Tampereen yliopisto

Esa Lakaniemi

KATTAVUUSPOHJAINEN FUZZ-TESTAUS JA AFL++

Kandidaatintyö
Informaatioteknologian ja viestinnän tiedekunta
Tarkastaja: Maarit Harsu
Maaliskuu 2022

TIIVISTELMÄ

Esa Lakaniemi: Kattavuuspohjainen fuzz-testaus ja AFL++
Kandidaatintyö
Tampereen yliopisto
Tieto- ja sähkötekniikan kandidaattiohjelma
Maaliskuu 2022

Fuzz-testaus on erityisesti tietoturva-alalla käytetty testauksen menetelmä, jossa ohjelmia testataan satunnaisesti luoduilla virheellisillä syötteillä. Fuzzauksen tavoitteena on löytää testattavien ohjelmien sisältämiä virheitä. Tämän työn tarkoitus oli esitellä nykyaikainen fuzzaus ja mitä sillä voidaan saavuttaa suhteessa muihin testausmenetelmiin, millaisia tuloksia fuzzaamalla on saavutettu sekä millaisia fuzzauksen tulevaisuudennäkymät ovat. Näitä asioita tarkastellaan erityisesti AFL++-fuzzaustyökalun kannalta.

Työn alussa annetaan ohjelmistotestauksen määritelmä sekä selitetään työn kannalta oleellisia ohjelmistotestauksen käsitteitä. Ohjelmistotestauksen yleisten käsitteiden jälkeen esitellään fuzz-testauksen historiaa ja tekniikoita, erityisesti painottaen kattavuuspohjaisen fuzzauksen menetelmää. Näiden esittelyä seuraa AFL++-työkalun, sen käytön ja joidenkin sitä laajentavien työkalujen kuvaus. Lopuksi työ esittelee fuzzausprojekteja ja mitä fuzzauksella voidaan ja ei voida saavuttaa sekä fuzzauksen tulevaisuudennäkymiä.

Tämä työ antaa perustason kuvauksen nykyaikaisesta fuzzauksesta. Fuzzauksen todetaan olevan tehokas tapa virheiden löytämiseen ohjelmista. Nykyaikana erityisesti kattavuuspohjainen fuzzaus on tehokas ja suosittu fuzzauksen menetelmä, ja AFL++ on suosittu ja tunnettu kattavuuspohjaisen fuzzauksen työkalu. Fuzzauksen avulla voidaan löytää hankalasti ennakoitavia virheitä, joita ennalta suunnitelluilla testitapauksilla ei voida tehokkaasti löytää ja jotka voivat olla hyvinkin merkittäviä. Fuzzauksen avulla ei kuitenkaan voida varmistaa jonkin tietyn vian olevan korjattu tai jonkin tietyn vaatimuksen olevan täytetty. Tulevaisuudessa fuzztestausta saatetaan esimerkiksi yhdistää ohjelmistokehityksen testiautomaatioon, ja se vaikuttaa jatkossakin pysyvän suosittuna menetelmänä virheiden löytämiseen. Kattavuuspohjaisen fuzzauksen lisäksi myös vanhempia fuzzauksen menetelmiä käytetään edelleen, ja fuzzaukselle löydetään myös uusia käyttötarkoituksia.

Avainsanat: fuzzaus, fuzz-testaus, testaus, AFL, AFL++, kattavuuspohjainen fuzzaus

SISÄLLYSLUETTELO

1. JOHDANTO

Fuzz-testaus on erityisesti tietoturvamaailmassa suosittu testausmenetelmä, jossa käytetään satunnaisesti luotuja syötteitä. Fuzzaamalla on viime vuosikymmenen aikana löydetty useita merkittäviä tietoturvaongelmia monissa laajalti käytetyissä ohjelmistoissa ja kirjastoissa. Esimerkiksi suureen määrään verkkopalvelimia vuonna 2014 vaikuttaneen ja julkisuutta saaneen Shellshock-nimellä tunnetun tietoturvaongelman eri muodoista moni löydettiin fuzzaamalla [1].

Tämän työn tarkoitus on selvittää ja esitellä mitä nykyaikainen fuzzaus, erityisesti kattavuuspohjainen harmaalaatikkofuzzaus on, mitä sillä voidaan tai ei voida saavuttaa verrattuna eräisiin yleisiin testauksen menetelmiin sekä millaisia tuloksia fuzzauksen avulla on viime vuosina saavutettu. Näitä aiheita tarkastellaan pääasiassa AFL++-työkalun näkökulmasta.

AFL++ on kiinnostava tutkimuskohde nykyaikaisen fuzzauksen näkökulmasta, sillä AFL ja sen jatkokehitteet ovat oman arvioni mukaan suosituimpia nykyaikaisista fuzzaustyökaluista. Niitä käytetään suurissa jatkuvan fuzzauksen projekteissa, ja alkuperäisen AFL:n sekä sen jatkokehitteiden ja sen päälle rakennettujen työkalujen avulla on viime vuosikymmenen aikana löydetty useita virheitä monista tunnetuista ja laajalti käytetyistä avoimen lähdekoodin kirjastoista ja sovelluksista. Lisäksi fuzzausta ja erityisesti tätä työkalua käsittelevää tai siihen jotenkin perustuvaa uutta tutkimusta julkaistaan jatkuvasti.

Työ jakaantuu johdantoon, varsinaisiin aihelukuihin ja yhteenvetoon. Luvussa 2 käsitellään yleisesti ohjelmistojen testausta, luvussa 3 fuzzausta ja sen historiaa ja menetelmiä. Luvussa 4 esitellään AFL++-työkalu sekä sen edeltäjät ja kerrotaan työkalun historiasta ja ominaisuuksista. Luvussa 5 esitellään fuzzauksen käyttöä tosimaailmassa, vastataan esimerkkien avulla tutkimuskysymykseen ja tarkastellaan fuzzauksen tulevaisuudennäkymiä.

2. OHJELMISTOTESTAUS

2.1 Ohjelmistotestauksen määritelmä

Ohjelmistotestaus on kokonaisuutena laaja, ja ohjelmistojen testaukseen käytetään monenlaisia eri menetelmiä. Näiden menetelmien avulla pyritään esimerkiksi todentamaan ohjelman vastaavan vaatimusmäärittelyjään, etsimään siitä tuntemattomia virheitä tai varmistamaan jonkin tunnetun virheen olevan korjattu. Ohjelmistoja testataan usein niiden kehityksen yhteydessä, mutta testaus voi myös tapahtua kehityksen ulkopuolella. Testaus voi olla automatisoitua tai manuaalista, ja automaattitestauksen automaation aste voi vaihdella.

Myers et al. määrittelevät ohjelmistotestauksen prosessina, jossa ohjelmaa suoritetaan tavoitteena löytää virheitä [2, luku 2]. Tästä näkökulmasta ohjelmassa oletetaan lähtökohtaisesti olevan virheitä, ja ohjelmaa testataan nimenomaan näiden virheiden löytämiseksi. Testausta käytetään myös varmistamaan ohjelman toimivan halutulla tavalla, mutta kirjoittajien mukaan paras tapa varmistaa tämä on pyrkiä löytämään virheitä [2, luku 2]. Tämän työn kannalta oleellista ohjelmistotestausta onkin erityisesti automaattinen, virheiden löytämiseen tähtäävä testaus.

2.2 Ohjelmistotestauksen käsitteitä

2.2.1 Testitapaukset

Testitapaus on yksittäisen testin kuvaus, joka sisältää testin suorittamiseen vaadittavat syötteet sekä testin odotetun tuloksen [2, luku 2][3, s. 466]. Testitapaus voi olla muodollisesti määritelty tai epämuodollinen kuvaus testin suorituksesta, ja se voi sisältää edellä mainittujen osien lisäksi erilaisia lisätietoja. Testitapaus voi koostua jopa pelkästä automaattitestistä, eli automaattitestauksen työkalun syötteestä. Yksittäinen testitapaus on usein tarkoitettu testaamaan yhtä tai useampaa ohjelman vaatimusta [3, s. 466]. Testitapaus voi olla myös esimerkiksi osa virheraporttia, jolloin testitapauksen tarkoitus on kuvata, miten raportoitu virhe voidaan toistaa.

Onnistuneesti suoritettu testitapaus kertoo sen testaaman vaatimuksen olevan toteutettu tai sen kuvaaman vian olevan korjattu. Tällaista testitapausta voidaan jatkossa käyttää

regressiotestinä. Regressiotestauksella pyritään löytämään regressioita eli vikoja ohjelman osissa, jotka ovat aiemmin toimineet [2, luku 4, 6]. Näin siis testitapausta toistamalla muutosten jälkeen voidaan varmistua, ettei ohjelman kehitys ole aiheuttanut vikoja osiin, joiden on aiemmin testattu onnistuneesti toimivan.

