TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

NIKO LAPPALAINEN

KOHINAINJEKTION TOTEUTUS VIRHEENETSINTÄOHJELMALLA

Kandidaatintyö

Tarkastaja: Yliopisto-opettaja Marko Helenius
Jätetty tarkastettavaksi
7. lokakuuta 2018

TIIVISTELMÄ

NIKO LAPPALAINEN: Kohinainjektion toteutus virheenetsintäohjelmalla
Tampereen teknillinen yliopisto
Kandidaatintyö, 21 sivua, 6 liitesivua
Lokakuu 2018
Tietotekniikan kandidaatin tutkinto-ohjelma
Pääaine: Ohjelmistotekniikka
Tarkastaja Marko Helenius

Avainsanat: rinnakkaisuus, testaaminen, kohinainjektio, virheenetsintäohjelmat, GDB

Tässä työssä käsitellään rinnakkaisten C ja C++ kielisten ohjelmien testaamista käyttäen epädeterminististä kohinainjektiotekniikkaa. Työssä osoitetaan, että tekniikka on mahdollista toteuttaa käyttäen samoja periaatteita, joihin virheenetsintäohjelmat perustuvat.

Työn alussa on kerätty tietoa kohinainjektiosta useista eri lähteistä, jotta tekniikan vaatimuksia ja rajoituksia voitaisiin ymmärtää. Työssä esitetään myös lyhyesti GNU-virheenetsintäohjelman, eli GDB:n toimintaa. Työssä esitetään ehdotuksia, kuinka virheenetsintäohjelmilla voitaisiin toteuttaa samoja ominaisuuksia kuin muissa kohinainjektiototeutuksissa.

Työn yhteydessä toteutettiin kohinainjektiokehyksen prototyyppi käyttäen apuna GDB-virheenetsintäohjelman Python-rajapintaa. Prototyypillä saadut tulokset eivät olleet täysin positiivisia, mutta ne kuitenkin vahvistavat tässä työssä esitettyjen ehdotusten soveltuvuutta.

SISÄLLYSLUETTELO

.

LYHENTEET JA MERKINNÄT

GDB	engl. GNU debugger, debug-ohjelma ja komentotulkki
GCC	engl. GNU Compiler Collection, kääntäjäkokoelma
IBM	engl. International Business Machines, teknologiayhtiö
IPC	engl. Inter-process communication, protokollatyyppi

1. JOHDANTO

Monisäikeisten ohjelmien suunnittelu on vaikeaa, koska monisäikeisten ohjelmien suoritusjärjestystä ei voida ennustaa tarkasti. Suunnitteluvirheet monisäikeisissä ohjelmissa voivat johtaa uuden tyyppisiin virheisiin, joista voi seurata vakaviakin seuraamuksia. Näin kävi esimerkiksi Therac-25 tapauksessa [1], jossa säteilyhoitolaitteen ohjelmiston kilpailutilanne aiheutti tappavia säteilymyrkytyksiä. Rinnakkaisuutta on kuitenkin vaikeaa testata sen ennustamattomuuden vuoksi. Koska ohjelman suoritusjärjestystä ohjaava vuorottaja (scheduler) toimii tyypillisesti käyttöjärjestelmätasolla, suoritusjärjestys riippuu ajoympäristöstä ja ympäristön tilasta ajohetkellä. On mahdollista, että ohjelma käyttäytyy aivan eri lailla sovelluskehittäjän henkilökohtaisella tietokoneella, testaamiseen tarkoitetulla palvelimella ja lopullisessa tuotantoympäristössä. Rinnakkaisuuden luotettava testaaminen vaatii omat tekniikkansa.

Kohinainjektio (noise injection) on yksi tapa testata rinnakkaisuutta. Sen tarkoituksena on tuottaa monia eri ajojärjestyksiä häiritsemällä vuorottajan toimintaa, mikä tehdään tuottamalla satunnaisia tai osittain satunnaisia viiveitä ja säievaihtoja ohjelman ajon aikana. Letkon [2] mukaan kohinainjektiolla on monia etuja verrattuna muihin tekniikoihin. Hän kertoo tekniikan olevan riittävän luotettava, mutta toisaalta myös verrannollisen kevyt resurssivaatimuksiltaan. Hän myös kuvaa tekniikkaa riittävän kypsäksi sovellustuotannon käyttöön, mikä todennäköisesti tarkoittaa, että tekniikkaa on hänen mielestään tutkittu tarpeeksi.

Kohinainjektion toteuttaminen ei ole yksinkertaista erityisesti konekielisten ohjelmien testaamiseen. Tässä työssä tutkitaan, kuinka kohinainjektiota voitaisiin toteuttaa käyttämällä virheenetsintäohjelman (debugger) ominaisuuksia. Lisäksi haluttiin selvittää toteutustavan haittoja ja hyötyjä verrattuna muihin toteutuksiin. Tutkimus suoritettiin toteuttamalla yksinkertainen kohinainjektioprototyyppi, joka kirjoitettiin käyttämällä GNU-virheenetsintäohjelman ohjelmointirajapintaa. Prototyypin toteuttaminen vaati ensin kohinainjektion toiminnan ja vaatimuksien selvitystä, joka tehtiin etsimällä tietoa kohinainjektiota käsittelevistä tutkimusartikkeleista ja muista julkaisuista.

Aikaisemmat kohinainjektiota hyödyntävät työkalut toteuttavat säikeiden ohjauksen ja analysoinnin lisäämällä omaa koodiaan testattavaan ohjelmaan. Samoin toimii myös Fiedorin ja Vorjnarin kirjoittama ANaConDA-kehystyökalu [3], joka käyttää kohinainjektiota C ja C++ kielisten ohjelmien testaamiseen. Työkalua käytetään tässä työssä vertailupisteenä.

Toteutettua prototyyppiä ei saatu toimimaan täysin toivotulla tavalla. Prototyypin tulosteista nähdään, että se tuottaa kohinaa tarkoituksen mukaisesti yksittäisissä testiajoissa, mutta sitä ei voida käyttää testitapausten toistuvaan ajoon lukkiutumisongelman takia. Tästä johtuen sen toimintaa ei voitu verrata ANaConDA-kehystyökalun toimintaan käytännössä. Prototyyppi kuitenkin vahvistaa, että ehdotettuja tekniikoita voitaisiin käyttää kohinainjektion toteuttamiseksi. Toteutustavan suurimman edun arvioitiin olevan virheenetsintäohjelman valmis ympäristöriippumaton rajapinta, mikä mahdollisti kohinainjektion toteutuksen helposti pienellä koodimäärällä.

Luvussa 2 kuvataan kohinainjektion toimintaa ja määritetään vaatimuksia kohinatestiohjelmalle. Luku 3 kuvailee kohinainjektion käytännön toteutustapoja. Luvussa 4 esitellään lyhyesti GNU-virheenetsintäohjelmaa ja sen oleellisia ominaisuuksia Luvussa 5 esitetään mahdollisia tapoja käyttää ominaisuuksia kohinainjektion toteuttamiseksi. Lisäksi luvussa 5.3 kuvataan toteutetun prototyypin toimintaa. Tutkimuksessa saatuja tuloksia kerätään ja arvioidaan luvussa 6.

2. KOHINAINJEKTIO

Tässä luvussa käsitellään kohinainjektion toimintaa. Aluksi selitetään kohinainjektion perusperiaatteet, ja seuraavaksi kerrotaan tekniikan oleellisista ongelmista ja niiden ratkaisuista liittyen kohinan tuottamiseen ja testauksen kattavuuteen.

2.1 Toimintaperiaatteet

Rinnakkaisessa ohjelmassa mahdollisten lomittaisuuksien (interleaving) määrä on hyvin suuri eli säikeillä on monta tapaa vuorovaikuttaa toisiinsa riippuen suoritusjärjestyksestä. Tietyt lomittaisuudet voivat olla todennäköisempiä kuin toiset. Rinnakkaisuusvirheet voivat kuitenkin esiintyä vain harvinaisemmissa lomittaisuuksissa, jotka eivät ole helposti tuotettavissa normaalissa testitapaustestauksessa. Kohinainjektion tarkoituksena on tuottaa monipuolisia lomittaisuuksia, jolloin voitaisiin löytää myös harvinaisemmat virhetilanteet. Käytännössä eri lomittaisuudet saadaan aikaan ajamalla samaa testitapausta toistuvasti tuottamalla samalla satunnaisia tai osittain satunnaisia viiveitä ja suoritusvuorovaihtoja eri säikeiden välillä. [2]

Kohinainjektio on verrattavissa deterministisiin rinnakkaisuuden testaamisen tekniikoihin, joiden pääperiaatteena on täysin hallita testattavan ohjelman vuorotusta. Deterministiset tekniikat pitävät kirjaa kaikista mahdollisista lomittaisuuksista ja suorittavat systemaattisesti uusia lomittaisuuksia. Systemaattinen testaaminen tuottaisi satunnaista kohinainjektiota parempia tuloksia, mutta sen resurssivaatimukset tekevät siitä tehottoman vaihtoehdon laajempien kokonaisuuksien testaamisessa. [4] Deterministiset tekniikat sopivat siksi paremmin yksikkötestaamiseen. Koska kohinainjektion resurssivaatimukset ovat huomattavasti pienemmät kuin deterministisen tekniikoiden [4], se sopii paremmin integraatiotestaukseen.

