TARTU ÜLIKOOL
Arvutiteaduse instituut
Informaatika õppekava

Raner Lebbin

Programmianalüsaator Goblinti hindamine Juliet testkomplektiga

Bakalaureusetöö (9 EAP)

Juhendaja: Vesal Vojdani, PhD

Tartu 2021

Programmianalüsaator Goblinti hindamine Juliet testkomplektiga

Lühikokkuvõte: Staatiline programmianalüüs võimaldab programme uurida käitusaja väliselt ja aitab avastada lähtekoodis leiduvaid turvanõrkuseid. Andmevooanalüüsi rakendav analüsaator Goblint, millega analüüsitakse staatiliselt mitmelõimelisi C-keelseid programme, suudab tuvastada andmejooksul põhinevaid turvanõrkuseid. Käesoleva töö eesmärk on hinnata Goblinti analüüsi võimekust ning pakkuda välja võimalusi analüsaatori tegevusvälja laiendamiseks. Töös kasutatakse Goblinti testimiseks Juliet testkomplekti, mis on kogum erinevaid nõrkuseid sisaldavatest testprogrammidest. Selle protsessi tarvis luuakse skript (Pythoni keeles), mis automatiseerib Goblinti rakendamist suurel hulgal testfailidel ja kuvab tulemused mugavaks ülevaatuseks HTML-failis. Tulemuste põhjal analüüsitakse Goblinti sooritust. Edasiarenduseks pakutakse välja turvanõrkused, mille kohta analüsaator võiks anda hoiatusi, ning luuakse uued testid, mille alusel implementeerida uute nõrkuste tuvastamist.

Võtmesõnad: staatiline analüüs, Goblint, Juliet testkomplekt, andmejooks, turvanõrkused

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

Evaluating Static Analyzer Goblint on the Juliet Test Suite

Abstract: Static program analysis is used outside of program runtime to discover potential security vulnerabilities. Goblint is a static analyzer based on data-flow analysis that examines multithreaded C programs and can detect flaws caused by data races. This thesis aims to evaluate Goblints' current capacity and provide ways to expand its functionality. The Juliet Test Suite, a set of test cases covering various vulnerabilities, is used to benchmark the analyzer. A script is written (in Python) that automatically runs Goblint on multiple test cases and generates an HTML file to display the results conveniently. Based on these results, this thesis highlights vulnerabilities that Goblint could detect and provides simplified regressions tests to support the suggested developments.

Keywords: static analysis, Goblint, Juliet Test Suite, data race, security vulnerabilities

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

Sisukord

Sissejuhatus **4**

1. Staatiline programmianalüüs **6**

1.1. Staatilise analüüsi tugevused 8

1.2. Staatilise analüüsi nõrkused 9

1.3. Goblint 9

2. Goblinti analüüs Juliet testkomplekti abil **12**

2.1. Juliet testkomplekti kirjeldus 12

2.2. Automaatse mõõtlustesti skripti arendamine 15

2.3. Tulemused: Goblinti hetkeseisu hindamine 22

3. Uued turvanõrkused ja nende testprogrammid **25**

3.1. Täisarvu ületäitumine (CWE-190) 26

3.2. Täisarvu allakadu (CWE-191) 28

3.3. Avaldise väärtus alati väär (CWE-570) 29

3.4. Avaldise väärtus alati tõene (CWE-571) 30

Kokkuvõte **32**

Viidatud kirjandus **33**

Lisad **36**

Sissejuhatus

Lähtekoodi analüüsimine on oluline protsess tarkvaraarenduses, mis võimaldab tuvastada uuritavas tarkvaras erisuguseid vigu ning seeläbi ennetada turvanõrkuste teket valmisproduktis [1]. Vigade enneaegne avastamine on kriitilise tähtsusega, sest vea esiletõus arenduse hilisemas faasis võib olla äärmiselt kulukas nii rahaliselt kui ka tööajas, mis kulub vea likvideerimisele [2]. Üheks selliseks analüüsi meetodiks on staatiline analüüs, mis kogub programmi käivitamata lähtekoodist informatsiooni, mille põhjal hinnatakse programmi käitumist erinevates seisundites ning seeläbi tuvastatakse potentsiaalseid vigu [1].

Staatiline analüüs võimaldab tuvastada kergemaid arendajate eksimusi nagu üleliigsed koodijupid või mittevastavusi valdkonna standarditega, kuid samuti aitab avastada suurema riskifaktoriga nõrkuseid nagu andmejooks või täisarvu ületäitumine [3].

Programmianalüüsi teostatakse kahel viisil: manuaalselt või automaatselt. Manuaalsel analüüsil on keskseks tegevuseks koodi läbivaatused (ingl *code review*), mida on võimalik sooritada arendajal üksinda või parema tulemuse saavutamiseks teeb seda grupp kvalifitseeritud inimesi. Manuaalne analüüs on kulukas, ajanõudlik ja ebastabiilne, sest arendajate tase avastada lähtekoodist vigu varieerub. Seetõttu kasutatakse koodianalüüsis eraldiseisvaid programme, mis teostavad analüüsi automaatselt ja on efektiivsemad [1, 4].

Goblint [5] on Tartu Ülikooli ja Müncheni Tehnikaülikooli koostööna valminud ja arendatav staatiline analüsaator, mille eesmärk on analüüsida mitmelõimelisi C-keelseid programme. Täpsemini kasutab Goblint andmevooanalüüsi (üks staatilise analüüsi meetoditest) selleks, et tuvastada andmejooksust (*data race*) tingitud turvanõrkuseid. Andmejooks esineb siis, kui mitu lõime üritavad üheaegselt samale mälukohale ligi pääseda ning see võib põhjustada andmete kadu või riknemist [6, 7].

Analüsaatori kasulikkus väljendub selles kui korrektsed (*sound*) on sooritatavad analüüsid ehk kas avastatakse kõik lähtekoodis esinevad vead ning mitut erinevat turvanõrkust on analüsaator võimeline identifitseerima. Goblint on korrektne analüsaator, kuid keskendub peamiselt andmejooksudele ja võib sealjuures tuvastada ka valepositiivseid vigu (analüsaator leidis vea, mida programmis tegelikult ei eksisteeri) [1, 7].

Töö eesmärkideks on hinnata Goblinti olemasolevat funktsionaalsust ja pakkuda suunitlusi selle täiustamiseks ning viia läbi eeltöö, mille abil on Goblinti arendajatel kergem implementeerida uute turvanõrkuste tuvastamist.

Eesmärkide saavutamiseks kasutatakse testkomplekti *Juliet Test Suite v1.3 for C/C++* (edaspidi Juliet testkomplekt) [8]. See on mastaapne kogum turvanõrkuseid sisaldavatest C-keelsetest testfailidest. Juliet komplekt valiti eelkõige tema suuruse tõttu - kokku on 64295 testfaili 118 erinevas CWE (*Common Weakness Enumeration*) kategoorias. CWE on üldtunnustatud loend erinevatest tark- ja riistvaraga seonduvatest nõrkustest, mida kasutatakse laialdaselt tarkvaraarenduses kui nõrkuste identifitseerimise standardit [9].

Olemasoleva funktsionaalsuse hindamiseks kirjutatakse skript, mis integreerib Juliet testkomplekti mugava ja automatiseeritud kasutuse Goblinti mõõtlustestide (*benchmark*) keskkonda. Skripti abiga saab katsetada Goblintit sobilikul testprogrammide kogumil (terve Juliet komplekt või selle mingi alamkomplekt). Käesoleva töö raames rakendatakse skripti andmejooksude tuvastamise protsessi hindamiseks ja parendamiseks Goblintis, kuid see on mõeldud arendajatele kasutamiseks ka selle töö väliselt.

Eeltöö uute turvanõrkuste avastamiseks on Goblinti arenduses oluline, sest see aitaks laiendada analüsaatori tegevusvälja. Selleks valitakse vastavalt CWE kategooriatele teatud turvanõrkused, mis on olemas Juliet komplektis ja mida on seega tänu skriptile kerge katsetada arenduse jooksul. Eeltöö raames määratakse turvanõrkused ning luuakse uued testprogrammid, mis oleksid Goblinti regressioonitestidele sobivama struktuuriga. Iga test sisaldab üht valitud nõrkust ja need on aluseks Goblinti arendajatele uute turvanõrkuste tuvastamise protsessi implementeerimisel analüsaatorisse.

Töö esimeses peatükis antakse ülevaade staatilist analüüsist, selle erinevatest meetoditest, tugevustest ja nõrkustest ning tutvustatakse automaatsete analüraatorite tööd ja täpsemalt seda, mis on Goblint. Teises peatükis kirjeldatakse lähemalt Juliet testkomplekti struktuuri, skripti ülesehitust ja funktsionaalsust ning antakse hinnang Goblinti hetkeseisule. Kolmandas peatükis määratakse uued turvanõrkused ja kirjeldatakse nende tarvis loodud testprogramme.

1. Staatiline programmianalüüs

Staatiline programmianalüüs[1] on arvutiteadusest tuntud tegevus, mille eesmärk on lähtekoodist koguda programmi semantilist informatsiooni ning selle põhjal siluda uuritava arvutiprogrammi lähtekoodi [10, 11]. Analüüs võimaldab hinnata programmi töökindlust, turvalisust ja optimeeritust ning aitab ennetada või vähendada käitusajal ilmnevate potentsiaalsete tõrgete arvu [12]. Staatilise programmianalüüsi eripära seisneb selles, et analüüsi teostatakse väljaspool programmi käitusaega (kompileerimisajal), tema vastandiks on dünaamiline programmianalüüs, kus lähtekoodi uuritakse keset programmi tööaega [1]. Staatilist analüüsi hakati rakendama alates 1960-ndate algusest, kuid siis oli selle funktsionaalsus rohkem piiritletud ning põhiline kasutusala oli kompilaatorites programmide optimeerimine [10].