2.2.2 Automaattitestaus

Nykyaikaisessa ohjelmistokehityksessä testaus on usein automatisoitua, eli siinä käytetään työkaluja, jotka sekä suorittavat testin että tarkistavat automaattisesti, oliko sen tulos odotetunlainen. Näitä työkaluja voidaan käyttää suoraan ohjelmistokehittäjien omissa kehitysympäristöissä, mutta usein myös testiajojen suorittaminen itsessään on automatisoitu. Ohjelmiston kääntämisen ja testien suorittamisen automatisoivia järjestelmiä kutsutaan joskus *jatkuvan integraation* (continuous integration) järjestelmiksi. Jatkuva integraatio terminä viittaa pelkkää testaamista laajempaan kokonaisuuteen [4], jonka muita osia ei käsitellä tässä.

Ehkä tyypillisin automatisoidun testin tyyppi on *yksikkötesti* (unit test). Yksikkötestillä tarkoitetaan ohjelman yksittäisten osien, esimerkiksi luokkien tai funktioiden, testaamista [2, luku 5]. Yksikkötesteillä pyritään testaamaan muun muassa onko osa toteutettu vaatimustensa mukaisesti, ja löytämään siitä mahdollisia virheitä [2, luku 5][3, s. 490]. Yksikkötesti ei yleensä käytä muita osia ohjelmasta varsinaisen testattavan osan ulkopuolella, jolloin testejä voi suorittaa riippumatta muiden ohjelman osien toimivuudesta. Näin yksikkötestejä on käytännöllistä suorittaa jo kehityksen aikana.

Myös *integraatiotestejä*, joissa testataan useita ohjelman osia samanaikaisesti [3, s. 231], voidaan automatisoida. Nämä testit testaavat osien toimimista yhteen, jolloin ne testaavat paremmin ohjelman toiminnallisuutta, mutta myös vaativat suuremman osan ohjelmasta toimiakseen.

2.2.3 Testikattavuus

Testikattavuudella (test coverage) tarkoitetaan sen osan suuruutta testattavasta ohjelmasta, jota jollakin tietyllä testijoukolla pystytään testaamaan. Kattavuus ilmaistaan yleensä prosenttilukuna. [3, s. 466] Kaikkien mahdollisten suorituspolkujen kattaminen on usein käytännössä mahdotonta [2, luku 4], joten kattavuutta mitataan tyypillisesti erilaisilla käytännöllisemmillä mittareilla.

Tämän työn kannalta tyypillisin kattavuuden mittari on *haarakattavuus* (branch coverage), joka mittaa kuinka suuri osuus ohjelman haaroista on testattu [2, luku 4]. Haaroja ovat esimerkiksi ehtolauseet, joissa yksi suorituksen haara suoritetaan ehtolauseen ollessa tosi ja toinen sen ollessa epätosi. Myös silmukoissa on kaksi suorituksen haaraa: paluu silmukkaan ja jatkaminen silmukasta eteenpäin.

Haarakattavuus ei mittaa täydellisesti suorituspolkujen kattavuutta. Esimerkiksi ehtolausetta, joka sisältää usean eri ehdon, tarvitsee testata vain koko ehtolauseen ollessa tosi ja koko ehtolauseen ollessa epätosi riippumatta siitä, millaisia ehtoja se sisältää. Jos halutaan mitata yksittäisten ehtojen kattavuutta, voidaan mittarina käyttää *ehtokattavuutta* (condition coverage), jossa kattavuuden saavuttamiseksi testataan jokaisen ehdon olevan tosi ja epätosi ainakin kerran [2, luku 4].

Ehtokattavuus ei kuitenkaan takaa haarakattavuutta, joten sekä haara- että ehtokattavuuden saavuttamiseksi käytetään ajoittain *moniehtokattavuuden* mittaria. Tällä mittarilla kattavuuden saavuttamiseksi testataan jokaisen ehdon olevan sekä tosi että epätosi, minkä lisäksi testataan kaikkien ehtojen arvojen yhdistelmät [2, luku 4]. Tällöin sama haara tai suorituspolku saatetaan joutua testata useampaan kertaan: Esimerkiksi ehtolauseen "$a < 1$ JA $b < 1$" moniehtokattavuuden saavuttamiseksi tarvitaan neljä erilaista testitapausta:

(a) $a < 1$, $b < 1$ (tosi)
(b) $a < 1$, $b \geq 1$ (epätosi)
(c) $a \geq 1$, $b \geq 1$ (epätosi)
(d) $a \geq 1$, $b < 1$ (epätosi)

Suorituspolkuja ja haaroja on kuitenkin vain kaksi: haara ehtolauseen ollessa tosi, ja haara ehtolauseen ollessa epätosi. Haarakattavuuden täyttämiseen näistä testitapauksista riittäisi tapaus a ja mikä tahansa kolmesta epätodesta tapauksesta, ja ehtokattavuuden täyttämiseen joko tapaukset a ja c tai tapaukset b ja d.

2.2.4 Valko-, musta- ja harmaalaatikkotestaus

Eräs tapa jaotella testauksen menetelmiä on määritellä ne sen mukaan, kuinka suuri näkyvyys niillä on testattavaan ohjelmistoon. Tällaisessa jaottelussa näkyvyys jaetaan karkeasti kolmeen eri kategoriaan, jotka ovat valkolaatikko, mustalaatikko sekä harmaalaatikko. Menetelmää, joka käsittelee testattavaa ohjelmistoa valkolaatikkona, kutsutaan valkolaatikkotestauksen menetelmäksi tai valkolaatikkomenetelmäksi, ja vastaavasti kahta muuta tyyppiä edustavia menetelmiä mustalaatikko- ja harmaalaatikkomenetelmiksi.

Valkolaatikko tarkoittaa järjestelmää, jonka sisäinen toteutus tunnetaan [3, s. 199]. Ohjelmiston tapauksessa tällöin on yleensä kyse siitä, että sovelluksen lähdekoodi on saatavilla [5, s. 82, 144]. Mustalaatikko taas merkitsee järjestelmää, jonka toteutusta ei tunneta, vaan ainoastaan sen ulkoisesti havaittava käytös [3, s. 47]. Ohjelmistosta puhuttaessa tämä tarkoittaa yleensä ainoastaan ohjelman käännetyn muodon olevan saatavilla [5, s. 85, 144].

Harmaalaatikko merkitsee jotain kahden edellä mainitun väliltä: järjestelmää, jonka sisäinen rakenne tunnetaan vain osittain, tai jota voidaan tai halutaan vain osittain hyödyntää. Sitä voidaan myös ajatella valkolaatikon ja mustanlaatikon yhdistelmänä, jossa

järjestelmän sisäistä rakennetta hyödynnetään esimerkiksi käyttämällä sitä mustalaatikkotestien suunnittelussa [5, s. 82]. Tässä työssä harmaalaatikkotestaus on näistä kolmesta oleellisin, ja sillä tarkoitetaan erityisesti testausta, jossa lähdekoodia käytetään rajoitetusti hyväksi ohjelmiston testauksen ohjaamisessa.

2.3 Testauksen rajoitteita

Jotta ohjelman voi testata tekevän mitä sen on tarkoitus tehdä, tulee testata oikean muotoisten ja odotetunlaisten syötteiden tuottavan oikeita tuloksia. Virheiden löytämisen kannalta on kuitenkin erityisen oleellista testata myös odottamattomia ja virheellisiä syötteitä [2, luku 2].

Monia virheitä on kuitenkin hankala testitapausta suunnitellessa ennakoida. Edellä esitellyt testikattavuuden mittarit saattavat auttaa testien suunnittelussa, mutta monimutkaisessa ohjelmassa jo esimerkiksi korkean haarakattavuuden saavuttaminen vaatii suuren määrän työtä. Korkea moniehtokattavuus saattaa olla käytännössä mahdotonta saavuttaa vaadittujen testitapausten suuren määrän takia. Lisäksi kehittäjien on ylipäätään usein hankala löytää omasta koodistaan virheitä muun muassa siksi, ettei itse kirjoittamaansa koodiin ole helppo ottaa virheitä etsivää näkökulmaa [2, luku 2].

Jos testejä halutaan toistaa automaattisesti, tulee niiden käytännössä olla deterministisiä. Tämä tarkoittaa, että niiden pitäisi tuottaa samoilla syötteillä ja samoilla ohjelman versioilla aina sama tulos. Näin esimerkiksi satunnaisuuden käyttö ei välttämättä ole hyväksyttävää – satunnaisuutta käyttävä testi saattaisi onnistua sattumalta, kun sen olisi pitänyt epäonnistua, tai päinvastoin.

