatud analüsaatoreid ehk analüüsi sooritamise vahendeid: Goblint, Locksmith ja Relay.

2.1 Tarkvara verifitseerimine

Tarkvara verifitseerimise kirjeldamisel toetun Dirk Beyeri ja Heike Wehrheimi ülevaateartiklile [2].

Tarkvara verifitseerimine (ingl *software verification*) on oluline osa tarkvaraarenduse protsessist, mis aitab kindlustada tarkvara turvalisuse ja nõuetekohase toimimise. Tarkvara verifitseerimiseks leidub tänapäeval suur valik töövahendeid. Need võivad varieeruda nii skaleeritavuse, täpsuse, kui ka programmeerimiskeelte funktsioonide käsitluse poolest. Täpse tulemuse saavutamiseks kombineerivad mitmed tööriistad erinevaid lähenemisviise ühtsesse raamistikku. Seda enamasti kas tihedalt, mis nõuab uute tööriistade rakendamist iga uue integreeritud tehnika jaoks, või lõdvalt, mis tähendab, et lähenemisviiside vahel puudub kommunikatsioon ja tulemuse saab kokku panna juba valmis olevatest osadest. Lõputöö raames loodud töövahend nimega CoOpeRace kasutab nende kahe meetodi keskteed - verifitseerimiskoostööd (ingl *cooperative verification*).

Beyeri ja Wehrheimi sõnul on verifitseerimiskoostöö eesmärk panna tööriistad tööle ühe eesmärgi nimel, jättes need ise enamjaolt puutumata. Analüsaatorid suhtlevad omavahel, aga uue tehnika integreerimiseks on vaja implementeerida ainult meetod selle tehnika tulemustest arusaamiseks, et neid oleks võimalik võrrelda teiste verifitseerijate väljunditega.

Lisaks tulemuste väljatöötamisele on vajalik ka nende optimaalne esitamine. Esitluse formaat sõltub sellest, kas tulemusi loeb inimene või masin, aga mõlema puhul on oluline väljastada ka detailsemat infot otsusele jõudmise kohta. See on ka CoOpeRace'i lõppeesmärk, keskendudes just inimesele vajaliku info edasi andmisele.

2.2 Staatiline programmianalüüs ja andmejooks

Programmi kirjutades on lihtne teha vigu nii teadmatuse kui ka tähelepanematuse tõttu. Arenduskeskkonnad toovad nähtavamad vead kasutajale automaatselt esile ja tihti pakuvad kohe ka lahenduse. Enamik koodis leiduvaid turvaauke tuleb aga välja vaid kindlates tingimustes, mistõttu võib neid ilmneda isegi aastaid pärast programmi valmimist [10].

Ühe sellise turvanõrkusena on välja toodud andmejooks (ingl *data race*) [3]. See esineb juhul, kui kaks või enam lõime proovivad ilma sünkroniseeriva operatsiooni juhtimiseta saada ligi ühele mäluaadressile ja vähemalt üks ligipääsu toimingutest on kirjutamine. Andmejooks võib tekitada programmi ettearvamatut käitumist ja andmete kadu või riknemist, mistõttu see on üks tõsisemaid mitmelõimeliste programmide defekte [3]. Üks viis andmejooksu tuvastamiseks on staatiline koodianalüüs.

Staatilise analüüsi eesmärk on koodi kompileerimise ajal koguda sellest semantilist informatsiooni ja tunda ära võimalikud turvaaugud. Programmi käitumise ennustused on tihti vaid hinnangud, kaugel kindlatest otsustest. See tähendab, et staatilise analüüsi põhiolemus on hinnanguliste kuid usaldusväärsete tulemuste arvutamine. Kuna analüüsi teostatakse väljaspool käitusaega, on võimalik alustada probleemide otsimist ja parandamist enne kui programm on valmis [3, 11].

Andmejooksude staatiline tuvastamine toimub nende välistamise kaudu - kui välistada ei saa, siis on olemas potentsiaalne oht. Esmalt on vaja teha kindlaks, kas lõimed kirjutavad samale mäluaadressile. See ei ole lihtne ülesanne, kuna viitade (ingl *pointer*) informatsiooni tuleb jälgida igas võimalikus kontekstis, mis pole suuremates programmides skaleeritav [9]. Teiseks tuleb kontrollida, kas kaks lõime tegutsevad sellel mäluaadressil samaaegselt. See ei saa juhtuda, kui on olemas automaatsed operatsioonid, mis kontrollivad lõimede tegevust või kui üks lõim tegutseb selgelt enne või pärast teist, näiteks tänu ajalisele barjäärile. Samal ajal ühel mäluaadressil tegutsemist ennetab ka see, kui lõimedel on jagatud mäluaadressi jaoks vähemalt üks muteks (ingl *mutex*), mida hoitakse antud mäluaadressile ligi pääsedes.

Staatilise analüüsiga andmejooksude tuvastamiseks tuleb kindlaks teha, et iga jagatud mäluaadressi jaoks leidub vähemalt üks muteks, mida hoitakse samal ajal, kui lõim antud mäluaadressile ligi pääseb. Mitu lõime ei saa samaaegselt hoida ühte muteksit, mis välistab andmejooksude tekkimise võimaluse [9].

2.3 Lõputöös kasutatud analüsaatorid

CoOpeRace kasutab koodi verifitseerimiseks kolme staatilise analüüsi tööriista: Goblint, Locksmith ja Relay. Paljude teiste andmejooksude leidmise tööriistade eesmärk on leida üks viga ja seejärel näidata kasutajale, kuidas programmi täitmine võib selleni jõuda. Sellist lähenemist võib CoOpeRace' tulevikus integreerida. Lõputööks valitud analüsaatorid aga keskenduvad programmi korrektsuse tõestamisele. Need väljastavad kõikide potentsiaalsete ohtude kohta informatsiooni ja see annab võimaluse tulemusi omavahel lihtsal viisil võrrelda.