Kohinainjektion satunnaisesta luonteesta ja lomittaisuuksien suuresta joukosta seuraa ongelma, että kaikkia mahdollisia lomittaisuuksia ei ole mahdollista tuottaa testatessa. Toisin sanoen testit eivät voi saavuttaa täyttä kattavuutta. Kattavuuden mittaaminen on tärkeää, jotta testiajot osattaisiin lopettaa, kun riittävä kattavuus on saavutettu [4]. Kattavuuden mittaamiseksi on kehitetty mittareita.

Jotta satunnaista kohinaa voitaisiin tuottaa, ainakin kaksi kohinan ominaisuutta täytyy määritellä. Yksi oleellinen ongelma on kohinan sijoitus (placement), joka määrittelee, mihin ja milloin viiveitä tuotetaan. Toiseksi täytyy myös määritellä, millä mekanismeilla kohinaa tulisi tuottaa ja kuinka kauan sen tulisi kestää eli kuinka kohinaa satunnaiste-

taan (seeding). [4] Molempien ongelmien ratkaisemiseksi on esitetty heuristiikkoja [5], joiden tavoitteena on tuottaa mahdollisimman paljon oleellisia lomittaisuustilanteita.

2.2 Kattavuuden mittaaminen

Ajettujen testien kattavuus pitää olla mitattavissa, jotta voidaan saada tietoa testitulosten luotettavuudesta. Perinteisessä testaamisessa tietyn testin suorittaminen yleensä tarkoittaa, että kyseinen testi on katettu. Kohinatesteissä samaa testiä ajetaan monta kertaa, jotta eri suoritusjärjestyksiä saataisiin testattua. Pelkkä testin uudelleen ajaminen ei lisää kattavuutta, koska säikeet saattavat käyttäytyä lähes täysin samalla tavalla eri ajokerroilla. Rinnakkaisuustestien kattavuuden mittaamiseksi on kehitetty mittareita, jotka voidaan karkeasti jakaa kahteen [4] luokkaan: yleisiin mittareihin ja virheenhavaitsemisalgoritmeihin perustuviin mittareihin.

Yleiset kattavuusmittarit ovat laajennettuja versioita perinteisistä mittareista. Mittareissa on määritelty kattavuustehtäviksi tiettyjä ohjelmasijainteja, kuten synkronoituja lohkoja. Kun testin suoritus saapuu näihin sijainteihin, testi raportoi säikeen kokemista rinnakkaisuustapahtumista, kuten vaihdoista tai synkronoinneista. Kun testiajossa saavutetaan uusi ohjelmasijainnin ja rinnakkaisuustapahtuman yhdistelmä, testiajon kattavuus kasvaa. [4] Heuristiikat vaativat, että testit voivat tunnistaa ja raportoida rinnakkaisuustapahtumista.

Toinen tapa mitata kattavuutta perustuu ohjelman dynaamiseen analyysiin. Rinnakkaisuuden testaamiseksi on kehitetty työkaluja, jotka analysoivat ohjelman toimintaa ja havaitsevat virhetilanteita dynaamisesti ohjelman ajon aikana. Näiden työkalujen huono puoli on se, että ne voivat raportoida virheen, vaikka virhettä ei olisi tapahtunut, mikä vaikeuttaa niiden käyttöä rinnakkaisuuden testaamiseen sellaisenaan. Samoja algoritmeja voidaan kuitenkin käyttää mittaamaan kattavuutta, jolloin jokainen algoritmin tunnistama virhealtis ohjelmakohta kasvattaa kattavuutta. [4] Näiden heuristiikkojen vaatimukset riippuvat käytetystä algoritmista, mutta yleisesti ne vaativat ilmoituksia ainakin muistin käsittelystä.

Koska kohinainjektiotestaus on luonteeltaan satunnaista, testit eivät voi saavuttaa täyttä kattavuutta, vaikka niitä ajettaisiin äärettömän monta kertaa. Testitapaukset voitaisiin katsoa testatuiksi tietyn ajan kuluttua, mutta se ei antaisi takuuta testien kattavuudesta. Toisaalta on vaikeaa määritellä mitään tiettyä kattavuusmittareihin perustuvaa kynnystä, koska sitä ei välttämättä koskaan saavutettaisi tekniikan satunnaisuuden vuoksi. Sherman et al. [6] esittävät ongelmaan ratkaisuksi saturaatioon perustuvan testauksen, missä testaaminen lopetetaan, kun sen kattavuus ei enää merkittävästi kasva. Fiedorin et al. [4] suorittaman kokeen mukaan saturaation saavutus voi kuitenkin kestää hyvin kauan riippuen kattavuusmittareista. Joka tapauksessa kohinainjektiotestaus asettaa perinteistä testaamista merkittävämpiä aikavaatimuksia.

2.3 Sijoitusongelma

Jotta oleellisia lomittaisuuksia saataisiin tuotettua, kohinaa täytyy tuottaa oikeissa ohjelmasijainneissa. Kohinan lisääminen kaikkiin mahdollisiin sijainteihin ei ole tehokasta ja tuottaa vain vähän relevantteja lomittaisuuksia [5]. Lisäksi on tärkeää määritellä, millä ajokerroilla kohinaa tuotetaan.

Yksinkertaisin sijoitusheuristiikka on täysin satunnainen, mitä Fiedor et al. [4] kutsuvat *random-all* -heuristiikaksi. Täysi satunnaisuus tarkoittaa, että jokaisessa ohjelmasijainnissa, kuten funktiossa, on yhtä suuri todennäköisyys tuottaa kohinaa. Kun Fiedor et al. [4] suorittivat kokeen, jossa vertailtiin eri heuristiikkojen tehokkuutta löydettyjen virheiden kannalta, *random-all* suoriutui suhteellisen hyvin, ja se onnistui löytämään monipuolisia rinnakkaisuusvirheitä. Heuristiikka vaatii mahdollisuuden kaapata ohjelman suorituksen jokaisen funktiokutsun yhteydessä.

Toinen yksinkertainen Fiedor et al. kuvaama sijoitusheuristiikka on *SharedVar-all*, mikä on laajennus *random-all*-heurisiikasta. Siinä kohinaa tuotetaan vain jaettujen muuttujien käytön yhteydessä. [4] Sen avulla on mahdollista löytää rinnakkaisuusvirheitä, jotka liittyvät muuttujien käsittelyyn, kuten kilpailutilanteita. Heuristiikka vaatii, että ohjelmasta voidaan tunnistaa säikeiden yhteisessä käytössä olevat jaetut muuttujat.

2.4 Satunnaistamisongelma

Satunnaistamisheuristiikat määrittelevät ne tavat, joilla säievaihtoja saadaan aikaan. Heuristiikasta riippuen, suoritusvuorossa oleva säie voi yksinkertaisesti luovuttaa suoritusvuoronsa, se voidaan laittaa nukkumaan tietyksi aikaa tai se voidaan jättää silmukkaan tietyksi määräksi iteraatioita. [4] Suurin osa säiemallia toteuttavista kirjastoista ja kielistä tarjoavat valmiit keinot suoritusvuoron luovuttamiseen (yield) ja nukuttamiseen (sleep), mutta ne tyypillisesti vaativat, että niitä kutsutaan siitä säikeestä, johon halutaan vaikuttaa.

Fiedor et al. [4] esittämien heuristiikkojen nimistä voidaan päätellä, että ne on suoraan johdettu yleisistä säiekirjastojen funktionimistä. Heuristiikat toteutetaan siten, että testattava ohjelma kutsuu kyseisiä funktioita. Siksi ne vaativat, että testattava koodi instrumentoidaan. Instrumentointi tarkoittaa, että kohinaa toteuttava koodi lisätään testattavan ohjelman koodiin.