Testauksen avulla voidaan siis harvoin löytää kaikki ohjelmassa olevat viat. Edellä esitellyt etukäteen suunniteltuihin testitapauksiin perustuvat menetelmät ovat yleisiä, ja niiden avulla voidaan saada varmuutta ohjelman oikeasta toimivuudesta. Ne kuitenkin kattavat lähinnä virhetilanteet, jotka testien kehittäjä on osannut etukäteen ennakoida. Uusien ja tuntemattomien virheiden etsimisen kannalta ne eivät kuitenkaan ole tehokkaimpia mahdollisia, vaan tähän tarvitaan erilaisia menetelmiä.

3. FUZZ-TESTAUS

3.1 Fuzzauksen historiaa

Fuzz-testaus eli fuzzaus on erityisesti tietoturva-alalla käytetty testauksen menetelmä, jossa ohjelmista etsitään virheitä virheellisten syötteiden avulla [5, s. 1]. Nämä syötteet luodaan automaattisesti satunnaisuutta hyväksikäyttäen, ja niiden avulla pyritään löytämään ongelmia, joita on muilla menetelmillä haasteellista löytää.

Termi "fuzz" on peräisin 1990-luvulla toteutetusta projektista, jossa kehitetyn työkalun avulla tutkittiin erilaisten UNIX-työkalujen luotettavuutta [6][5, s. xv–xvii]. Projekti toteutettiin vuonna 1988 Barton Millerin opettamalla kurssilla University of Wisconsin-Madisonissa, ja sen pohjalta kirjoitettu artikkeli julkaistiin vuonna 1990 [6][5, s. xv–xvii, 22–23]. Projektin aiheen taustalla oli Millerin havainto, että satunnaiset virheet ohjelmien syötteessä paljastavat ohjelmissa olevia virheitä: käyttäessään tietokonetta etäyhteydellä kohisevan puhelinlinjan yli hän havaitsi kohinan aiheuttavan komentorivin syötteisiin satunnaisia virheitä, jotka ajoittain jopa kaatoivat käytettävän sovelluksen [6][5, s. xv]. Termin "fuzz" onkin myöhemmin nähty viittaavan nimenomaan tähän puhelinlinjan kohinaan [7][5, s. 1].

Projektin toteuttamaa `fuzz`-työkalua pidetään ensimmäisenä tai ainakin parhaiten tunnettuna aikaisena fuzzaustyökaluna [5, s. 22–23]. Samaa `fuzz`-työkalua käytettiin myöhemmin myös vuonna 1995 julkaistussa jatkotutkimuksessa, joka toisti ja laajensi alkuperäisen julkaisun tuloksia [8]. 1990-luvulla fuzzausta alettiin tutkia myös muualla, muun muassa Oulun yliopiston Oulu University Secure Programming Group -tutkimusryhmässä [5, s.25].

2000- ja 2010-luvuilla kehitettiin useita fuzzaustyökaluja. Nämä työkalut edustavat aiempien mustalaatikkomenetelmien lisäksi erilaisia valkolaatikkomenetelmiä, joita voitiin käyttää tietokoneiden suoritustehon kasvaessa. Esimerkki tällaisesta valkolaatikkofuzzausjärjestelmästä on Microsoftin sisäiseen käyttöönsä kehittämä SAGE [9], joka käyttää hyväkseen symbolista suoritusta.

Erityisen merkittävä askel fuzzauksen suosion ja käytettävyyden kasvussa oli kuitenkin Michał Zalewskin vuonna 2014 julkaistu AFL [10][5, s. 24], joka oli ensimmäinen merkittävä kattavuuspohjainen fuzzaustyökalu [5, s. 25]. AFL:ää seurasivat Honggfuzz, jon-

ka ensimmäinen kattavuuspohjaista fuzzausta hyödyntävä versio 0.5 julkaistiin vuonna 2015 [11], sekä LLVM:n niin ikään vuonna 2015 julkaistu libFuzzer [12][5, s. 24]. Tässä työssä keskitytään näihin kattavuuspohjaisiin työkaluihin ja erityisesti AFL++:aan, joka on edellä mainitun AFL:n jatkokehite.

3.2 Fuzzauksen tekniikoita

3.2.1 Satunnaisuus

Fuzzaukselle tyypillistä on nimenomaan satunnaisuuden käyttö testauksessa, ja fuzzattavalle ohjelmalle annettavat syötteet ovat yksinkertaisimmillaan puhtaasti satunnaisia merkkijonoja. Tämä oli ensimmäisiä fuzzauksen menetelmiä, ja sitä käytti tehokkaasti jo edellä alaluvussa 3.1 mainittu `fuzz`-työkalu [6].

Jo tuo aikainen julkaisu havaitsi tämän menetelmän myös varsin tehokkaaksi: sen avulla alustasta riippuen 24–33 % työkaluista joko kaatuivat tai jumiutuivat [6]. Myös jatkotutkimus, joka julkaistiin vuonna 1995, sai samankaltaisia tuloksia – osa testatuista työkaluista oli viiden vuoden jälkeen luotettavampia, mutta monissa oli edelleen jopa samoja aiemmin löydettyjä ongelmia [8].

Tällainen satunnaisia merkkijonoja käyttävä menetelmä sopii käytännössä kaikkiin ohjelmistoihin, jotka pystyvät lukemaan tekstisyötettä. Syöte voidaan lukea esimerkiksi tiedostosta, komentoriviltä tai internet-yhteyden kautta. Satunnaisesti luodut syötteet voidaan myös taltioida, ja jos jonkin syötteen havaitaan kaatavan tai jumittavan testattavan ohjelman, voidaan siitä tehdä testitapaus. Näin voidaan helposti testata monenlaisia eri ohjelmia. Testattavien ohjelmien sisäisestä toiminnasta ei tarvita mitään tietoa, vaan ainoastaan niiden käytöstä syötteen seurauksena tarkkaillaan. Tämä tekee menetelmästä tyypillisen mustalaatikkotestauksen menetelmän.

3.2.2 Mutaatiot

Edellä mainitun puhtaasti satunnaisiin merkkijonoihin pohjautuvan menetelmän yksinkertaisuudessa piilee kuitenkin ongelma: suuri osa sen luomista syötteistä on rakenteeltaan virheellisiä. Tällöin testattava ohjelma saattaa hylätä syötteet jo aivan alkuvaiheessa [5, s. 138][13]. Jos testattava ohjelmisto hylkää suoraan rakenteellisesti virheelliset syötteet, on täysin satunnaisia merkkijonoja luovan työkalun hyvin hankala koskaan pystyä testaamaan sitä syötteen käsittelyä pidemmälle. Tällaisten ongelmien takia moni myöhempi fuzzaustyökalu on ottanut käyttöön *mutaatioon* pohjautuvia menetelmiä.

Mutaatioita käytettäessä aloitetaan jostakin olemassaolevasta syötteestä [5, s. 137]. Yleensä alkusyöte on rakenteellisesti oikeanlainen: esimerkiksi kuvia käsittelevää kirjastoa testatessa valittaisiin ensimmäiseksi syötteeksi jokin oikea kuva. Tähän ensimmäiseen

syötteeseen fuzzaustyökalu sitten tekee muutoksia, joita kutsutaan mutaatioiksi, ja näiden mutaatioiden toteuttamiseen käytettäviä menetelmiä kutsutaan *mutaattoreiksi* [13].

Mutatoitu syöte syötetään testattavalle ohjelmalle, ja jos se aiheuttaa esimerkiksi ohjelman kaatumisen, voidaan se tallettaa ja ottaa myöhemmin uudeksi alkusyötteeksi. Näin voidaan löytää syötteitä, joita ohjelma ei hylkää suoralta kädeltä, mutta jotka saavat sen käyttäytymään uudella tavalla [13]. Tässä työssä tällaista automaattisesti löydettyä syötettä, joka aiheuttaa jonkin uuden havaittavan vaikutuksen, kutsutaan *kiinnostavaksi* syötteeksi.

Yksi mutaattori toteuttaa aina yhden tietynlaisen mutaation. Tyypillisiä mutaatioita ovat esimerkiksi yksittäisen bitin arvon vaihtaminen syötteessä ja yhden tavun (tai merkin jos kyseessä on teksti) lisääminen syötteeseen tai poistaminen syötteestä [13]. Erikoisempia käytettyjä menetelmiä ovat esimerkiksi useamman bitin arvon vaihtaminen kerrallaan, aritmeettiset operaatiot syötteen tavuilla ja tavujen korvaaminen tietyillä tunnetuilla arvoilla [14]. Lisäksi useampaa mutaattoria voidaan käyttää samanaikaisesti sen sijaan, että niitä käytettäisiin suorituskertojen välillä vain yhtä kerrallaan.