Staatilise analüüsi erinevaid vorme liigitatakse vastavalt läbiviidava analüüsi tööprotsessile. Anders Møller and Michael I. Schwartzbach on oma akadeemilises väljaandes "Static Program Analysis" esile toonud mitmeid erinevaid staatilise analüüsi meetodeid, mida on võimalik teostada automaatselt (puudub programmeerija poolne suunamine). Järgnevalt on lähemalt tutvustatud mõnda neist analüüsi meetoditest selleks, et anda põhjalikum ülevaade analüsaatorite tööst [10]:

1. Andmevooanalüüs (*data-flow analysis*) - Analüüsitava programmi võimalikest seisunditest koostatakse juhtvoograaf (*control-flow graph*) ja lõpliku pikkusega täielik võre (*lattice*). Võre kirjeldab abstraktset informatsiooni, mida soovitakse omistada erinevatele juhtvoograafi tippudele, iga tipp defineerib mingit võrrandisüsteemi muutujat ning serv sümboliseerib operatsiooni selle muutuja suhtes. Andmevooanalüüsis vaadeldakse iga täitmise sammu järel tipu seisundit ning selle põhjal valideeritakse programmi tööd [12, 13].

2. Juhtvooanalüüs (*control-flow analysis*) - Juhtvooanalüüsi (või käsuvooanalüüsi) ülesandeks on alalhoidlikult lähendada alamosakeste (kõrgemat järku funktsioonid, objektorienteeritud keeltel meetodikutsed) vahelist juhtvoogu. Teisisõnu luuakse kutsegraaf (*call graph*), kus on graafiliselt kujutatud hierarhilised seosed erinevate

---

[1] Käesoleva töö autor on kirjutanud antud teemat hõlmava Vikipeedia artikli ("Programmi analüüs", https://et.wikipedia.org/wiki/Programmi_anal%C3%BC%C3%BCs).

kutsungite vahel või nende seoste puudumised. Juhtvooanalüüs eristub andmevooanalüüsist enim selle poolest, et juhtvooanalüüsi võrrandisüsteemid on üldiselt keerulisemad [12].

3. Tüübianalüüs (*type analysis*) - Tüübianalüüs (samuti ka tüübikontroll või tüübisüsteem) uurib, kas koodis deklareeritud andmetüüpidega muutujaid on erioperatsioonides kasutatud korrektselt ning kas väljundite tüübid vastavad ootustele. Näiteks muutujale, mis on deklareeritud *boolean*-tüübiks, ei tohiks omistada mitte ühelgi programmi käitusel teist talle sobimatut muutuja tüübiga väärtustust. Keeli, kus muutujate mittetriviaalseid tüüpe deklareeritakse, nimetatakse tüübitud keelteks ja keeli, mis muutujate tüüpe ei piira, tüüpimata keelteks [14].

Programmi staatilise analüüsi kohta võib väita, et see on matemaatiliselt mittelahenduv (*undecidable*) probleem [10, 11]. See tuleneb Rice'i teoreemist, mis ütleb, et "ei leidu Turingi masinat, mis suudaks alati vastata õigesti küsimustele programmi mitte-triviaalsete semantiliste omaduste kohta" ja mida on võimalik tõestada lähtudes Turingi masinate peatumise probleemist (*halting problem*) [15:4]. Staatilise analüüsi jaoks tähendab see sisuliselt seda, et tulemuste leidmiseks tuleb kasutada vastuste lähendamist (*approximation*) ning peab arvestama, et "ideaalset" staatilist analüsaatorit ei eksisteeri. Lähendamise optimeerimine ning lähendatud vastuste täpsuse parandamine on analüsaatori loomise ja arendamise protsessi juures ühed olulisemad võtmekohad [10].

Programmianalüüsi on võimalik teostada manuaalselt, kuid oluliselt efektiivsem on seda teha automaatselt kasutades selleks mõnda otstarbelist tööriista ehk analüsaatorit [16]. Analüsaator on uuritavast programmist eraldiseisev tarkvara ja selle üldisem eesmärk on viia lähtekood vastavusse valdkonnas levinud standarditega ning avastada võimalikud turvanõrkused. Iga analüsaator on spetsialiseerunud teatud programmeerimiskeelele või -keeltele ning suudab avastada vaid talle teadaolevaid turvanõrkuseid. Selliseid tööriistu kasutatakse laialdaselt tarkvara testimisel, arendamisel ja kvaliteedikontrollis [17].

Automaatset staatilist analüüsi teostavad töövahendid skaneerivad vaadeldava programmi lähtekoodi ning sõeluvad eelsätestatud reeglite ja mustrite põhjal välja potentsiaalsed koodijupid, mis võivad endas sisaldada otsitavaid nõrkuseid. Tööprotsess on mingil määral sarnane viirusetõrjetarkvarale, mis samuti fikseeritud mustrite abil otsib ohtusid. Pärast seda,

kui automaatne analüsaator on lähtekoodil oma töö lõpetanud ning raporteerinud võimalikud vead, tuleb arendajal valideerida leitud vigade staatus (määrata, kas tegu oli valepositiivsega või mitte) ning seejärel implementeerida parandused. Mõningad automaatsed tööriistad lihtsustavad arendajate tööd soovitades muudatusi või pakkudes lahendusi vigade eemaldamiseks [3, 16].

1.1. Staatilise analüüsi tugevused

Põhjuseid, miks staatilist programmianalüüsi kasutada tarkvaraarenduses ja miks seda võidakse olenevalt olukorrast eelistada dünaamilisele analüüsile, on mitmeid.

Staatiline analüüs leiab lähtekoodist nõrkuseid ilma, et oleks vaja käivitada sellel lähtekoodil põhinevat programmi. Selline baitkoodi tasandil lähenemine eemaldab otsese seose lähtekoodi ja tema käivituskeskkonna vahel ning samuti sõltuvuse erinevate teekidest, s.t lähtekoodi võib vaadata kui eraldiseisvat koodijuppi. Lisaks võib olla oht, et välisest allikast pärit kood on pahatahtlikku loomusega ning selle täitmine toob esile ootamatuid probleeme [13, 18].

Abstraktne interpretatsioon on meetod, mis tõlgendab programmi funktsionaalsust lihtsustatud ja üldistatud kujule. See võimaldab lähtekoodi semantilise informatsiooni põhjal hinnata programmi või mõne spetsiifilisema tehte õigsust. Staatilises analüüsis kasutatakse abstraktseid sisendparameetreid, mille ülesandeks on lihtsustada tehted abstraktsele kujule ja eemaldada vajadus testida koodi kõikvõimalike sisendparameetritega (tüüpiline probleem dünaamilises analüüsis). Sõltuvalt sisendi tüübist võib abstraktsete parameetrite kasutamata jätmine muuta programmi testimise kulukaks või sisuliselt võimatuks [12, 13].

Staatiline analüüs aitab kiiremini leida selliseid potentsiaalseid vigu, mis võivad ajapikku tekitada süsteemis märkimisväärset ja kulukat kahju. Tarkvaratehnika ekspert Steve McConnell on oma raamatus “Code Complete” rõhutanud, et defektide varajane avastamine on tarkvara arenduses tohutult oluline, sest hilisemates faasides (täissüsteemi proovilepanek, pärast tarkvara avaldamist) võib vea parandamine ning koodi ülekirjutamine olla 10 kuni 100

korda resursinõudlikum (põhineb Hewlett-Packard, IBM, Hughes Aircraft, TRW ja teiste organisatsioonide tehtud uuringutel) [2].

1.2. Staatilise analüüsi nõrkused

Programmide staatilisel analüüsil tuleb arvestada ka mõningate puudujääkidega. Lisaks sellele, et staatiline analüüs on oma olemuselt mittelahenduv probleem ning seega ei võimalda täieliku veendumusega programmi korrektsust hinnata, tekivad raskused analüüsi tulemuste vääras identifikatsioonis ja nende kehvas tõlgendamises arendajate poolt.

Tulenevalt staatilise analüüsi mittelahenduvusest, on vajalik analüüsitava programmi tööd üldistada ja lähendada. See toob endaga paratamatult kaasa selle, et analüsaator jätab programmis tuvastamata vead, mis seal tegelikult esinevad (ehk valenegatiivsed), ning teisalt tuvastab vigu, mida tegelikult programmis ei eksisteeri (ehk valepositiivsed) [19].

Valenegatiivsed võivad põhjustada märkimisväärset kahju, sest vea tuvastamata jätmine tekitab arendajas ekslikult turvatunde, et programm justkui peaks töötama õigesti ja kui see viga lõpuks ilmneb, siis võib selle parandamine olla kulukas [19]. Valepositiivsete mõju on samuti kehvapoolne. Kui Google disainis endale uut staatilise analüüsi keskkonda, siis nad avastasid, et valepositiivsed tekitavad arendajatele stressi ja liigset ajakulu. Selle tulemusena hakkavad arendajad kergekäeliselt analüsaatori raporteeritud vigu ignoreerima pidades neid valepositiivseteks ilma, et uuritaks põhjalikumalt vea tegelikku olemust. Tänu sellele pääsevad programmi sisse vead olenemata sellest, et analüsaator suutis neid tuvastada [20].

1.3. Goblint

Goblint valmis Müncheni Tehnikaülikooli ja Tartu Ülikooli vahelise koostöö tulemusena, samad institutsioonid vastutavad ka selle programmi arendamise eest. Goblinti avatud lähtekoodile pääseb ligi GitHubi repositooriumis (https://github.com/goblint/analyzer) ning seda tohib kasutada vastavalt MIT-i litsentsi direktiividele [5].

Goblint on andmevooanalüüsil põhinev automaatne staatilise analüüsi töövahend, mis rakendab abstraktset interpretatsiooni selleks, et üle-lähendada programmi olemust. See tähendab, et määratakse ligikaudselt programmi kõikvõimalikud seisundid. Goblinti raamistik on kirjutatud OCaml programmeerimiskeeles. Analüsaator on peamiselt mõeldud mitmelõimeliste (*multi-threaded*) C-keelsete programmide uurimiseks, et neis avastada andmejooksuga (*data race*) seotud turvanõrkuseid [6, 21].

