Tampereen yliopisto

Akseli Käppi

KATSAUS WEBASSEMBLYYN JA SEN NYKYTILANTEESEEN

Informaatioteknologian ja viestinnän tiedekunta (ITC)
Kandidaatintyö
Toukokuu 2020

TIIVISTELMÄ

Akseli Käppi: Katsaus WebAssemblyyn ja sen nykytilanteeseen
Kandidaatintyö
Tampereen yliopisto
Tieto- ja sähkötekniikan TkK-tutkinto-ohjelma
Toukokuu 2020

Tämän kandidaatintyön tarkoituksena oli perehtyä WebAssemblyyn ja luoda hyvä yleiskuva WebAssemblysta ja sen nykytilanteesta perehtymällä olemassaolevaan kirjallisuuteen. Työn päätavoitteena olikin selvittää mikä ja mitä on WebAssembly, miksi se on olemassa, ja mikä on WebAssemblyn nykytilanne.

WebAssembly on matalan tason tavukoodi, joka voidaan myös esittää ohjelmointikielenä, joka muistuttaa assembly-ohjelmointikieltä. Se on tarkoitettu käännöskohteeksi muille ohjelmointikielille, ja siihen kääntämistä tukevatkin useat eri ohjelmointikielet, esimerkiksi C/C++ -ohjelmointikielet. Sen päätavoitteet ovat turvallisuus, suorituskyky, siirrettävyys ja tiiviys. WebAssemblysta erityisen tekee se, että sen suorittamista tukee suurin osa käytössä olevista internetselaimista, jotka aikaisemmin ovat tukeneet ainoastaan JavaScriptin suorittamista. WebAssemblyn suunnittelun takana ovatkin suurimmat selainvalmistajat Google, Apple, Mozilla ja Microsoft.

Websovelluksina tarjotaan nykyään yhä monimutkaisempia sovelluksia, joista osa suorittaa paljon laskentaa. WebAssembly pyrkiikin vastaamaan tähän tarpeeseen. WebAssembly ei ollut ensimmäinen teknologia, joka on yrittänyt suorituskykyisen koodin suorittamisen mahdollistamista internetselaimessa. Kuitenkaan yksikään WebAssemblyn edeltäjistä ei onnistunut täyttämään kaikkia WebAssemblyn päätavoitteita. WebAssemblyn MVP-versio tarjoaa huomattavasti JavaScriptia paremman suorituskyvyn, mutta ei kuitenkaan konekieleen käännetyn C-koodin tasoista suorituskykyä.

WebAssemblyn turvallisuus puolestaan paljasti ongelmia sillä yli puolet internetsivustoista, jotka käyttävät WebAssemblya, käyttävät sitä haitallisiin tarkoituksiin, kuten kryptovaluutan louhintaan. Haavoittuvuuksia WebAssemblyssa on kuitenkin ollut suhteellisen vähän ja sen suorittaminen hiekkalaatikon sisällä mahdollistaa epäluotettavan koodin turvallisen suorittamisen. WebAssemblyn siirrettävyys ja tiiviys ovat hyvällä tasolla tutkitun kirjallisuuden perusteella. WebAssemblyn suorittamiseen internetselaimen ulkopuolella on olemassa useita itsenäisiä runtime-ympäristöjä ja WebAssembly-koodin koko on vertailuissa ollut vastaavan asm.js-koodin ja konekieleen käännetyn C-koodin kokoja pienempiä.

Avainsanat: WebAssembly, ohjelmointikielet, web-ohjelmointi, JavaScript, asm.js

Tämän julkaisun alkuperäisyys on tarkastettu Turnitin OriginalityCheck –ohjelmalla.

SISÄLLYSLUETTELO

1. JOHDANTO....1
2. INTERNET ENNEN WEBASSEMBLYA....2
2.1 Internet ja JavaScript....2
2.2 WebAssemblyn edeltäjät....3
3. WEBASSEMBLY OHJELMOINTIKIELENÄ....7
3.1 Yleistä....7
3.2 Käsitteet....9
3.3 Semanttiset vaiheet....12
3.4 Kääntäminen lyhyesti....13
4. KATSAUS WEBASSEMBLYN NYKYTILANTEESEEN....15
5. JOHTOPÄÄTÖKSET....22
6. YHTEENVETO....24
LÄHTEET....25

1. JOHDANTO

Internetin alkuaikoina internetsivut olivat yksinkertaisia ja saattoivat koostua pelkästään HTML:stä. Vuonna 1995 esiteltiin JavaScript, jonka tarkoituksena oli mahdollistaa interaktio HTML:n Document Object Modelin kanssa, mutta oletuksena oli se, että JavaScriptia käytettäisiin vain lyhyiden skriptien kirjoittamiseen [27, 44]. Internetin suosion kasvun myötä internetsivuista on tullut yhä monimutkaisempia ja suurempia [34]. Myöskin JavaScriptin määrä on lisääntynyt merkittävästi [34], mistä johtuen internetsivujen suorittamisen vastuu on siirtynyt etenevässä määrin palvelimelta asiakkaan eli internetselaimen vastuulle. Erityisesti suosiotaan kasvattaneet yhden sivun websovellukset (engl. single-page application) nojaavat täysin internetselaimessa suoritettavaan JavaScriptiin [5,29,55]. Websovelluksina tarjotaankin nykyään yhä monimutkaisempia ja kokonaisvaltaisempia sovelluksia, joista osa vaatii huomattavan määrän suorituskykyä, mutta JavaScript ei kuitenkaan ole suorituskykykeskeinen ohjelmointikieli.

WebAssembly on uusi matalan tason tavukoodi, jota voidaan myös tarkastella assembly-kielen kaltaisena ohjelmointikielenä ja jonka suorittamista tukevat useat internetselaimet. WebAssemblyn yleinen käyttöperiaate on se, että korkean tason ohjelmointikielillä, kuten C/C++ -ohjelmointikielet, kirjoitetut ohjelmat voidaan kääntää suoraan WebAssemblyksi. WebAssemblyn tarkoituksena on mahdollistaa JavaScriptia huomattavasti paremman suorituskyvyn saavuttaminen. Toisin sanoen WebAssembly mahdollistaa muun muassa C/C++ -ohjelmien suorittamisen tehokkaasti suoraan internetselaimessa. Työn päämääränä on perehtyä kattavasti siihen, mikä ja mitä on WebAssembly, erityisesti ohjelmointikieli-näkökulmasta. Samalla selvitetään WebAssemblyn edeltäjiä hyödyntämällä syyt siihen, miksi se on kehitetty. Lopuksi pyritään selvittämään, mikä on WebAssemblyn pienin toimiva tuote (engl. Minimum Viable Product, MVP) -version nykytilanne. Vastaukset näihin tutkimuskysymyksiin selvitetään olemassaolevan kirjallisuuden avulla.

Luvussa 2 tutustutaan internetin ja JavaScriptin suhteeseen sekä WebAssemblyn edeltäjiin. Luvussa 3 tutustutaan WebAssemblyn rakenteeseen ja toimintaan. Luvussa 4 tehdään lyhyt katsaus WebAssemblyn nykytilanteeseen WebAssemblyn tavoitteiden kautta. Tämän jälkeen luvussa 5 esitetään johtopäätöksiä WebAssemblysta esitetyn tiedon pohjalta ja lopulta luvussa 6 on yhteenveto työstä.

2. INTERNET ENNEN WEBASSEMBLYA

2.1 Internet ja JavaScript

JavaScript oli pitkään ainoa natiivisti tuettu ohjelmointikieli internetselaimissa, ja sen suosio on noussut web-ohjelmoinnin lisääntymisen myötä [11,26]. Myöskin Software as a Service -mallin eli SaaS-mallin suosion nousun myötä yhä enemmän sovelluksia jaetaan käyttäjille suoraan internetselaimen välityksellä [25]. Näiden internetselaimen välityksellä jaettavien websovelluksien määrän ja monimuotoisuuden kasvu lisää websovellusten toiminnallisia vaatimuksia, mikä puolestaan vahvistaa JavaScriptin asemaa entisestään.

JavaScript on korkean tason tulkattava tai ajonaikana käännettävä ohjelmointikieli, joka on oliopohjainen ja dynaamisesti tyypitetty. JavaScript-koodia suoritetaan JavaScript-moottorilla, joka suorittaa myös mahdollisen ajonaikaisen käännöksen tekemisen. JavaScriptin yleisin käyttökohde websovelluksissa on käyttäjän synnyttämään tapahtumaan, esimerkiksi klikkaukseen, reagoiminen. Klikkauksen seurauksena JavaScriptilla voidaan esimerkiksi muuttaa HTML-dokumentin sisältöä tai rakennetta muokkaamalla HTML-dokumentin Document Object Modelia eli DOM:ia. [39] Nykyään JavaScriptia käytetään myös paljon palvelimissa Node.js:n käytön yleistymisen takia [60].

JavaScriptia käytettiin vuoteen 2008 asti pelkästään tulkattavana ohjelmointikielenä [15]. Vaikka tulkattavuus yleisesti tuokin etuja esimerkiksi laitteistoriippumattomuuden muodossa, vähentää se ohjelman suorituskykyä selvästi verrattuna konekieleen käännetyn ohjelman suorituskykyyn [6,52]. Google julkaisi vuonna 2004 Google Maps -websovelluksen, joka vaati huomattavan määrän suorituskykyä ja toimikin sen aikaisilla internetselaimilla hyvin hitaasti [59]. Googlen liiketoimintasuunnitelman olennaisin osa olivat Googlen websovellukset, joista pyrittiin tekemään parempia. Tämä kuitenkin vaati myös sen, että internetselaimista piti tehdä parempia, minkä takia vuonna 2008 Google esitteli Chrome-internetselaimen ja sitä varten kehitetyn JavaScript-moottori V8:n. [2]

V8 oli yksi ensimmäisistä JavaScript-moottoreista, joka hyödynsi ajonaikaista kääntämistä. Se suoritti ajonaikaisen kääntämisen kääntämällä suoritettavan JavaScript-koodin ennen suorittamista konekieleen. Mozillan Firefoxia varten toteutettu ensimmäinen ajonaikainen kääntäjä TraceMonkey puolestaan seurasi ohjelman suoritusta ja käänsi ainoastaan useasti suoritettavan koodin konekielelle. [8,18,38] Ajonaikaisella kääntämi-

sellä saavutettiin merkittäviä parannuksia JavaScriptin suorituskyvyssä, joka joissain testeissä yli 20-kertaistui. [36,38,51,54]

Nykyään JavaScript-moottorit sisältävät yleensä kaksi ajonaikaista kääntäjää: yhden lähtötasokääntäjän (engl. baseline compiler), joka kääntää nopeasti JavaScriptin konekieleksi, ja yhden optimoivan kääntäjän, joka tekee optimoidun konekielisen käännöksen useasti suoritettavasta koodista. Esimerkiksi Mozillan SpiderMonkey-JavaScript-moottori ja Googlen V8-JavaScript-moottori toimivat näin. [23,41] JavaScript-moottoreiden suorituskyky onkin parantunut vuosien mittaan, mutta suorituskyvyn parantuminen alkoi hidastumaan vuosien 2013–2015 aikana [19,49]. JavaScript-moottoreiden suorituskyvyn parantumisesta huolimatta on JavaScriptin suorittaminen edelleen huomattavasti käännettyä konekieltä hitaampaa [6,52].