Useimmat satunnaistamisheuristiikat vaativat parametrinaan kohinan voimakkuuden [2]. Voimakkuus viittaa heuristiikasta riippuen siihen, kuinka monta säikeenvaihtoa tehdään, kuinka kauan säie on nukkuvassa tilassa tai kuinka monta iteraatiota säie on silmukassa [4]. Koetulokset [4] osoittavat, että kohinan voimakkuudella voi olla suuri merkitys virheiden löytämisen todennäköisyyteen, ja useissa tapauksissa voimakkaampi kohina antaa parempia tuloksia. Kohinan voimakkuutta on vaikeaa määritellä ennen

ajoa, sillä ei ole olemassa yleisesti parasta voimakkuusarvoa, koska optimaalinen voimakkuus todennäköisesti riippuu testattavasta ohjelmasta. Fiedor et al. [4] suorittamissa kokeissa esimerkiksi *sleep* heuristiikan voimakkuus oli enimmillään muutaman kymmenen millisekunnin luokkaa, mutta testattavan ohjelman optimaalinen viive voi olla sitä pidempikin.

2.5 Heuristiikkojen valinta

Kuten aikaisemmin viitattiin, eri heuristiikat soveltuvat parhaiten tietyn tyyppisten rinnakkaisuusvirheiden etsimiseen. Normaalissa sovelluskehityksessä ei kuitenkaan todennäköisesti voida ennustaa, onko koodissa atomisuusvirheitä, nälkiintymistä vai muita rinnakkaisuusvirheitä. Ohjelman testaaminen yhdellä heuristiikalla ei riitä kattavaksi testiksi. Lisäksi testaaja ei voi itse suorittaa optimaalista parametrisointia. Parhaat tulokset saadaan, jos valittuja heuristiikkoja ja parametreja säädetään testaamisen aikana [4].

Heuristiikkojen asettaminen on hakuongelma, johon voidaan käyttää hakualgoritmeja [4]. Hakualgoritmien toteuttaminen vaatisi, että kohinatestiohjelma pystyisi vaihtamaan käytettyjä heuristiikkoja vapaasti. Tämä asettaa omia vaatimuksiaan kohinaohjelmalle. Sijoitus- ja satunnaistamisheuristiikkojen toteutukset on oltava toisistaan riippumattomia, ja ohjelman pitää tukea myös mahdollisuutta käyttää useita samantyyppisiä heuristiikkoja.

3. KOHINAINJEKTION TOTEUTTAMINEN

Kohinainjektion toteuttaminen ei ole helppoa, koska se vaatii epätyypillisiä toimintoja testeiltä. Ohjelmointikielten säiekirjastot eivät tyypillisesti tarjoa ominaisuutta muokata säikeen toimintaa sen ulkopuolelta. Jotta säikeen toimintaa voitaisiin ohjata ja sen toiminnasta saataisiin tietoa, säikeen täytyy normaalin ajonsa yhteydessä kutsua kohinainjektiota toteuttavia rutiineja (routine). Nämä rutiinit eivät todennäköisesti ole toivottavia ohjelman tuotantoversiossa.

Fiedor et al. [4] mukaan paras tapa toteuttaa kohinainjektiota on instrumentoida koodi, eli lisätä tarvittavat rutiinit testattavan ohjelman koodiin. Instrumentaatiota voi tehdä suoraan lähdekoodiin, käännettyyn binäärikoodiin, tai käännettyyn tulkittavaan koodiin, kuten Javan tavukoodiin (bytecode). Fiedor et al. [4] esittävät, että instrumentointi tulisi suorittaa joko valmiiksi käännettyyn tulkittavaan koodiin tai binäärikoodiin, koska siten vältettäisiin kääntäjän optimointien vaikutus. Optimointi voidaan kuitenkin ottaa kokonaan pois käytöstä ainakin GCC kääntäjästä. Toisaalta käännettyä koodia voidaan instrumentoida, vaikka alkuperäistä lähdekoodia ei olisi saatavilla [4]. Tämä on merkittävä etu, koska se mahdollistaa kohinan tuottamisen myös kolmannen osapuolen valmiiksi käännetyissä kirjasoissa.

Käännettyä koodia on suhteellisen helppo instrumentoida, kun on kyse Javan konekoodin tapaisista tulkittavista koodeista. Koodia analysoimalla saadaan tietoa esimerkiksi siitä, onko tietty muuttuja säikeiden yhteisessä käytössä. IBM:n ConTest-työkalu toteuttaa kohinainjektiota Java-ohjelmille. Binäärikoodin analysointi on vaikeampaa. Esimerkiksi, ohjelman pinossa (stack) tallennetuista muuttujista on vaikeaa päätellä, onko muuttuja kokonaisluku, vai osoitin yhteiseen tietueeseen. [4] Fiedor ja Vojnar [7] ehdottivat, että tässä analyysissa voisi käyttää apuna debug-informaatiota. Fiedor ja Vojnar [3] ovat kirjoittaneet C-kielille kohinainjektiota toteuttavan ANaConDA-kehystyökalun.

Koodin instrumentointi voi olla staattista, eli ennen ajoa tapahtuvaa, tai dynaamista, mikä tapahtuu ajon aikana. Staattinen instrumentointi on tehokkaampaa, mutta se ei pysty reagoimaan muuttuvaan koodiin [7]. Lisäksi instrumentointi voi aiheuttaa ongelmia dynaamisesti linkitettyjen kirjastojen kanssa. Dynaamisen instrumentaation huono puoli on sen hitaus [7]. ANaConDA:a käytettäessä ohjelma toimii merkittävästi hitaammin [3]. Toisaalta kohinan tuottaminen tarkoittaa aina testattavan ohjelman hidastumista, koska siinä säikeitä tarkoituksella estetään suorittamasta koodiaan.

Säikeiden hallinta ja analyysi eivät välttämättä tarvitse monimutkaista instrumentaatiota. Esimerkiksi Linux-ympäristöt tarjoavat ptrace-systeemikutsun, jolla voidaan pysäyt-

tää aliprosessin säikeitä, analysoida ja muuttaa niiden muistia ja rekistereitä ja rajoittaa niiden suoritusta yhteen konekäskyyn kerrallaan. Koska ANaConDA perustuu Pin-työkaluun [3], myös se käyttää ptrace-toiminnallisuutta välillisesti [8]. Se ei kuitenkaan hyödynnä toimintoa muuhun kuin koodin instrumentointiin.

Kohinainjektio ei suoraan ota kantaa, kuinka virhetilanteet tunnistetaan. Sen tarkoitus on vain kasvattaa virhetilanteen mahdollisuutta. Virhetilanteet täytyy tunnistaa muilla tavoin. Yksi yleisesti käytetty testaustekniikka on testitapausmalli (test case), jossa tiettyä toiminnallisuutta kutsutaan tietyillä syötteillä, jonka jälkeen saatuja tuloksia verrataan odotettuihin tuloksiin. Yksinkertaisin tapa toteuttaa kohinainjektiotestejä on suorittaa testitapauksia toistuvasti syöttäen samalla järjestelmään kohinaa. Yksi merkittävä kohinainjektion etu on, että siihen voidaan käyttää valmiita testitapauksia ilman merkittäviä muutoksia [2].

4. GNU-VIRHEENETSINTÄOHJELMA

GDB on Free Software Foundation, Inc. -yhtiön hallitsema vapaan lähdekoodin virheenetsintäohjelma, joka on julkaistu GNU-hankkeen yleisen lisenssin alaisuuteen. GDB mahdollistaa testattavaa ohjelmaa suorittavan prosessin pysäyttämisen, analysoimisen, sekä hallitsemiseen. GDB-yhteydessä tähän prosessiin viitatessa puhutaan aliprosessista (inferior). GDB tarjoaa myös toimintoja monisäikeisten ohjelmien käsittelemiseen. [9]

GDB on tyypillinen virheenetsintäohjelma, jonka toiminta perustuu aliprosessin pysäyttämiseen silloin, kun aliprosessi tuottaa käsittelemättömän virhesignaalin. Kun aliprosessin yksikkö on pysäytetty, GDB voi lukea sen rekistereitä ja pinoa. [9] Tämä mahdollistaa esimerkiksi muuttujien arvon tarkastelun. Lisäksi GDB:llä on mahdollista ajaa aliprosessia yksi konekäsky tai koodirivi kerrallaan [9].

GDB:llä on mahdollista ajaa monisäikeistä ohjelmaa kahdella eri tavalla. Yksi tapa pysäyttää kaikki säikeet, kun pysäytystapahtuma havaitaan. [9] Tämä ei ole toivottavaa kohinainjektion kannalta, koska sillä ei saataisi aikaan merkittäviä viive-eroja säikeiden välillä. Toinen tapa pysäyttää vain sen säikeen, johon pysähdystapahtuma kohdistui, ja antaa muiden säikeiden jatkaa normaalisti. GDB:n termi jälkimmäiselle on Non-Stop-moodi, ja se aktivoidaan kutsumalla [9] GDB-komentoja ohjelman 1 mukaisesti.

```
(gdb) set target-async 1
(gdb) set pagination off
(gdb) set non-stop on
```

Ohjelma 1. *Non-Stop moodin käyttöönotto.*

Vaikka Non-Stop-moodi olisi aktivoitu, GDB käsittelee komentoja oletusarvoisesti synkronisesti [9]. Tämä tarkoittaa, että jos pysähtynyttä säiettä jatketaan normaalilla `continue`-käskyllä, GDB ei ota vastaan muita käskyjä ennen kuin kyseinen säie pysähtyy uudelleen tai lopettaa suorituksensa. GDB:lle voi kuitenkin erikseen määritellä, että komentoja halutaan kutsua asynkronisesti lisäämällä &-kirjain komentomerkkijonon perään [9]. Tämä mahdollistaa säikeiden jatkamisen toisistaan riippumatta.