Andmejooks on turvanõrkus, mis esineb programmis siis, kui mitu omavahel sünkroonimata lõime üritavad üheaaegselt samale mälukohale kirjutada mingit väärtust. Selle tulemusena võib tekkida andmete riknemist või kadu ning programmi käitumine on seeläbi ettearvamatu [22]. Lihtsustatud näidet sellise olukorra kohta võib näha joonisel 1. Peafunktsioonis luuakse kaks lõime (read 13-14), mis kutsuvad mõlemad välja funktsiooni `helperBad`, et uuendada globaalse muutuja `gBadInt` väärtust. Kuna lõimed on sünkroonimata, siis pole teada kumb lõim, mis ajahetkel funktsiooni rakendab. Selles programmis raporteerib Goblint andmejooksu real 5, kus toimub muutujate väärtuse ülekirjutamine.

```
1    static int gBadInt = 0;
2
3    static void helperBad(void *args)
4    {
5        gBadInt = gBadInt + 1; // DATA RACE
6    }
7
8    void main()
9    {
10       stdThread threadA = NULL;
11       stdThread threadB = NULL;
12       // Creating threads, calling helperBad
13       if (!stdThreadCreate(helperBad, NULL, &threadA)) threadA = NULL;
14       if (!stdThreadCreate(helperBad, NULL, &threadB)) threadB = NULL;
15       // Closing threads
16       if (threadA && stdThreadJoin(threadA)) stdThreadDestroy(threadA);
17       if (threadB && stdThreadJoin(threadB)) stdThreadDestroy(threadB);
18   }
```

Joonis 1. *Andmejooksu sisaldav näidis koodiosa, aluseks Juliet testkomplekti fail “CWE366_Race_Condition_Within_Thread__global_int_01.c” (lihtsustatud) [8].*

Goblinti analüüs andmejooksude puhul on korrektne, mis tähendab, et tuvastatakse kõik veaohtlikud kohad (sh valepositiivsed). Selle saavutamiseks teostab Goblint analüüsi vastastikku välistavatele (*mutex*) lukkudele, mille käigus kogutakse informatsiooni globaalsete muutujate ja lukkude vaheliste seoste kohta. Selleks, et lõimed oleksid sünkroonitud ning andmejooksu ei esineks, peab ühele globaalsele muutujale vastama erinevatel juhtumitel ühine lukk [22].

Goblint otsib üles kõik juhtumid, kus ühe muutuja väärtust uuendatakse, ning koostab iga juhtumi kohta hulga, milles on kõik konkreetse juhtumiga seotud lukud. Seejärel leitakse nende hulkade ühisosa - kui see on tühi ehk ühiseid lukke ei leidu, siis järelikult esineb andmejooks ja Goblint annab selle kohta hoiatuse [22].

2. Goblinti analüüs Juliet testkomplekti abil

Goblinti rakendamiseks Juliet testkomplekti peal kirjutatakse skript, mis viib automaatselt läbi vajalikud testid ning koondab testide tulemused ühte HTML-faili, kus neid on mugav uurida. Skript kirjutatakse Pythoni programmeerimiskeeles, kuid esmalt tuleks tutvuda Juliet testkomplekti struktuuriga ning testprogrammide sisuga, mida tehakse peatükis 2.1. Skriptist ja selle kirjutamisel ilmnenud probleemidest antakse ülevaade peatükis 2.2. ning viimaks analüüsitakse skripti abiga Goblinti sooritust Juliet komplektil peatükis 2.3. Skript asub töö autori GitHubi harutatud repositooriumis (https://github.com/RanerL/bench) kaustas “juliet”.

2.1. Juliet testkomplekti kirjeldus

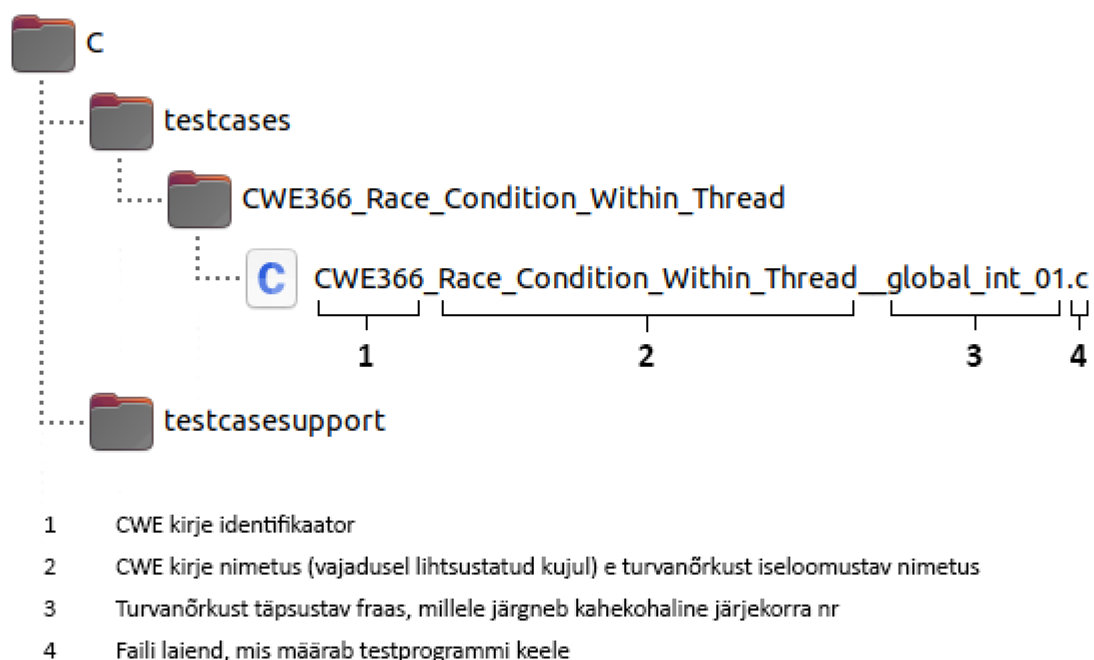
Tehislikuks (*artificial*) testkomplektiks (*test suite*) nimetatakse kogumit sihtotstarbeliselt konstrueeritud programmidest, kus iga programm on üksik eraldiseisev test, mis üritab jäljendada mingisugust potentsiaalse turvaohuga koodiosa. Tema vastandiks on loomulik (*natural*) testkomplekt, mis koosneb päris programmidest (Juliet on tehislik komplekt). Testkomplekte kasutatakse staatilises analüüsis analüsaatorite soorituse hindamiseks või mõõtmiseks (*benchmarking*). Igas testis ei pruugi ilmtingimata esineda turvanõrkust, kuid sellegipoolest on tema ülesehitus sarnane seda turvanõrkust sisaldavale testile - sellise testi eesmärk on katsetada, kas analüsaator tuvastab sealt vea ehk valepositiivse või mitte [8, 23].

Juliet testkomplekt loodi Ameerika Ühendriikide Riikliku Julgeolekuagentuuri (NSA) allüksuse Center for Assured Software (CAS) poolt eesmärgiga hinnata staatilist analüüsi teostavate tööriistade võimeid. Testkomplekt koosneb kahest osast: esimeses on test-programmid C ja C++ keeles ning teises on need kirjutatud Javas [24].

Käesoleva töö raames on lähema vaatluse all ainult C-keelsed programmid, mis pärinevad testkomplekti kõige uuemast (avaldatud 2017. a) versioonist, mille ametlik nimetus on *Juliet Test Suite v1.3 for C/C++*. Komplektiga kaasa tulnud dokumentatsiooni põhjal on selles versioonis kokku testprogramme ehk testfaile 64295 ning need hõlmavad 118 erinevat CWE (*Common Weakness Enumeration*) põhjal kategoriseeritud turvanõrkust. Juliet testkomplekti suur maht ja kerge kättesaadavus olid peamisteks põhjusteks antud komplekti valimiseks.

Järgnevalt on selgitatud Juliet testkomplekti failisüsteemi ülesehitust. Testkomplekti allatõmbamisel tekib kaust “C” (edaspidi juurkaust) ning sealt on järgmised tähtsad kaustad “testcases” ja “testcasesupport”, ülejäänud failid või kaustad, mis paiknevad juurkaustas, ei leia töö raames kasutust. Kaust “testcases” sisaldab 118 CWE kirjega märgistatud alamkausta, millest igaüks vastab ühele CWE kategooriate alusel identifitseeritud turvanõrkusele. Kaust “testcasesupport” sisaldab C-keelele omaseid päise faile (*header files*) ja samuti kahte C-keelset abifaili, mis sisaldavad testprogrammide tööks vajalikke abifunktsioone. Testkomplekti lihtsustatud struktuuri võib näha joonisel 2.

CWE kaustad, mis asetsevad “testcases” kaustas, koosnevad testprogrammidest, mille uurimine Goblintiga on töö keskseks osaks. Testprogrammid paiknevad üldiselt otse CWE kaustades, kuid mõned CWE kaustad võivad veel olla sisemiselt jaotatud alamkaustadeks (alamkaustade nimed on stiilis “sXX”, kus “XX” tähistab kahekohalist numbrit) sõltuvalt konkreetse turvanõrkuse natuurist. Failisüsteemi struktuuri seisukohalt tähendab see seda, et testprogrammid asuvad sellisel juhul üks aste sügavamal.



Joonis 2. *Teekond juurkaustast ühe testfailini ja testfaili nimemääramise vorm.*

Joonisel 2 on kujutatud lihtsustatud näide teekonnast, mis viib juurkaustast kuni andmejooksu sisaldavale testfailini nimega "CWE366_Race_Condition_Within_Thread__global_int_01.c". Nagu joonisel 2 on näha, siis testprogrammide nimetamisel järgiti loetavust ja järjepidevust silmas pidades kindlaid reegleid. Faili nimes CWE põhised terminid (ehk 1. ja 2. osas) ühtivad alati CWE kausta nimega. Lisaks võib komplektis leiduda veel selliseid testprogramme, mis on mõeldud jooksutamiseks ainult Windowsi operatsioonisüsteemiga masinates - sellisel juhul asetseb nime 3. osas sõne "w32". See on oluline, sest Goblintit on mõeldud kasutama eelkõige Linuxi arhitektuuriga arvutites. 3. osa kahekohalise numbri järel võib asetseda ka tähekarakter (nt "01a" ja "01b"), see märgib ühe testi erinevaid variatsioone.