JavaScriptin yksi suurimmista vahvuuksista on se, että samaa JavaScript-koodia voidaan suorittaa suurella määrällä erilaisia laitteita. Kuitenkin JavaScriptia suoritetaan websovellusten tapauksessa lähes aina internetselaimella, joita on useita erilaisia ja jokaisella niistä on oma toteutus JavaScript-moottorista. Esimerkiksi Mozilla käyttää SpiderMonkey-JavaScript-moottoria Firefox-internetselaimessaan, ja Google käyttää V8-JavaScript-moottoria Chrome-internetselaimessaan. Erilaisten JavaScript-moottoreiden käyttö on johtanut siihen, että saman JavaScript-koodin suorituskyky voi vaihdella merkittävästi käytettävästä selaimesta riippuen. Varsinkin JavaScript-koodin optimointi tuottaa erilaisia tuloksia eri JavaScript-moottoreilla. Pahimmassa tapauksessa toista JavaScript-moottoria varten tehty optimointi heikentää suorituskykyä toisella JavaScript-moottorilla. [7,56]

Internetin vallankumouksen myötä on syntynyt tarve saada muilla ohjelmointikielillä kirjoitetut ohjelmat jaettavaksi websovelluksina. Perusteluita tälle voi olla esimerkiksi se, että jokin ohjelma on jo kirjoitettu jollain toisella kielellä tai ohjelmoija haluaa ohjelmoida toisella ohjelmointikielellä ja on tuottavampi sillä. [72] Tämän seurauksena JavaScriptista on tullut käännöskohde muille ohjelmointikielille, vaikka JavaScriptiksi kääntäminen sisältää useita ongelmia. Tämän lisäksi JavaScriptin heikko suorituskyky ja suoritusnopeuden vaihtelevuus tekevät siitä epäoptimaalisen ohjelmointikielen suorituskykynäkökulmasta. [7,26]

2.2 WebAssemblyn edeltäjät

WebAssembly ei ole ensimmäinen teknologia, joka pyrkii mahdollistamaan suorituskykyisen matalan tason koodin suorittamisen suoraan internetselaimessa. Aikaisempia teknologioita ovat olleet muun muassa Microsoftin ActiveX, Googlen Native Client ja

Mozillan asm.js. Internetissä aikaisemmin yleisiä olleet Java ja Flash eivät puolestaan koskaan ole pyrkineet matalan tason koodin tehokkaaseen suorittamiseen internetselaimessa. [26]

Microsoftin ActiveX -teknologia mahdollistaa x86-binäärien suorittamisen suoraan Microsoftin Internet Explorer -internetselaimessa [26]. Tämä johtaa luonnollisesti siihen, että ActiveX -teknologiaa hyödyntävät websovellukset eivät toimi kaikilla laitteilla. Tietoturvasta muodostui kuitenkin yksi ActiveX:n merkittävimmistä ongelmista. ActiveX-sovelluksilla oli pääsy Windows-käyttöjärjestelmään samalla tavalla kuin natiiveillakin sovelluksilla. Tietoturva perustuikin täysin luottamukseen digitaalisten allekirjoituksien ja käyttäjän suostumuksen kautta, minkä takia ActiveX:stä aiheutui jatkuvasti tietoturvaongelmia. [1,20,26,64,71] Tietoturvaongelmien lisäksi HTML5-teknologian julkaisun myötä Microsoft koki, että ActiveX:lle ei ole enää tarvetta, minkä takia Microsoftin nykyinen internetselain Microsoft Edge ei tue ollenkaan ActiveX-teknologiaa [42].

Googlen Native Client -teknologia eli NaCl-teknologia mahdollistaa epäluotettavien x86-binäärien suorittamisen Googlen Chrome-internetselaimessa hyödyntäen ohjelmallista vian eristämistä (engl. software fault isolation tai software-based fault isolation). NaCl-teknologian toteutus perustuu kahteen hiekkalaatikkoon (engl. sandbox). Sisempi hiekkalaatikko on vastuussa epäluotettavien x86-binäärien validoinnista, joka suoritetaan staattisella analyysilla. Ulompi hiekkalaatikko puolestaan on vastuussa järjestelmäkutsujen välittämisestä. NaCl-sovellusten suorituskyky on hyvin lähellä natiiveja sovelluksia. [26,71]

Vaikka NaCl-sovellukset ovatkin siirrettäviä käyttöjärjestelmien välillä, ne eivät kuitenkaan toimi kaikilla laitteilla suoritinarkkitehtuurin takia [64]. Tämän lisäksi ainoa keino jakaa NaCl-sovelluksia on Chrome-internetselaimen sovelluskaupan kautta, mikä edelleen johtaa siihen, että NaCl-sovellukset eivät ole erityisen hyvin siirrettävissä [46]. Chrome-internetselaimen vaatimat rajoitteet aiheuttivat myös sen, että NaCl-sovellukset eivät pysty synkronisesti kutsumaan JavaScript- tai web-rajapintoja [26].

Erityisesti eri suoritinarkkitehtuurien takia Google kehitti Portable Native Client -teknologian eli PNaCl-teknologian, joka lisää NaCl -teknologian ominaisuuksiin suoritinarkkitehtuuririippumattomuuden. Tämä tarkoittaa kuitenkin sitä, että PNaCl -sovelluksia ei voida jakaa suoraan konekielenä, minkä takia PNaCl perustuu LLVM-välikieleen (engl. Low-Level Virtual Machine intermediate representation), joka on suoritinarkkitehtuurista riippumaton. PNaCl-sovellukset jaetaankin LLVM-välikielisinä käyttäjille, jotka ovat vastuussa sovelluksen kääntämisestä NaCl-sovellukseksi ennen ohjelman suorittamista. [21,46]

Suoritinarkkitehtuuririippumattomuuden lisäksi PNaCl parantaa siirrettävyyttä pelkkään NaCl-teknologiaan verrattuna myös siten, että PNaCl -sovelluksia voidaan jakaa käyttäjille suoraan verkkosivun kautta eli Chromen sovelluskauppaa ei ole pakko käyttää sovelluksien jakamisessa [46]. PNaCl ei kuitenkaan parantanut sovelluksien tiiviyttä verrattuna NaCl-sovelluksiin ja suoritinarkkitehtuuririippumattomuudesta huolimatta se paljastaa yhä alustaan ja kääntäjään liittyvää informaatiota, kuten kutsupinon rakenteen. PNaCl sekä NaCl molemmat toimivat ainoastaan Chrome-internetselaimella, mikä rajoittaa niillä toteutettujen sovelluksien siirrettävyyttä huomattavasti. [26]

Vuonna 2017 Google ilmoitti luopuvansa PNaCl-teknologiasta Chrome-sovelluksia ja -laajennuksia lukuun ottamatta. Syy tähän oli se, että PNaCl:n käyttöaste oli tarpeeksi matalalla, ja WebAssemblya tuettiin jo Chromessa ja Firefoxissa. Erityisesti internetselainten välisen natiivin tuen luoma parempi ekosysteemi WebAssemblylle vaikutti Googlen päätökseen. [47]

Mozillan kehittämä asm.js on JavaScriptin täsmällinen osajoukko, joka pyrkii tarjoamaan suorituskykyisen matalan tason käännöskohteen muilla ohjelmointikielillä tehtyjä ohjelmia varten. Toisin sanoen asm.js on JavaScriptin alakieli. Suurin ero tavalliseen JavaScriptiin on se, että asm.js on staattisesti tyypitetty verrattuna tavallisen JavaScriptin dynaamiseen tyypitykseen. Koska asm.js on tarkoitettu muiden ohjelmointikielien käännöskohteeksi, tarjoaa se abstraktion, joka on rakennettu saman tapaisesti kuin C/C++ -virtuaalikone. [30]

Yksi suurimmista asm.js:sän vahvuuksista on se, että se on JavaScriptin osajoukko, minkä takia se toimii suoraan kaikilla selaimilla. Kuitenkaan kaikki JavaScript-koodi ei ole validia asm.js-koodia, minkä takia asm.js-koodille suoritetaan validointi ennen sen valmistamista suoritusta varten. Jos validointi epäonnistuu, suoritetaan kyseinen koodi normaalina JavaScriptina eli tulkitsemalla tai ajonaikaisesti kääntämällä. Validoinnin onnistuminen kuitenkin mahdollistaa asm.js-koodin ennenaikaisen kääntämisen (engl. ahead-of-time compilation, AOT) tehokkaaksi konekieleksi, koska ennenaikainen kääntäjä pystyy jättämään pois esimerkiksi ajonaikaiset tyypin tarkistukset sekä automaattisen roskienkeräyksen. Ennenaikaisen käännöksen suorituskyky on myös hyvin ennustettavissa. [30] Ennenaikainen kääntäminen kuitenkin vaatii sen, että käytössä oleva JavaScript-moottori tukee nimenomaan asm.js-koodin ennenaikaista kääntämistä [26]. Vuoden 2019 alkupuolella ainoastaan Edge- ja Firefox-internetselaimet tukevat erikseen asm.js:sän ennenaikaista kääntämistä [12,43,62].

Vaikka asm.js:sän yksi suurimmista vahvuuksista on se, että se on JavaScriptin osajoukko, on se samalla myös yksi sen suurimmista heikkouksista. JavaScriptiin sidonnaisuudesta seuraa se, että jos asm.js:sään halutaan lisätä uusia ominaisuuksia, täytyy

ne ensin lisätä JavaScriptiin ja sitten hyväksyä osaksi asm.js:sää. Esimerkiksi 64-bittisten kokonaislukujen lisääminen jouduttaisiin suorittamaan näin, ja hyvän suorituskyvyn saavuttaminen olisi haastavaa. [26] Asm.js on kuitenkin pohjimmiltaan JavaScriptia, joten sen jäsentämiseen (engl. parsing) kuluu edelleen huomattavasti aikaa ja suorituskyky ei muutenkaan ole samaa tasoa kuin natiivien ohjelmien suorituskyky [74,76].