GDB on komentotulkki, eli se määrittelee oman ohjelmointikielensä. Kielessä on määritelty muun muassa muuttujat, ehtolauseet sekä silmukat [9]. Kieli on kuitenkin hyvin yksinkertainen ja rajoittunut. GDB:n toiminnallisuutta on mahdollista laajentaa käyttämällä muita ohjelmointikieliä. Tässä työssä keskitytään GDB:n tarjoamaan Python-ohjelmointirajapintaan. Jos GDB käännetään lähdekoodista, Python-tuki täytyy ottaa käyttöön konfiguraatiota tehdessä optiolla `--with-python` [9]. Joissakin Linux-

distribuutioissa toiminnallisuus on valmiiksi käytössä paketinhallintaohjelmalla asennettavassa versiossa. Python-komentosarjoja voi käyttää GDB:ssä käynnistämällä GDB optiolla `-x polku_skriptiin.py` tai ajon aikana kirjoittamalla GDB-komentotulkkiin ”`python`” [9].

GDB:n Python-tulkki ei etsi Python-moduuleja normaalin Pythonin tapaan samasta hakemistosta, jossa käynnistetty komentosarja sijaitsee. Käyttöohjeessa [9] suositellaan, että uudet Python-komentosarjat asennettaisiin tiettyyn hakemistoon, josta GDB osaa hakea moduuleita. Testiympäristössä hakemisto oli ”`usr/share/gdb/python/`”. Kehitysvaiheessa moduuleita voi olla kuitenkin helpompaa muokata kehittäjän määrittelemässä hakemistossa. Testitoteutuksessa todettiin, että yksinkertaisin tapa ottaa omia moduuleita käyttöön oli lisätä moduulien hakemisto Pythonin hakupolkuun.

5. KOHINATOTEUTUS GDB:LLÄ

Tässä luvussa kuvataan, kuinka GDB:n komennoilla voidaan toteuttaa kohinainjektiotyökalu. Fiedor ja Vojnar [7] kuvasivat joitakin perusvaatimuksia kohinainjektio- ja analyysityökalulle. Tämän luvun lopussa tarkastellaan kirjoitetun GDB-kohinatestausprototyypin toteutuksen onnistumista. Luvussa keskitytään tarkastelemaan *random-all* ja *sleep* heuristiikkojen toteutuksia, koska ne ovat suhteellisen yksinkertaisia toteuttaa GDB:llä.

5.1 Kohinan tuottaminen

GDB:n toiminta perustuu aliprosessin pysäyttämiseen. Kun Non-Stop -moodi on käytössä, pysäyttäminen kohdistuu vain yhteen aliprosessin säikeistä kerrallaan. Pysäytetyn säikeen käyttäytymistä ei ole selkeästi kuvattu ainakaan GDB:n käyttöohjeissa tai Linuxin manuaalisivuilla. On oletettavaa, että pysäytetty säie vapauttaa prosessoriresurssinsa muiden säikeiden käyttöön. Jos resursseja ei vapautettaisi, testaava prosessori saataisiin lukkiutumaan pysäyttämällä yhtä monta säiettä, kuin prosessorilla on ytimiä. Lukkiutuminen ei ole kuitenkaan mahdollista, sillä tätä työtä varten tehdyssä testiympäristössä onnistuttiin pysäyttämään kymmeniä säikeitä samanaikaisesti ilman ongelmia. Pysäytetyn säikeen tila on siten verrattavissa säikeeseen, joka on kutsunut sleep-funktiota. Säikeen pysäyttäminen GDB:llä mahdollistaa siten *sleep*-heuristiikan toteuttamisen.

GDB mahdollistaa ainakin neljä eri tapaa pysäyttää aliprosessin ajo. Ennalta määritellyt pysäytys- (breakpoint) ja tarkkailupisteet (watchpoint) pysäyttävät ohjelman, kun jokin ehto toteutuu. [9] Näitä kahta tapaa käsitellään myöhemmin tässä luvussa tarkemmin. Kolmas tapa on tuottaa pysäytys ajon aikana ilman varsinaisia ehtoja. Jos aliprosessia suoritetaan asynkronisesti, prosessin voi pysäyttää `interrupt`-komennolla. Lisäksi GDB pysäyttää ohjelman aina kun ohjelmasta vuotaa ulos käsittelemätön signaali. [9] Tämä ei ole kuitenkaan helposti halittava pysäytystapa.

Pysäytyspisteet ovat todennäköisesti sopivin tapa kohinainjektio-ohjelmalle, koska ne mahdollistavat sijoitus- ja satunnaistamisheuristiikkojen erottamisen toisistaan. Sijoitusheuristiikat voivat yksinkertaisesti valita sijainnit, joihin pysäytyspisteitä sijoitetaan, minkä jälkeen satunnaistamisheuristiikat voivat määritellä pysäytyspisteiden käyttäytymisen pysähdystapahtumissa. Tämän vuoksi ne valittiin prototyypin toteutukseen pysäytysmekanismiksi.

5.1.1 Pysäytys- ja tarkkailupisteet

Yksi hyvin yleisesti käytetty GDB:n ominaisuus on pysäytyspisteet, jotka pysäyttävät ohjelman tai säikeen ajon, kun säie saapuu tietylle koodiriville. GDB Python API:ssa pysäytyspisteitä kuvaa luokka, jonka rakentajalle annetaan argumenttina pysäytyspisteen sijainti. Jos luokasta peritään aliluokka, sille voidaan määritellä pysähdystilanteessa suoritettavaa logiikkaa. [9]

Pysäytyspisteet eivät välttämättä liity tiettyyn koodiriviin. GDB mahdollistaa myös ohjelman pysäytyksen, kun ohjelma käsittelee tiettyä tietuetta. Tällaista pysäytyspistettä kutsutaan tarkkailupisteeksi. Tarkkailupiste voi pysäyttää ajon silloin, kun tietueen arvoa luetaan, siihen kirjoitetaan, tai molemmissa tapauksissa. [9] Toimintoa testattaessa havaittiin, että tarkkailupisteet toimivat myös viittauksia ja osoittimia käytettäessä.

GDB:n pysäytystoiminnot eivät merkittävästi vaikuta testattavan ohjelman tehokkuuteen. Pysäytyspisteet eivät perustu ohjelman dynaamiseen analyysiin, vaan virhesignaalien kaappaamiseen. Tarkkailupisteet puolestaan käyttävät nopeita laitteistotason toteutuksia sovellustoteutuksen sijasta niissä ympäristöissä, joissa se on mahdollista. Tarkkailupisteet toimivat oikein monisäikeisissä ohjelmissa vain, jos ne ovat laitteistotasoisia. [9]

Valitettavasti, tarkkailupisteet pysäyttävät ohjelman vasta silloin, kun muuttujan arvo on luettu tai kirjoitettu. Tätä selvitettiin testaamalla toimintoa ohjelmalla, jossa yksi säie asetti muuttujan arvon ja toinen luki sen. Vaikka kirjoittava säie pysäytettiin laitteistotarkkailupisteellä, lukijasäie näki muuttujan muutoksen jälkeisen arvon. Tarkkailupisteet eivät siksi sovi sellaisenaan kilpailutilanteiden testaamiseksi ja *shared-all*-heuristiikan toteuttamiseksi. Niitä voidaan kuitenkin käyttää etsimään niitä koodirivejä, joissa käsitellään jaettuja muuttujia. Tämä tarkoittaa, että testit täytyy ajaa ainakin kerran ilman kohinaa, jotta siitä löydettäisiin jaettuja muuttujia käsittelevät koodirivit, ja niihin voitaisiin lisätä pysäytyspisteet. Tarkkailupisteet soveltuvat kuitenkin hyvin kattavuusmittareiden toteuttamiseen.