Tähtis on aru saada ka testprogrammide (eriti C-keelsete) disainist, sest vastasel juhul võib neid Goblintile uurimiseks ette andes esineda ootamatuseid. Üldjuhul on igas testprogrammis olemas turvanõrkust sisaldav koodiosa kui ka seda mitte sisaldav petliku loomusega koodiosa. Turvanõrkust kätkev kood asub testprogrammi "halvas" funktsioonis ning nii öelda puhas kood asub "heas" funktsioonis. Lähtudes varem näiteks toodud programmist, oleksid vastavate funktsioonide nimed järgnevad:

```
CWE366_Race_Condition_Within_Thread__global_int_01_bad(),
CWE366_Race_Condition_Within_Thread__global_int_01_good().
```

Lisaks eelmainitud kahele funktsioonile leidub testprogrammis veel abifunktsioone ning ka peafunktsioon (*main*). Dokumentatsiooni põhjal on peafunktsioon mõeldud selle jaoks, et üksikut testprogrammi jooksutades oleks võimalik läbida korraga nii halb kui ka hea funktsioon. Töö seisukohalt pole peafunktsioon vajalik, sest Goblintiga uuritakse halba ja head funktsiooni eraldi, et uurimistulemusi oleks mugavam analüüsida. Üksikutel testprogrammidel puudub hea funktsioon, sest ei olnud võimalik luua halva funktsiooniga sarnast programmi, mis ei sisaldaks endas turvanõrkust põhjustavaid vigu.

2.2. Automaatse mõõtlustesti skripti arendamine

Skripti eesmärk on lihtsustada ja automatiseerida Goblinti töö uurimist Juliet komplekti testprogrammidel. Goblintit on võimalik jooksutada üksikute testprogrammide peal, kuid see ei annaks adekvaatset ülevaadet Goblinti seisundist ning ilmselgelt oleks iga testprogrammi eraldi läbivaatamine manuaalselt ebaefektiivne. Testprogrammide uurimisel on oluline mõnikord muuta Goblinti sisendparameetreid ja seejärel tuleb analüsaatori jooksutamist korrata, mis on skripti abil oluliselt kiirem, sest siis on võimalik ühekorde käivitusega läbi käia kõik huvipakkuvad testfailid.

Skript on kirjutatud Pythoni programmeerimiskeeles ning selle käivitamine toimub operatsioonisüsteemi (töö tegemisel kasutati Linuxil baseeruvat Ubuntu OS-i) käsurealt. Selleks, et skripti oleks võimalik kasutada tulevikus ka teistel osapooltel, on koodi põhjalikult kommenteeritud ning iga funktsiooni ees on selle tööd selgitav tekst. Kuna Goblinti arenduses töötavad inimesed eri rahvusest, siis on kogu kood ja sellega kaasas käivad kommentaarid inglise keeles. Järgnevalt on kirjeldatud skripti peamiseid tööülesandeid, skripti ja failisüsteemi ülesehitust ning täpsemalt selgitatud kriitilisemaid koodiosasid.

2.2.1. Peamised ülesanded

Skripti peamised ülesanded jagunevad kaheks: Goblinti jooksutamine testprogrammidel ja saadud tulemuste kuvamine lähemaks läbivaatamiseks.

Goblinti jooksutamisel on võtmekohaks käsk, mida rakendatakse uuritavatel testprogrammidel. Käsk koosneb muutujatest nagu Goblinti täitmisfaili või abifailide kausta asukoht kataloogis või sisendparameetritest nagu peameetodi määramine või lisavõimaluste lubamine. Lihtsustatud (muutujad asendatud selgitustega) ja peamiselt andmejooksude analüüsimiseks mõeldud kujul oleks see järgnev (eri osad on eraldatud püstkriipsuga):

> Goblinti failitee | testprogrammi failitee | C-keelsete abifailide failitee | **-I** | abifailide kausta failitee | **--sets "mainfun[+]"** | testprogrammi funktsioon (hea/halb) | **--enable dbg.debug** | **--enable printstats**.

Jämedas kirjas on näidatud fikseeritud sisendparameetrid. “-I” tähendab Goblinti jaoks seda, et järgnev muutuja määrab testprogrammide päises kasutatavate include abifailide kausta asukoha. “--sets ‘mainfun[+]’” määrab, et järgnev muutuja on uuritava programmi põhifunktsioon, millest alustatakse analüüsimist (juhtvoograafi juurtipp). “--enable dbg.debug” ja “--enable printstats” on lubatud lisavõimalused, mille abil saab analüüsi aruandest rohkem informatsiooni. Siin on illustratiivne näide sellest, milline näeks üks käsk välja päriselt skripti kasutamise ajal, kui Goblint analüüsib varasemalt näiteks toodud testprogrammi head funktsiooni:

```
../../analyzer/goblint
"C/testcases/CWE366_Race_Condition_Within_Thread/
CWE366_Race_Condition_Within_Thread__global_int_01.c"
"C/testcasesupport/*.c"  -I  "C/testcasesupport"  --sets  "mainfun[+]"
"CWE366_Race_Condition_Within_Thread__global_int_01_good"   --enable
dbg.debug --enable printstats
```

Tulemuste kuvamiseks luuakse HTML-fail, mille sisu täidetakse jooksvalt koos testprogrammide analüüsimisega. Valmival HTML-lehel on olemas päis koos legendiga, milles selgitatakse analüüsi olekut väljendavaid märgendeid, sisukord, mille abil on võimalik kiiresti liikuda kindla CWE kausta tulemusteni, ja viimaks põhiline element ehk tabel(id), kus on iga uuritud testprogrammi analüüsi tulemused. Lisaks on võimalik klikkida tabelis igale testprogrammi nimele, et näha põhjalikumalt Goblinti analüüsi olemust. HTML-lehe kujundust on näidatud lisas I, selle jaoks jooksutati skripti vähendatud testkomplektil.

2.2.2. Ülesehitus

Skripti koodiosa võib vaadata kolmes erinevas osas (lähtekoodis on parema loetavuse huvides osad eraldatud pealkirjade ehk kommenteeritud tekstiga):

1. Ülesseadmise ja alustamise osa - deklareeritakse esimesed muutujad koos vaikimisi väärtustega, näiteks veebilink Juliet testkomplekti allatõmbamiseks või Goblinti täitmisfaili asukoht kataloogis. Esmalt kontrollitakse, kas testkomplekt on juba paigaldatud korrektsesse kausta ja kui pole, siis seda tehakse. Kasutajale antakse käsureal tagasisidet selle kohta, kui URL-i vaikeväärtus ei toimi (nt veebileht maas). Lisaks saab kasutaja anda skripti käivitamisel kaasa 1 argumendi, millega muudetakse

testfaile sisaldava kausta faililteed. See on kasulik, sest vaikimisi rakendatakse Goblintit kõikidel komplekti testprogrammidel, kuid paljudel juhtudel on vajalik Goblinti rakendamine näiteks ainult ühe CWE kausta testfailidel.

2. Funktsioonid - skriptis on kokku 4 erinevat funktsiooni. Põhjalikum kirjeldus igast funktsioonist on leitav peatükis “2.2.3. Detailsem ülevaade koodist”.
3. Peamised protseduurid - teostatakse peamised ülesanded ehk HTML-faili loomine ja selle sisu kokkupanemine (legend, sisukord, tabelid) ning organiseeritakse kaustade automaatset läbikäimist ja Goblinti jooksutamist testfailidel. Testide läbikäimise üksikasjalikum kirjeldus asub peatükis “2.2.3. Detailsem ülevaade koodist”.

Selleks, et skripti edukalt käivitada, on oluline fikseerida Goblinti täitmisfaili asukoht. Üldiselt on ettenähtud, et analüsaatori peamine kaust “analyzer”, kus asub ka täitmisfail, ning mõõtlustestide kaust “bench” asuvad failisüsteemis samal sügavusel. Mõlemad kaustad on kättesaadavad Goblinti või töö autori (skriptiga täiendatud kaust “bench”) repositoorimist. Täitmisfaili asukoha vaikeväärtus saab skriptis vajadusel muuta, kuid see nõuaks lähtekoodi püsiprogrammeerimist (*hard coding*).

Juliet testkomplekti juurkaust “C” peab asuma skriptiga samas kaustas. See ei tohiks valmistada probleeme, sest juurkausta puudumisel laeb skript komplekti automaatselt alla õigesse kohta.

2.2.3. Detailsem ülevaade koodist

Skripti täpne nimetus on “juliet_summary.py” ja selle lähtekood leitav töö autori GitHubi repositooriumist (https://github.com/RanerL/bench/blob/master/juliet/juliet_summary.py). Eelnevalt kirjeldati seda, milleks skripti vaja on ja mis ülesandeid ta selleks täitma peab. Lisaks anti ülevaade lähtekoodi struktuurist ning sellest, kus peavad asuma kataloogis failid selleks, et tagada skripti edukas käivitamine. Selles peatükis kirjeldatakse süvitsi funktsioone, ühte peamist protseduuri, milleks on Juliet komplektis testprogrammide läbitöötamine, ja viimasena seda, kuidas skripti jooksutada.

Skripti peamised protseduurid toimivad nelja funktsiooni toel, nende nimetused, sisend- ja väljundparameetrid ning ülesanded on järgmised:

- `def check_path(filepath)`

Sisendiks on failitee, mis viitab mingile kaustale. Funktsiooni eesmärk on uurida etteantud failiteel asuvaid faile ja alamkaustasid ning välja filtreerida edasiseks töötlemiseks sobilikud.

Regulaaravaldiste (*regex*) abil sõelutakse uuritavast kausta välja failid, mis ei ole kirjutatud C keeles ja/või on mõeldud ainult Windowsi operatsioonisüsteemis katsetamiseks. Analüüsimiseks sobivad testprogrammi failid ja kõik leitud alamkaustad lisatakse eraldi massiividesse. Funktsioon tagastab mõlemad massiivid.

- `def goblint_cmd(filepath, filename, mode)`

Sisendiks on failitee, mis viitab testprogrammi asukohale, testprogrammi nimi ja märgend sellest, mis liiki funktsiooni analüüsitakse. Märgendil on seega kaks võimalikku teineteist välistavat väärtus, need on "_good" ja "_bad".

Funktsiooni eesmärk on konstrueerida käsk, millega rakendatakse Goblintit etteantud faili ühel funktsioonil. Terminali käsureal antakse kasutajale jooksvalt teada, millist funktsiooni uuritakse. Regulaaravaldisega asendatakse testprogrammi nime lõpp ".c" märgendi väärtusega, et saada funktsiooni nimi, mida kasutatakse sõne kujul käsu konstrueerimiseks. Sõne kujul käsu realiseerimiseks kasutatakse Pythoni tarkvaramoodulit "subprocess" [25], mis võimaldab skripti siseselt jooksutada süsteemi käsureal käske ning salvestada tekkiv väljund. Funktsioon tagastab Goblinti analüüsi tulemused sõne kujul.