Kuigi suuremas plaanis on CoOpeRace mõeldud kui üldine verifitseerimiskoostöövahend, siis on üks töö eesmärkidest aidata arendajatel võrrelda enda analüsaatorite tulemusi teiste omadega. Goblint oli 2024. aasta rahvusvahelisel verifitseerimisvõistlusel SV-COMP [12] parim andmejooksude analüsaator ja on lõputöö loomise hetkel aktiivses arenduses, mistõttu keskendub töö just Goblinti arendajate aitamisele.

Goblint [13, 14] on Müncheni Tehnikaülikooli ja Tartu Ülikooli koostöös valminud staatiline programmianalüsaator, mida arendatakse ja edendatakse ka praegu. Kuigi Goblintil on nüüdseks mitmeid kasulikke omadusi, oli see algselt andmejooksude leidmise raamistik ja on peamiselt mõeldud mitmelõimeliste C-keelsete programmide analüüsimiseks.

Goblint baseerub Patrick ja Radia Cousot' esitatud Abstraktse Interpretatsiooni kontseptsioonil [15]. See on rajatundlik (ingl *path-sensitive*) ja viib läbi lõim-modulaarse (ingl *thread-modular*) analüüsi arvutades ligikaudselt kõik võimalikud lõime põimumised, kasutades jagatud andmete globaalseid muutujaid. Lisaks andmejooksude tuvastamisele suudab Goblint kontrollida, kas teatud väite kehtivus on antud programmipunktis garanteeritud või kas teatud programmipunkt on kättesaamatu ja kinnitada, et programmis pole täisarvude ületäitmisi (ingl *integer overflow*).

Kuna Goblint pigem ülehindab potentsiaalseid turvaohtusid, võib positiivse tulemuse korral olla kindel, et kood on korrektne. See omadus võib aga paratamatult põhjustada mitmeid valehoiatusi [16].

Locksmith [17, 14] on analüsaator, mis töötab tüübi- ja efektisüsteemi (ingl *type and effect system*) rakendamisega, mis sisuliselt määrab koodi elementidele tüübid ja jälgib nende koostööd. Locksmith seostab juurdepääsud andmetele selle juurdepääsu ajal hoitud muteksitega. Antud seos pannakse kirja võrratusena. Kuna kirjutamine võib toimuda läbi viitade, on võrratustes ka seosed viitade ja nende võimaliku sisu vahel. Lahendades võrratuse süsteemi saab leida, kas muutujatel on ühine lukk. Programmide keerukusega toimetulekuks kasutab Locksmith nii vootundlikke (ingl *flow-sensitive*) kui ka -tundetuid tehnikaid. Vootundlik koodianalüüs arvestab programmi täitmise järjekorraga, vootundetu aga kõigi võimalike programmi olekutega olenemata täitmisjärjekorrast.

Relay [18, 14] kasutab suhtelise lukuhulga (ingl *relative lockset*) kontseptsiooni, et kirjeldada lukkude hulkades funtksioonide sisenemiskohtade suhtes toimuvaid muutusi. Koos juurdepääsetavate mäluaadresside kogumisega annab see tehnika võimaluse teha kokkuvõtte iga funktsiooni mõjust andmetele. Iga funktsiooni jaoks tehakse kolm analüüsi: sümboolne funktsiooni läbiviimine, suhtelise lukuhulga analüüs ja valvega juurdepääsu analüüs. Sümboolne täitmine jälgib muutuvaid väärtusi funktsiooni parameetrite suhtes, suhtelise lukuhulga analüüs selgitab välja, milliseid mutekseid kasutatakse ja lõpuks otsitakse konflikte valvatud juurdepääsudes, et väljastada kasutajale hoiatused.

3. Valminud lahendus

Lõputöö eemärk oli luua veebipõhine rakendus CoOpeRace, mis kombineeriks kolm staatilist koodianalüsaatorit ja lihtsustaks nende analüsaatorite väljundite lugemist.
Kasutaja laeb üles C-keelse koodiga faili, mille sisu analüüsitakse ja selle analüüsi tulemused esitatakse kasutajale mugavalt loetavas formaadis. Esmalt saab kasutaja tabeli, milles on välja toodud sisestatud koodi read, kus on potentsiaalne andmejooksu oht. Seejärel saab kasutaja soovi korral lugeda, mida iga analüsaator täpsemalt selle spetsiifilise rea kohta väitis.
Programm on kirjutatud programmeerimiskeeles Python ja analüsaatorite tulemustest vajaliku informatsiooni leidmiseks on kasutatud regulaaravaldisi (ingl *regular expression*). Veebirakenduse loomiseks on kasutatud aastal 2011 loodud veebiraamistikku Flask. Flask valiti antud töö loomiseks, kuna 2023. aasta Stack Overflw küsitluse andmetel on see maailmas 15 populaarseima raamistiku hulgas [19] ja varasemad kogemused Flaski kasutamisega on olnud positiivsed.

3.1 Tööprotsess

Rakenduse loomise protsessi alguses oli selles kasutusel ainult kaks kolmest analüsaatorist: Goblint ja Locksmith. Nende kahe väljundite baasil oli vaja otsustada, milline informatsioon on kasutajale oluline ning milliste omaduste baasil on võimalik analüsaatorite tulemusi võrrelda. Rakenduse algversioonides olid Locksmithi ja Goblinti väljunditest olulise leidmiseks eraldiseisvad funktsioonid, loodud ainult ühe kindla vormistusega tekstist informatsiooni leidmiseks.

Lisades koodi analüüsimiseks ka Relay muutsin rakenduse ülesehituse universaalsemaks. Iga analüsaatori käivitamiseks ja tulemuste analüüsimiseks on ühised funktsioonid, mis vajavad igale analüsaatorile omaseid muutujaid. Selle tulemusena on loodud rakenduse koodi mõistmine ja tulevikus uute analüsaatorite lisamine muudetud lihtsamaks.