3.2.3 Geneettiset algoritmit

Mutaatioiden yhteydessä puhutaan ajoittain myös *geneettisten algoritmien* laajemmasta käsitteestä. Geneettisillä algoritmeilla tarkoitetaan menetelmiä, jotka perustuvat abstraktiin malliin biologisesta evoluutiosta [15, luku 1.1]. Niissä ratkaistavan ongelman ratkaisuja mallinnetaan yksilöinä, joiden ominaisuuksia säätelee tietokoneen käsiteltävään muotoon koodattu perimä. Tällaisista yksilöistä koostuvan populaatioon sovelletaan erilaisia operaattoreita, kuten mutaatiota, geenien vaihtoa kahden yksilön välillä (*crossover*) ja valintaoperaattoria, jolla valitaan kelpoisimmat yksilöt seuraavan sukupolven perustaksi [15, luku 1.1]. Näin voidaan ikään kuin mallintaa luonnonvalintaa, jossa kelpoisimmat yksilöt lisääntyvät, ja käyttää sitä hyväksi erilaisten ongelmien ratkaisuun.

Geneettisten algoritmien tunnusomaisena ominaisuutena pidetään nimenomaan geenien vaihtoa [15, luku 5.5]. Siinä kahden yksilön perimät katkaistaan ja yhdistetään uuden yksilön uudeksi perimäksi [15, luku 5.5]. Tällä menetelmällä voidaan luoda yksilöitä, joita pelkkien mutaatioiden avulla ei olisi helposti luotavissa.

Fuzzauksen tapauksessa yksilöitä ovat testisyötteet. Niiden perimä on itse syötteen sisältö, mutaatio toimii kuten alaluvussa 3.2.2 on kuvattu ja kelpoisuutta vastaa niinikään alaluvussa 3.2.2 esitetty syötteen kiinnostavuus. Myös kahden syötteen yhdistäminen on fuzzauksessa tunnettu operaattori [14]. Edellä kuvattuja mutaatioon pohjautuvia fuzzauksen menetelmiä voidaan siis pitää eräänlaisina geneettisten algoritmien sovellutuksina tai niihin verrattavina menetelminä.

3.3 Kattavuuspohjainen fuzzaus

Useat nykyaikaiset fuzzaustyökalut, kuten AFL++ ja libFuzzer, käyttävät käännösaikaista instrumentaatiota ohjaamaan toimintaansa [10][16][12]. Instrumentaatio merkitsee ohjelmaan lisättäviä käskyjä, joiden avulla sen toimintaa voidaan jollakin tavoin seurata [3, s. 230]. Käännösaikaisella instrumentaatiolla tarkoitetaan näiden käskyjen lisäämistä ohjelmaan sitä käännettäessä, ja käännösaikainen instrumentaatio on edellä mainittujen työkalujen pääasiallinen tapa seurata ohjelman suoritusta. Esimerkiksi AFL++:ssa on mahdollista käyttää myös dynaamista instrumentaatiota, jossa nämä käskyt lisätään suoritusaikaisesti [17].

Tällaisissa fuzzaustyökaluissa instrumentaation avulla seurataan ohjelman suorituspolkua tavoitteena selvittää onko nykyinen suorituspolku uusi vai jo nähty, ja näin automaattisesti kasvattaa testien kattavuutta löytämällä uusia testitapauksia [10][12]. Tätä menetelmää kutsutaan nimellä *kattavuuspohjainen harmaalaatikkofuzzaus* (Coverage-based Greybox Fuzzing, GCF)[18], ja menetelmä on suosittu erityisesti tehokkuutensa ja helppokäyttöisyytensä takia.

Menetelmä on tehokas molemmissa sanan merkityksissä – se sekä löytää paljon kiinnostavia testitapauksia (englanniksi *effective*) että käyttää suhteellisen vähän aikaa niiden löytämiseen (englanniksi *efficient*). Erityisesti jälkimmäinen on kattavuuspohjaisen fuzzauksen etu verrattuna valkolaatikkomenetelmiin, kuten symboliseen suoritukseen. Valkolaatikkomenetelmät voivat olla tehokkaampia ensimmäisessä merkityksessä, eli voivat tuottaa parempia tuloksia [18]. Toisaalta ne usein vaativat suuren määrän aikaa, joten ne ovat jälkimmäisessä mielessä tehottomampia [18].

Kattavuuspohjaisessa fuzzauksessa testattavalle sovellukselle annetaan jokin syöte, seurataan instrumentaation avulla löytyikö tällä syötteellä uusi polku ohjelman läpi ja tallennetaan uuden polun löytänyt syöte. Tallennettuja syötteitä taas muutetaan hieman mutaattorien avulla, syötetään ne jälleen sovellukselle ja toistetaan edellä mainittu toimenpide. Näin testauksen aikana luodaan jatkuvasti uusia testisyötteitä, jotka taas voivat toimia pohjana tuleville syötteille. [18]

Edellä kuvattu menetelmä on lähes sama kuin aiemmin kuvattu mutaation käyttö mustalaatikkofuzzauksessa, mutta se on huomattavasti tehokkaampi. Jokaista uuden suorituspolun aiheuttanutta syötettä voidaan pitää kiinnostavana, ja jokainen tällainen syöte voidaan ottaa uudeksi alkusyötteeksi.

Näin uusia ja erityisen kiinnostavia syötteitä voidaan löytää hämmästyttävän tehokkaasti: esimerkiksi `djpeg`-nimistä työkalua testattaessa AFL kykeni jopa luomaan rakenteellisesti oikean JPEG-muotoisen kuvan aloittaen vain merkkijonosta `hello` [19]. AFL siis löysi kirjastosta suorituspolkuja, joiden löytämiseksi syötteen tuli olla rakenteellisesti oikeanlainen JPEG-kuva. Vaikka alkuperäinen syöte ei edes etäisesti muistuttanut JPEG-muotoista

kuvaa, uusia suorituspolkuja löytäneet muutokset siihen muuttivat syötteen vähä vähältä lähemmäs tällaista muotoa.

Kattavuuspohjaisen fuzzauksen avulla voidaan siis kiinnostavia syötteitä löytää hyvin tehokkaasti verrattuna mustalaatikkofuzzaukseen. Tämä on merkittävä osa menetelmän edellä mainittua helppokäyttöisyyttä: hyödyllisten testitapausten luomiseen ei tarvitse käyttää merkittävästi aikaa, eivätkä ne lopu kesken fuzzaustyökalun löytäessä uusia tapauksia automaattisesti [5, s. 25]. Näin fuzzausta voidaan suorittaa jopa jatkuvasti.

4. AFL++

4.1 Työkalun kuvaus

AFL++ on avoimen lähdekoodin kattavuuspohjainen fuzzaustyökalu. Se on jatkokehitelmä aiemmasta työkalusta nimeltä *american fuzzy lop* [10], joka tunnetaan yleisesti lyhenteellä AFL, ja joka on eräs käytetyimpiä kattavuuspohjaisen fuzzauksen työkaluja [20]. AFL++ on AFL:n käyttäjäyhteisön kehittämä: alkuperäisen AFL-työkalun kehittäjä ei ole aktiivisesti päivittänyt AFL:ää vuoden 2017 jälkeen [10][16][20]. AFL++ sisältääkin alkuperäisen AFL:n ominaisuuksien lisäksi monia tuon jälkeen kehitettyjä ominaisuuksia, joita ei koskaan lisätty osaksi alkuperäistä työkalua [16][20].

AFL++ käyttää suorituspolun seuraamiseen käännösaikaista instrumentaatiota [10]. Testitapausten luomiseen käytetään AFL:n dokumentaation mukaan geneettisiä algoritmeja [10], joskaan sen käyttämä menetelmä ei automaattisesti hylkää aiempia testitapauksia parempien löytyessä [21, luku 3]. Kattavuuden mittarina kattavuuspohjaisessa fuzzauksessa AFL++ käyttää eräänlaista haarakattavuuden hybridimuotoa [21, luku 1][20].

4.2 Fuzzaaminen AFL++:lla

4.2.1 Tyypillinen työkalun käyttö

AFL++:aa käytetään ensisijaisesti kattavuuspohjaiseen fuzzaukseen käyttäen käännösaikaista instrumentaatiota. Tällöin testaukseen tarvitaan kolme asiaa: kääntämiseen ja fuzzaukseen käytetyt työkalut, sovelluksen lähdekoodi ja ainakin yksi syöte. Sovelluksen tulee olla kirjoitettu C- tai C++-ohjelmointikielillä, ja lähtösyöte ei saa suoraan kaataa sovellusta. AFL++ tukee Unixin kaltaisia käyttöjärjestelmiä kuten Linux, eri BSD:t sekä MacOS, mutta ei tue esimerkiksi Windowsia [22].

Fuzzattavan sovelluksen kääntämiseen käytetään joko GCC- tai Clang-kääntäjää [22]. Sovellusta käännettäessä fuzzausta varten käytetään suoraan kääntäjän ajamisen sijaan AFL++:n mukana toimitettavaa `afl-cc`-apuohjelmaa [23, osio 1]. Tämä apuohjelma suorittaa varsinaisen kääntäjäohjelman ja lisää samalla instrumentaation käännettävään ohjelmaan.