- `def files_output_to_HTML(testcases, filepath, HTML_info)`

Sisendiks on massiiv testfailide nimedega, jooksev failitee, kus asuvad ka testfailid, ning kaheelemendiline massiiv, milles esimeses on HTML-faili tabel sõnena ja teises sisukorra jaoks "href" aadresse sisaldav massiiv.

See funktsioon täidab teiste funktsioonidega võrreldes rohkem erinevaid ülesandeid. Esiteks luuakse HTML-failile uus tabel, mida täidetakse analüüsitavate testfailidega. Teiseks kutsutakse välja funktsioon `goblint_cmd`, millega rakendatakse Goblinti käsku ja mille väljund salvestatakse testprogrammi nimelisse tekstifaili. Tekstifail lingitakse vastava testfailiga tabelist, et selle peale vajutades näeks kasutaja Goblinti aruannet. Kolmandaks

otsitakse aruandest regulaaravaldisega üles sobiv märksõna, millega identifitseeritakse, kas testprogrammi funktsioonis leiti turvanõrkus või mitte ja saadud tulemus märgitakse seejärel tabelisse. Funktsioon tagastab kaheelemendilise HTML-i massiivi, milles on täiendust saanud tabel ning aadresside massiiv.

- `def goblint_analyse(current_directory, HTML_info)`

Koodiosa võib näha joonisel 3. Sisendiks on jooksev failitee ja kaheelemendiline massiiv HTML-faili loomiseks vajaliku informatsiooniga (tabeli alus ja "href" aadressid).

Funktsiooni eesmärk on koondada funktsioonide väljakutsungid ühte kohta ning parandada seeläbi peamiste protseduuride osa loetavust, sest antud funktsioonita esineks koodis palju kordust. Funktsiooni `check_path` abiga määratakse jooksvas kaustas Goblinti jaoks sobivad failid ning alamkaustad, mida uuritakse hiljem (rida 52). Kui sobivaid faile leidus, siis rakendatakse nendel funktsiooni `files_output_to_HTML` (read 54-57). Funktsioon tagastab massiivi alamkaustadest, mida uurida järgmisel sammul, ja täiendust saanud kaheelemendilise massiivi, milles on analüsaatori tulemuste esitamiseks vajalik HTML-faili sisu.

```
43  # The definitive function that performs all the main operations:
44  #       - Takes current working directory and HTML contents as input
45  #       - Finds valid C files for Goblint in that dir and establishes
46  #       other potential sub-directories for the future
47  #       - Runs Goblint on valid test cases if they exist
48  #       - Generates HTML table with new results from those cases
49  #       - Returns potential directories and new HTML contents
50  def goblint_analyse(current_directory, HTML_info):
51          # Getting valid C files and potential directories for further work
52          valid_files, pot_directories = check_path(current_directory)
53          # In case there are valid C files to analyse with Goblint
54          if len(valid_files) > 0:
55                  os.system('echo ' + current_directory) # Feedback to the user
56                  # Generating HTML table with new results from Goblint
57                  HTML_info[0] = files_output_to_HTML(valid_files,
    current_directory, HTML_info[0])
58                  HTML_info[1].append(current_directory)
59          return pot_directories, HTML_info
```

Joonis 3. *Funktsioon "goblint_analyse".*

Funktsiooni `goblint_analyse` (joonisel 3) rakendamine leiab aset peamiste protseduuride osas, kus toimub Juliet testkomplekti kaustade läbikäimine. Funktsiooni on välja kutsutud kolmel korral. Asjakohast koodiosa võib näha joonisel 4.

```
148 # Running Goblint and generating content for HTML
149
150 # Starting with files in the running directory
151 current_dir = path
152 # Running Goblint in current directory and getting new potential
    sub-directories
153 directories, HTML = goblint_analyse(current_dir, HTML)
154
155 # Going through the main CWE directories if present
156 directories.sort()
157 for d in directories:
158         current_dir = path + '/' + d
159         # Running Goblint, getting new sub-directories
160         sub_directories, HTML = goblint_analyse(current_dir, HTML)
161         # When the main CWE dir is split into sub-directories (s01, s02,
    ...)
162         sub_directories.sort()
163         for s in sub_directories:
164                 current_subdir = current_dir + '/' + s
165                 # Running Goblint, max depth achieved so no sub-directories
166                 goblint_analyse(current_subdir, HTML)
```

Joonis 4. *Kaustade läbikäimine ja jooksvalt HTML-faili sisu loomine.*

Kõigepealt deklareeritakse muutuja `current_dir` (rida 151), millele omistatud väärtus on kas käsureal sisendina antud failitee või vaikeväärtusena Juliet komplekti kausta "testcases" asukoht. Etteantud failiteel ehk sügavusel null toimub esmane `goblint_analyse` kutsung, sisendiks sama failitee ja esialgu tühi HTML-faili sisu massiiv (rida 153).

Seejärel, kui sobivatel C-keelsetel failidel on rakendatud Goblintit, sorteeritakse kaustade massiiv tähestikulises järjekorras (rida 156), et hiljem oleks tulemuste HTML-lehel navigeerimine kergem. Sorteeritud kaustad käiakse ükshaaval läbi *for*-tsüklis (read 157-160) ning toimub protsess, mis on tegevuste poolest identne eelmises lõigus kirjeldatuga. Erinevus seisneb selles, et tsüklis uuritakse etteantud failitee alamkausta ehk ollakse tase sügavamal (sügavus üks). Igal tsükli iteratsioonil toimub HTML-faili sisu täiendamine ning lisaks uuritakse tsükliliselt (read 163-166) uuel failiteel asuvate alamkaustade sisu (sügavus kaks).

Seega esineb joonisel 4 näidatud koodiosas teatud kordus ning funktsiooni `goblint_analyse` kutsutakse välja kolmel korral ja kaustasid läbikäies jõutakse kuni sügavusele kaks (alustades sügavusest null). Kaheastmelise *for*-tsükli kasutamine oli ajendatud Juliet komplekti failisüsteemi maksimaalsest sügavusest ehk sellest, kus võib kõige sügavamal asuda Goblintile analüüsimiseks sobiv C-keelne testprogramm.

Selline disain teeb skripti vähem paindlikuks ning teine võimalus oleks olnud kasutada rekursiooni. Siiski on skript mõeldud töötamiseks vaid Juliet komplektil ning komplekti puudumisel paigaldatakse see automaatselt korrektsesse asukohta, seega on failisüsteemi võimalik maksimaalne sügavus alati konstante. Murekoht võib tekkida, kui failitee sisendiks on antud drastiliselt teistsuguse struktuuriga kataloog, kuid sellist kasutust pole ettenähtud. *For*-tsüklite eeliseks oli veel see, et neid oli lihtsam disainida ning skripti käsitsi katsetamisel ja silumisel oli kergem avastada vigu.

Skripti jooksutamine toimub operatsioonisüsteemi terminali käsurealt, soovitatud on kasutada Linuxil põhinevast süsteemi nagu Ubuntu või MacOS. Skripti jooksutamiseks peavad kindlasti olema täidetud kolm järgmist tingimust:

1. masinasse peab skripti käivitamiseks olema paigaldatud Python 3+;
2. Goblinti täitmisfaili tee peaks skripti suhtes olema "../../analyzer/goblint" (ehk kaks kausta kõrgemale ja siis üks sügavamale), vajadusel võib modifitseerida skripti lähtekoodi, et anda täitmisfaili teele uus väärtus;
3. Goblint peab olema ülesseatud korrektselt (järgida juhendit "analyzeri" GitHubi repositooriumis https://github.com/goblint/analyzer/blob/master/README.md).

Kasutajal on võimalik skripti käivitamisel kaasa anda üks argument, milleks on failitee testprogrammide kaustani. Käsureal sisestatud käsk võiks sellisel juhul olla "python3 juliet_summary.py C/uued_testid" (see etteantud kaust on fiktiivne ja eeldatakse, et käsureal asutakse skriptiga samas kaustas). Kui argumenti ei anta, siis arvestab skript vaikeväärtusega, mis on tee Juliet komplekti "testcases" kaustani (ehk kõik testprogrammid). Käsk näeks käsureal sellisel juhul välja lihtsalt "python3 juliet_summary.py".

2.3. Tulemused: Goblinti hetkeseisu hindamine

Goblinti hetkeseisu hindamiseks loodi Juliet komplekti juurkausta “C” ajutiselt uus kaust “testcases_new”. Skripti jooksutamine tervel testkomplektil on ajakulukas (sõltub arvuti riistvarast, töö autori virtuaalmasinas kulus üle 5 tunni) ning ebavajalik, sest Goblint on spetsialiseeritud leidmaks andmejooksu ja ignoreerib seega teisi turvanõrkuseid. Uus kaust sisaldab viite otstarbeliselt välja valitud CWE kausta, need on:

- “CWE366_Race_Condition_Within_Thread”,
- “CWE190_Integer_Overflow”,
- “CWE191_Integer_Underflow”,
- “CWE570_Expression_Always_False”,
- “CWE571_Expression_Always_True”.

Andmejooksude kaust (algusega “CWE366…”) valiti sellepärast, et seda turvanõrkust peaks eelduste kohaselt Goblint tuvastama edukalt ning seega saab selle abil hinnata analüsaatori hetkeseisu. Ülejäänud turvanõrkuste puhul on teada, et Goblint ei väljasta nende ilmnemisel hoiatusi või ei oskagi neid analüüsides leida. Neid nelja turvanõrkust kirjeldatakse lähemalt järgmises peatükis “3. Uued turvanõrkused ja nende testprogrammid”.

Seega võib eeldada, et skripti jooksutamisel raporteerib Goblint turvanõrkustest andmejooksu testfailides, kuid ülejäänud nelja kausta failide puhul turvanõrkustest ei teavitata.