WebAssemblyn tavoitteena on mahdollistaa turvallisen, nopean, siirrettävän ja tiiviin matalan tason koodin hyödyntäminen internetissä [26]. Yksikään WebAssemblyn edeltäjistä ei ole onnistunut täyttämään kaikkia näistä vaatimuksista. ActiveX-teknologia ei esimerkiksi täyttänyt turvallisen ja siirrettävyyden vaatimuksia, kun taas NaCl- ja PNaCl-teknologiat eivät myöskään olleet siirrettäviä tai erityisen tiiviitä. Asm.js:sän sidokset JavaScriptiin puolestaan heikentävät sen laajennusmahdollisuuksia, ja asm.js:sän saama heikko internetselaintuki vähentää teknologian suorituskykyä internetselaimesta riippuen. WebAssemblyn kehityksessä on ollut mukana kaikkien WebAssemblyn edeltäjien kehittäjien edustajia, mistä on ollut hyötyä WebAssemblyn kehityksessä [26,74].

3. WEBASSEMBLY OHJELMOINTIKIELENÄ

3.1 Yleistä

WebAssembly voidaan määritellä useilla eri tavoilla. Se voidaan määritellä binääriseksi suoritusformaatiksi (engl. binary execution format) tai binääriseksi käskyformaatiksi (engl. binary instruction format), mutta se voidaan myöskin määritellä matalan tason tavukoodiksi tai virtuaaliseksi käskykanta-arkkitehtuuriksi (engl. virtual instruction set architecture) [26,68,69,74]. Vaikka WebAssembly onkin binäärinen koodiformaatti, voidaan se esittää myös ohjelmointikielenä, jolla on assembly-kielen kaltainen syntaksi ja rakenne. Sen päätavoitteena on korkeaa suorituskykyä vaativien websovellusten mahdollistaminen. Sen tarkoituksena ei ole korvata JavaScriptia, vaan se tulee täydentämään JavaScriptia erityisesti suorituskykykeskeisissä käyttötapauksissa [37]. WebAssemblya ei ole tarkoitus kirjoittaa käsin, vaan se on matalan tason käännöskohde muille ohjelmointikielille, esimerkiksi C/C++ -ohjelmointikielille. [26] WebAssemblyn tavukoodin binäärimuotoa ja sitä vastaavaa tekstimuotoa on havainnollistettu ohjelmassa 2, joka on ohjelmassa 1 esitetyn C++-ohjelman WebAssembly-käännöksen tulos.

```
int factorial(int n) {
  if (n == 0)
    return 1;
  else
    return n * factorial(n-1);
}
```

Ohjelma 1*: C++-ohjelmointikielellä kirjoitettu rekursiivinen kertomafunktio [68].*

```
20 00                                get_local 0
42 00                                i64.const 0
51                                   i64.eq
04 7e                                if i64
42 01                                    i64.const 1
05                                   else
20 00                                    get_local 0
20 00                                    get_local 0
42 01                                    i64.const 1
7d                                       i64.sub
10 00                                    call 0
7e                                       i64.mul
0b                                   end
```

Ohjelma 2*: Ohjelma 1 esitettynä WebAssemblyn binäärimuodossa heksaluvuilla ja tekstimuodossa [68].*

WebAssembly on avoin standardi, jota kehittää World Wide Web Consortiumin eli W3C:n yhteisöryhmä, joka pitää sisällään muun muassa Microsoftin, Mozillan, Googlen ja Applen edustajia [68]. WebAssembly toimii tällä hetkellä natiivisti muun muassa Microsoftin Edge, Mozillan Firefox, Googlen Chrome, Applen Safari ja Opera -internetselaimilla [47,53,68]. Vaikka yksi WebAssemblyn päätavoitteista on suorituskykyisten websovellusten mahdollistaminen, niin WebAssembly itsessään ei sisällä mitään ominaisuuksia, jotka rajaisivat sen käyttömahdollisuudet ainoastaan internetin ja internetselaimien piiriin. WebAssemblyn kehittäjät odottavatkin, että WebAssemblylle löytyy useita käyttötarkoituksia internetin ulkopuolella. [26,69] Niitä voisivat olla esimerkiksi epäluotettavan koodin suorittaminen palvelimella ja ylipäätänsä palvelimella suoritettavat ohjelmat. Kuitenkin internetsivustojen tukeminen on tärkeää WebAssemblylle, minkä takia WebAssembly pyrkii ylläpitämään internetin kaltaista versiottomuutta ja taaksepäin yhteensopivuutta. [68]

WebAssembly pyrkii toteuttamaan nopean, turvallisen ja siirrettävän semantiikan. Sen suorituskyvyn tavoitteena on lähes natiivin suorituskyvyn saavuttaminen. [69] Turvallisuus puolestaan on internetissä erityisen tärkeätä, koska WebAssembly-koodia vastaanotetaan usein epäluotettavista lähteistä. Epäluotettavan koodin turvallinen suorittaminen on perinteisesti toteutettu suoritusympäristön avulla erityisesti tulkattavien ohjelmointikielien tapauksessa. Tämä lähestymistapa on toiminut, koska tulkattavat kielet varmistavat muistiturvallisuuden estämällä koodin pääsyn sellaisiin muistiosoitteisiin, joihin koodin ei kuulu päästä. Tällaisia muistiosoitteita voivat olla esimerkiksi järjestelmän tilaan liittyvät muistiosoitteet. Kuitenkaan tällaista ominaisuutta ei löydy C/C++ -ohjelmointikielien kaltaisista käännettävistä ohjelmointikielistä, joilla kirjoitettuja ohjelmia on tarkoitus kääntää WebAssemblyksi. [26] WebAssembly ratkaisee turvallisuusongelman validoimalla suoritettavan koodin ennen suoritusta ja tekemällä itse koodin suorituksen muistiturvallisen hiekkalaatikon sisällä. Turvallisuutta parantaa se, että WebAssembly määrittelee tarkasti ja kokonaisvaltaisesti validin ohjelman ja sen käyttäytymisen. [69]

Siirrettävyyden WebAssembly mahdollistaa siten, että se on täysin suoritusympäristön laitteistosta riippumaton eli WebAssembly-koodia voidaan suorittaa perinteisillä tietokoneilla, mobiililaitteilla ja sulautetuilla järjestelmillä. Koska WebAssembly on tarkoitettu käännöskohteeksi muille ohjelmointikielille, se pyrkii myös ohjelmointikielestä riippumattomuuteen: se ei suosi esimerkiksi tiettyjä ohjelmointikieliä tai ohjelmointimalleja (engl. programming model). Lisäksi WebAssemblyn on tarkoitus toimia internetselaimien ulkopuolella, minkä vuoksi se pyrkii olemaan alustariippumaton, mikä mahdollistaa WebAssemblyn suorittamisen esimerkiksi erillisessä virtuaalikoneessa tai sulautet-

tuna johonkin muuhun järjestelmään kuin internetselaimeen. Tästä johtuen WebAssembly pyrkii mahdollistamaan kommunikoinnin suoritusympäristönsä kanssa universaalilla ja yksinkertaisella tavalla. [69]

Selkeän semantiikan lisäksi WebAssembly pyrkii tehokkaaseen ja siirrettävään esitysmuotoon. Erityisen tärkeätä internetkontekstissa siirrettävyyden kannalta on ohjelman koko ja erityisesti koodin tiiviys, jotta siirto internetin yli tapahtuisi mahdollisimman nopeasti. Tämän takia WebAssemblyn esitysmuoto on binäärinen, koska se mahdollistaa tavallista tekstiä tai koodia pienemmän koon saavuttamisen. Siirrettävyyden parantamiseksi WebAssembly-ohjelmat ovat myös modulaarisia eli ne voidaan jakaa pienempiin osiin, joita voidaan käsitellä yksitellen esimerkiksi siirron, suorittamisen ja välimuistiin tallentamisen yhteydessä. WebAssemblyn semantiikan tapaan WebAssemblyn esitysmuoto ei myöskään tee mitään oletuksia suoritusympäristön laitteistosta. [69]

WebAssemblyn suorituskykyteema näkyy myös sen esitysmuodossa, joka on suunniteltu mahdollisimman suorituskykyiseksi. Esimerkiksi WebAssembly-koodin purkaminen (engl. decoding), validointi ja kääntäminen pystytään suorittamaan yhdellä läpäisyllä. Kääntäminen onnistuu sekä ajonaikaista että ennenaikaista käännösstrategiaa käyttäen. Suorituskykyä tehostaa myös se, että WebAssemblyn esitysmuoto mahdollistaa näiden tehtävien suorittamisen ennen kuin kaikki koodi on saatavilla. Tämän lisäksi nämä tehtävät pystytään rinnakkaistamaan itsenäisiksi tehtäviksi. [69]

3.2 Käsitteet

WebAssembly voidaan myös määritellä ohjelmointikieleksi, joka pitää sisällään useita käsitteitä, joiden ympärille WebAssembly rakentuu. WebAssemblyn käsitteitä ovat arvot (engl. values), käskyt (engl. instructions), ansat (engl. traps), funktiot, taulut (engl. tables), lineaarinen muisti (engl. linear memory), moduulit ja sulauttaja (engl. embedder). [69]

WebAssembly sisältää ainoastaan neljä eri arvotyyppiä. Näitä ovat kokonaisluvut ja liukuluvut sekä 32- että 64-bittisinä. Yleisesti käytössä olevan laitteiston tapaan arvon tyyppi ei ota kantaa siihen, että onko kyseessä etumerkillinen vai etumerkitön arvo, vaan käytettävät käskyt määrittelevät arvojen tulkinnan. 32-bittiset kokonaisluvut toimivat myös muistiosoitteina, totuusarvoina ja funktiotaulukkojen indekseinä. WebAssembly tarjoaa muunnosoperaatiot kaikkien tyyppien välille sekä näiden tyyppien tavalliset operaatiot. Funktiot tallentavat WebAssemblyssa arvoja paikallisiin muuttujiin ja moduulit globaaleihin muuttujiin. [26,69]