Kääntämiseen käytetyn apuohjelman avulla voidaan voidaan myös esimerkiksi vaihtaa kattavuutta seuraavan instrumentaation tyyppiä [23, osiot 1–2]. Lisäksi sillä voidaan lisätä muunlaista testausta auttavaa instrumentaatiota, esimerkiksi muistivirheitä tarkistavat ASAN ja MSAN [23, osio 1]. Tällainen lisäinstrumentaatio voi esimerkiksi tarkoituksellisesti kaataa ohjelman, kun siinä havaitaan muistinkäsittelyvirhe, jolloin virheen aiheuttava syöte voidaan löytää fuzzaamalla.

Itse fuzzaus suoritetaan `afl-fuzz`-ohjelmalla. Se vaatii kolme pakollista parametria, joista ensimmäinen on lähtösyötteet sisältävä hakemisto, toinen fuzzauksen tulosten tallettamiseen käytettävä hakemisto ja kolmas fuzzattavan sovelluksen käynnistämiseen käytettävä komentorivi. AFL++ aloittaa syötehakemistossa olevista lähtösyötteistä ja taltioi löytämänsä kiinnostavat uudet syötteet tuloshakemiston alihakemistoon `queue`. Lisäksi tuloshakemistoon taltioidaan fuzzausistuntoon liittyviä tilatietoja, ja omiin alihakemistoihinsa `hangs` ja `crashes` uniikit, virheitä aiheuttaneet syötteet. Näistä `hangs` sisältää ohjelman jumittaneet syötteet ja `crashes` ohjelman kataneet syötteet. Fuzzausistunnon päättymisen jälkeen näitä tuloksia voidaan esimerkiksi jatkokäsitellä erilaisilla työkaluilla.

4.2.2 Fuzzaus ilman lähdekoodia

Jos fuzzattavan ohjelman lähdekoodi ei ole saatavilla, tai se on kirjoitettu ohjelmointikielellä jota AFL++ ei tue, ei käännösaikaista instrumentaatiota voida lisätä edellä kuvatulla tavalla. Osassa näistäkin tapauksista AFL++:aa kuitenkin voidaan käyttää, sillä se sisältää useita erilaisia moodeja, jotka mahdollistavat testauksen ilman käännösaikaista instrumentaatiota tai edes ohjelman lähdekoodia.

Yleisimmin ilman lähdekoodia suoritettavaan fuzzaukseen AFL++:lla käytetään QEMU-moodia, jossa sovellusta suoritetaan QEMU:n prosessoriemulaattorin [24] avulla. Tällöin suoritusta seurataan emulaattorin kautta, eikä käännösaikaista instrumentaatiota ohjelman sisällä tarvita. QEMU-moodin haittapuolena se on kuitenkin huomattavasti hitaampi, jopa 50 % hitaampi kuin käännösaikaisen instrumentaation käyttö [17]. AFL++, toisin kuin alkuperäinen AFL, tukee kuitenkin myös QEMU-moodissa niin sanottua *pysyvää moodia* (persistent mode), jossa koko ohjelman sijaan testattavasta ohjelmasta suoritetaan vain tietty haluttu osa silmukassa [17][25], ja joka nopeuttaa fuzzausta huomattavasti.

AFL++ tarjoaa myös joitakin harvemmin käytettyjä moodeja ilman lähdekoodia tapahtuvaan fuzzaukseen. Esimerkiksi FRIDA-moodin avulla voidaan testata kirjastoja, ja se toimii hyvin MacOS-käyttöjärjestelmässä [17]. Lisäksi on mahdollista käyttää erilaisia ulkoisia työkaluja instrumentaation lisäämiseen. Esimerkiksi RetroWrite kykenee lisäämään tietynlaisiin sovelluksiin instrumentaation ilman alkuperäistä lähdekoodia purkamalla sovellusbinäärin takaisin assembly-koodiksi [26].

4.3 AFL++:aa laajentavat työkalut

AFL++ sisältää monia ominaisuuksia, jotka eivät kuulu alkuperäiseen AFL-työkaluun. Sen lisäksi on myös olemassa lukuisia lisätyökaluja, joilla esimerkiksi AFL++:n tuloksia voidaan jatkokäsitellä, tai joilla se saadaan tukemaan alustoja tai ohjelmointikieliä, joita pelkkä AFL++ itsessään ei tue.

4.3.1 Tuki muille ohjelmointikielille ja alustoille

AFL++ tukee lähinnä C- ja C++-ohjelmointikieliä, jotka ovat varsin yleisiä. Moni ohjelma ei kuitenkaan ole kirjoitettu kummallakaan näistä ohjelmointikielistä, jolloin niitä ei voida suoraan testata AFL++:lla. Lisäksi AFL++ ei suoraan tue Windows-käyttöjärjestelmää lainkaan. Tällaisia tapauksia varten on kehitetty sekä erilaisia muokattuja AFL:n tai AFL++:n versioita että ulkoisia työkaluja, joilla se saadaan tukemaan muita alustoja ja ohjelmointikieliä.

Esimerkki työkalusta, jonka avulla AFL++ saadaan tukemaan muita ohjelmointikieliä, on python-afl [27]. Tämä työkalu laajentaa AFL++:n toimimaan Python-ohjelmointikielisten ohjelmien kanssa. Python-koodia ei tarvitse etukäteen kääntää, mutta myös se tarvitsee instrumentoida, jotta AFL++ voi seurata sen suorituspolkua. Tätä varten python-afl sisältää kirjaston, jota tulee kutsua testattavasta sovelluksesta ennen fuzzattavan osan alkua, sekä `py-afl-fuzz`-skriptin, joka suoritetaan `afl-fuzz`:n sijaan [27].

AFL++ käyttää Linux- ja Unix-järjestelmien ominaisuuksia, joita Windows ei tarjoa, eikä siis tue Windowsia [28]. AFL:stä on kuitenkin kehitetty useita Windowsia tukevia versioita, huomionarvoisena erityisesti Ivan Fratricin ylläpitämä WinAFL [28][17], joka perustuu DynamoRIO-instrumentaatioalustaan [29]. Näiden Windowsia tukevien AFL:n versioiden yhteinen ongelma on se, että ne ovat hitaita, jopa 99 % hitaampia kuin normaali suoritus [17]. Ne kuitenkin mahdollistavat AFL:n käytön fuzzaamiseen alustalla, jolla se ei muuten olisi mahdollista.

4.3.2 Tulosten jatkokäsittely

AFL++ yrittää jo suoritusaikaisesti tunnistaa, mitkä ohjelman kaatumiset ja jumiutumiset ovat uniikkeja. Useat "uniikit" syötteet saattavat silti paljastaa yhden ja saman ongelman, joka vain löydettiin usealla eri suorituspolulla. Ohjelman kaataneiden ja jumittaneiden syötteiden joukko halutaan tyypillisesti minimoida niin, että jokaista ohjelmassa olevaa virhettä kohti olisi vain yksi uniikki testitapaus, joka aiheuttaa kyseisen ongelman. Tästä syystä sen tulosten jatkokehittelyyn on olemassa useita työkaluja.

Tuloksia voidaan jatkokäsitellä jo AFL++:n itsensä sisältämillä apuohjelmilla, joista `afl-tmin` on tarkoitettu yksittäisen testitapauksen minimointiin ja `afl-cmin` koko testita-

pausten joukon karsintaan [16]. Näiden lisäksi tutkijat ovat kehittäneet erilaisia kehittyneempiä työkaluja tulosten käsittelyyn.

Eräs tällainen tuloksia jatkokäsittelevä työkalu on Igor, joka on AFL++:n tuloksena löydettyjen uniikkien virheiden joukon kaventamiseen tehty työkalu [30]. Igorin avulla tätä joukkoa voidaan kaventaa niin, että se vastaa paremmin todellista löydettyjen virheiden määrää, ja on siten hyödyllisempi esimerkiksi bugien raportointiin. Työkalu minimoi ensin jokaisen yksittäisen syötteen aiheuttaman suorituspolun mahdollisimman yksinkertaiseksi niin, että se kuitenkin aiheuttaa saman virheen. Tämän jälkeen minimoidut syötteet ryhmitetään niiden aiheuttaman virheen mukaan. Näin jokaista eri virheen juurisyytä kohden jää joukko yksinkertaistettuja syötteitä, ja tutkijoiden mukaan Igor pystyykin pienentämään tunnistettujen uniikkien virheiden määrää jopa kertaluokkaa pienemmäksi kuin sitä edeltäneet työkalut. [30]

5. FUZZAUS MAAILMALLA

5.1 Fuzzausprojektit

Kuten luvussa 3.3 mainittiin, kattavuusohjatun fuzzauksen avulla voidaan toteuttaa jatkuvaa fuzzausta, eli fuzzata samaa sovellusta automaattisesti pitkiä aikoja luomatta käsin uusia syötteitä. Tämä on mahdollistanut fuzzausprojektit, jotka voivat näin käyttää suurta määrää suoritusaikaa kriittisten ohjelmistojen fuzz-testaukseen löytääkseen niistä mahdollisia tietoturvaongelmia.