CWE kaust	Test-programmide arv	Turvanõrkus funktsioonis “_good”	Turvanõrkus funktsioonis “_bad”
CWE366_Race_Condition_Within_Thread	36	36	36
CWE190_Integer_Overflow	3420	-	-
CWE191_Integer_Underflow	2622	-	-
CWE570_Expression_Always_False	16	-	-
CWE571_Expression_Always_True	16	-	-

Tabel 1. *Goblinti analüüsi tulemused.*

Skriptil kulus viie kausta läbikäimiseks töö autori masinas natukene üle 35 minuti. Tulemused, mis saadi skripti jooksutamisel Goblinti väljundist, on kokku koondatud ja nähtavad tabelist 1. Sidekriips "-" tähistab, et Goblinti väljundist ei leitud, et neis funktsioonides oleks esinenud turvanõrkuseid.

Osaliselt vastasid tulemused eeldustele, sest kui tegemist pole andmejooksuga, siis Goblint ei olegi häälestatud nendest teavitama või neid leidma. Seda võis täheldada nelja andmejooksuga mitteseonduva kausta puhul. See osa tulemustest ilmestab hästi ka seda, kuidas näeks töö kirjutamise hetkel välja skripti jooksutamine kogu testkomplektil.

Goblinti arhitektuuris on siiski olemas mõningad kontrollid, mida teostatakse jooksvalt andmevooanalüüsi raames. Näiteks, kui programmis esineb täisarvu ületäitumine, siis Goblint analüüsi jätkamiseks arvestab, et ületäituva muutuja väärtus on määramata. Probleem on lihtsalt selles, et kasutajale selle kohta midagi ei väljastata.

Andmejookse puudutavad tulemused on mingis mõttes ootamatud. Hetkeseisu järgi leiti kõigis 36 testprogrammis mõlemas funktsioonis (nii heas kui halvas) turvanõrkus. Tulemus justkui viitaks sellele, et analüsaator on korrektne, sest ühtegi viga ei jäetud tuvastamata, kuid samas väga ebatäpne, sest vead tuvastati ka funktsioonides, kus neid ei olnud (ehk valepositiivsed). Kuna vead tuvastati ühtemoodi kõikides testides, siis oli alust arvata, et midagi läheb analüüsis valesti. Joonisel 5 on näidis koodisa probleemist, mis esineb testkomplekti programmides.

```
1    struct { pthread_mutex_t mutex; } *mylock;
2    int myglobal;
3
4    int main(void) {
5        mylock = malloc(sizeof(*mylock));
6        pthread_mutex_init(&mylock -> mutex, NULL);
7
8        pthread_mutex_lock(&mylock -> mutex);
9        myglobal= myglobal + 1;
10       pthread_mutex_unlock(&mylock -> mutex);
11   }
```

Joonis 5. *Näidis koodiosa andmejooksu testprogrammides tekkivast probleemist.*

Lähemal testprogrammide uurimise selgus, et Juliet komplekti failides kasutatakse dünaamilisi lukke ning Goblint ei suuda verifitseerida, et tegemist oleks igal poole täpselt sama lukuga. Goblinti vaatest kasutatakse seega enne globaalse muutuja ümberväärtustamist eri lõimedes erinevaid lukke ja seetõttu tuvastatakse andmejooks.

Seda olukorda võib näha joonisel 5. Testprogrammis luuakse struktuuri-tüüpi muutuja `mylock`, mis sisaldab lukuvälja nimega `mutex` (rida 1). Struktuur `mylock` on seejärel dünaamiliselt allokeeritud ja algväärtustatud (read 5-6). Probleem tekib real 8, kus on kasutatud luku viita, et lukustada järgnev operatsioon ühele lõimele. Goblint ei ole viidatud välja lugemisel kindel, et see lukk, mis dünaamiliselt allokeeriti, on sama lukk igal pool mujal programmis. Seetõttu ignoreerib Goblint luku võtmise käsku ja tuvastab andmejooksu. Olukorra lahendamiseks tuleks Goblintis implementeerida unikaalsuse analüüs, mis suudaks kinnitada, et 5. real on allokeeritud ainult üks objekt ja eristataks seda vajadusel teistest samalaadsetest.

3. Uued turvanõrkused ja nende testprogrammid

Goblint on orienteeritud leidmaks andmejooksu põhiseid turvanõrkuseid ning nagu eelmises peatükis kinnitati, siis teisi nõrkuseid ei tuvastata või vähemalt nendest ei teavitata. Liisa Sakerman ja Rain Hallikas on oma teadustöös [4] uurinud staatilise analüüsi tööriistu. Nad on ühe miinusena välja toonud asjaolu, et arendajad on sunnitud kasutama mitut töövahendit korraga, et avastada võimalikult palju vigu. Analüsaatori arenduse seisukohalt on seega oluline, et tegevusväli suureneks, sest muidu on analüüsatorist kasu vaid teatud juhtudel.

Uute turvanõrkuste tuvastamise implementeerimine on keeruline protsess ning üheks käesoleva töö eesmärgiks oli viia läbi selleks vajalikud ettevalmistused. Eeltöö uute nõrkuste avastamiseks seisneb selleks, et esiteks identifitseeritakse uued turvanõrkused, mida Goblinti analüüs leidma hakkab, ja teiseks luuakse arenduse tarvis testprogrammid.

Uued turvanõrkused valiti koos selle lõputöö juhendajaga, kes on Goblinti analüsaatori üks juhtivatest arendajatest (V. Vojdani). Uued turvanõrkused koos vastavate CWE kategooria ID-numbritega on järgnevad:

- CWE-190: täisarvu ületäitumine (*overflow*),
- CWE-191: täisarvu allakadu (*underflow*),
- CWE-570: avaldise väärtus alati väär,
- CWE-571: avaldise väärtus alati tõene.

Põhjused, miks valiti just need nõrkused, jagunevad peamiselt kaheks. CWE-190 ja CWE-191 on ohtlikumad nõrkused, mis võivad põhjustada tõsisemaid turvaauke. Nende lisamine Goblintisse oleks tugev täiendus senisele funktsionaalsusele just programmide turvalisuse aspektist. CWE-570 ja CWE-571 on kergemad nõrkused, mis kahjustavad koodi kvaliteeti või pärsivad programmi ettenähtud toimimist (tingitud arendajate loogikavigadest). Turvanõrkuste täpsemad kirjeldused on leitavad neile vastavatest alapeatükkidest.

Iga turvanõrkuse kohta kirjutatakse mõned testprogrammid. Nende abil saavad arendajad teha algust uute nõrkuste avastamise võimekuse loomisega ning kasutada neid kui regressiooniteste, millega hinnata Goblinti seisundit erinevates iteratsioonides.

Uued loodavad testprogrammid kujutavad endast väiksemat ja kompaktsemat versiooni Juliet testkomplekti ühest alamhulgast (milleks on neli erinevat CWE kategooriat). Erinevus võrreldes testkomplekti programmidega on see, et uusi programme on standardiseeritud ja on lihtsustatud nende loetavust.

See tähendab, et on eemaldatud sõltuvus Juliet komplektile eripärastest abifailidest (näiteks "std_testcases.h"), nende asemel kasutatakse vajadusel üldlevinumaid teeke. Lisaks on uutes testprogrammides ainult "halb" funktsiooni ehk siis igas programmis on üks peafunktsioon, kus kutsutakse esile turvanõrkus. Esialgse arenduse juures pole petlikud "head" funktsioonid niivõrd olulised, sest korrektne analüsaator peab suutma eelkõige tuvastada kõik veaohtlikud olukorrad. Omades ainult üht põhifunktsiooni ja sõltumatust teatud abifailidest, on Goblinti analüüsi mugavam initsieerida, sest pole vaja kasutada nii palju sisendparameetreid.

Uued testprogrammid asuvad GitHubi repositooriumis (https://github.com/RanerL/bench) ja on leitavad kaustast "juliet/testcases_demo". Kõik testprogrammide nimetused on nähtavad lisas II, kus iga programmi juures on ka lühikene kirjeldus turvanõrkuse põhjustajast.

3.1. Täisarvu ületäitumine (CWE-190)

Täisarvu ületäitumine[2] esineb programmides aritmeetilise tehte tulemusena (nagu liitmine või korrutamine) siis, kui täisarvu väärtus ületab andmetüübile määratud maksimaalse väärtuse. Andmetüübi maksimaalne väärtus sõltub sellest, kui palju on antud tüübile eraldatud mäluruumi. Näiteks tüüp, millele on eraldatud 8-bitine mäluruum, suudab salvestada $2^8 = 256$ erinevat seisundit. Kui see tüüp on märgita (*unsigned*) arv, siis on võimalik jäädvustada täisarve vahemikus 0 kuni 255 või märgiga (*signed*) arvu puhul -128 kuni 127 [26, 27].

Täisarvu ületäitumine leiab aset, kui näiteks 8-bitisele märgita täisarvu maksimaalsele väärtusele 255 liita juurde täisarv 1. Probleemi olemus tuleb selgelt välja kasutades binaarset arvusüsteemi. Täisarvu 255 väärtus oleks 1111 1111 (8 bitti) ning seda suurendades ühe ühiku võrra, saame täisarvu 256 ehk binaaris 1 0000 0000 (9 bitti), mis ei mahu 8-bitisesse

[2] CWE lehekülg "Integer Overflow or Wraparound": https://cwe.mitre.org/data/definitions/190.html

mäluruumi. Leiab aset ümberlähtestamine (wraparound) ja mäluruumi salvestatakse binaararv 0000 0000, mille täisarvuline väärtus on 0.

Märgita täisarvu puhul on ümberlähtestamine C keele standardi järgi defineeritud tegevus ning seda võidakse kasutada taotluslikult. Ületäitumine märgiga täisarvude korral on määramata. Turvaanalüüsi seisukohalt tuleks siiski mõlemal juhul hoiatada, sest igasugune ületäitunud väärtus võib olla sinna tekkinud arendajale ootamatult ning põhjustada seega turvaauke [28].

2021. aasta CWE 25 kõige ohtlikuma tarkvara nõrkuse edetabelis [29] on täisarvu ületäitumine 12. kohal. Selle on põhjustanud käesoleva turvanõrkuse lai levik ning see, et täisarvu ületäitumine võib viia teiste tõsiste turvanõrkusteni nagu kuhja (*heap*) või puhvri (*buffer*) ületäitumine [26]. Seega oleks Goblintile oluline prioritiseerida antud turvanõrkuse tuvastamise implementeerimist.