WebAssemblyn laskentamalli perustuu pinokoneeseen. Tämä tarkoittaa sitä, että WebAssembly-koodi koostuu käskyjen sarjasta, joka käsittelee arvoja, jotka ovat implisiittisessä operandipinossa (engl. implicit operand stack). Arvoja voidaan siis poimia pinosta ja tuloksia lisätä pinoon. Operandipinon asetelma voidaan kuitenkin staattisesti määritellä missä tahansa kohtaa koodia tyyppijärjestelmän ansiosta, minkä takia WebAssemblyn toteutukset pystyvät käytännössä hoitamaan käskyjen välisen datavirran suoraan ilman operandipinon luomista. [26] Pinokoneeseen pohjautuvan laskentamallin valinta tehtiin sen takia, että saataisiin tiiviimpiä ohjelmia, sillä pinokoneella on mahdollista saavuttaa tiiviimpi ohjelman esitysmuoto kuin rekisterikoneella [26,57]. Tämä parantaa edelleen WebAssembly-koodin siirrettävyyttä. WebAssemblyn kehittäjät kokeilivat myös pakattuja tavukoodattuja abstrakteja syntaksipuita (engl. abstract syntax tree, AST) sekä esi- että jälkijärjestyksessä, mutta havaitsivat, että pinokone mahdollistaa paremman tiiviyden. [26]

WebAssemblyssa on kahdenlaisia käskyjä: yksinkertaisia käskyjä ja hallintakäskyjä. Edellä kuvatut käskyt ovat yksinkertaisia käskyjä, jotka siis suorittavat yksinkertaisia operaatioita operandipinon arvoille. [69] Hallintakäskyt puolestaan vaikuttavat ohjelman suoritusjärjestykseen, joka on WebAssemblyssa strukturoitu, mikä on epätavallista ohjelmointikielille [67,69]. Tämä tarkoittaa sitä, että ohjelman suoritusjärjestystä hallitaan lohkojen, silmukoiden ja ehtojen avulla [69]. Toisin sanoen perinteisiä branch ja goto -operaatioita ei ole toteutettu ollenkaan [67]. Strukturoidulla suoritusjärjestyksellä varmistetaan se, että pelkistymättömiä silmukoita (engl. irreducible loops) ei voi muodostua ja koodin suoritus ei voi siirtyä lohkoihin, joissa on yhteensopimaton pinon korkeus, eikä myöskään keskelle monitavuista käskyä. Juurikin tämä mahdollistaa WebAssembly-koodin validoinnin ja kääntämisen yhdellä läpäisyllä. Strukturoitu suoritusjärjestys tuo myös hyötyjä erityisesti internetkontekstissa, sillä internetissä käyttäjät ovat tottuneet internetsivujen lähdekoodin tutkimiseen, ja strukturoidun suoritusjärjestyksen omaavan koodin lukeminen on ihmiselle helpompaa. [26]

Joidenkin käskyjen tuloksena voi syntyä ansa. Tällaisia käskyjä ovat esimerkiksi muistialueen ulkopuolelle pääsyn yrittäminen ja tyyppien yhteensopimattomuuden aiheuttaminen. Ansa keskeyttää ohjelman suorittamisen välittömästi ja sitä ei pystytä käsittelemään WebAssemblyn sisällä. Ansa voidaan kuitenkin siepata sulauttavassa järjestelmässä, jonka jälkeen siihen voidaan reagoida asiaan kuuluvalla tavalla. Esimerkiksi internetkontekstissa JavaScript-koodissa pystytään sieppaamaan WebAssemblyn synnyttämä ansa. JavaScriptissa ansa näkyy tavallisena poikkeuksena, joka pitää sisällään sekä WebAssemblyn että JavaScriptin pinokehykset. Ansojen käsittely jää siis sulauttavan ympäristön vastuulle. [26, 69]

WebAssemblyssa koodi sijoitetaan erillisiin funktioihin, joista jokainen ottaa parametreinaan sarjan arvoja ja myös palauttaa tuloksenaan sarjan funktion tyypin mukaisia arvoja. Funktiot pystyvät kutsumaan toisiansa ja myös rekursiivinen kutsuminen on mahdollista, mikä muodostaa implisiittisen kutsupinon, jota ei pysty suoraan käsittelemään. [69]

WebAssemblyssa taulu on taulukko, joka pitää sisällään tietyn elementin tyyppisiä läpinäkymättömiä arvoja. Taulun arvoihin ei siis suoraan pääse käsiksi, vaan niitä voi käsitellä epäsuorasti dynaamisen indeksioperandin avulla. WebAssemblyn julkaisussa 1.0 ainoa sallittu elementin tyyppi on tyypittämätön funktioviite. Tämä mahdollistaa funktio-osoittimien toiminnan matkimisen taulujen avulla dynaamista indeksioperandia hyödyntämällä. [69]

Lineaarinen muisti on WebAssembly-ohjelman pääasiallinen tallennustila. Se on suuri, jatkuva ja muokattavissa oleva raakoja tavuja sisältävä taulukko. Jokainen moduuli voi määrittää enintään yhden lineaarisen muistin, joka voidaan jakaa joko tuomalla (engl. import) tai viemällä (engl. export) muiden ilmentymien kanssa. Lineaarisella muistilla on alustava koko, mutta sitä voidaan dynaamisesti kasvattaa tarpeen niin vaatiessa. Lineaarista muistia käsiteltäessä muistin yksikkönä käytetään sivua (engl. page), jonka kooksi on määritelty 64 kibitavua (KiB). Koon valinta perustuu siihen, että 64 KiB on modernien laitteistojen sivujen minimikokojen pienin yhteinen jaettava. Sivun koko on pysyvä eikä järjestelmäkohtainen, jotta vältytään siirrettävyysongelmilta. Nykyaikainen laitteisto vaikuttaisi myös suosivan little endian -tavujärjestystä, minkä takia WebAssemblyn lineaarinen muisti käyttää sitä. Alustat, jotka käyttävät big endian -tavujärjestystä, joutuvat käyttämään eksplisiittisiä muunnoksia tavujärjestyksen takia, mutta niiden tekemisessä voidaan hyödyntää kääntäjän tekemää optimointia. [26,69]

WebAssembly-ohjelma pystyy käsittelemään mitä tahansa kohtaa lineaarisesta muistista luku- ja kirjoitusoperaatioilla. Kaikki luku- ja kirjoitusoperaatiot tarkistetaan dynaamisesti, sillä yritys päästä kielletylle muistialueelle johtaa ansan synnyttämiseen. Koska suoritettava WebAssembly-ohjelma, erityisesti internetissä, on yleensä peräisin epäluotettavasta lähteestä, on lineaarisen muistin turvallisuus tärkeä tekijä. Lineaarinen muisti onkin täysin erillään varsinaisesta ohjelmakoodista, kutsupinosta ja sulauttavan ympäristön tietorakenteista. Tämä johtaa siihen, että WebAssembly-ohjelma ei pysty korruptoimaan muuta kuin oman muistinsa eli lineaarisen muistin ja näin ollen se pystytään suorittamaan turvallisesti samassa osoiteavaruudessa muun koodin kanssa. Tällainen prosessin sisäinen eristäminen oli pakollinen vaatimus WebAssemblylle, jotta WebAssembly-ohjelmat pystyvät kommunikoimaan epäluotettavan JavaScriptin ja muiden web-rajapintojen kanssa tehokkaasti. Tästä myös seuraa se, että WebAssembly-ohjel-

masta voidaan luoda useita itsenäisiä ilmentymiä, joilla on oma muistinsa ja jotka elävät saman prosessin sisällä. [26]

WebAssembly-binäärit esitetään moduuleina, jotka sisältävät määritelmät funktioille, globaaleille muuttujille, tauluille ja lineaarisille muisteille. Näitä määritelmiä pystytään sekä tuomaan että viemään, ja tuotuja määritelmiä pystytään edelleen viemään eteenpäin. Tuominen vaatii moduuli-nimi -parin ja sopivan tyypin määrittämisen, kun vieminen puolestaan vaatii yhden tai useamman nimen määrittelemistä vietävälle määritelmälle. Lisäksi tuomisen avulla on mahdollista kommunikoida suoritusympäristön kanssa. Tällä tavalla esimerkiksi internetselaimessa suoritettavat WebAssembly-moduulit voivat kutsua JavaScript-funktioita. Moduulit pystyvät myös määrittelemään alustusdataa lineaarista muistia tai tauluja varten sekä automaattisesti suoritettavan aloitusfunktion (engl. start function). [26,69]

Sulauttaja on WebAssemblyn varsinainen suoritusympäristö, johon WebAssemblyn toteutus sulautetaan. Sulauttaja määrittelee esimerkiksi kuinka moduulit käynnistetään ja kuinka tuonnit sekä viennit käsitellään. [69] Muita sulauttajan vastuita ovat muun muassa ansojen käsittely ja siirräntäoperaatiot (engl. input/output, I/O) [26]. Internetsivustojen tapauksessa WebAssemblyn sulauttajana toimii yleensä internetselaimen JavaScript-moottori [16,26].

3.3 Semanttiset vaiheet

WebAssemblyn suorittaminen voidaan jakaa kolmeen vaiheeseen. Ensimmäinen vaihe on WebAssemblyn binäärimuodon purkaminen. Toinen vaihe on ohjelman validointi ja kolmas vaihe on ohjelman suorittaminen, joka voidaan edelleen jakaa kahteen erilliseen vaiheeseen: ilmentymän tuottaminen (engl. instantiation) ja kutsuminen (engl. invocation). [69]

WebAssembly-ohjelmia jaetaan binäärimuodossa, minkä takia niitä ei voida suorittaa ennen binäärikoodin purkamista. WebAssemblyn binäärimuotoiset moduulit voidaan purkaa suoritusympäristöstä riippuvaan sisäiseen esitystapaan, mutta ne voidaan myös kääntää suoraan konekieleksi. [69] Esimerkiksi V8-JavaScript-moottorin nopea WebAssemblyn lähtötasokääntäjä Liftoff kääntää WebAssembly-binäärit suoraan konekieleen yhdellä läpäisyllä [28].

Purettu WebAssembly-moduuli täytyy validoida ennen sen suorittamista. Validoinnissa tarkistetaan moduulin turvallisuus ja oikeellisuus käyttäen validointisääntöjä, jotka perustuvat yksinkertaiseen tyyppijärjestelmään. Tyyppijärjestelmä on suunniteltu siten, että validointi onnistuu yhdellä läpäisyllä ja samanaikaisesti WebAssembly-moduulin

purkamisen ja kääntämisen kanssa. [26,69] Esimerkiksi V8:n Liftoff-kääntäjä toimii näin [28]. WebAssemblyn validoinnin käyttämä tyyppijärjestelmä on myös todistettu eheäksi (engl. soundness), mikä tarkoittaa sitä, että WebAssemblyn suoritussemantiikka on määritelty täysin eli määrittelemätöntä käyttäytymistä ei voi tapahtua [26,67]. Tämä implikoi sen, että esimerkiksi muistiturvallisuus ja kutsupinon saavuttamattomuus ovat varmistettu [26].