Eräs merkittävä tällainen fuzzausprojekti on Googlen johtama OSS-Fuzz-projekti. Projektin tavoitteena on etsiä tietoturvaongelmia yleisesti käytössä olevista avoimen lähdekoodin sovelluksista ja kirjastoista, ja sen avulla on löydetty yhteensä kymmeniä tuhansia virheitä sadoista eri projekteista [31]. Projekti käyttää fuzzaukseen AFL++:n lisäksi Honggfuzz- ja libFuzzer-työkaluja [31].

OSS-Fuzz-projekti käyttää testien suorittamiseen Googlen kehittämää ClusterFuzz-alustaa, joka hallinnoi testaamiseen käytettyjä virtuaalikoneita, minimoi löydetyt testitapaukset, ja kykenee monissa tapauksissa raportoimaan automaattisesti löytämänsä virheet. Google käyttää alustaa myös omassa sisäisessä projektissaan omien tuotteidensa fuzzaamiseen 30 000 virtuaalikoneen avulla. [32]

Aiemmin huomattava projekti oli myös Hanno Böckin johtama The Fuzzing Project, joka aloitettiin sen jälkeen, kun fuzzaamalla oli löydetty useita tietoturvaongelmia vuonna 2014 [33]. Projekti on kuitenkin sittemmin ilmeisesti lakannut toimimasta, eikä sen blogia ole päivitetty vuoden 2019 jälkeen.

5.2 Mitä fuzzauksella voidaan saavuttaa

Kuten luvussa 2 on mainittu, etukäteen suunnitelluilla testitapauksilla on usein hankala ennakoida kaikkia mahdollisia ongelmakohtia. Tämä voi johtua monesta seikasta, kuten testattavan ohjelman monimutkaisuudesta tai esimerkiksi siitä, ettei ohjelmiston kehittäjä tai testien suunnittelija osannut tunnistaa virheen mahdollisuutta.

Tämän seurauksena moneen ohjelmaan jää hankalasti löydettäviä ja odottamattomia ongelmia, ja tällaiset ongelmat voivat säilyä ohjelmissa huomaamattomina hyvinkin pitkiä

aikoja. Erityisen hyvä esimerkki on Shellshock-nimellä tunnettu merkittävä haavoittuvuus bash-komentotulkissa: löytyessään vuonna 2014 haavoittuvuus oli ollut olemassa ohjelman koodissa ainakin 25 vuotta [34]. Bash on laajalti käytetty avoimen lähdekoodin ohjelma, jota oli myös kehitetty koko tuon ajan, joten tällaisen haavoittuvuuden löytyminen oli erittäin yllättävää.

Shellshock on tämän työn kannalta huomattava tapaus myös siksi, että fuzzaamista AFL:llä käytettiin apuna haavoittuvuuden korjaamiseksi. Alkuperäiset korjaukset eivät poistaneet haavoittuvuutta kokonaan [34], vaan uusia, edelleen hyväksikäytettäviä muotoja samasta haavoittuvuudesta löydettiin pian. Näistä vioista usean löysivät tunnetut tietoturvatutkijat, mutta ainakin kaksi muuten havaitsematonta vikaa löysi AFL:n avulla Michał Zalewski, AFL:n alkuperäinen kehittäjä [1].

Shellshock löytyi alle vuosi AFL:n julkaisun jälkeen, joten tämä tapaus oli osaltaan tekemässä työkalua tunnetuksi. Sillä löydettiin samana vuonna myös muita tietoturvaongelmia, esimerkiksi `strings`-työkalussa olevia mahdollisesti hyväksikäytettäviä haavoittuvuuksia [35], jotka nekin olivat monille yllättäviä. Fuzzauksen ja erityisesti AFL-työkalun käyttö tietoturvaongelmien löytämisessä sai näin vuoden 2014 lopulla paljon huomiota. Osittain juuri tästä syystä perustettiin edellä alaluvussa 5.1 mainittu The Fuzzing Project [36].

Monet tällaiset viat on helppo löytää fuzzaamalla, vaikka ne olisivat olleet ohjelmistossa huomaamatta hyvinkin pitkiä aikoja. Fuzzauksen vahvuutena voidaan siis pitää erityisesti ennakoimattomien, yllättävien vikojen löytämistä. Tällaisia vikoja ei ole välttämättä löydetty edes vuosikymmenien aikana, ja ne voivat olla hyvin merkittäviä. Esimerkiksi edellä käsitelty Shellshock vaikutti hyvin suureen osaan kaikista verkkopalvelimista. Fuzzaaminen myös tuottaa tällaiset ongelmat löytäessään valmiita testisyötteitä, jotka voidaan ottaa käyttöön ohjelman testauksessa.

5.3 Mitä fuzzauksella ei voida saavuttaa

Fuzzaus ei kuitenkaan sovellu sellaiseen ohjelmien testaukseen, jossa testien tulee varmistaa jonkin tietyn vian olevan korjattu tai jonkin vaatimuksen olevan täytetty, sillä se perustuu pitkälti satunnaisuuteen eikä ennalta määriteltyihin testitapauksiin. Esimerkiksi Besseyn et al. mukaan käyttäjät haluavat käyttää virheitä etsiviä työkaluja usein sen varmistamiseen, ettei vikoja ole ollenkaan [37]. Lisäksi työkaluilla halutaan saada ajosta ajoon sama tulos, jolloin satunnaisuutta ei voida käyttää [37].

Vian löytämiseen fuzzaamalla voi kulua suurikin määrä aikaa. Sillä ei voida taata ohjelman täydellistä kattavuutta, ja tietynlaiset ongelmat voivat olla myös kattavuuspohjaiselle fuzzaukselle ylitsepääsemättömiä tai vähintään hyvin hankalia. Esimerkiksi jos virheen aiheuttaa yksittäinen syöte, joka ei ole lähellä mitään muuta kiinnostavaa syötettä, sen

löytämiseen voi kulua erittäin kauan.

Fuzzattavaa ohjelmaa joudutaan usein myös muokkaamaan fuzzaamista varten, sillä moni ohjelma ei ota syötettään komentorivillä annetusta tiedostosta tai standardisyötteestä. Lisäksi ohjelman osa, jota halutaan testata, ei voi vaatia syötteen olevan salattu tai kryptografisesti varmennettu. Tällaisen ohjelman testaamista varten fuzzaustyökalun pitäisi onnistuneesti purkaa salaus tai varmennus voidakseen luoda toimivan syötteen. Vaikka fuzzaus itsessään on nykyaikaisilla työkaluilla melko yksinkertaista, voi sen aloittaminen joissain tapauksissa vaatia siis huomattavan määrän työtä.

5.4 Tulevaisuudennäkymät

Vaikka fuzz-testaus ei olekaan sellaista toistettavaa, determinististä testausta johon testiautomaatiota usein käytetään, on nykyään kuitenkin tarjolla tapoja yhdistää fuzzaus esimerkiksi nykyaikaisessa ohjelmistokehityksessä usein käytettyihin jatkuvan integraation järjestelmiin. Eräs esimerkki on Microsoftin OneFuzz [38], joka on suunniteltu nimenomaan tätä tarkoitusta varten. Se käyttää Microsoftin Azure-palvelua fuzzaukseen käytettävien virtuaalikoneiden luomiseen, ja sen avulla fuzzausajon voi käynnistää helpoimmillaan vain yhdellä komentorivikomennolla [39]. Näin se on helppo liittää Microsoftin palveluja käyttävään järjestelmään.

Tällaisten helposti olemassaoleviin järjestelmiin yhdistettävien työkalujen saatavuus mahdollistaa helposti jo ohjelmistokehityksen aikaista jatkuvaa fuzzauksen käyttöä. Erityisen merkittäviä tällaiset pilvipohjaiset työkalut ovat organisaatioille, joilla ei ole valmista kapasiteettia tai halua fuzzausjärjestelmän ylläpitoon. Nähtäväksi kuitenkin jää, tuleeko tällaisesta fuzzauksesta todellisuudessa merkittävä osa tulevaisuuden automaattista ohjelmistotestausta.

Kattavuuspohjaisen fuzzauksen suosio ei ainakaan vaikuta heikkenevältä. Artikkelissaan vuonna 2020 AFL++:n kehittäjiin kuuluvat Fioraldi et al. totesivat AFL:n olevan eräs suosituimmista ja menestyneimmistä kattavuuspohjaisista fuzzaustyökaluista [20]. Kirjoittajat toivoivat AFL++:n toimivan tulevaisuuden fuzzaustutkimuksen perustana [20], ja AFL++:aan viittaavia tai siihen pohjautuvia julkaisuja onkin tämän jälkeen ilmestynyt useita [40]. Esimerkiksi aiemmin mainittua Igor-työkalua käsittelevä artikkeli [30] julkaistiin tämän työn kirjoittamisen aikana. Aiemmin mainitut OneFuzz- ja ClusterFuzz -alustat tukevat myös erityisesti kattavuuspohjaisia työkaluja [39][32].