Järgnevalt on esitletud üht uut testprogrammi (joonis 6), mis on aluseks täisarvu ületäitumise tuvastamise protsessi arendamiseks Goblintis (CWE-190 teste loodud kokku 3 tk).

```
1   #include<stdio.h>
2
3   void main()
4   {
5       char data;
6       fscanf(stdin, "%c", &data);
7       {
8           char result = data + 1;  // WARN: potential overflow
9           printf("%hhd\n", result);
10      }
11  }
```

Joonis 6. *Testprogramm "CWE190_Integer_Overflow__01.c".*

Joonisel 6 on näha testprogrammi, kus võib esineda täisarvu ületäitumist. Real 5 on deklareeritud muutuja `data`, mille andmetüübiks on `char`. See on tüüpiliselt 8-bitise salvestusruumiga andmetüüp, milles hoiustatakse tähemärke, kus igale tähemärgile vastab mingi kood (näiteks binaararvudes või märgiga detsimaalsüsteemi arvudes).

Programmi käivitamisel sisestab kasutaja käsureal tähemärgi ja see omistatakse muutujale `data` (rida 6). Täisarvu ületäitumise oht tekib real 8, kus toimub aritmeetiline tehe `data + 1`. Näiteks, kui kasutaja sisestab sümboli, mille detsimaalsüsteemi väärtus on 127, siis muutuja `data` uus väärtus peale programmi tööd oleks -128.

3.2. Täisarvu allakadu (CWE-191)

Täisarvu allakadu[3] võib käsitleda kui ületäitumisele vastastikust protsessi. Täisarvu allakadu esineb siis, kui aritmeetilise tehte järel tekkiv tegelik tulemus on suurem kui eeldatav tulemus. Täpsemini tähendab see seda, et andmetüübi kõige väiksemast võimalikust väärtusest minnakse arvu vähendava operatsiooni abil läbi maksimaalse piirväärtuse [27].

Täisarvu allakadu ja ületäitumine on oma loomuselt kaks väga sarnast turvanõrkust ning selle tõttu on ka riskifaktorid neil sama suured [27]. Analüsaatori seisukohast on oluline eristada kahte protsessi, sest see aitab arendajal kergemini leida lahendus tuvastatud nõrkusele.

Järgnevalt on esitletud üht uut testprogrammi (joonis 7), milles võib esineda täisarvu allakadu (CWE-191 teste loodud kokku 3 tk).

```
1   #include<stdio.h>
2
3   void main()
4   {
5       char data;
6       fscanf(stdin, "%c", &data);
7       if(-data < 0) // avoid potential overflow
8       {
9           char result = -data * 2; // WARN: potential underflow
10          printf("%hhd\n", result);
11      }
12  }
```

Joonis 7. T*estprogramm "CWE191_Integer_Underflow__02.c".*

[3] CWE lehekülg "Integer Underflow (Wrap or Wraparound)": https://cwe.mitre.org/data/definitions/191.htm.

Joonisel 7 on testprogramm, milles võib esineda täisarvu allakadu. Deklareeritakse 8-bitine `char`-tüüpi märgiga muutuja `data` (rida 5) ning kasutaja sisendist võetakse sümbol, mis salvestatakse `char`-ina mäluruumi (rida 6). *if*-tingimuslauses (rida 7) kontrollitakse, et `data` vastandväärtus oleks väiksem kui 0. See on vajalik, sest `data` vastandväärtust korrutatakse 2-ga (rida 9) ning kontrolli puudumisel võib arvu uus väärtus (muutuja `result`) olla väiksem kui eeldatav ehk tegu oleks ületäitumisega. Täisarvu allakadu esineb seega real 9 siis, kui sisestatud sümboli detsimaalsüsteemi vastandväärtus on väiksem kui -64.

3.3. Avaldise väärtus alati väär (CWE-570)

Alati väär avaldise väärtus[4] on turvanõrkus, mille tehniline mõju asetseb CWE lehekülje järgi "kvaliteedi halvenemise" (*quality degradation*) kategoorias. See tähendab, et riskifaktor pole nii suur kui näiteks andmejooksul või täisarvu ületäitumisel, kuid siiski võib kehv koodi kvaliteet põhjustada probleeme. Arendajale on koodi loetavus raskendatud ja silumine võtab kauem aega, tavakasutaja võib märgata programmi tavapäratut või ebaloogilist käitumist ning ründajale võib avaneda võimalus leida ootamatutes kohtades turvaauki [30].

Avaldise konstante väär väärtus tekib *if*-tingimuslausetes, kus muutujaid on kasutatud sellisel viisil, et programmi täitmine ei jõua kunagi *if*-lause sees olevate käskudeni ehk sealne sisu on tühi kood (*dead code*). Selline olukord võib juhtuda erinevatel põhjustel, näiteks:

- `if (alwaysFalse)` - deklareeriti kahendmuutuja `alwaysFalse` tõeväärtusega "false" ja selle väärtust kordagi koodis ei muudetud;
- `if (a == (a + 1))` - matemaatiliselt ebakorrektne tehe, mille väärtus on alati väär;
- `if (!b) { if (b) }` - kahendmuutuja `b` tõeväärtusel pole vahet, sest tegu on lausearvutuse tehtega, mille väljund on alati väär [31].

Järgnevalt on esitletud üht uut testprogrammi (joonis 8), milles avaldise väärtus jääb igas seisus vääraks (CWE-570 teste loodud kokku 3 tk).

[4] CWE lehekülg "Expression is Always False": https://cwe.mitre.org/data/definitions/570.html.

```c
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   void main()
5   {
6       // (0 <= uInt < UINT_MAX), uInt is pseudo-random
7       unsigned int uInt = (unsigned int)(rand() * 2);
8
9       if (uInt < 0) // WARN: expression is always false
10      {
11          printf("Never prints");
12      }
13  }
```

Joonis 8. *Testprogramm "CWE570_Expression_Always_False__02.c".*

Joonisel 8 on testprogramm, milles esineb turvanõrkus "avaldise väärtus alati väär". Peameetodis real 7 deklareeritakse märgita täisarv `uInt`, mille omistatud väärtus on pseudo-juhuslik (testi kontekstis pole see probleemiks). Märgita täisarv tähendab seda, et puuduvad negatiivsed väärtused.

Seega peab `uInt` arvväärtus jääma vahemikku 0 … `UINT_MAX` (ehk positiivse täisarvu maksimaalne mäluruumi mahtuv väärtus). Võttes arvesse seda tingimust, siis tuleb välja, et real 9 olev *if*-lause tagastab alati väärtuse "false", sest `uInt` ei saa ühelgi juhul olla negatiivne täisarv. Selle tulemusena tuleks real 9 fikseerida turvanõrkus.

3.4. Avaldise väärtus alati tõene (CWE-571)

Alati tõene avaldise väärtus[5] on tehnilise mõju ja loomuse poolest väga sarnane eelmises peatükis (3.3.) käsitletud turvanõrkusele. Erinevus seisneb selles, et alati tõese väärtuse puhul jõuab programmi täitmine igas olukorras *if*-tingimuslause konstruktsiooni sees olevate käskudeni. Selliseid situtatsioone esineb näiteks siis, kui:

- `if (alwaysTrue)` - deklareeriti kahendmuutuja `alwaysTrue` tõeväärtusega "true" ja selle väärtust kordagi koodis ei muudetud;

[5] CWE lehekülg "Expression is Always True": https://cwe.mitre.org/data/definitions/571.html.

- `if (3 == (2 + 1))` - matemaatiliselt korrektne tehe, avaldise väärtus on alati tõene;
- `if (int rand() <= INT_MAX)` - kontrollitakse, kas juhuslik täisarv on väiksem või võrdne kui sama andmetüübi maksimaalne väärtus, avaldis on alati tõene [31].

Järgnevalt on esitletud üht uut testprogrammi (joonis 9), milles avaldise väärtus jääb igas seisus tõeseks (CWE-571 teste loodud kokku 3 tk).

```
1    #include <stdio.h>
2    #include <stdlib.h>
3    #include <limits.h>
4
5    void main()
6    {
7        int intRand = rand();
8        if (intRand <= INT_MAX) // WARN: expression is always true
9        {
10           printf("Always prints\n");
11       }
12   }
```

Joonis 9. *Testprogramm "CWE571_Expression_Always_True__03.c".*

Joonisel 9 on näha testprogrammi, kus esineb turvanõrkus "avaldise väärtus alati tõene". Real 7 on deklareeritud (märgiga) täisarv `intRand`, millele omistatakse pseudo-juhuslik väärtus ehk sisuliselt `intRand`-i puhul on tegemist mingi suvalise täisarvuga. Järelikult jääks see juhuslik täisarv vahemikku `INT_MIN` … `INT_MAX`.

Real 8 toimub kontroll, kus võrreldakse, kas `intRand` väärtus on väiksem või võrdne kui `INT_MAX`. Suvalise täisarvu väärtus on igas olukorras väiksem või võrdne kui selle andmetüübi maksimaalne võimalik väärtus ning seega on real 8 olev if-lause väärtus alati "true" ja seal tuleks fikseerida turvanõrkus.

Kokkuvõte

Goblint on korrektne analüsaator mitmelõimeliste C-keelsete programmide staatiliseks uurimiseks. Goblint rakendab andmevooanalüüsi meetodit ja keskendub andmejooksude avastamisele. See tähendab, et teisi turvanõrkuseid analüüsis ei otsita või nendest ei teavitata.

Töö väljundina kirjutati skript, mida kasutati Goblinti töö hindamiseks Juliet testkomplektil, ning saadud hinnangute põhjal viidi läbi eeltöö analüsaatori täiustamiseks nii, et edasistel arendajatel oleks kergem implementeerida uute turvanõrkuste avastamist.

Skripti abiga teostati automaatset analüüsi Juliet komplekti testprogrammidele. Töö raames analüüsiti kokku 6110 C-keelset testfaili, millest 36 olid andmejooksu põhised ja teised uued turvanõrkused neljast erinevast CWE kategooriast. Goblinti analüüsi tulemused olid osaliselt ootuspärased: andmejookse mittesisaldavatest testfailidest turvanõrkuseid ei tuvastatud või nendest ei teavitatud, kuid andmejooksu katsetes fikseeriti turvanõrkuse olemasolu kõikides testprogrammi funktsioonides (s.t ka seal, kus andmejooksu tegelikult ei esinenud).