Sekä pysäytyspisteille että tarkkailupisteille on mahdollista määritellä ehdollisuuksia ja käskyjä. Pysäytyspisteisiin liitetty ehdollisuus tarkoittaa, että ohjelma pysäytetään vain, jos ehdollisuuslauseke (expression) evaluoidaan todeksi. Pisteeseen lisätty käsky puolestaan saa ohjelman suorittamaan kyseisen lausekkeen pysähtymisen sijasta. [9] Molemmissa tapauksissa käytettävä lauseke kirjoitetaan aliprosessin lähdekoodin kielellä, ja se suoritetaan aliprosessissa. Tämä on yksi tapa toteuttaa instrumentaatiota GDB:llä, mikä mahdollistaisi myös muiden satunnaistamisheuristiikkojen kuin *sleep*-heuristiikan toteuttamisen. Koodin instrumentointi tarkoittaisi kuitenkin, että GDB joutuisi myös kääntämään koodia konekielelle testaamisen aikana [9], jolloin se kärsisi samasta tehokkuusongelmasta kuin esimerkiksi ANaConDA [3].

Kohinaa ei ole mielekästä tuottaa, kun testiohjelmassa on vain yksi säie ajossa riippumatta siitä, ovatko muut säikeet pysäytettyinä, vai onko ohjelma yksisäikeinen. Yksi tapa estää kohinan tuottaminen näissä tapauksissa ilman pysäytyspisteiden tuhoamista on ottaa pysäytyspisteet pois käytöstä (disable) [9]. Testikehyksen täytyy tietää kuinka monta säiettä aliprosessilla on käytössä, jotta se voisi tietää, onko ajossa vain yksi säie. GDB:n Python API:n avulla uusista säikeistä voidaan saada ilmoituksia toteuttamalla käsittelijä `events.new_thread`-tapahtumalle tai tarkastelemalla `Inferior.threads`-metodin tulostetta määräajoin [9].

5.1.2 Säikeen suorituksen jatkaminen

Säikeen pysäyttäminen ei riitä kohinan tuottamiseksi. Säikeiden suoritusta pitää pystyä jatkamaan, jotta testattava ohjelma ei pysähtyisi kokonaan. Lisäksi, jotta kohinaohjelma voisi toteuttaa *sleep*-satunnaistamisheuristiikkan eri voimakkuuksia, sen pitää noudattaa heuristiikan määrittelemiä viiveitä säikeitä herättäessään.

Jotta säikeiden ajon aloittamista voitaisiin hallita, kohinatyökalun täytyy tietää, koska säie on pysähtynyt. GDB API:n Breakpoint-luokalla on stop-metodi, jota GDB kutsuu, kun aliohjelma saapuu pysäytyspisteeseen [9]. Kohinatestiohjelmaan on mahdollista toteuttaa signalointiominaisuus ylikuormittamalla stop-metodi. Toinen mahdollisuus toteuttaa pysähdyksen havaitseminen olisi tapahtumankäsittelijöillä, jotka toimivat hyvin samalla tavalla kuin stop-metodi. Stop-metodi kuitenkin mahdollistaa myös pysäytyksen ehdollistamisen Python-koodilla, koska aliprosessi jatkaa ajoaan välittömästi, jos metodi palauttaa positiivisen totuusarvon. [9] Tämä todennäköisesti tarkoittaa, että stop-metodia kutsutaan välittömästi ohjelman havaittua pysähdystapahtuman sen sijaan, että odotettaisiin tapahtumankäsittelijäsilmukkaa. Metodi ei kuitenkaan anna tietoa pysähtyneestä säikeestä, joten pysähdyksen seurantaan täytyy käyttää tapahtumia. Stop-metodilla toteutetaan vain pysäytyksen ehdollistaminen.

Stop-metodin ja pysähdystapahtumankäsittelijän täytyy olla suoritusajaltaan hyvin nopeita, jotta muut säikeet eivät ehtisi pysäytyspisteisiin samaan aikaan. Jos kaikki säikeet ehtisivät pysähtyä aina ennen kuin edellinen säie on käynnistetty uudelleen, mahdollisten lomittaisuuksien määrä vähenisi. Metodien hitaus ei kuitenkaan suoraan vaikuttaisi testattavan ohjelman nopeuteen, sillä se vain lisäisi kohinan voimakkuutta.

Kun GDB suorittaa stop-metodin tai pysähdystapahtumankäsittelijän koodia, se ei käsittele muita tapahtumia. Kaikki GDB:n tekemät kutsut Python-koodiin tehdään peräkkäin yksisäikeisesti. [9] Säikeiden jatkamista ei voida kuitenkaan odottaa yksisäikeisesti, koska silloin säikeitä voitaisiin jatkaa vain siinä järjestyksessä kuin ne pysähtyvät. Jotta mahdollisimman moni eri lomittaisuus olisi mahdollinen, säikeitä pitää pysytä pysäyttämään ja jatkamaan toisistaan riippumatta. Siksi viiveen odotus on tehtävä omassa Python-säikeessään.

GDB:n API ei nähtävästi tarjoa omaa rajapintaa aliprosessin säikeiden ajon jatkamiseen. Myös tämä ominaisuus on toteutettava execute-funktiolla. Kuten normaalissa komentokehoteajossa, ensin täytyy valita tietty aliprosessin säie komennolla `thread n`, missä n on kyseisen säikeen numero. Seuraavaksi säikeen ajoa voidaan jatkaa asynkronisesti komennolla `continue&`. Execute-funktio ei ole kuitenkaan säieturvallinen, mikä aiheutti ongelmia, kun sitä yritettiin käyttää prototyypissä. Funktiota täytyy kutsuta GDB:n pääsäikeestä, tai GDB voi lukkiutua. GDB:n API:ssa ongelmalle on valmis ratkaisu: `post_event`-funktiolla voidaan lisätä kutsuttava olio GDB:n tapahtumajonoon, jolloin GDB:n pääsäie kutsuu oliota myöhempänä ajanhetkenä sen normaalissa tapahtumankäsittelysilmukassa [9]. Kun aliprosessin säie herätetään kutsuttavassa oliossa, säieturvallisuus ei aiheuta ongelmia.

5.2 Debug-symbolidatan lukeminen

Fiedor ja Vojnar [7] mukaan yksi oleellinen vaatimus kohinainjektiotyökalulle on jaettujen muuttujien tunnistaminen. Ainakin *shared-all* -heuristiikka sekä jotkin kattavuuden mittarit vaativat tiedon siitä, koska käsitellään jaettuja muuttujia. Lisäksi ohjelmasta täytyy pystyä tunnistamaan säikeen saapuminen ohjelmasijaintiin, kuten funktioon. Ohjelmaa käännettäessä tallennetut debug-symbolit sisältävät tietoa ohjelman muuttujista ja funktioista. GDB:n avulla niistä on mahdollista tunnistaa kohinaheuristiikoille relevantin tietueet.

5.2.1 Jaetut muuttujat

Jaettu muuttuja on näkyvissä kahdelle tai useammalle säikeelle. Rinnakkaisissa ohjelmissa kilpailutilannevirheet liittyvät jaettujen muuttujien rinnakkaiseen käsittelyyn. C ja C++ Kielissä muuttuja voidaan jakaa säikeiden kesken ainakin kolmella eri tavalla: Muuttuja voi olla globaali, jolloin se ei ole määritelty minkään koodilohkon sisällä ja se on näkyvissä kaikkialta ohjelmasta. Staattiset muuttujat käyttäytyvät globaalin tietueen tavoin, mutta ne on määritelty koodilohkon sisällä, kuten funktiossa tai luokassa. Säikeen aloittanut emosäie voi myös jakaa oman lokaalin muuttujansa tai dynaamisesti varatun muuttujan lapsisäikeelleen antamalla sille argumenttina osoittimen tai viittauksen. Viimeistä tapaa ei kuitenkaan oteta huomioon prototyypissä, koska sen käsittelylle ei löydy yksinkertaista ratkaisua GDB:n rajapinnasta.

GDB näyttää ohjelman ylimmällä tasolla näkyvät globaalit ja staattiset muuttujat komennolla `info variables`. GDB ei toistaiseksi tarjoa vastaavaa toimintoa Python-rajapinnassa. Jotta jaettujen muuttujien nimet saataisiin luettua, Python-komentosarjan täytyy kutsua kyseistä komentoa `gdb.execute`-funktiolla, joka palauttaa komennon tulostuksen merkkijonona. [9] Tarvittavat tiedot saadaan merkkijonosta parsimalla.