Myös perinteisen, satunnaisiin merkkijonoihin perustuvan mustalaatikkofuzzauksen tulevaisuus näyttää yllättävän valoisalta. Miller toisti alkuperäisen `fuzz`-työkalun esitelleen tutkimuksen [6] tuloksia ensimmäisen kerran vuonna 1995 [8], ja on sen jälkeen edelleen toistanut eri tutkijoiden kanssa noin viiden vuoden välein joko samankaltaisen tai hieman erilaisen fuzzausprojektin. Viimeisimmässä, vuonna 2020 julkaistussa tutkimuksessa tä-

mä 90-luvulta peräisin ollut työkalu pystyi edelleen kaatamaan tai jumittamaan alustasta riippuen 12–19 % testatuista työkaluista [41].

Fuzzaus löytää edelleen myös uudenlaisia käyttötarkoituksia. Esimerkiksi Jattke et al. ovat kehittäneet Blacksmith-nimisen työkalun, joka fuzzaamalla löytää tapoja kiertää tietokoneiden DRAM-muistin ominaisuuksia, joilla pyrittiin korjaamaan niin sanottu Rowhammer-haavoittuvuus [42]. Se siis fuzzaa tietokoneen fyysistä muistia haavoittuvuuden löytämiseksi ohjelmiston avulla.

6. YHTEENVETO

Ohjelmistotestauksessa käytetään useita eri menetelmiä, ja eräs sen oleellisimmista tavoitteista on löytää ohjelmissa olevia virheitä. Ohjelmia testataan usein ennakkoon määriteltyjen testitapausten avulla, ja niitä voidaan suorittaa sekä manuaalisesti että automaattisesti. Testikattavuus kertoo, kuinka suurta osaa ohjelmasta testitapauksilla voidaan testata, ja kattavuutta voidaan mitata useilla eri mittareilla. Testausmenetelmiä voidaan jaotella esimerkiksi valko-, musta- ja harmaalaatikkotestauksen menetelmiin. Ennalta suunniteltuihin testitapauksiin perustuvan testauksen avulla voidaan harvoin löytää kaikkia ohjelmassa olevia ongelmia, vaan tähän tarvitaan muunlaisia menetelmiä.

Fuzz-testaus on tehokas tapa löytää ohjelmissa olevia virheitä satunnaisuutta hyväksikäyttäen, ja fuzzausta on käytetty tähän jo 1990-luvulta asti. Fuzzausta käytetään erityisesti tietoturva-alalla. Nykyaikainen fuzzaus perustuu suurimmaksi osaksi jo pitkään tunnettuihin menetelmiin, kuten satunnaisuuteen ja mutaatioiden käyttöön testitapausten luomisessa. Aikaisimmat fuzzauksen menetelmät ovat pääasiassa mustalaatikkomenetelmiä.

Nykyfuzzaukselle oleellinen edistysaskel on eräs harmaalaatikkomenetelmä, kattavuuspohjainen fuzzaus. Kattavuuspohjaisen fuzzauksen avulla voidaan löytää ohjelmissa olevia virheitä huomattavasti aiempia mustalaatikkomenetelmiä tehokkaammin, ja se mahdollistaa tehokkaan fuzzauksen ilman suurta määrää valmiita testitapauksia. Sen avulla voidaan fuzz-testausta suorittaa jopa jatkuvasti.

AFL oli ensimmäisiä kattavuuspohjaisia fuzzaustyökaluja, ja on niistä myös tunnetuimpia. AFL:n sekä sen jatkokehitettyjen versioiden avulla on löydetty useita merkittäviä bugeja tunnetuista ohjelmistoista. AFL++ on käyttäjäyhteisön AFL:sta jatkokehittämä versio, joka sisältää useita parannuksia ja uusia ominaisuuksia.

AFL++ käyttää pääasiassa käännösaikaista instrumentaatiota, mutta kykenee myös muunlaiseen suorituspolun seurantaan. AFL++ tukee pääasiassa C- ja C++-ohjelmointikieliä ja Unixin kaltaisia käyttöjärjestelmiä, mutta on olemassa työkaluja, joilla se voidaan laajentaa toimimaan muiden ohjelmointikielien ja käyttöjärjestelmien kanssa. Sen tuloksia halutaan usein jatkokäsitellä uniikkeja virheitä löytävien syötteiden tuottamiseksi, ja tällaiseen jatkokäsittelyyn on olemassa useita työkaluja.

Kattavuuspohjainen fuzzaus on mahdollistanut jatkuvan fuzzauksen, ja tällaisista jatku-

van fuzzauksen projekteista ehkä merkittävin esimerkki on OSS-Fuzz. Tämän projektin tavoitteena on etsiä yleisesti käytössä olevista avoimen lähdekoodin ohjelmistoista tietoturvaongelmia. Aiemmin huomattava on ollut myös The Fuzzing Project.

Fuzzauksen avulla voidaan löytää ohjelmista ongelmia, joita on muilla menetelmillä hankala löytää. Esimerkiksi bash-komentotulkissa olleen Shellshock-haavoittuvuuden eri muotojen löytämisessä käytettiin hyväksi fuzzausta. Samoin `strings`-työkalussa olevat haavoittuvuudet löydettiin fuzzaamalla. Fuzzauksella löydettävät ongelmat ovat usein sellaisia, joita on vaikeaa ennakoida, ja siten niiden löytämiseksi on hankala suunnitella testejä.

Fuzzaus tuottaa löytämilleen virheille valmiita testisyötteitä. Fuzzaus ei sovellu vaatimusten varmistamiseen, tai sen tarkistamiseen ettei ohjelmassa ole lainkaan ongelmia. Lisäksi joitakin vikoja on hankala löytää fuzzauksen avulla, ja testattava ohjelma saattaa vaatia huomattavan määrän muutoksia, jotta sitä voitaisiin fuzzata.

Fuzzaaminen näyttää myös tulevaisuudessa oleelliselta menetelmältä ohjelmistojen testauksessa, ja sen käyttö testiautomaation yhteydessä saattaa yleistyä. Niin nykyaikainen fuzzaus kuin vanhemmat mustalaatikkomenetelmätkin ovat edelleen ajankohtaisia, mutta erityisesti kattavuuspohjaisen fuzzauksen tutkimus on pinnalla.

Työ vastasi pinnallisella tasolla tutkimuskysymykseen. Fuzzaus ja mitä sillä voidaan saavuttaa suhteessa muunlaiseen ohjelmistotestaukseen esiteltiin, samoin kuin AFL++-työkalun käyttö ja joitakin AFL++:aa laajentavia työkaluja. Helposti rajattavia esimerkkejä siitä, mitä tuloksia fuzzauksen avulla on saavutettu, oli hankala löytää, mutta muutama sellainen esiteltiin luvuissa 3.3, 5.2, ja 5.4.

Ongelmia esiintyi tulosesimerkkien löytämisen lisäksi myös tiiviin mutta tarpeeksi kattavan ohjelmistotestauksen määritelmän kokoamisessa, ja oleellisten testauksen käsitteiden esittelemisessä. Suuri osa työn lähteistä oli pakon sanelemana verkkolähteitä, sillä esimerkiksi AFL++-työkalun kuvauksesta suurin osa on julkaistu ainoastaan verkossa.

Tämän työn kaltaista perustason kuvausta fuzzauksesta ei ainakaan tutkimuksen aikana löytynyt, mutta se on varsin pintapuolinen. Työn tuloksia voisi syventää nykyaikaisen fuzzauksen kannalta laajemmalla, järjestelmällisellä katsauksella viime vuosien julkaisuihin fuzzauksen alalla. Lisäksi työtä voisi laajentaa käytännön kuvauksella AFL++-fuzzaustyökalun käytöstä.

Aiheeseen voi syventyä tässä työssä lähteinä käytetyn materiaalin perusteella. Laajemmin itse fuzzauksen alaa käsittelee esimerkiksi *Fuzzing for Software Security Testing and Quality Assurance* [5]. Fuzzauksen ja fuzzaustyökalujen toteutusta käytännössä käsittelee verkossa julkaistu *The Fuzzing Book* [43]. AFL++-työkalusta syventävää tietoa löytyy AFL++:n verkkosivuilta [16], sekä Fioraldin et al. julkaisusta AFL++: Combining Incremental Steps of Fuzzing Research [20].

LÄHTEET

[1] Zalewski, M. *Bash bug: the other two RCEs, or how we chipped away at the original fix (CVE-2014-6277 and '78)*. 1. lokakuuta 2014. URL: https://lcamtuf.blogspot.com/2014/10/bash-bug-how-we-finally-cracked.html (viitattu 25. 10. 2021).