Esmapilgul võis neid tulemusi pidada valepositiivseteks, kuid komplekti testprogrammide lähemal uurimisel selgus, et probleem on tehnilisem. Juliet komplekti failides kasutatakse dünaamilisi lukke ja Goblint ei saa kinnitada, et tegu oleks kõikjal mujal programmis ühe kindla lukuga. Lahenduseks pakuti välja, et Goblintisse tuleks lisada unikaalsuse analüüs, millega oleks võimalik eristada dünaamilisi lukke teistest samalaadsetest.

Goblinti täiustamise eeltöö jaoks valiti välja neli uut turvanõrkust, mille puhul Goblint peaks hoiatama: täisarvu ületäitumine, täisarvu allakadu, avaldise väärtus alati väär ja avaldise väärtus alati tõene. Iga turvanõrkuse põhjal loodi uued testprogrammid, mille põhjal saavad Goblinti arendajad täiendada analüsaatorit, et tuvastataks rohkem nõrkuseid. Võrreldes olemasolevate Juliet komplekti testidega on uued kompaktsemad ja ei sõltu Juliet komplekti abifailidest. Kokku loodi 12 uut testprogrammi.

Nii skript kui ka uued programmid on mõeldud Goblinti projektis pikaajaliseks kasutamiseks regressioonitestimisel või olla uute edasiarenduste aluseks.

Viidatud kirjandus

[1] Ivo Gomes, Pedro Morgado, Tiago Gomes, Rodrigo Moreira. “An overview on the Static Code Analysis approach in Software Development,” 2009.

[2] Steve McConnell. “Code Complete: A Practical Handbook of Software Construction, Second Edition,” *Microsoft Press*, 2004.

[3] Jiang Zheng, Laurie Williams, Nachiappan Nagappan, Will Snipes, John P. Hudepohl, Mladen A. Vouk. “On the Value of Static Analysis for Fault Detection in Software,” *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, 2006.

[4] Liisa Sakerman, Rain Hallikas. “Overview of the advantages and disadvantages of static code analysis tools,” *Tartu Ülikooli Arvutiteaduse instituut*, 2021.

[5] Goblinti kodulehekülg. Veebilink: https://goblint.in.tum.de/ (vaadatud: 20.04.2021).

[6] Vootele Rõtov. “Time Partitioning in Goblint: Extending region analysis with happens-before information,” *Tartu Ülikooli Arvutiteaduse instituut*, 2016.

[7] Simmo Saan. “Abstraktsete domeenide omaduspõhine testimine,” *Tartu Ülikooli Arvutiteaduse instituut*, 2018.

[8] Juliet Test Suite’i kodulehekülg. Veebilink: https://samate.nist.gov/SRD/testsuite.php (vaadatud: 13.07.2021).

[9] CWE (Common Weakness Enumeration) kodulehekülg. Veebilink: https://cwe.mitre.org/ (vaadatud: 13.07.2021).

[10] Anders Møller, Michael I. Schwartzbach. “Static Program Analysis,” *Aarhus University, Denmark*, 2020.

[11] William Landi. “Undecidability of Static Analysis,” *Letters on Programming Languages and Systems, Vol. 1, No. 4*, 1992.

[12] Flemming Nielson, Hanne R. Nielson, Chris Hankin. “Principles of program analysis,” *Springer*, 2005.

[13] Mirjam Iher. “Nõrgima eeltingimuse staatiline analüüs pinukeeltele,” *Tartu Ülikooli Arvutiteaduse instituut*, 2019.

[14] Luca Cardelli. “Type Systems,” *Digital Equipment Corporation Systems Research Center*, 1996.

[15] Andre Sinisalu. “Java programmide staatiline intervallanalüüs raamistikus Põder,” *Tartu Ülikooli Arvutiteaduse instituut*, 2019.

[16] Darko Stefanovic, Danilo Nikolic, Dusanka Dakic, Ivana Spasojevic, Sonja Ristic. “Static Code Analysis Tools: A Systematic Literature Review,” *Proceedings of the 31st International DAAAM Symposium*, 2020.

[17] Alexander S. Gillis. “Static analysis (static code analysis),” *TechTarget,* 2020. Veebilink https://searchsoftwarequality.techtarget.com/definition/static-analysis-static-code-analysis (vaadatud: 06.07.2021).

[18] V. Benjamin Livshits, Monica S. Lam. “Finding security vulnerabilities in java applications with static analysis,” *In Proceedings of the 14th conference on USENIX Security Symposium - Volume 14 (SSYM'05)*, 2005.

[19] Gary McGraw. “Static analysis for security,” *IEEE Security & Privacy*, 2004.

[20] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, Ciera Jaspan. “Lessons from Building Static Analysis Tools at Google,” *Communications of the ACM (CACM)*, 2018.

[21] Vesal Vojdani, Kalmer Apinis, Vootele Rõtov, Helmut Seidl, Varmo Vene, and Ralf Vogler. “Static race detection for device drivers: the Goblint approach,” *In Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, 2016.

[22] Vesal Vojdani, Varmo Vene. “Goblint: Path-Sensitive Data Race Analysis,” *Tartu Ülikooli Arvutiteaduse instituut*, 2009.

[23] Shin'ichi Shiraishi, Veena Mohan, Hemalatha Marimuthu. “Test suites for benchmarks of static analysis tools,” *IEEE International Symposium on Software Reliability Engineering Workshops*, 2015.

[24] Andreas Wagner, Johannes Sametinger. “Using the Juliet Test Suite to compare Static Security Scanners,” *11th International Conference on Security and Cryptography*, 2014.

[25] Subprocess’i moodul. Veebilink https://docs.python.org/3/library/subprocess.html (vaadatud: 27.07.2021).

[26] Yang Zhang, Xiaoshan Sun, Yi Deng, Liang Cheng, Shuke Zeng, Yu Fu, Dengguo Feng. “Improving Accuracy of Static Integer Overflow Detection in Binary,” *Research in Attacks, Intrusions, and Defenses (RAID)*, 2015.

[27] Robert Auger. “Integer Overflows,” *The Web Application Security Consortium*, 2009. Veebilink http://projects.webappsec.org/w/page/13246946/Integer%20Overflows (vaadatud: 30.07.2021).

[28] Will Dietz, Peng Li, John Regehr, Vikram Adve. “Understanding Integer Overflow in C/C++,” *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, 2012.

[29] “2021 CWE Top 25 Most Dangerous Software Weaknesses”. Veebilink https://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html (vaadatud: 30.07.2021).

[30] “Dead Code: Expression is Always false,” *Fortify Taxonomy: Software Security Errors*. Veebilink https://vulncat.fortify.com/en/detail?id=desc.structural.java.dead_code_expression_is_always_false (vaadatud: 02.08.2021).

[31] **“**V560. A part of conditional expression is always true/false,” 2012. Veebilink https://pvs-studio.com/en/docs/warnings/v560/ [vaadatud: 02.08.2021].

Lisad

I. HTML-faili vaade brauseriaknas (vähendatud testkomplekt)

RESULTS

X Vulnerabilities detected

- No vulnerabilities detected

? Function not found, error

C/testcases/CWE190_Integer_Overflow
C/testcases/CWE366_Race_Condition_Within_Thread

Folder: C/testcases/CWE190_Integer_Overflow Go to top

Testcase	Good	Bad
CWE190_Integer_Overflow__int_rand_add_01.c	-	-
CWE190_Integer_Overflow__int_rand_add_02.c	-	-

Folder: C/testcases/CWE366_Race_Condition_Within_Thread Go to top

Testcase	Good	Bad
CWE366_Race_Condition_Within_Thread__global_int_01.c	X	X
CWE366_Race_Condition_Within_Thread__global_int_02.c	X	X

Go to top

II. Tabel uutest testprogrammidest koos lühikirjeldusega vea põhjustajast

Täisarvu ületäitumine (CWE-190)
CWE190_Integer_Overflow__01.c - *muutuja väärtuse suurendamine liitmistehtega*
CWE190_Integer_Overflow__02.c - *muutuja väärtuse (> 0) korrutamine juhusliku väärtusega (> 0)*
CWE190_Integer_Overflow__03.c - *muutuja väärtus võetakse ruutu, sisaldab lisaks CWE-570 nõrkust*
Täisarvu allakadu (CWE-191)
CWE191_Integer_Underflow__01.c - *muutuja väärtuse vähendamine lahutamistehtega*
CWE191_Integer_Underflow__02.c - *muutuja vastandväärtuse (< 0) suurendamine korrutustehtega*
CWE191_Integer_Underflow__03.c - *muutuja juhusliku negatiivse väärtuse suurendamine korrutustehtega for-tsüklis*
Avaldise väärtus alati väär (CWE-570)
CWE570_Expression_Always_False__01.c - *globaalsele kahendmuutujale omistatud väärtus "false", mida ei uuendata, tingimuslauses kontrollitakse, kas väärtus on "true"*
CWE570_Expression_Always_False__02.c - *märgita juhusliku täisarvu suurendamine korrutustehtega, tingimuslauses kontrollitakse, kas väiksem kui 0*
CWE570_Expression_Always_False__03.c - *tingimuslauses matemaatiliselt ebakorrektne võrdus (a = a - 1)*
Avaldise väärtus alati tõene (CWE-571)
CWE571_Expression_Always_True__01.c - *globaalsele kahendmuutujale omistatud väärtus "true", mida ei uuendata, tingimuslauses kontrollitakse, kas väärtus on "true"*
CWE571_Expression_Always_True__02.c - *märgita juhusliku täisarvu suurendamine korrutustehtega, tingimuslauses kontrollitakse, kas suurem/võrdne kui 0*
CWE571_Expression_Always_True__03.c - *tingimuslauses kontrollitakse, kas juhuslik täisarv on väiksem/võrdne kui maksimaalse täisarvu väärtus*

Lihtlitsents lõputöö reprodutseerimiseks ja üldsusele kättesaadavaks tegemiseks

Mina, **Raner Lebbin**,

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

“**Programmianalüsaator Goblinti hindamine Juliet testkomplektiga**,”
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 3.0, 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.

Raner Lebbin

04.08.2021