3.2 Analüsaatorite väljundid

Andmejooksu analüsaatorite väljunditel puudub standardne vormistus ja kahjuks puudub Goblintil, Locksmithil ja Relay-l ülevaatlik dokumentatsioon, mis teeb analüüsi tulemustest aru saamise keerukaks.

Selle peatüki eesmärk on selgitada väljundite sisu, kasutades näidisfaili "c_fail.c" (joonis 1), mida on analüüsitud Goblinti, Locksmithi ja Relay poolt. Peatükid 3.2.1 - 3.2.3 kirjeldavad iga analüsaatori väljundit. Väljundite joonistelt on eemaldatud osad, mis ei ole seotud andmejooksudega, et vältida informatsiooni üleküllust.

```
3  |  int myglobal;
4  |  pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
5  |  pthread_mutex_t mutex2 = PTHREAD_MUTEX_INITIALIZER;
6  |
7  |  void *t_fun(void *arg) {
8  |    pthread_mutex_lock(&mutex1);
9  |    myglobal = 2;
10 |    pthread_mutex_unlock(&mutex1);
11 |    return NULL;
12 |  }
13 |  int main(void) {
14 |    pthread_t id;
15 |    int x;
16 |    pthread_create(&id, NULL, t_fun, NULL);
17 |    pthread_mutex_lock(&mutex2);
18 |    x = myglobal + 5;
19 |    pthread_mutex_unlock(&mutex2);
20 |    pthread_join (id, NULL);
21 |    return 0;
22 |  }
```

Joonis 1. *Faili "c_fail.c" sisu*

Antud kood defineerib real 4 jagatud muutuja **myglobal** ning initsialiseerib ridadel 4 ja 5 kaks muteksit, **mutex1** ja **mutex2**. Funktsioon **t_fun**, defineeritud real 7, hoiab muteksit **mutex1**, määrab muutuja **myglobal** väärtuseks 2 ja seejärel vabastab luku. Põhifunktsioon (**main**), defineeritud real 13, loob kõigepealt real 16 uue lõime, kus täidetakse funktsiooni **t_fun** paralleelselt põhifunktsiooniga. Siis määrab real 18 muutuja **x** väärtuseks globaalse muutuja **myglobal** väärtuse, hoides samal ajal muteksit **mutex2**. Sellel programmil on kalduvus andmejooksule, kuna kaks lõime pääsevad samaaegselt ligi globaalsele muutujale **myglobal**.

3.2.1 Goblint

Goblinti väljundis on informatsiooni nii andmejooksude kui ka teiste probleemide kohta, mis koodis võib leiduda. Näiteks toob Goblint välja defineerimata muutujad ja funktsioonid, kehtetud või ebatäpsed väljendused ja surnud koodi (ingl *deadcode*). Joonisel 2 on Goblinti "c_fail.c" analüüsi tulemus.

```
5  | [Warning][Race] Memory location myglobal@c_fail.c:3 (race
     with conf. 110):
6  |   write with [mhp:{tid=[main, t_fun@c_fail.c:16]},lock:
       {mutex1}, thread:[main, t_fun@c_fail.c:16]] (conf. 110)
       (c_fail.c:9)
7  |   read with [mhp:{tid=[main]; created={[main,
       t_fun@c_fail.c:16]}}, lock:{mutex2}, thread:[main]] (conf.
       110) (c_fail.c:18)
8  | [Info][Race] Memory locations race summary:
9  |   safe: 0
10 |   vulnerable: 0
11 |   unsafe: 1
12 |   total: 1
```

Joonis 2. *Goblinti analüüsi väljund.*

Väljundi 4. rida algab andmejooksu hoiatusega "[Warning][Race]". Seejärel on selgitus, et antud andmejooks on seotud globaalse muutujaga **myglobal**, mis algselt defineeriti faili "c_fail.c" real 3. Rida 6 algusega "**write with**" annab informatsiooni asukoha kohta koodis, kus kirjutatakse jagatud mäluaadressile ja rida 7 algusega "**read with**" asukoha kohta, kus jagatud mäluaadressilt loetakse. "**mhp**" (*may happen in parallel*, est võib juhtuda paralleelselt) sisaldab informatsiooni, mida analüsaator on korjanud, et otsustada, kas antud tegevus toimub mõne teisega paralleelselt. "**mhp**" komponendid on "**tid**" (*thread identifier*, est lõime identifikaator) ja "**created**" (est loodud). "**tid**" kirjeldab lõime, milles tegevus toimub. Staatiliselt eristatakse lõimesid nende loomise ajaloo järgi.

Näiteks real 6 on kirjas, et kirjutamist teostas faili 16. real põhimeetodis ehk **main** meetodis loodud funktsiooni **t_fun** täitev lõim. "**created**" viitab lõimedele, mida tegevuse ajaks antud lõim on loonud. Real 7 on lugemise ajaks **main** meetod juba loonud teise lõime, kus teostatakse kirjutamist ning sellest järeldab Goblint, et lugemine ja kirjutamine võivad toimuda samaaegselt. "**lock**" on lukk ehk muteks, mida lõim kirjutades või lugedes parajasti hoiab, antud kontekstis vastavalt **mutex1** või **mutex2.** "**thread**" on lõim, kus tegevus toimub.

"**conf.110**" väljendab kindlust, et kirjutati antud muutujale. Hetkel on see arv kõrge, kuna kirjutati otse muutujasse, aga kaudsemal kirjutamisel muutub see madalamaks. Rea lõpus on välja toodud fail ja faili rida, kus mäluaadressile kirjutamine või sellelt lugemine toimub. Antud väljundis on mõlemad tegevused failis "c_fail.c", kirjutamine toimub real 9 ning lugemine real 18. Väljundi lõpus annab Goblint ülevaate kõigist andmejooksu hoiatustest: turvalisi mäluaadresse, mida kood kasutas on 0, haavatavaid 0, ebaturvalisi 1 ja jagatud mäluaadresse kokku 1.