[2] Myers, G. J., Sandler, C. ja Badgett, T. *The Art of Software Testing*. 3. painos. Wiley Publishing, 2011. ISBN: 1118031962.

[3] *ISO/IEC/IEEE 24765:2017. ISO/IEC/IEEE International Standard - Systems and software engineering – Vocabulary*. 2017. DOI: 10.1109/IEEESTD.2017.8016712.

[4] Fowler, M. *Continuous Integration*. 1. toukokuuta 2006. URL: https://martinfowler.com/articles/continuousIntegration.html (viitattu 31. 12. 2021).

[5] Takanen, A., DeMott, J., Miller, C. ja Kettunen, A. *Fuzzing for Software Security Testing and Quality Assurance*. 2. painos. Boston, MA: Artech House, 2018. ISBN: 9781608078509.

[6] Miller, B. P., Fredriksen, L. ja So, B. An Empirical Study of the Reliability of UNIX Utilities. *Communications of the ACM* 33.12 (joulukuu 1990), s. 32–44. ISSN: 0001-0782. DOI: 10.1145/96267.96279.

[7] Oehlert, P. Violating assumptions with fuzzing. *IEEE Security & Privacy* 3.2 (2005), s. 58–62. DOI: 10.1109/MSP.2005.55.

[8] Miller, B. P., Koski, D. R., Lee, C. P., Maganty, V., Murthy, R., Natarajan, A. ja Steidl, J. Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services (1995). URL: http://digital.library.wisc.edu/1793/59964.

[9] Godefroid, P., Levin, M. Y. ja Molnar, D. SAGE: Whitebox Fuzzing for Security Testing. *Communications of the ACM* 55.3 (maaliskuu 2012), s. 40–44. ISSN: 0001-0782. DOI: 10.1145/2093548.2093564.

[10] Zalewski, M. *american fuzzy lop*. URL: https://lcamtuf.coredump.cx/afl/ (viitattu 31. 12. 2021).

[11] *CHANGELOG*. URL: https://github.com/google/honggfuzz/blob/2.4/CHANGELOG (viitattu 31. 12. 2021).

[12] *libFuzzer – a library for coverage-guided fuzz testing. — LLVM 13 documentation*. URL: https://llvm.org/docs/LibFuzzer.html (viitattu 07. 11. 2021).

[13] Zeller, A., Gopinath, R., Böhme, M., Fraser, G. ja Holler, C. Mutation-Based Fuzzing. *The Fuzzing Book*. CISPA Helmholtz Center for Information Security,

2021. URL: https://www.fuzzingbook.org/html/MutationFuzzer.html (viitattu 04. 11. 2021).

[14] Zalewski, M. *Binary fuzzing strategies: what works, what doesn't*. 8. elokuuta 2014. URL: https://lcamtuf.blogspot.com/2014/08/binary-fuzzing-strategies-what-works.html (viitattu 23. 11. 2021).

[15] Mitchell, M. *An introduction to genetic algorithms*. Cambridge, MA, USA: MIT Press, 1996. ISBN: 0-262-13316-4.

[16] *AFL++ Overview*. URL: https://aflplus.plus/ (viitattu 03. 10. 2021).

[17] *Fuzzing binary-only programs with AFL++*. URL: https://aflplus.plus/docs/binaryonly_fuzzing/ (viitattu 08. 11. 2021).

[18] Böhme, M., Pham, V.-T. ja Roychoudhury, A. Coverage-Based Greybox Fuzzing as Markov Chain. *IEEE Transactions on Software Engineering* 45.5 (2019), s. 489–506. DOI: 10.1109/TSE.2017.2785841.

[19] Zalewski, M. *Pulling JPEGs out of thin air*. 7. marraskuuta 2014. URL: https://lcamtuf.blogspot.com/2014/11/pulling-jpegs-out-of-thin-air.html (viitattu 04. 10. 2021).

[20] Fioraldi, A., Maier, D., Eißfeldt, H. ja Heuse, M. AFL++: Combining Incremental Steps of Fuzzing Research. *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, elokuu 2020. URL: https://www.usenix.org/conference/woot20/presentation/fioraldi (viitattu 21. 12. 2021).

[21] *Technical "whitepaper" for afl-fuzz*. URL: https://aflplus.plus/docs/technical_details/ (viitattu 31. 12. 2021).

[22] *Installation instructions*. URL: https://aflplus.plus/docs/install/ (viitattu 12. 01. 2022).

[23] *Environmental variables*. URL: https://aflplus.plus/docs/env_variables/ (viitattu 12. 01. 2022).

[24] *About QEMU — QEMU documentation*. URL: https://www.qemu.org/docs/master/about/index.html (viitattu 08. 11. 2021).

[25] *How to use the persistent mode in AFL++'s QEMU mode*. URL: https://github.com/AFLplusplus/AFLplusplus/blob/3.14c/qemu_mode/README.persistent.md (viitattu 21. 12. 2021).

[26] Dinesh, S., Burow, N., Xu, D. ja Payer, M. RetroWrite: Statically Instrumenting COTS Binaries for Fuzzing and Sanitization. *2020 IEEE Symposium on Security and Privacy (SP)*. 2020, s. 1497–1511. DOI: 10.1109/SP40000.2020.00009.

[27] *GitHub - jwilk/python-afl: American Fuzzy Lop fork server and instrumentation for pure-Python code*. URL: https://github.com/jwilk/python-afl (viitattu 12. 01. 2022).

[28] *GitHub - googleprojectzero/winafl: A fork of AFL for fuzzing Windows binaries*. URL: https://github.com/googleprojectzero/winafl (viitattu 14. 11. 2021).

[29] *Home*. URL: https://dynamorio.org/ (viitattu 14. 11. 2021).

[30] Jiang, Z., Jiang, X., Hazimeh, A., Tang, C., Zhang, C. ja Payer, M. Igor: Crash Deduplication Through Root-Cause Clustering. *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 2021, s. 3318–3336. ISBN: 9781450384544. DOI: 10.1145/3460120.3485364.

[31] Google. *OSS-Fuzz - OSS-Fuzz*. URL: https://google.github.io/oss-fuzz/ (viitattu 03. 10. 2021).

[32] *ClusterFuzz - ClusterFuzz*. URL: https://google.github.io/clusterfuzz/ (viitattu 15. 11. 2021).

[33] *The Fuzzing Project - about*. URL: https://fuzzing-project.org/about.html (viitattu 17. 01. 2022).

[34] Leyden, J. Patch Bash NOW: 'Shellshock' bug blasts OS X, Linux systems wide open. *The Register* (24. syyskuuta 2014). URL: https://www.theregister.com/2014/09/24/bash_shell_vuln/ (viitattu 17. 01. 2022).

[35] Zalewski, M. *PSA: don't run 'strings' on untrusted files (CVE-2014-8485)*. 24. lokakuuta 2014. URL: https://lcamtuf.blogspot.com/2014/10/psa-dont-run-strings-on-untrusted-files.html (viitattu 17. 01. 2022).

[36] *The Fuzzing Project - Misc*. URL: https://fuzzing-project.org/background.html (viitattu 17. 01. 2022).

[37] Bessey, A., Block, K., Chelf, B., Chou, A., Fulton, B., Hallem, S., Henri-Gros, C., Kamsky, A., McPeak, S. ja Engler, D. A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World. *Communications of the ACM* 53.2 (helmikuu 2010), s. 66–75. DOI: 10.1145/1646353.1646374.

[38] *GitHub - microsoft/onefuzz: A self-hosted Fuzzing-As-A-Service platform*. URL: https://github.com/microsoft/onefuzz (viitattu 21. 12. 2021).

[39] *Getting started using OneFuzz*. URL: https://github.com/microsoft/onefuzz/blob/4.0.0/docs/getting-started.md (viitattu 21. 12. 2021).

[40] *Papers*. URL: https://aflplus.plus/papers/ (viitattu 30. 12. 2021).

[41] Miller, B., Zhang, M. ja Heymann, E. The Relevance of Classic Fuzz Testing: Have We Solved This One? *IEEE Transactions on Software Engineering* (2020), s. 1–1. DOI: 10.1109/TSE.2020.3047766.

[42] Jattke, P., van der Veen, V., Frigo, P., Gunter, S. ja Razavi, K. BLACKSMITH: Rowhammering in the Frequency Domain. *IEEE Symposium on Security & Privacy 2022*. Marraskuu 2021. URL: https://comsec.ethz.ch/wp-content/files/blacksmith_sp22.pdf. Esijulkaistu.

[43] Zeller, A., Gopinath, R., Böhme, M., Fraser, G. ja Holler, C. *The Fuzzing Book*. CISPA Helmholtz Center for Information Security, 2021. URL: https://www.fuzzingbook.org/ (viitattu 12. 03. 2021).