Kun komentoa kutsutaan ennen ohjelman suorituksen aloitusta, sen tuloste on samaa muotoa kuin ohjelmassa 2 on esitetty.

```
(gdb) info variables
All defined variables:

File symbolit.cpp:
nimialue::luokka *osoitin;
int globaali;
int nimialue::luokka::staattinen_jasen;
int nimialue::nimi_alue_muuttuja;

Non-debugging symbols:
0x00000000000007a0  _IO_stdin_used
0x00000000000007a4  __GNU_EH_FRAME_HDR
0x0000000000000954  __FRAME_END__
0x0000000000200db8  __frame_dummy_init_array_entry
0x0000000000200db8  __init_array_start
0x0000000000200dc0  __do_global_dtors_aux_fini_array_entry
0x0000000000200dc0  __init_array_end
0x0000000000200dc8  _DYNAMIC
0x0000000000201000  _GLOBAL_OFFSET_TABLE_
0x0000000000201020  __data_start
0x0000000000201020  data_start
0x0000000000201028  __dso_handle
0x0000000000201030  __TMC_END__
0x0000000000201030  __bss_start
0x0000000000201030  _edata
0x0000000000201030  completed
0x0000000000201050  funk()::funk_staattinen
0x0000000000201058  _end
```

Ohjelma 2. *Komento globaalien muuttujien tulostamiseksi ja sen tulostus.*

Kuten nähdään, muuttujasta saadaan tietää sen sisältävä kooditiedosto, tyyppi, nimiavaruus, luokka ja nimi. Muuttujan tyyppiä ei tässä työssä pidetä oleellisena. Kaksi eri tiedostoa jaetaan yhdellä tyhjällä rivillä. Muuttujan näkyvyystiedot jaetaan kahdella kaksoispisteellä. Luokkien julkisia muuttujia ei näytetä listassa. Komento kuitenkin listaa globaalit oliot ja muuttujat olioihin, joiden avulla luokan jäsenten käyttöä voitaisiin valvoa.

Suurin osa komennon tulosteesta on testaamiselle epärelevantteja symboleita. Näiden symbolien joukossa oli kuitenkin globaalin funktion sisällä määritellyn staattisen muuttujan nimi. Koska tällaiset muuttujat ovat hyvin harvinaisia, ei niitä oteta huomioon tässä työssä. Tulostetta parsiessa ei-debug-symbolit ohitetaan.

5.2.2 Ohjelmasijainnit

Ohjelmasijainnit C++ kielessä ovat käytännössä funktioita tai metodeja. Periaatteessa myös lambdat voitaisiin laskea ohjelmasijainneiksi, mutta niitä ei oteta huomioon tässä työssä.

Ohjelmasijaintien lukeminen GDB:llä toimii samalla tavalla kuin jaettujen muuttujien lukeminen. Komento `info functions` tulostaa listan eri tiedostoissa sijaitsevista funktioista. Tulosteessa on listattu funktioiden nimiavaruus ja luokka. Lisäksi tulosteessa on epärelevantteja symbolimerkintöjä, jotka voidaan jättää huomiotta.

Kohinatestausohjelmalle voi olla oleellista, että ohjelmasijainneista erotetaan tietyt funktiot ja metodit. Jotkin heuristiikat voivat vaatia erityisiä toimintoja esimerkiksi synkronointikutsujen yhteydessä. Näitä ohjelmasijainteja ei ole mahdollista tunnistaa pelkästä debug-symbolidatasta, koska funktioiden nimet riippuvat käytetystä kirjastosta. Jos heuristiikat vaativat tietyn tyyppisten ohjelmasijaintien tunnistamista, testaajan on määriteltävä näiden sijaintien nimet ennen testaamisen aloittamista. Myös ANaConDA:ssa [3] on sama puutteellisuus, joten tämä ei tarkoita suurta haittaa muihin toteutuksiin verrattuna.

5.2.3 Ulkopuoliset moduulit

Käytännön ohjelma käyttää todennäköisesti ulkopuolisia kirjastoja, mikä kasvattaa symboleiden määrää huomattavasti. Kohinaa ei välttämättä ole toivottavaa tuottaa tiettyjen kirjastojen sisällä. Erityisesti, jos testaamisessa käytetään testauskehystä, itse kehyksen koodia ei ole mielekästä tutkia kohinainjektiolla, sillä se ei vaikuta ohjelman toimintaan. Olisi siis hyvä pystyä tunnistamaan epäoleelliset ohjelmasijainnit ja ottamaan kohina pois käytöstä niiden ajossa. GDB-toteutuksella epäoleellisten moduulien tiedostonimiä voitaisiin käyttää suodattamaan kaikki moduulien symbolit pois tietorakenteesta. Epäoleellisten moduulien tiedostonimien tunnistaminen ei välttämättä ole yksinkertaista, koska kyseiset moduulit voivat edelleen käyttää muita moduuleita. Symbolien suodatuksen katsotaan olevan tämän työn laajuuden ulkopuolella.

Muissa tapauksissa voi olla oleellista tuottaa kohinaa myös ulkopuolisissa kirjastoissa. Tämä ei välttämättä ole mahdollista, jos kirjastoon ei ole saatavissa debug-dataa. Erityisesti kaupallisesti myydyt kirjastot todennäköisesti eivät sisällä debug-dataa. Tapauksissa, joissa symbolidataa ei ole saatavissa, voitaisiin käyttää samoja dynaamisia analyysitekniikoita kuin ANaConDA:ssa [3]. Tämänkin katsotaan olevan työn laajuuden ulkopuolella.

5.3 Prototyypin toteutus

Tämän työn yhteydessä kirjoitettiin yksinkertainen kohinainjektio-ohjelman prototyyppi, jonka tarkoituksena oli varmistaa, että työssä ehdotetut tekniikat ovat toteutettavissa GDB:n Python-rajapinnalla. Prototyypissä haluttiin toteuttaa ainakin *random-all* ja *sleep* -heuristiikkoja, koska ne olivat yksinkertaisimmat toteuttaa GDB:n ominaisuuksien avulla. Prototyypin lähdekoodi on liitteessä A.

Prototyypin voidaan sanoa olevan onnistunut, koska se suorittaa testattavan ohjelman loppuun asti ilman poikkeuksia ja myös ilmeisesti aiheuttaa säikeiden pysähdyksiä. Prototyypin toiminnan varmistamisessa luotetaan prototyypin ja GDB:n tulostuksiin, koska muuta luotettavaa varmistustapaa ei ole, tai sellainen olisi kohtuuttoman vaikeaa ottaa käyttöön tämän työn laajuudessa. On kuitenkin oletettavaa, että ainakin GDB:n omat tulostukset säikeiden tapahtumista kuvaavat ohjelman toimintaa riittävällä tarkkuudella.

Prototyypissä sijoitusheuristiikka *random-all*-heuristiikka toteutettiin yksinkertaisella säännöllisen lausekkeen (regular expression) ratkaisulla. Lausekkeita etsitään GDB:n funktiolistaussyötteestä, ja jokaiseen tulokseen asetetaan pysäytyspisteolio. Kun pysäytyspistettä luodaan, sille arvotaan numero ignore_count, joka määrittelee kuinka monta kertaa pysäytyspiste jätetään ohjelman ajossa huomiotta. Ignore_count [9] on GDB:n pysähdyspisteiden luontainen ominaisuus, joten se todennäköisesti aiheuttaa vähemmän kohinaa kuin pysähdyksen käsitteleminen stop-metodissa. Numeroa vähennetään yhdellä joka kerta kun aliohjelma ohittaa pisteen kunnes se on nolla, jonka jälkeen pisteeseen saapunut säie pysäytetään normaalisti. Numero arvotaan uudelleen joka kerta kun stop-metodia kutsutaan. Numero arvotaan siten, että vain tietty prosenttimäärä pysäytyspisteiden ohituksista aiheuttaa pysäytyksen, kuten *random-all*-heuristiikka määrittelee.

Sleep-heuristiikka toteutettiin kirjoittamalla käsittelijäfunktio GDB:n pysähdystapahtumille. Käsittelijä tallentaa tapahtuman parametrina annetun aliprosessin säikeen sekä halutun herätysajan kekotietorakenteeseen, joka järjestetään herätysaikojen mukaan pienin ensin. Erillinen Python-säie odottaa säikeen lisäämistä kekoon ehtomuuttujan avulla. Kun uusi säie lisätään kekoon, ehtomuuttuja vertaa sen herätysaikaa sen hetkiseen aika-arvoon. Jos herätysaika on ohitettu, säie nostaa säikeen herätystapahtuman, jonka GDB käsittelee tapahtumasilmukassaan. Jos aikaa on jäljellä, Python-säie odottaa joko aikaeron nollaantumista tai kunnes uusi säie lisätään kekoon.

Prototyyppi pyrkii myös ottamaan huomioon ajossa olevien säikeiden määrän kohinaa tuottaessaan. Jos aliprosessissa on vähemmän kuin kaksi pysäyttämätöntä säiettä, pysäytyspisteen stop-metodista pyritään palauttamaan negatiivinen totuusarvo mahdollisimman nopeasti. Prototyyppi ei kuitenkaan toistaiseksi ota huomioon tilanteita, joissa aliprosessin säie suorittaa ohjelmansa loppuun ja lopettaa suorituksensa. Tämä oli tietoinen päätös johtuen siitä, että GDB:n API:sta ei löydetty sopivaa tapahtumaa tilanteen käsittelyyn, ja muu toteutus ei olisi kovin hyödyllinen työmäärään nähden. Pahimmassa tapauksessa tämän puutteen takia aliprosessiin tuotetaan kohinaa, vaikka sillä olisi vain yksi säie käytössä.