3.2.2 Locksmith

Sarnaselt Goblintile annab ka Locksmithi väljund (joonis 3) informatsiooni rohkemate probleemide kohta kui andmejooksud. Lisaks on väljundis kirjas kui kaua analüüsimise protsess aega võttis.

```
8  | Warning: Possible data race: &myglobal:c_fail.c:3 is not
```

```
    protected!
9   |  references:
10  |   dereference of &myglobal:c_fail.c:3 at c_fail.c:9
11  |     &myglobal:c_fail.c:3
12  |   locks acquired:
13  |     concrete mutex1:c_fail.c:4
14  |     mutex1:c_fail.c:4
15  |     mutex1:c_fail.c:4
16  |   in: main at c_fail.c:13 -> c_fail.c:16
17  |
18  |   dereference of &myglobal:c_fail.c:3 at c_fail.c:18
19  |     &myglobal:c_fail.c:3
20  |   locks acquired:
21  |     concrete mutex2:c_fail.c:5
22  |     mutex2:c_fail.c:5
23  |     mutex2:c_fail.c:5
24  |   in: main at c_fail.c:13
```

Joonis 3. *Locksmithi analüüsi tulemus.*

Väljundi real 8 on võimaliku andmejooksu hoiatus: muutuja **myglobal**, mis deklareeriti faili "c_fail.c" real 3, ei ole kaitstud. Viiteid (ingl *dereference*) antud muutujale on kaks. Esimene on faili "c_fail.c" real 9. Muutujale **myglobal** ligipääsedes hoiab see lõim, kus ligipääsemine toimub, muteksit **mutex1**, mis defineeriti analüüsitud faili real 4. Väljundi rida 16 ütleb, et antud viitamine algas faili 13. real defineeritud põhimeetodis ehk **main** meetodis. Noolega on viidatud, et selles kutsuti välja uus funtksioon, mis on defineeritud real 16 ja selles toimubki viitamine käesolevale muutujale **myglobal**.

Teise viite informatsioon algab väljundi real 18 ning sarnaneb esimesele, aga selle puhul toimub viitamine real 18, lõim hoiab muteksit **mutex2**, mis defineeriti "c_fail.c" real 5 ja viitamine toimub põhimeetodis.

3.2.3 Relay

Relay väljud (joonis 4) on võrreldes eelneva kahe analüsaatroi väljundiga märgatavalt mahukam. Väljundi alguses on informatsioon sisestatud faili sisu dubreerimise kohta, seejärel loodud duplikaadi kompileerimise ning kompileerimisel tekkinud probleemide kohta. Kõige lõpuks viib Relay läbi analüüsi ning esitab selle tulemused.

```
192 | Possible race between access to:
193 |
194 | myglobal @ c_fail.c:3 and
195 | myglobal @ c_fail.c:3
196 |      Cluster ID: 0
197 |      Escapes? true / true
198 |      Accessed at locs:
199 |      [c_fail.c:9:(10.f)] and
200 |      [c_fail.c:18:(13.f)]
201 |
202 |      Confidence: Syntactic
203 |
204 | LS for 1st access:
205 | L+ = {mutex1#g:8} (1)
206 | LS for 2nd access:
207 | L+ = {mutex2#g:9} (1)
208 |      Th. 1 spawned: c_fail.c:16 52225 w/ func: t_fun
209 |      Th. 2 spawned: #entry_point:0 0 w/ func: main
```

Joonis 4. *Relay analüüsi tulemus*

Väljundi read 192-195 hoiatavad, et kahes sisestatud koodi asukohas on juurdepääs globaalsele muutujale **myglobal**, mis defineeriti faili “c_fail.c” real 3. Real 196 on kindlale kontekstile või tingimuste hulgale antud numbriline identifikaator, mis aitab programmi käitumist paremini jälgida. Rida 197 väidab, et mõlemad juurdepääsud muutujale **myglobal** toimuvad programmi täitmisvoo sees. Muutujale on juurdepääs kahes asukohas: faili “c_fail.c” ridadel 9 ja 18. Väljundi rida 202 ütleb, et Relay väite usaldusväärsus, et ridadel 9

ja 18 toimunud mälupöördumised puudutasid sama mäluaadressi on süntaktiline (ingl *synctactic*), sest mälupöördumised toimusid süntaktiliselt identse avaldise, konkreetse muutuja **myglobal** kaudu.

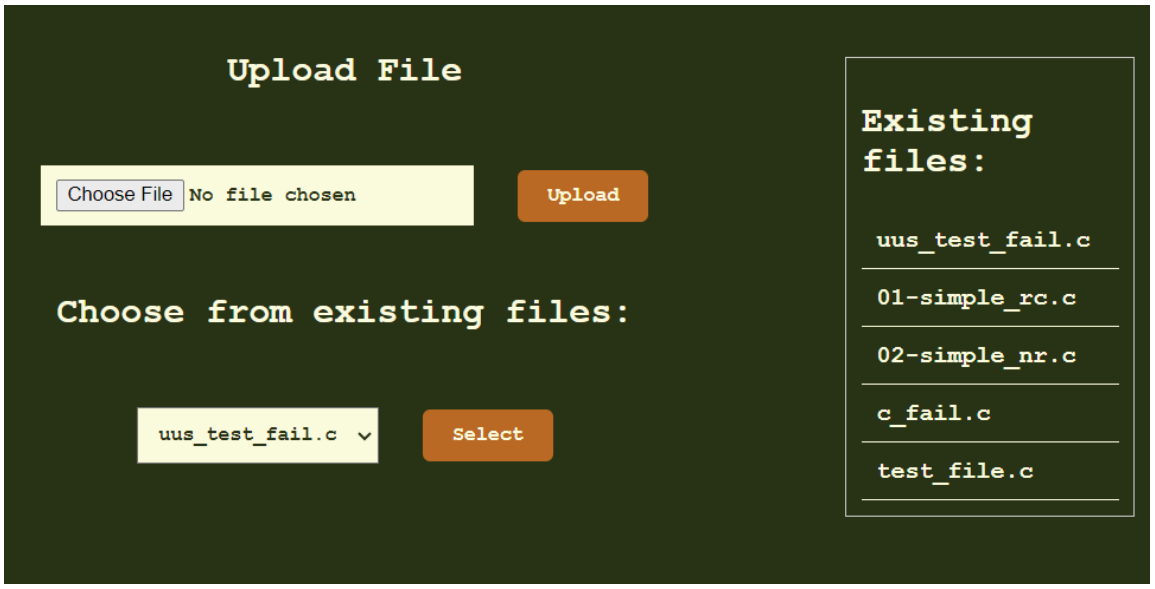
Väljundi ridadel 204-207 on informatsioon juurdepääsudel hoitud muteksite hulkade (LS - *Lock Set*) kohta. Esimene lõim hoiab juurdepääsul muteksit **mutex1** ja teine lõim muteksit **mutex2**. Väljundi read 208-209 ütlevad, et esimene lõim on loodud failis "c_fail.c" 16. real funktsiooniga **t_fun** ja teine lõim programmi käivitumisel põhimeetodiga ehk **main** meetodiga.