WebAssembly-moduulin suorittaminen alkaa ilmentymän tuottamisella eli staattisesta moduulista tuotetaan sen dynaaminen esitys eli ilmentymä. Ilmentymällä on oma tila ja kutsupino. Sille alustetaan globaalit muuttujat, lineaarinen muisti ja taulut. Moduuliin tuodut funktiot saavat myös määritelmät. Ilmentymän tuottamisen lopputuloksena on moduulista vietyjen määritelmien ilmentymät. Lisäksi ilmentymän tuottamisen yhteydessä suoritetaan aloitusfunktio, jos sellainen on määritetty. [69]

WebAssemblyn suorittaminen onnistuu kutsumalla ilmentymässä määriteltyä funktiota. Funktion kutsuminen onnistuu normaalisti antamalla oikeat parametrit, minkä seurauksena funktio suoritetaan, ja se palauttaa tuloksensa. Kuitenkin sekä ilmentymän tuottaminen että sen kutsuminen ovat sulauttajan määrittelemiä operaatioita, mistä johtuen niiden toteutusyksityiskohdat voivat vaihdella sulauttajasta riippuen. [69]

3.4 Kääntäminen lyhyesti

WebAssembly on matalan tason käännöskohde muille korkeamman tason ohjelmointikielille. Toisin sanoen WebAssemblya ei ole tarkoitus koodata suoraan, vaan toisella ohjelmointikielellä tehty ohjelma on tarkoitus kääntää WebAssemblyksi.

WebAssemblyyn käännettävissä olevia ohjelmointikieliä on paljon enemmän kuin pelkästään C/C++ -ohjelmointikielet. Näiden joukkoon kuuluvat muun muassa Rust ja Java. [4] WebAssemblyyn kääntämisen toteutus riippuu käytettävästä ohjelmointikielestä. Esimerkiksi C/C++ -ohjelmien WebAssemblyksi kääntämiseen käytetään Emscriptenia [68]. Emscripten on kääntäjä, joka pystyy kääntämään LLVM-bittikoodin JavaScriptiksi [72]. Emscripteniä käytettiinkin WebAssemblyn edeltäjän asm.js:sän tuottamiseen [26]. Asm.js liittyy myös olennaisesti WebAssemblyn käännösprosessiin, sillä Emscripten kääntää C/C++ -koodin ensin Clang-kääntäjän avulla LLVM-bittikoodiksi, sitten LLVM-bittikoodin asm.js:sään ja lopuksi hyödyntäen Binaryen-käännöskirjastoa se kääntää syntyneen asm.js-koodin WebAssemblyksi [22,73]. Rust-koodi puolestaan voidaan kääntää WebAssemblyksi ilman Emscriptenin käyttämistä, mutta Emscripteniakin voidaan hyödyntää [75].

Emscriptenin tekemän käännöksen tuloksena syntyy .wasm-tiedosto ja .js-tiedosto [22]. Binäärimuotoisen WebAssemblyn suositeltu tiedostotyyppi on ”.wasm”, kun taas WebAssemblyn tekstimuodon suositeltu tiedostotyyppi on ”.wat” [69]. Emscriptenin tuottama .wasm-tiedosto pitääkin sisällään varsinaisen WebAssembly-koodin. Käännöksen tuloksena syntynyt ”.js”-tiedosto puolestaan on vastuussa ”.wasm”-tiedoston sisältämän WebAssemblyn alustamisesta. Se esimerkiksi hoitaa WebAssemblyssa määritetyt viennit ja tuonnit. Kyseinen ”.js”-tiedosto on erityisen hyödyllinen internetselain-ympäristöä varten, sillä se mahdollistaa esimerkiksi sen, että WebAssembly-ohjelma pystyy tulostamaan suoraan internetselaimen konsoliin. [22]

4. KATSAUS WEBASSEMBLYN NYKYTILANTEE-SEEN

Luvussa 3 perehdyttiin WebAssemblyyn ohjelmointikielenä ja sen tavoitteisiin, mutta tavoitteiden saavuttamiseen ei otettu kantaa. WebAssemblyn päätavoitteet voidaan tiivistää neljään eri kategoriaan: turvallisuus, suorituskyky, siirrettävyys ja tiiviys [26]. Tässä luvussa tehdään katsaus WebAssemblyn nykytilanteeseen näiden kategorioiden kautta hyödyntäen olemassaolevaa kirjallisuutta. Työn tarkoituksena ei ole perehtyä tarkasti yksittäisiin asioihin, vaan luoda kokonaiskuva siitä, miten WebAssembly onnistuu näissä kategorioissa.

Turvallisuus on WebAssemblyssa toteutettu validoinnin ja muistiturvallisen hiekkalaatikon avulla. Tämän lisäksi WebAssemblyn rakenne estää useiden haavoittuvuuksien periytymisen kehitykseen käytetystä ohjelmointikielestä WebAssemblyyn käännöksen kautta [24,40]. Kuitenkin joitain perinteisiä haavoittuvuuksia on mahdollista hyväksikäyttää myös WebAssemblyssa. Näitä ovat esimerkiksi kokonaislukujen ja puskureiden ylivuodot, jotka saattavat olla vaarallisia. WebAssemblya pystytään mahdollisesti käyttämään cross site scripting eli XSS -hyökkäyksien tekemiseen. Esimerkiksi puskurin ylivuotoa voidaan hyödyntää tällaisen hyökkäyksen toteuttamisessa. [40] WebAssembly-ohjelman suoritusjärjestyksen kaappaaminen on myös mahdollista tyyppihämmennystä (engl. type confusion) hyödyntämällä [24].

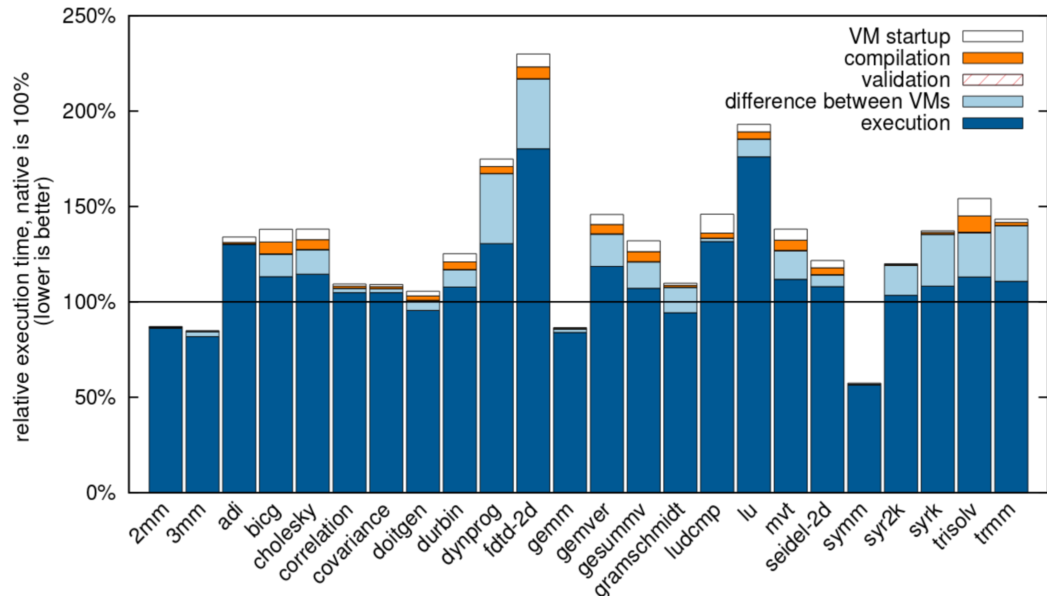
Edellä mainitut hyökkäykset on mahdollista estää kirjoittamalla turvallista koodia ja valitsemalla oikeanlaiset käännösparametrit [24,40]. Kuitenkin WebAssemblyn binäärimuoto aiheuttaa luonnostaan turvallisuusongelmia internetissä, koska sen tulkitseminen on huomattavasti haastavampaa kuin JavaScript-lähdekoodin tulkitseminen. Tästä johtuen haitallisten WebAssembly-ohjelmien havaitseminen on vaikeata. WebAssemblya pystytään hyödyntämään esimerkiksi teknisen tuen huijauksissa (engl. tech support scam) ja näppäilytallentimissa (engl. keylogger), koska ainoa WebAssemblyn jättämä jälki on itse WebAssembly-koodi binäärimuodossa. Myös WebAssemblyn suorituskyky houkuttelee haittaohjelmien tekemistä WebAssemblylla. Sen suorituskyvyn takia se soveltuu hyvin käytettäväksi esimerkiksi selainpohjaisissa kryptovaluuttalouhijoissa. [37]

WebAssemblyn turvallisuuden kannalta on kriittistä, että myös WebAssemblyn sulauttajan toteutus on turvallinen. Internetselaimissa tämä tarkoittaa JavaScript-moottorei-

den turvallisuutta. JavaScript-moottoreiden WebAssembly-toteutuksista onkin löydetty useita bugeja, jotka ovat voineet aiheuttaa tietoturvaongelmia. Suurin osa näistä bugeista liittyy WebAssemblyn binäärimuodon jäsentämiseen. Myöskin WebAssemblyn suorituksessa on ilmennyt ongelmia, joissa WebAssemblysta generoitu koodi on ollut väärin. Väärin generoitu koodi ei kuitenkaan ole aiheutunut selkeätä tietoturvariskiä. Tämän lisäksi WebAssemblyyn liittyviä bugeja on löytynyt vähemmän kuin muihin internetselainten uusiin ominaisuuksiin liittyviä bugeja, mikä todennäköisesti johtuu WebAssemblyn ja sitä varten toteutetun muistinhallinnan yksinkertaisuudesta. [58]