Prototyypissä on toteutettu vain hyvin yksinkertainen virheenhavainnointimekanismi. Pysähdystapahtumaa käsitellessä tarkistetaan, onko tapahtuma `gdb.BreakpointEvent`-luokan instanssi. Jos se ei ole, kaikki aliprosessin säikeet pysäytetään komennolla `interrupt -a` [9]. Pysäytyksen jälkeen testaaja voisi tutkia virhetilannetta käyttäen ainakin joitakin GDB:n normaaleja toimintoja, kuten suorituspinon tarkastelemista. Tulevai-

suudessa prototyyppiin voitaisiin toteuttaa mekanismi, jolla testaaja voisi ottaa kohinan pois käytöstä kokonaan, mikä mahdollistaisi myös pysäytyspisteiden käytön virhetilanteen tutkimiseksi.

Prototyyppi ei saanut tuotettua virhetilannetta, kun sitä testattiin tietyllä ohjelmalla, jossa tiedettiin olevan kilpailuehto. Prototyyppi lukkiutui aina usean virheettömän testiajon jälkeen. Lukkiutuminen johtui todennäköisesti prototyypin ohjelmointivirheestä liittyen joko monisäikeiseen herätyksen odottamiseen tai GDB-käskyjen kutsumiseen tavalla, joka ei ollut säieturvallinen. Vian oletetaan liittyvän uuden testiajon käynnistämiseen Python-koodista, koska lukkiutuminen tapahtui aina kahden testiajon välissä. Virheen syytä ei kuitenkaan löydetty. Lukkiutumisesta johtuen prototyypistä ei voida varmuudella sanoa, olisiko virheen havaitseminen sen avulla mahdollista. Fiedor et al. suorittamissa kokeissa [4] testattavaa ohjelmaa ajettiin toistuvasti 20 minuutin ajan, mistä huolimatta kaikkia virheitä ei aina löydetty. Prototyypillä testiä voidaan ajaa vain muutaman kymmenen sekunnin ajan ilman lukkiintumista. Toisaalta Fiedor et al. suorittamissa testeissä ilmeisesti testattiin eri ohjelmaa, joten vertailu ei ole kovin luotettavaa.

Suurimman ongelmat prototyypin kirjoittamisessa johtuivat GDB:n säieturvallisuusrajoituksista. Testiohjelma saatiin toistuvasti lukkiutumaan ennen kuin käyttöohjetta [9] lukiessa löydettiin tiedot `post_event`-funktiosta ja `execute`-komentojen asynkronisesta suorituksesta. Yksi jatkotutkimuksen aihe voisi olla toteuttaa kohinainjektio-ohjelma muokkaamalla GDB:n lähdekoodia, mikä mahdollistaisi käyttötarkoitukseen paremmin sopivan rinnakkaisen tapahtumankäsittelyn suorittamisen.

6. YHTEENVETO

Kohinainjektio on toimiva ratkaisu testata rinnakkaisia ohjelmia integraatiotasolla. Sen luotettavuus erityyppisten rinnakkaisuusvirheiden etsimisessä riippuu käytetyistä heuristiikoista. GDB:llä on kaikki tarvittavat ominaisuudet, joita tarvitaan ainakin joidenkin kohinaheuristiikkojen toteuttamiseksi.

Prototyypin toimivuutta on mahdotonta todentaa täydellä varmuudella, koska sille ei voi määritellä vertailupistettä. Rinnakkaisuuden epädeterministisyyden takia ei voida päätellä, millainen ohjelman ajojärjestys olisi ollut ilman kohinaa. Jos kuitenkin oletetaan säikeiden käyttäytyvän määritellyllä tavalla pysäytyspisteissä, voimme olettaa ohjelman aiheuttaneen viiveitä, joita normaaliajossa ei olisi ilmentynyt. Tämä todistaa, että kohinainjektiota on mahdollista toteuttaa virheenetsintäohjelmilla.

Kohinan tuottamisessa tehokkuus on hyvin tärkeää, jos halutaan tarkasti toteuttaa satunnaistamisalgoritmien asettamia voimakkuuksia. Kohinainjektion toteuttaminen komentotulkilla ei välttämättä ole paras ratkaisu, koska aikaväli tapahtuman havaitsemisen sen käsittelemisen välillä voi olla pidempi kuin mitä säikeen oli tarkoitus odottaa. Parempi ratkaisu olisi toteuttaa kohinaohjelma muokkaamalla GDB:n lähdekoodia. Toisaalta edes teoreettinen täydellisen nopea herätysmekanismi ei pystyisi herättämään säikeitä täsmälleen oikealla viiveellä, koska herätyksen jälkeen säikeen täytyy myös odottaa suoritusvuoroa vuorottajalta. Itse kohinaohjelman tehokkuus ei kuitenkaan suoraan vaikuta testaamiseen kuluvaan aikaan, sillä se vaikuttaa vain kohinan voimakkuuteen.

Toteutettu prototyyppi on hyvin puutteellinen verrattuna esimerkiksi ANaConDA:aan. Näiden kahden oleellisia eroja on listattu taulukossa 1.

Taulukko 1. *ANaConDA:n [3] ja GDB-prototyypin oleellisia eroja.*

ANaConDA	**GDB-prototyyppi**
Toimii myös kolmannen osapuolen moduuleissa.	Vaatii debug-symbolidataa toimiakseen moduulissa.
Instrumentoitavan koodin kääntäminen aiheuttaa testiohjelman hidastusta.	Pullonkauloja ovat komentojen tulkitseminen, GDB-tapahtumakäsittelijäsilmukat ja IPC.
Tarjoaa rajapinnan virheanalyysille.	Toteuttaa vain kohinainjektiota.

Prototyyppiä ei kuitenkaan kirjoitettu ratkaisemaan kohinainjektio-ongelmaa, tai kilpailemaan muiden ratkaisuiden kanssa, vaan todistamaan kohinainjektion olevan toteutet-

tavissa virheenetsintätekniikoilla. Optimaalinen GDB-toteutus mahdollistaisi teoriassa samoja ominaisuuksia, mukaan lukien ulkopuolisten moduulien käsittelyn ja virheanalyysin, koska se tarjoaa hyvin matalan tason rajapinnan käsitellä prosessin muistia.

GDB-toteutuksella on teoriassa muutamia etuja verrattuna instrumentaatiototeutuksiin. Ensinnäkin se tarjoaa ympäristöriippumattoman abstraktiokerroksen ohjelman analysointiin ja hallintaan. Koska GDB tukee monia eri ohjelmointikieliä, kuten C, Fortran ja Pascal, samaa kohinaohjelmaa voitaisiin periaatteessa käyttää kaikkien tuettujen kielien testaamiseen. Toiseksi sen pysäytysmekanismit eivät vaikuta ohjelman toimintaan silloin kun ne eivät tuota kohinaa, jos pysäytyspisteet otetaan aina niissä tapauksissa pois käytöstä. Tämä voi edelleen vaikuttaa mahdollisten lomittaisuuksien määrään. Toteutuksen tarkkuutta on vaikea arvioida tai verrata, sillä GDB:n toiminta perustuu prosessien väliseen viestintään (Inter-process communication IPC), joka aiheuttaa lisää satunnaisia viiveitä pysähdyksien käsittelyssä.

Eri toteutusmalleja on turhaa verrata testaamiseen kuluvan ajan kannalta. Kohinainjektio luonnollisesti kärsii suoritusnopeusongelmasta, koska sen toiminta perustuu siihen, että säikeiden suorituksia keskeytetään useasti testaamisen aikana. Ainoa tapa vaikuttaa suoritusnopeuteen olisi kehittää ja ottaa käyttöön heuristiikkoja, jotka tuottaisivat enemmän lomittaisuuksia pienemmillä kohinan voimakkuuksilla.

Rinnakkaisuuden testaamisen kehitys tulee todennäköisesti jatkumaan jatkossakin, koska rinnakkaisuuden testaamiselle on tarvetta. Rinnakkaisuuden testaaminen mallintarkastustekniikoilla (model check) on jo useita vuosia ollut tutkimuksen kohteena [4], ja siitä voi kehittyä korvaava vaihtoehto kohinainjektiolle. Toistaiseksi Kohinainjektio on kuitenkin ainoa tekniikka, joka kykenee riittävän luotettavasti testaamaan ohjelmaa integraatiotasolla ilman suuria resurssivaatimuksia.

LÄHTEET

[1] N.G. Leveson, C.S. Turner, An investigation of the Therac-25 accidents, Computer, Vol. 26, Iss. 7, 1993, pp. 18–41.

[2] Z. Letko, Analysis and Testing of Concurrent Programs, Information Sciences and Technologies, Vol. 6, Iss. 1, 2014, pp. 28.