3.3 Veebirakendus CoOpeRace

Veebirakenduse CoOpeRace kasutamise eesmärk võib olla nii koodi analüüsimine kui ühe analüsaatori tulemuste võrdlemine teistega. Rakenduse loomise eesmärkideks oli, et kasutaja saaks olulisema info analüüsi tulemustest, saaks laadida üles uusi faile või vaadata eelnevalt üles laetud failide analüüse ja saaks soovi korral võimalikult mugavalt lisada rakendusse ka uue analüsaatori.

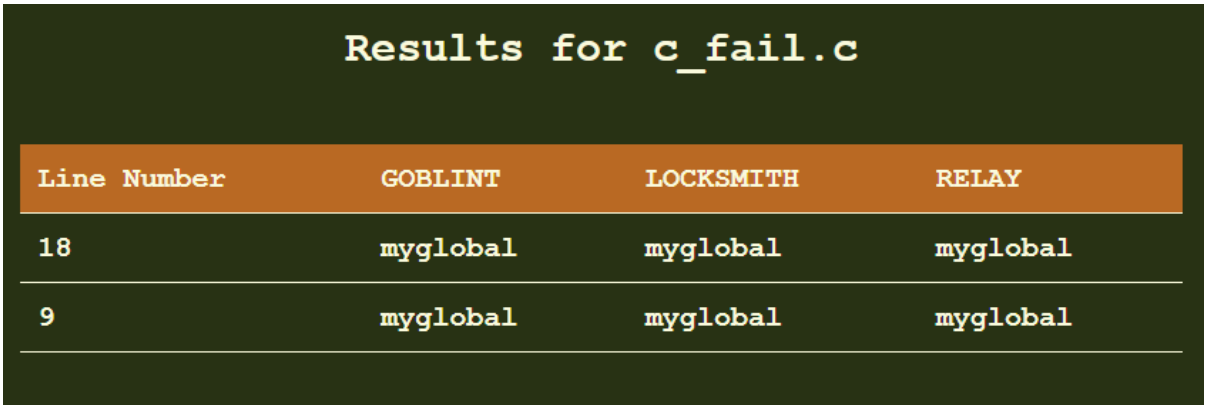
3.3.1 CoOpeRace' kasutamine

Rakenduse avalehel (joonis 4) on kasutajal võimalus laadida üles uus fail või valida etteantud nimekirjast varasemalt salvestatud fail.