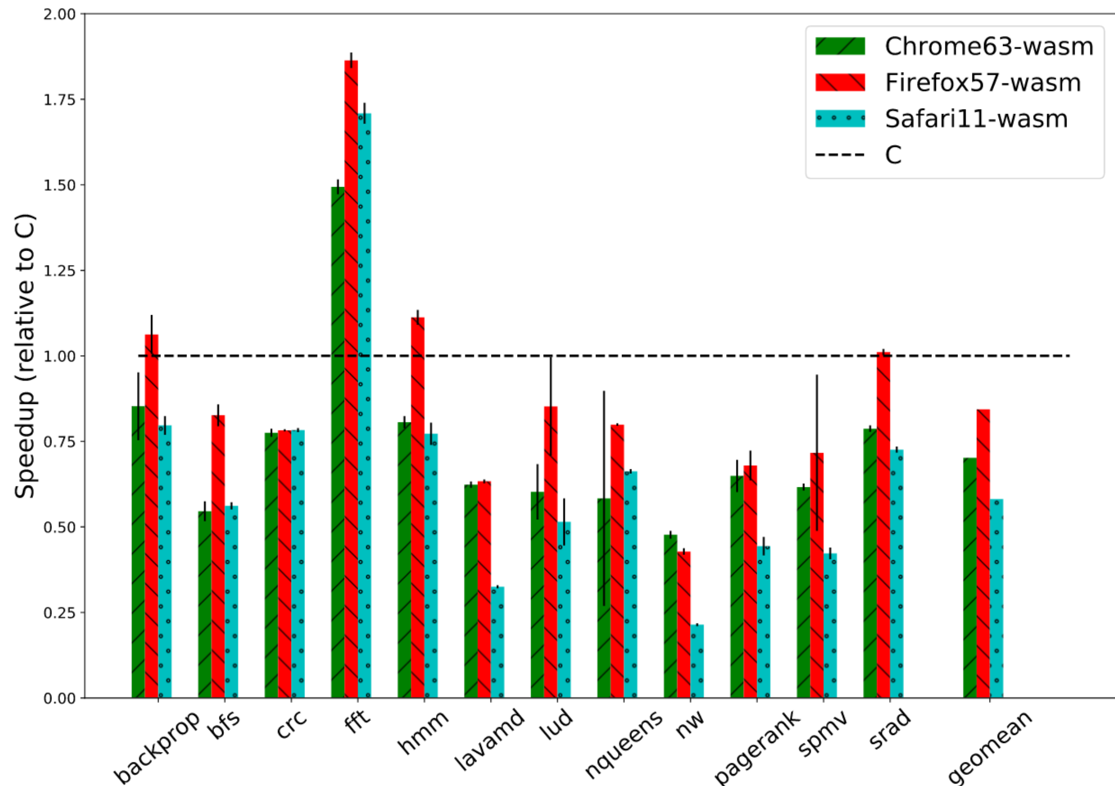
Jatkossa WebAssemblya tullaan mahdollisesti hyödyntämään internetselainten haavoittuvuuksien hyväksikäyttämisessä ja vihamielisten verkkosivujen uudelleenohjausketjujen toteuttamisessa [37]. Tämä on jo tällä hetkellä nähtävissä, sillä Musch et al. havaitsivat tutkimuksessaan, että 56% WebAssemblya käyttävistä sivustoista Alexa Top 1 Million -listauksessa käyttävät sitä haitallisiin tarkoituksiin, kuten kryptovaluutan louhintaan tai obfuskointiin [45]. WebAssemblylla toteutetun kryptovaluutan louhijan havaitsemiseen ja siltä suojautumiseen on kehitetty jo useita eri metodeja [9,10,35]. Myöskin WebAssemblyyn tulevat uudet ominaisuudet tuovat mukanaan todennäköisesti tietoturvaongelmia. Erityisesti säikeiden ja automaattisen roskienkeräyksen toteuttaminen aiheuttavat todennäköisesti haavoittuvuuksia. [40,58]

WebAssemblyn suorituskyvyn tavoitteena on natiivin koodin suorituskyky [33,74]. Ainakaan vielä se ei saavuta kyseistä tavoitetta, mutta sen suorituskyky on kuitenkin huomattavasti JavaScriptin suorituskykyä parempi [26,31,33,61]. Haas et al. analysoivat WebAssemblyn suorituskykyä työssään PolyBenchC-suorituskykytestikokonaisuudella (engl. benchmark suite). Testien tuloksien vertailu perustui testin suorittamiseen kuluneeseen aikaan. Testikohteina olivat C-koodista käännetty natiivi koodi, asm.js ja WebAssembly. Testeissä WebAssemblyn ja asm.js:sän suorittamiseen käytettiin sekä V8- että SpiderMonkey-JavaScript-moottoreita. Saatujen tuloksien mukaan WebAssembly oli testeissä keskimäärin 33,7% nopeampi kuin asm.js. Kuitenkin WebAssemblyn suoritusaika oli suurimmillaan yli kaksinkertainen verrattuna vastaavan natiivin koodin suorittamiseen. Kuvassa 1 on esitelty WebAssemblyn suhteelliset testitulokset normalisoituna natiivin koodin suhteen. Kuvassa tuloksista esitetään myös V8:n ja SpiderMonkeyn välisiä eroja. [26]



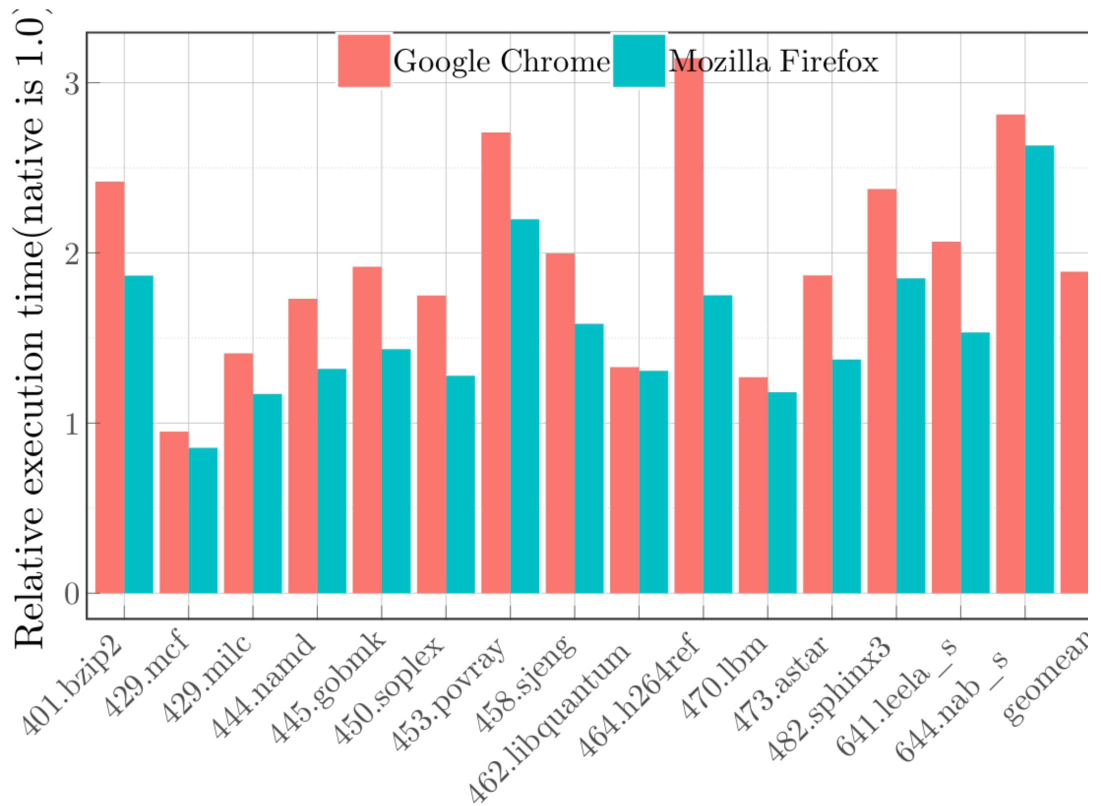
Kuva 1. *WebAssemblyn suhteellinen suoritusaika normalisoituna C-koodista käännetyn natiivin koodin suoritusaikaan PolyBenchC-suorituskykytestikokonaisuuden testeissä. [26]*

Haas et al. työssä saatuja tuloksia tukevat myös Herrera et al. tutkimuksessa saadut tulokset. Herrera et al. perehtyivät työssään WebAssemblyn hyödyntämiseen numeerisessa laskennassa. Työssä verrattiin natiivin koodin, WebAssemblyn ja JavaScriptin suorituskykyä hyödyntämällä Ostrich-suorituskykytestikokonaisuutta, joka sisältää useita numeerista laskennan tehokkuutta mittaavia testejä. Työssä käytettiin testien suorittamiseen useita erilaisia suoritusympäristöjä. Esimerkiksi testejä suoritettiin muun muassa iPhoneX-puhelimella ja MacBookPro 2018 -kannettavalla tietokoneella. Tämän lisäksi testien suorittamiseen käytettiin Chrome, Firefox ja Safari -internetselaimia. Tuloksena havaittiin, että WebAssembly on huomattavasti JavaScriptia nopeampi numeerisessa laskennassa, yleensä 1,5-2 kertaa nopeampi. Kuitenkin jälleen havaittiin myös, että WebAssembly on hitaampaa kuin C-koodista käännetty natiivi koodi. MacBookPro 2018 -kannettavalla tietokoneella suoritettujen Ostrich-suorituskykytestikokonaisuuden testien mukaan Chrome, Firefox ja Safari -internetselaimilla WebAssembly on keskimäärin 1,41, 1,20 ja 1,76 kertaa hitaampaa kuin vastaava natiivi koodi. Kuvassa 2 on esitetty tarkemmin näihin keskimääräisiin arvoihin johtaneiden testien tulokset. [31]



Kuva 2. *WebAssemblyn suorituskyky suhteessa C-kielestä käännettyyn natiiviin koodin suorituskykyyn Ostrich-suorituskykytestikokonaisuuden testeissä. [31]*

Sekä Haas et al. että Herrera et al. käyttivät tutkimuksissaan testejä, jotka ovat pieniä tieteelliseen laskentaan keskittyviä testejä, minkä takia niiden tulokset eivät välttämättä kuvasta selaimeen tarkoitettujen ohjelmien suorituskykyä. Tämän takia Jangda et al. ottivat työnsä tavoitteeksi tutkia WebAssemblyn suorituskykyä suurempien ohjelmien kanssa ja valitsivat suorituskykytestikokonaisuudekseen SPEC CPU -paketin, jonka versioita 2006 ja 2017 käytettiin työssä. SPEC CPU sisältää laajoja testejä, jotka on kehitetty todellisia ohjelmia hyödyntäen, mistä johtuen sen avulla pystytään tekemään perusteellisempi arvio suorituskyvystä. Jangda et al. vertailivat työssään WebAssemblyn suorituskykyä ams.js:n ja natiivin koodin suorituskykyyn. Työssä käytettiin WebAssemblyn ja asm.js:sän suoritusympäristöinä Chrome ja Firefox -internetselaimia. Työn tuloksena saatiin, että WebAssembly on keskimäärin 30% nopeampi kuin asm.js. Lisäksi WebAssemblyn ja natiivin koodin suorituskyvyn vertailun tuloksena oli se, että WebAssemblyn suorittaminen oli keskimäärin 1,5 kertaa hitaampaa Firefoxissa ja 1,9 kertaa hitaampaa Chromessa verrattuna natiivin koodin suorittamiseen. Kuvassa 3 on esitetty tarkemmin WebAssemblyn suhteelliset suoritusajat SPEC CPU -suorituskykytestikokonaisuuden testeistä normalisoituna natiivin koodin suoritusaikoihin. [33]