[3] J. Fiedor, T. Vojnar, ANaConDA: A Framework for Analysing Multi-threaded C/C++ Programs on the Binary Level, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 35–41.

[4] J. Fiedor, V. Hrubá, B. Křena, Z. Letko, S. Ur, T. Vojnar, Advances in noise-based testing of concurrent software, Software Testing, Verification and Reliability, Vol. 25, Iss. 3, 2015, pp. 272–309.

[5] B. Křena, Z. Letko, T. Vojnar, Noise Injection Heuristics for Concurrency Testing, Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, pp. 123–135.

[6] E. Sherman, M. Dwyer, S. Elbaum, Saturation-based testing of concurrent programs, Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering, ACM, pp. 53–62.

[7] J. Fiedor, T. Vojnar, Noise-based testing and analysis of multi-threaded C/C++ programs on the binary level, Proceedings of the 2012 Workshop on parallel and distributed systems: testing, analysis, and debugging, ACM, pp. 36–46.

[8] Pin 3.4 User Guide, Intel Corporation, web page. Available (accessed 08.07.2018), Saatavissa: https://software.intel.com/sites/landingpage/pintool/docs/97438/Pin/html/index.html.

[9] Debugging with GDB, Free Software Foundation, Inc., web page. Available (accessed 08.07.2018), Saatavissa: https://sourceware.org/gdb/onlinedocs/gdb/index.html.

LIITE A: KOHINAINJEKTIOPROTOTYYPIN LÄHDEKOODI

noiseinjector.py:

```
1  import sys
2  import os
3  import threading
4  import gdb
5  from time import time
6  import random
7  random.seed()
8
9  # Liätään omien moduulien hakemisto hakupolkuun.
10 sys.path.append(os.getcwd() + "/python/")
11
12 import symbolparser
13 from threadhandler import *
14
15
16 class RandomAll:
17     def __init__(self, breakpointFacotry=None):
18         self._factory = breakpointFacotry
19         if breakpointFacotry == None:
20             self._factory = RandomAll.Breakpoint
21
22         self._symbols = symbolparser.Parse(gdb.execute("info func-
tions", to_string=True))
23
24         self._breakpoints = []
25         for filename, funcnames in self._symbols.items():
26             for name in funcnames:
27                 location = filename + ": " + name
28                 print(location)
29                 self._breakpoints.append(self._factory(location))
30
31     class Breakpoint(gdb.Breakpoint):
32         enabled = False
33
34         def __init__(self, *args):
35             super().__init__(*args)
36             self.ignore_count = self.get_ignore_count()
37
38         def stop(self):
```

```
39              if (not self.enabled):
40                  return False
41
42              self.ignore_count = self.get_ignore_count()
43
44              print("pysähdyspiste pysäyttää ohjelman")
45              return True
46
47          stop_probability = 0.8
48
49          @classmethod
50          def get_ignore_count(cls):
51              ignorecount = 0
52              while random.uniform(0, 1) > cls.stop_probability:
53                  ignorecount += 1
54
55              return ignorecount
56
57  # RandomAll loppu
58
59
60  class Sleep:
61      def __init__(self):
62          self._threadQueue = ThreadStartingQueue(Sleep.start_thread)
63          gdb.events.stop.connect(self.stop_handler)
64
65      def stop_handler(self, event):
66          ThreadHandler.running_inferior_threads_increment(-1)
    self._threadQueue.add_stopped_thread(StoppedThread(event.inferior_threa
67  d, self.get_wake_time()))
68
69      @staticmethod
70      def get_wake_time():
71          return time() + random.uniform(0.002, 0.02)
72
73      class StartEvent:
74          def __init__(self, thread):
75              self._thread = thread
76
77          def __call__(self):
78              gdb.execute("thread " + str(self._thread.num), True)
79              gdb.execute("continue&", True)
80              ThreadHandler.running_inferior_threads_increment(1)
81              print("jatketaan säiettä " + str(self._thread.num))
82
83      @staticmethod
```

```
84      def start_thread(thread):
85          gdb.post_event(Sleep.StartEvent(thread))
86
87
88  class ThreadHandler:
89      _runningInfThreads = 0
90
91      @classmethod
92      def new_thread_handler(cls, event):
93          cls.running_inferior_threads_increment(1)
94
95      @classmethod
96      def running_inferior_threads_increment(cls, increment):
97          cls._runningInfThreads += increment
98
99          if cls._runningInfThreads < 2:
100             RandomAll.Breakpoint.enabled = False
101         else:
102             RandomAll.Breakpoint.enabled = True
103
104 gdb.events.new_thread.connect(ThreadHandler.new_thread_handler)
105
106 gdb.execute("set target-async 1")
107 gdb.execute("set pagination off")
108 gdb.execute("set non-stop on")
109
110 paclement = RandomAll()
111 seeding = Sleep()
112
113
114 def run_inferior():
115     gdb.execute("run&")
116
117
118 def exit_handler(event):
119     if event.exit_code != 0:
120         print("virhekoodi " + str(event.exit_code))
121         return
122     else:
123         gdb.post_event(run_inferior)
124
125
126 gdb.events.exited.connect(exit_handler)
```

symbolparser.py:

```
from enum import Enum, auto
from collections import defaultdict
from re import *
import sys


class _States(Enum):
    TOP = auto()
    FILE = auto()
    IGNORE = auto()


def Parse(string: str, fileRE='^File (.+):$', nameRE='^.+ \*?&?([^
;\(\)]+(\(.*\))?);$') -> defaultdict:
    results = defaultdict(list)
    filename = None
    blockStart = int()
    blockEnd = int()
    state = _States.TOP
    fileRegex = compile(fileRE, MULTILINE)
    nameRegex = compile(nameRE, MULTILINE)
    emptyRegex = compile('^\s*$', MULTILINE)
    searchPos = int(0)

    while True:
        if state == _States.TOP:
            # Etsitään tiedoston nimeä.
            searchResults = fileRegex.search(string, searchPos)

            if searchResults is None: return results

            filename = searchResults.group(1)
            state = _States.FILE

            #print(filename)

            searchPos = searchResults.end()

            endPos = emptyRegex.search(string, searchPos)

            blockStart = searchResults.end()
            if endPos is None:
                blockEnd = len(string)
            else:
                blockEnd = endPos.start()
```

```
45
46          elif state == _States.FILE:
                searchResults    =    nameRegex.search(string,    blockStart,
47  blockEnd)
48              #print(string[blockStart:blockEnd])
49
50              if searchResults is None:
51                  state = _States.TOP
52                  continue
53
54              #print("NameFound")
55
56              if searchResults.end() > blockEnd:
57                  state = _States.TOP
58                  continue
59
60              results[filename].append(searchResults.group(1))
61              blockStart = searchResults.end()
```

threadhandler.py:

```
 1  import threading
 2  import heapq
 3  from time import time
 4
 5  class StoppedThread:
 6      def __init__(self, thread, startTime):
 7          self.thread = thread
 8          self.startTime = startTime
 9
10      def __lt__(self, other):
11          return self.startTime < other.startTime
12
13      def __gt__(self, other):
14          return self.startTime > other.startTime
15
16      def __eq__(self, other):
17          return self.startTime == other.startTime
18
19  class ThreadStartingQueue:
20      def __init__(self, threadStartCallable):
21          self._threadStartFunc = threadStartCallable
22          self._queue = []
23          self._queueLock = threading.Lock()
24          self._condVar = threading.Condition(self._queueLock)
25          self._waiterThread = threading.Thread(target=self._waiter)
```

```
            self._waiterThread.daemon = True    # Ei odoteta säikeen loppu-
26  mista ohjelman loputtua
27          self._waiterThread.start()
28          self._running = True
29          self._addedThread = False
30
31      def _waiter(self):
32          self._queueLock.acquire()
33          timeout = None
34          while True:
35              self._condVar.wait(timeout)
36
37              if not self._running:
38                  return
39
40              if len(self._queue) == 0:
41                  tiemout = None
42                  continue
43
44              if self._queue[0].startTime <= time():
45                  # Jatketaan säikeen ajoa.
46                  event = heapq.heappop(self._queue)
47                  self._threadStartFunc(event.thread)
48                  pass
49
50              if len(self._queue) == 0:
51                  timeout = None
52              else:
53                  timeout = self._queue[0].startTime - time()
54
55          # Vapautetaan lukko poistuessa.
56          self._queueLock.release()
57
58
59      def add_stopped_thread(self, stoppedThread):
60          self._queueLock.acquire()
61          heapq.heappush(self._queue, stoppedThread)
62          self._addedThread = True
63          self._condVar.notify_all()
64          self._queueLock.release()
65
66      def cleanup(self):
67          self._condVar.acquire()
68          self._running = False
69          self._condVar.notify_all()
70          self._condVar.release()
```