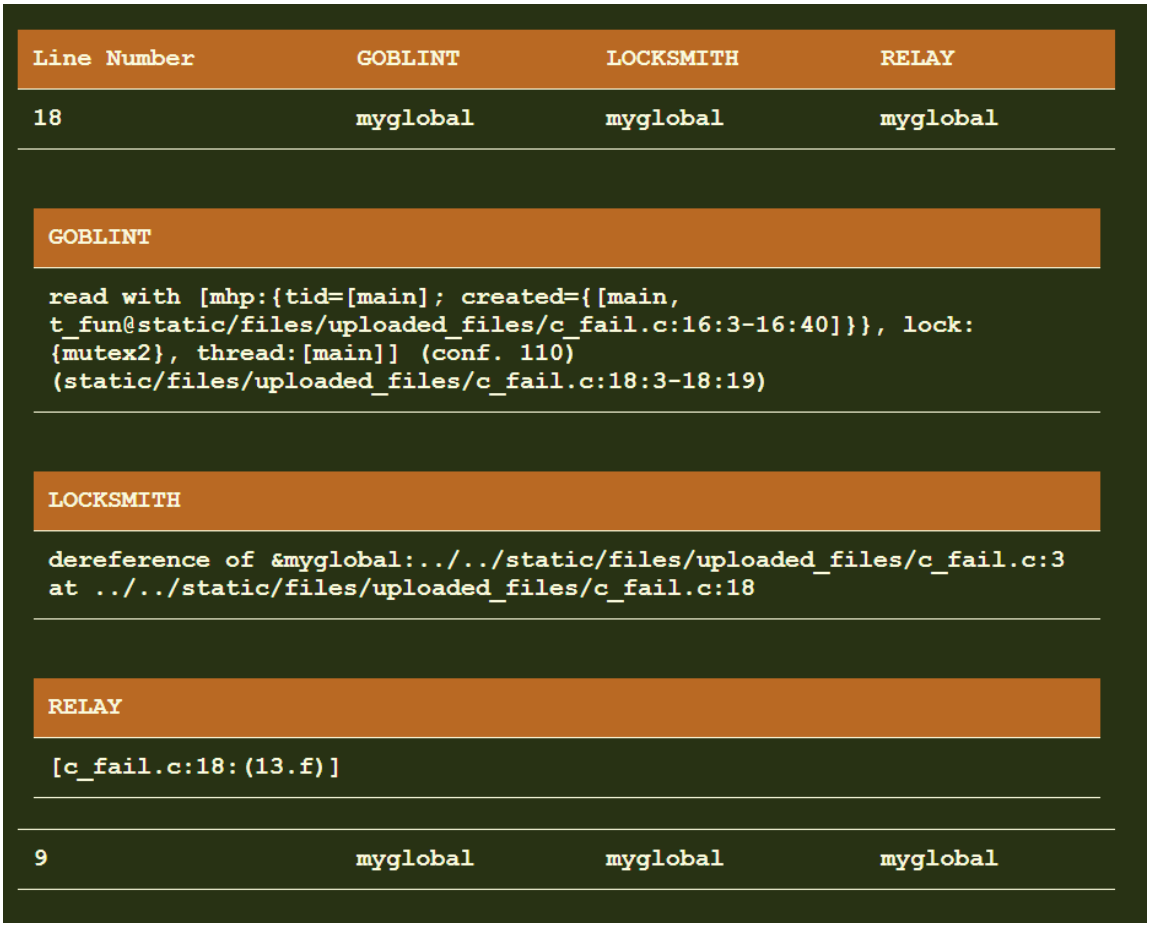
Joonis 4. *CoOpeRace' avaleht.*

Olemasoleva faili valimisel otsib programm varasemalt salvestatud väljundist tulemused koodi uuesti analüüsimata. Olemasolevate failide sisu saab näha tabelis valitud failile klikkides. Kui aga kasutaja laeb üles uue faili, on kaks varianti. Juhul, kui üles laetud faili nimega faili veel ei eksisteeri, analüüsitakse sisestatud koodi ning analüsaatorite väljundid salvestatakse uutesse failidesse. Kui samanimeline fail on juba olemas, küsitakse kasutajalt, kas ta soovib selle faili üle kirjutada, teha läbi analüüsiprotsessi ja salvestada uued tulemused vanade asemel, või soovib ta üles laetud failist loobuda ja vaadata eelnevalt salvestatud faili analüüsi tulemusi. Analüüsi tulemused on iga variandi puhul esitatud tabelina (joonis 5), kus on näha, milliseid rea numbreid on väljundites mainitud ja võimaluse korral ka millise muutujaga sellel real tekkiv andmejooks on seotud.



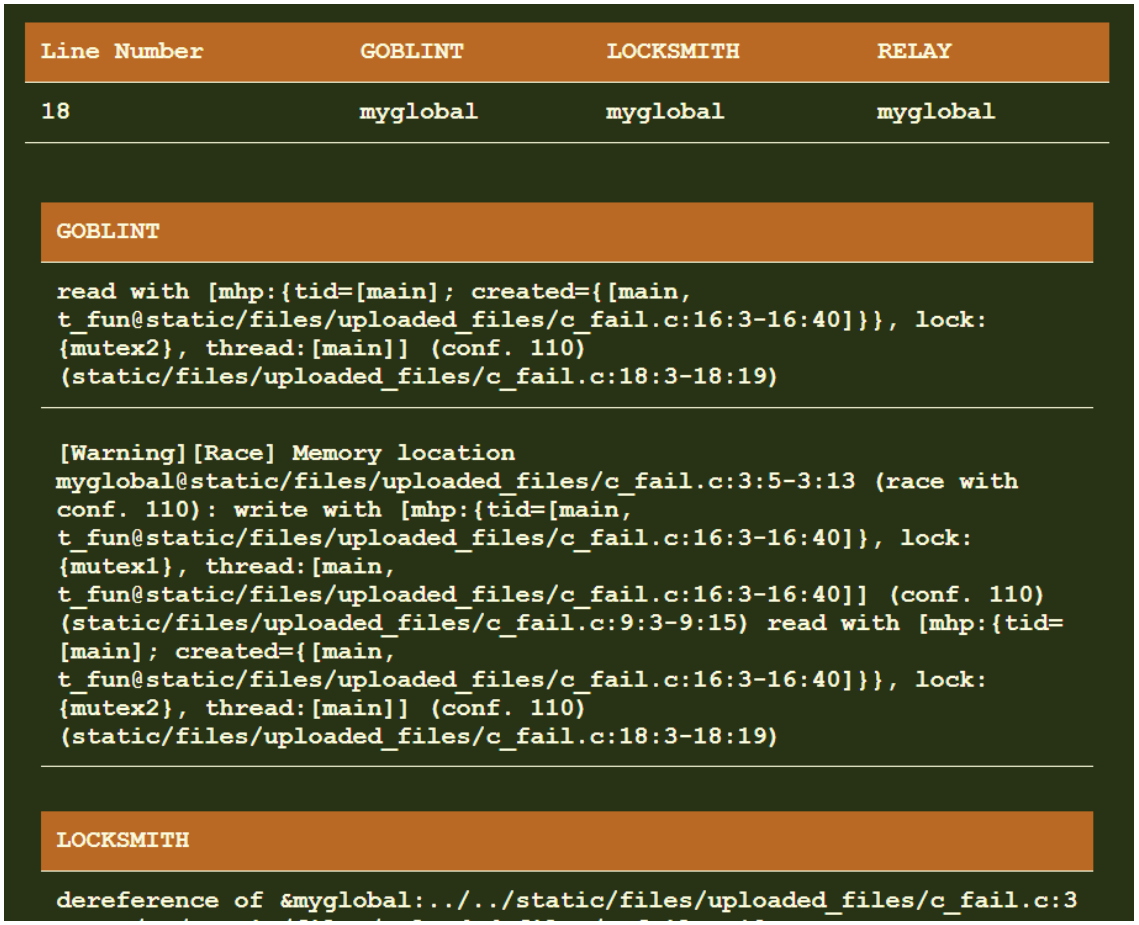
Joonis 5. *Tulemuste tabel rea numbritega.*

Klõpsates tabelis olevale reale, avaneb peidetud tabel, kus on täpsem info, mida iga analüsaator mainitud rea kohta väljastas (joonis 6).