Kuva 3. *WebAssemblyn suhteelliset suoritusajat SPEC CPU -suorituskykytesti-kokonaisuuden testeissä normalisoituna natiivin koodin suoritusaikoihin sekä Firefox että Chrome -internetselaimilla. [33]*

WebAssemblyn tarjoaman paremman suorituskyvyn takia useat palveluntarjoajat ovat ottaneet WebAssemblyn käyttöön tuotteissaan. Esimerkiksi eBay käyttää WebAssemblya websovelluksessaan viivakoodien skannaamiseen ja tunnistamiseen. JavaScriptiin perustuva kirjasto oli yksinään liian hidas ja WebAssemblyyn käännetty natiivi viivakoodinlukukirjasto toimikin 50-kertaa nopeammin. Lopulta eBay päätyi kuitenkin käyttämään rinnakkain kolmea eri viivakoodinlukijakirjastoa: kahta WebAssemblylla toimivaa ja yhtä JavaScriptilla toimivaa. [50] Figma on ottanut WebAssemblyn käyttöönsä käyttöliittymien suunnittelemiseen tarkoitetussa websovelluksessaan. Figma käytti aikaisemmin asm.js:sään perustuvaa ratkaisua, mutta korvasi asm.js:sän WebAssemblylla, mikä johti Figman websovelluksen latausnopeuden 3-kertaistumiseen. Latausnopeus mitattiin mittaamalla websovelluksen alustamiseen kuluva aika, johon lisättiin suunnittelutiedoston lataamiseen käytetty aika sekä sen lopulliseen renderöintiin käytetty aika. [63]

WebAssemblyn siirrettävyyteen liittyy olennaisesti sen suorittaminen erilaisilla laitteilla ja teoriassa sen suorittaminen useilla erilaisilla laitteilla on mahdollista. Vuoden 2020 huhtikuussa WebAssemblya pystytään käytännössä suorittamaan ainakin x86- ja AArch64-suoritinarkkitehtuureilla [3] ja 90,5% käytössä olevista selaimista tukee sen suorittamista [13]. Huolimatta siitä, että WebAssembly on suunniteltu hyvin siirrettäväk-

si ohjelmointikieleksi, liittyy siihen silti tiettyjä rajoitteita, jotta sen tehokas suorittaminen olisi mahdollista. Jos suoritusympäristö ei tue näitä rajoitteita, niin niitä täytyy emuloida, mikä voi johtaa huonoon suorituskykyyn. Nämä rajoitteet on esitetty taulukossa 1. [68]

Taulukko 1. WebAssemblyn optimaaliseen suorituskykyyn vaadittavat asiat. [68]

WebAssemblyn suoritusympäristöltä oletettavat asiat tehokasta suorittamista varten
8-bittiset tavut
Tavu-tarkkuudella osoitettavissa oleva muisti (engl. addressable at a byte memory granularity)
Tasaamaton muistissakäynti tai sen emulointi luotettavien ansojen avulla (engl. unaligned memory access or reliable trapping that allows software emulation thereof)
Kahden komplementti etumerkilliset kokonaisluvut (engl. two’s complement signed integers) 32-bittisenä ja vapaaehtoisesti 64-bittisenä
IEEE 754-2008 standardin mukaiset 32-bittiset ja 64-bittiset liukuluvut muutamaa poikkeusta lukuunottamatta.
Little endian -tavujärjestys
Muistialueet, joihin päästään tehokkaasti 32-bittisillä osoittimilla tai indekseillä. Tuettaessa 64-bittistä WebAssemblya tehokkaan pääsyn täytyy toimia myös 64-bittisillä osoittimilla ja indekseillä.
Turvallisen eristämisen pakottaminen samalla koneella suorituksessa olevien WebAssembly-moduulien ja muiden moduulien sekä prosessien välillä.
Edistymisen takaaminen (engl. forward progress guarantee) kaikille suorituksessa oleville säikeille.
Lukottomat atomiset muistioperaattorit luonnollisesti kohdistetuille 8-, 16- ja 32-bittisille muistissakäynneille sekä 64-bittistä WebAssemblya varten 64-bittisille muistissakäynneille.

WebAssemblyn siirrettävyys on hyvin ilmeistä käännöskohde-näkökulmasta. Vuoden 2020 huhtikuussa WebAssemblyyn kääntämistä tukivat muun muassa C, C#, C++, COBOL, Go, Lua ja Rust [4]. Tätä käännettävyyttä onkin hyödynnetty tuotantokäytössä asti. Esimerkiksi Autodesk julkaisi websovelluksen AutoCAD-ohjelmistostaan, jonka koodikanta on 30 vuotta vanha, WebAssemblyyn kääntämisen avulla [14].

WebAssemblyyn käännetyn koodin mahdollisiin suoritusympäristöihin kuuluu enemmän vaihtoehtoja kuin pelkästään internetselain, kuten WebAssemblyn kehittäjät uskoivatkin [68]. Palvelimella WebAssemblyn suorittamiseen löytyy useita vaihtoehtoja, joihin kuuluvat esimerkiksi Wasmer ja Wasmtime, jotka tarjoavat erillisen runtime-ympäristön WebAssemblylle, mikä pystytään sulauttamaan osaksi muilla ohjelmointikielillä tehtyjä ohjelmia. Tämä mahdollistaa siis WebAssembly-moduulien suorittamisen osana toisella ohjelmointikielellä tehtyä ohjelmaa. Suosittu JavaScript runtime-ympäristö Node.js tarjoaa myös mahdollisuuden WebAssemblyn suorittamiseen W3C:n WebAssembly -yhteisöryhmän kehittämän WebAssembly System Interface (WASI) -standardin kautta, jota myös Wasmer ja Wasmtime hyödyntävät. [17,48,65,66] Siirrettävyyden näkökulmasta tämä tarkoittaa sitä, että WebAssemblyyn käännettävissä olevaa koodia pystytään suorittamaan monenlaisissa ympäristöissä erilaisilla teknologioilla. Myös yksi

Dockerin luojista, Solomon Hykes, totesi, että Docker-konttiteknologiaa ei olisi tarvinnut luoda, jos WebAssembly ja WASI olisivat olleet olemassa jo vuonna 2008 [32].

Ohjelmien tiiviys on yksi WebAssemblyn tavoitteista, mitä Haas et al. mittasivat tutkimuksessaan vertaamalla WebAssemblyn kokoa vastaaviin x86-64 natiivin koodin ja asm.js:sän kokoihin. WebAssemblyn ja asm.js:sän koodin kokojen vertailussa käytettiin Unityn suorituskykytestejä ja WebAssemblyn ja natiivin koodin välisessä vertailussa käytettiin PolyBenchC- ja SciMark-suorituskykytestejä. Tulokseksi saatiin, että WebAssemblyn koko on keskimäärin 62,5% asm.js:sän koosta ja mediaani oli 68,6%. Natiiviin koodiin verrattaessa WebAssembly-koodin koko oli keskimäärin 85,3% natiivin koodin koosta ja mediaani oli 78%. Tapauksissa, joissa WebAssemblyn täytyy luoda varjopino, sen koodin koko oli natiivin koodin kokoa suurempi. [26]

Figman tekemässä testauksessa havaittiin hyvin samankaltaisia tuloksia WebAssemblyn tiiviydestä, kun WebAssembly-koodin kokoa verrattiin asm.js-koodin kokoon heidän omassa ohjelmassaan. Kuitenkin Figman tekemässä testauksessa havaittiin, että koodin pakkaamisen jälkeen WebAssembly- ja asm.js-koodin kokojen välinen suhteellinen ja absoluuttinen ero pienenivät huomattavasti. Figman testauksissa pakkaamiseen käytettiin gzip- ja brotli-pakkausteknologioita. WebAssemblyn tiiviydestä voidaankin todeta, että se on yleensä huomattavasti tiiviimpää kuin vastaava asm.js-koodi tai natiivi koodi, mutta yleisesti internetissä käytössä olevat pakkausteknologiat vähentävät WebAssemblyn tiiviydestä saatavia hyötyjä internet-kontekstissa. [63]

5. JOHTOPÄÄTÖKSET

WebAssembly on matalan tason tavukoodi, jolla on assembly-kielen kaltainen rakenne ja syntaksi, kun se esitetään ohjelmointikielenä [26]. Se on tarkoitettu käännöskohteeksi muille ohjelmointikielille [26], joista WebAssemblyyn kääntämisen motivaationa toimii WebAssemblyn tarjoama ohjelmistokieli-, laitteisto- ja alustariippumattomuus [69]. Erityisesti se, että yli 90% kaikista käytössä olevista internetselaimista tukee WebAssemblyn suorittamista [13], lisää sen houkuttelevuutta käännöskohteena. Myös WebAssemblyn pääperiaatteina toimivat turvallisuus, suorituskyky, siirrettävyys ja tiiviys [26] ovat usein haluttuja ominaisuuksia. Kyseiset ominaisuudet ovat kuitenkin myös syy siihen, että WebAssembly on olemassa, koska yksikään WebAssemblya edeltäneistä teknologioista ei täyttänyt kyseisiä vaatimuksia [26].

WebAssemblyn MVP-versio saavuttaa WebAssemblyn päätavoitteista pääosin siirrettävyyden ja tiiviyden, mutta suorituskyvyssä ja turvallisuudessa on vielä parannettavaa. Suorituskyky ei ole natiivin koodin tasolla [26,31,33] ja ei mahdollisesti ikinä tulekaan olemaan, koska WebAssembly-koodi täytyy kääntää ennen sen suorittamista, kun taas natiivi koodi on käännetty etukäteen ja käännöksen optimointiin on voitu käyttää paljon aikaa. Kuitenkin käännetty WebAssembly voidaan tallentaa väliaikaisesti, jolloin vältytään käännöksen tekemiseltä uudestaan. WebAssemblyn turvallisuus vaikuttaa MVP-toteutuksessa hyvältä, mutta tulevaisuudessa lisättävistä ominaisuuksista mahdollisesti aiheutuvia turvallisuuteen liittyviä ongelmia on syytä pitää silmällä [40,58]. Myöskin se, että yli 50% Alexa Top 1 Million -listauksessa olevista internetsivuista käyttävät WebAssemblya haitallisiin tarkoituksiin [45] on huolestuttavaa ja selainvalmistajien tulisi asiaan reagoida.