Joonis 6. *Tabel peale tabeli reale klõpsamist. Teksti read analüsaatorite väljunditest, kus valitud reanumbrit on mainitud.*

Klõpsates omakorda mõnel uue tabeli ridadest, näeb valitud analüsaatori poolt kogu ühe spetsiifilise andmejooksu kohta väljastatud infot (joonis 7).



Joonis 7. *Tabel peale avanenud infole klõpsamist. Kogu info potentsiaalse andmejooksu kohta, mis valitud koodireaga seotud.*

Klõpsates uuesti reale, mis varasemalt avas uue informatsioonirea, saab avatud sektsiooni sulgeda.

3.3.2 Uue analüsaatori lisamine

Rakendusele uue analüsaatori lisamiseks on vaja laadida alla programm ja salvestada uus analüsaator selleks ettenähtud kausta, kus on ka Goblint, Locksmith ja Relay. Lisatud analüsaatori jaoks on vaja luua analüsaatorite klasside kausta uus klass, kus on defineeritud funktsioon “run_analyzer()”, mis vajadusel navigeerib käsureal õigesse kohta, käivitab analüsaatori valitud failil ja naaseb algsesse asukohta ning muutujad, millega proramm saab leida vajaliku informatsiooni analüüsi tulemustest. Igal analüsaatoril on vaja analüsaatori nime ja kolme regulaaravaldist.

Esiteks on vaja avaldist suurema lõigu leidmiseks, kus on tavaliselt informatsioon ühe andmejooksu ohu kohta. Teine regulaaravaldis leiab väljundis iga eraldi rea, kus on mainitud ühte spetsiifilist sisestatud koodi rida, kust saab omakorda võtta välja selle ühe rea numbri tabeli koostamiseks. Viimane regulaaravaldis on jagatud mäluaadressi muutuja nime leidmiseks. See ei ole vajalik tabeli koostamiseks, kuid on siiski kasulik info.
Kui klass on loodud, on vaja uus analüsaator importida analüsaatorite initsialiseerimise faili ja lisada see ka sealsesse analüsaatorite nimekirja. Viimaks on vaja uus analüsaator lisada rakenduse põhifailis "app.py" asuvasse analüsaatorite nimekirja ning seejärel ongi uus analüsaator rakendusele lisatud. Analüsaatori eemaldamiseks piisab selle "app.py" failis asuvast nimekirjast eemaldamisest.

3.4 Edasiarendamise võimalused

CoOpeRace täidab seatud eesmärgi kombineerides kolm analüsaatorit ja lihtsustades nende väljundeid, et kasutajal oleks kergem mõista analüüsi tulemusi. Edasiarendamise võimalusi on sellel aga mitmeid.

Näiteks võib jääda kasutajale enamus väljundist siiski segaseks. Korralikku dokumnetatsiooni valitud analüsaatorite väljundite kohta internetis ei leidu ja praegune rakendus teeb vähe nende lahti selgitamiseks. Väljundite juurde saab tulevikus lisada täpsemad kirjeldused ja selgitused, mida iga element tähendab, mis aitaks ka analüsaatorite tulemusi omvahel paremini võrrelda.

Kasutajale võib anda ka valikuid näiteks erinevate tabelijärjestuste vahel, võimaluse kasutajaliidese kaudu muuta, milliste analüsaatorite tulemusi näha soovitakse ja ka näiteks võimaluse lisada uus analüsaator koodi kirjutamise asemel kasutajaliidese kaudu.

Analüüside läbiviimist on võimalik ühtlustada, väljunditest olulise info otsimist muuta täpsemaks kasutades regulaaravaldistest keerukamaid funktsioone.

4. Kokkuvõte

Tarkvara usaldusväärsus muutub maailmas iga tehnoloogia arengusammuga olulisemaks. Pidevalt luuakse uusi veriftseerimisvahendeid erinevate tarkvara probleemide ennetamiseks. Koostööverifitseerimine on lähenemine verifitseerimisele, mis ei nõua iga uue tehnika integreerimist juba eksisteerivasse süsteemi. Selle asemel pannakse süsteemi osad eraldi tööle ühise eesmärgi nimel.

Bakalaureusetöö eesmärk oli luua C-keelsest koodist andmejooksusid otsiva koostööverifitseerimise töövahend CoOpeRace. Töö ühendab kolm juba eksisteerivat koodi analüsaatorit - Goblint, Locksmith ja Relay - ning väljastab nende tulemused kasutajale loetavas formaadis.

Lõputöö raames valmis veebirakendus, kuhu on kasutajal võimalik üles laadida või valida eelnevalt salvestatud failide seast C-keelset koodi sisaldav fail, mida seejärel analüüsitakse. Analüüsi tulemustest leiab programm koodi asukohad, kus võib olla andmejooksu oht ning esitab need asukohad kasutajale tabelina. Soovi korral saab tabeli reale klõpsates näha väljundi konteksti, kus leitud asukoht on välja toodud. Loodud töövahendile on sarnase tööpõhimõttega analüsaatorite lisamine lihtne, selleks on vaja vaid regulaaravaldisi, et leida väljunditest otsitavat infot.

Töö kirjutamise suurimaks raskuseks osutus mõistete definitsioonide ja dokumentatsiooni leidmine. Suur osa töös välja toodud mõistetest ja terminitest on küll tarkvara verifitseerimise maastikul laialdaselt kasutusel, kuid neile ei leidu ametlikku definitsiooni ning neil puudub ka üldkasutatav eestikeelne tõlge. Goblinti, Locksmithi ja Relay' analüüsiväljundi kirjeldused põhinevad peamiselt töö juhendaja V. Vojdani ja minu enda kogutud teadmistel. Tänu nendele raskustele aga oli töö kirjutamine põnev. See andis võimaluse harida end teemal, millega on varasemat kokkupuudet vähe, proovida erinevaid lahendusi lõputöö praktilise osa loomisel ning kirjutada töö, mis aitab potentsiaalselt kaasa eest keele arengule informaatika vallas.

Töövahend CoOpeRace täidab kõik esmased seatud eesmärgid, kuid sellel on mitmeid aspekte, mida tulevikus saab edendada. Kasutajaliidesele saab luua uusi funktsioone, mis teeks tulemuste haldamise ja nendest paremini aru saamise veelgi lihtsamaks. Programmile

saab lisada praegustest erinevate tööpõhimõtete ja väljundimudelitega analüsaatoreid ning selle üldist ülesehitust saaks teha kompaktsemaks ja konkreetsemaks.

Viidatud kirjandus

[1] Beyer D., Lemberger T., Haltermann J., Wehrheim H. 2022. Decomposing Software Verification into Off-the-Shelf Components: An Application to CEGAR. 2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE), Pittsburgh, PA, USA. DOI: 10.1145/3510003.3510064.

[2] Beyer D., Wehrheim H. 2020. Verification Artifacts in Cooperative Verification: Survey and Unifying Component Framework. Lecture Notes in Computer Science, Vol. 12476. DOI: 10.1007/978-3-030-61362-4_8.

[3] Zhang Y., Yan J., Qiao L., Gao H. 2022. A novel approach of data race detection based on CNN-BiLSTM hybrid neural network. Neural Computing and Applications, Vol 34, Issue 18, pp 15441–15455. DOI: 10.1007/s00521-022-07248-8.

[4] IT terministandardi sõnastik. URL: https://www.eki.ee/dict/its/ (mai 2024)

[5] AKIT. URL: https://akit.cyber.ee/ (mai 2024)

[6] Møller A., Schwartzbach M. I. 2023. Department of Computer Science, Aarhus University. URL: https://cs.au.dk/~amoeller/spa/ (mai 2024)

[7] Pierce B. C. 2004. Advanced Topics in Types and Programming Languages. The MIT Press.

[8] Computer Dictionary, Terms and Glossary. URL: https://www.computerhope.com/jargon.htm (mai 2024)

[9] CWE-561: Dead Code (4.14). URL: https://cwe.mitre.org/data/definitions/561.html (mai 2024)

[10] McGraw G., Chess B. 2004. Static Analysis for Security. IEEE Computer Society. DOI: 10.1109/MSP.2004.111.

[11] Landi W. 1992. Undecidability of Static Analysis. Letters on Programming Languages and Systems, Vol. 1, Issue 4. DOI:10.1145/161494.161501.

[12] Beyer, D. 2017. Software Verification with Validation of Results. In: Legay, A., Margaria, T. (eds) Tools and Algorithms for the Construction and Analysis of Systems. TACAS 2017. Lecture Notes in Computer Science(), vol 10206. Springer, Berlin, Heidelberg. DOI: 10.1007/978-3-662-54580-5_20.

[13] Goblint. A static analyzer for multi-threaded C programs, specializing in finding concurrency bugs. URL: https://goblint.in.tum.de/home (detsember 2023)

[14] Vojdani V. 2009. Static Data Race Analysis for C. Doktoritöö, Tartu Ülikool. URL:http://hdl.handle.net/10062/15866

[15] Cousot P., Cousot R. 1977. Abstract Interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 238—252, Los Angeles, California. DOI: 10.1145/512950.512973.

[16] Vojdani V., Apinis K., Rõtov V., Seidl H., Vene V., Vogler R. 2016. Static race detection for device drivers: the Goblint approach. In Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering. DOI: 10.1145/2970276.2970337.

[17] Pratikakis, P., Foster, J.S., Hicks, M. 2006. Locksmith: Context-sensitive correlation analysis for detecting races. In: PLDI'06. pp. 320–331. ACM Press. DOI: 10.1145/1133981.1134019,

[18] Voung, J.W., Jhala, R., Lerner, S. 2007. RELAY: static race detection on millions of lines of code. ESEC/FSE'07. pp. 205–214, ACM Press. DOI: 10.1145/1287624.1287654,

[19] Stack Overflow Developer Survey 2023. https://survey.stackoverflow.co/2023/#section-admired-and-desired-other-frameworks-and-libraries (veebruar 2024)

Lisad

I. CoOpeRace

Töövahendi CoOpeRace programm on leitav aadressil https://github.com/epphaava/CoOpeRace .
Töö raames tehtud muudatuste ajalugu on leitav aadressil https://github.com/epphaava/loputoo

II. Litsents

Lihtlitsents lõputöö reprodutseerimiseks ja üldsusele kättesaadavaks tegemiseks

Mina, Epp Haavasalu,

1. annan Tartu Ülikoolile tasuta loa (lihtlitsentsi) minu loodud teose

Töövahendi CoOpeRace loomine,

mille juhendaja on Vesal Vojdani,

reprodutseerimiseks eesmärgiga seda säilitada, sealhulgas lisada digitaalarhiivi DSpace kuni autoriõiguse kehtivuse lõppemiseni.

2. Annan Tartu Ülikoolile loa teha punktis 1 nimetatud teos üldsusele kättesaadavaks Tartu Ülikooli veebikeskkonna, sealhulgas digitaalarhiivi DSpace kaudu Creative Commonsi litsentsiga CC BY NC ND 4.0, mis lubab autorile viidates teost reprodutseerida, levitada ja üldsusele suunata ning keelab luua tuletatud teost ja kasutada teost ärieesmärgil, kuni autoriõiguse kehtivuse lõppemiseni.

3. Olen teadlik, et punktides 1 ja 2 nimetatud õigused jäävad alles ka autorile.

4. Kinnitan, et lihtlitsentsi andmisega ei riku ma teiste isikute intellektuaalomandi ega isikuandmete kaitse õigusaktidest tulenevaid õigusi.

Epp Haavasalu
15.05.2024