Vaikka WebAssembly onkin teknologiana tuore, on se silti valmis tuotantokäyttöä varten, minkä todistavat useat sovellukset, jotka hyödyntävät sitä [14,50,63]. Merkittävä osa websovelluksista ei kuitenkaan sisällä ominaisuuksia, jotka pystyisivät hyödyntämään tehokkaasti WebAssemblya, joten se ei ole korvaamassa JavaScriptia internetissä. WebAssembly sopii käytettäväksi websovelluksissa, joissa selaimen vastuulle jää raskasta prosessointia, esimerkiksi selainpohjaiset videoeditorit, joissa selain on vastuussa videon renderöinnistä. On kuitenkin mahdollista, että WebAssemblyn myötä prosessointivastuuta voidaan siirtää palvelimelta selaimen vastuulle. Tähän voi kannustaa esimerkiksi se, että prosessoinnissa käytettyjä kirjastoja pystytään kääntämään

suoraan WebAssemblyksi, ja prosessoinnin siirtäminen palvelimelta selaimelle voi joissain tilanteissa vähentää kustannuksia tai parantaa käyttäjäkokemusta. Erityisesti kaikkien websovelluksien, jotka käyttävät asm.js:sää, tulisi tarjota myös WebAssembly-versio kyseisestä koodista, koska WebAssembly tarjoaa paremman suorituskyvyn [26,31,33] ja tiiviyden [26,63] ilman merkittäviä haittapuolia. Asm.js-versio olisi kuitenkin hyvä säilyttää käyttäjiä varten, jotka eivät jostain syystä pysty suorittamaan WebAssemblya.

WebAssemblyn hyödyntäminen verkon ulkopuolella pitää sisällään useita mielenkiintoisia mahdollisuuksia, joihin kuuluu esimerkiksi epäluotettavan koodin suorittaminen palvelimella [68]. Dockerin kehittäjän Solomon Hykesin kommentointi [32] viittaa myös siihen, että WebAssemblya pystyttäisiin mahdollisesti käyttämään tulevaisuudessa ohjelmien kontittamiseen. Myöskin ohjelmistokirjastojen tarjoajille WebAssembly voi olla houkutteleva kohde, koska WebAssemblyn nykyiset runtime-ympäristöt tarjoavat sulauttamismahdollisuuden useisiin ohjelmointikieliin [65,66], mikä mahdollistaisi kirjaston käyttämisen hyvin laaja-alaisesti laitteisto- ja alustariippumattomasti.

WebAssemblyn tulevaisuuden näkymät vaikuttavat hyvältä, koska se on aktiivisen kehityksen alla [70] sekä kaikki suuret selainvalmistajat ovat sen takana [68]. Myös se, että suuret yritykset ovat ottaneet WebAssemblyn käyttöön tuotannossa olevissa sovelluksissa vahvistaa WebAssemblyn asemaa. WebAssemblyyn tulevaisuudessa lisättävien ominaisuuksien vaikutus sen suorituskykyyn voi entisestään tehdä WebAssemblysta houkuttelevamman käännöskohteen. WebAssemblyn laajempaan adoptointiin vaikuttaa myös sen tuottamiseen ja ylläpitämiseen vaadittavasta työmäärästä aiheutuvat kustannukset.

6. YHTEENVETO

Työn aiheen valintaan vaikutti kiinnostus ohjelmointikielten väliseen tehokkuuteen. Erityisesti JavaScriptin kaltaisen tulkattavan ohjelmointikielen suorituskyky internetkontekstissa on ollut kiinnostava aihe varsinkin web-ohjelmoinnin ja JavaScriptin käytön suosion nousun myötä. WebAssembly tarjosi hyvän lähestymiskulman aiheeseen. Työssä selvitettiinkin, että mitä ja mikä on WebAssembly, miksi se on olemassa, ja mikä on sen MVP-version nykytilanne.

WebAssembly on suorituskykyinen matalan tason tavukoodi, joka on tarkoitettu käännöskohteeksi muille ohjelmointikielille. WebAssemblyn päätavoitteet ovat turvallisuus, suorituskyky, siirrettävyys ja tiiviys. WebAssemblysta houkuttelevan käännöskohteen tekeekin sen hyvä suorituskyky, tiiviys ja alusta- sekä laitteistoriippumattomuus. Yli 90% nykyään käytössä olevista internetselaimista tukee WebAssemblyn suorittamista natiivisti, mikä on mahdollistanut WebAssemblyn tuotantokäytön websovelluksissa, millä on saavutettu merkittäviä suorituskykyparannuksia. Tämä johtuu siitä, että WebAssembly tarjoaa JavaScriptia huomattavasti paremman suorituskyvyn, mikä on merkittävä motivaatiotekijä WebAssemblyn käyttämiselle internetkontekstissa. WebAssemblya voi kuitenkin käyttää myös internetselaimen ulkopuolella, missä sille on tarjolla useita itsenäisiä runtime-ympäristöjä, joiden avulla pystytään esimerkiksi suorittamaan epäluotettavaa koodia turvallisesti.

WebAssemblyn MVP-version tilanne on siis hyvä suorituskyvyn, siirrettävyyden ja tiiviyden perusteella katsottuna. Kuitenkin turvallisuuspuolella WebAssembly on aiheuttanut ongelmia, sillä yli 50% sivustoista, jotka käyttävät WebAssemblya, käyttävät sitä haitallisiin tarkoituksiin, kuten kryptovaluutan louhintaan. WebAssemblyn MVP-versiosta puuttuu myös ominaisuuksia, jotka ovat aktiivisen suunnittelun ja kehityksen alla, ja joiden puuttuminen voi joissain tilanteissa estää WebAssemblyn hyödyntämisen. Esimerkiksi säikeiden tukeminen ei ole mukana MVP-versiossa.

Työssä perehdyttiin tarkasti WebAssemblyn rakenteeseen ja siihen liittyviin käsitteisiin. Jatkotutkittavaksi jää ainakin WASI-standardiin perehtyminen, mutta esimerkiksi myös eri ohjelmointikielien käännettävyyttä WebAssemblyksi voisi vertailla, koska yksi WebAssemblyn suunnitteluperiaatteista on se, että se ei suosi tiettyjä ohjelmointikieliä tai ohjelmointimalleja.

LÄHTEET

[1] ActiveX Controls (Internet Explorer), Microsoft Docs, 2017. Saatavissa (viitattu 19.3.2019): https://docs.microsoft.com/en-us/previous-versions/windows/internet-explorer/ie-developer/platform-apis/aa751968(v%3dvs.85).

[2] M. Ahmed, Will every cloud have a Chrome lining? The man who built the V8 JavaScript engine from a Danish farm believes it will, writes Murad Ahmed, The Times (London, England), 2011.

[3] S. Akbary, Running WebAssembly on ARM, Medium, 2019. Saatavissa (viitattu 29.4.2020): https://medium.com/wasmer/running-webassembly-on-arm-7d365ed0e50c.

[4] S. Akinyemi, Awesome WebAssembly Languages, GitHub, 2019. Saatavissa (viitattu 28.4.2020): https://github.com/appcypher/awesome-wasm-langs.

[5] Angular, 2019. Saatavissa (viitattu 8.3.2019): https://angular.io/.

[6] S. B. Aruoba, J. Fernández-Villaverde, A comparison of programming languages in macroeconomics, Journal of Economic Dynamics and Control, Vol. 58 , 2015, 265-273, 10.1016/j.jedc.2015.05.009.

[7] R. Auler, E. Borin, P. De Halleux, M. Moskal, N. Tillmann, Addressing JavaScript JIT engines performance quirks: A crowdsourced adaptive compiler, Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), Vol. 8409, 2014, 218-237, doi:10.1007/978-3-642-54807-9_13.

[8] L. Bak, Google Chrome's Need for Speed, Chromium Blog, 2008. Saatavissa (viitattu 21.3.2019): https://blog.chromium.org/2008/09/google-chromes-need-for-speed_02.html.

[9] W. Bian, W. Meng, Y. Wang, Poster: Detecting WebAssembly-based Cryptocurrency Mining , Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, 2019, 2685-2687.

[10] W. Bian, W. Meng, M. Zhang, MineThrottle: Defending against Wasm In-Browser Cryptojacking , Proceedings of the Web Conference 2020, 2020, 3112-3118.

[11] K. Brown, JavaScript: How Did It Get So Popular? 2018. Saatavissa (viitattu 8.3.2019): https://news.codecademy.com/javascript-history-popularity/.

[12] Can I use asm.js, Can I use..., 2019. Saatavissa (viitattu 25.3.2019): https://caniuse.com/#feat=asmjs.

[13] Can I use WebAssembly, Can I use..., 2020. Saatavissa (viitattu 29.4.2020): https://caniuse.com/#feat=wasm.

[14] K. Cheung, AutoCAD & WebAssembly: Moving a 30 Year Code Base to the Web , QCon New York 2018, Sep 26, 2018.

[15] L. Clark, A cartoon intro to WebAssembly, 2017. Saatavissa (viitattu 8.3.2019): https://hacks.mozilla.org/2017/02/a-cartoon-intro-to-webassembly/.

[16] L. Clark, Calls between JavaScript and WebAssembly are finally fast □, Mozilla Hacks, 2018. Saatavissa (viitattu 10.4.2019): https://hacks.mozilla.org/2018/10/calls-between-javascript-and-webassembly-are-finally-fast-%F0%9F%8E%89/.

[17] L. Clark, Standardizing WASI: A system interface to run WebAssembly outside the web, 2019. Saatavissa (viitattu 28.4.2020): https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface/.

[18] J. Conrod, A tour of V8: full compiler, jayconrod, 2012. Saatavissa (viitattu 21.3.2019): https://www.jayconrod.com/posts/51/a-tour-of-v8-full-compiler.

[19] M. Coppock, The best web browsers for 2019, Digital Trends, 2019. Saatavissa (viitattu 21.3.2019): https://www.digitaltrends.com/computing/best-browser-internet-explorer-vs-chrome-vs-firefox-vs-safari-vs-edge/.

[20] A. Donnelly, How To Reduce Risks With ActiveX, Network Security, Vol. 2001, (2) , 2001, 5, 10.1016/S1353-4858(01)00212-4.

[21] A. Donovan, R. Muth, B. Chen, D. Sehr, PNaCl: Portable native client executables, Google White Paper , 2010.

[22] Emscripten Contributors, Emscripten Documentation, Emscripten Documentation, 2019. Saatavissa (viitattu 19.4.2019): https://emscripten.org/docs/.

[23] ExE-Boss, Jean-Yves-P, arai, mgaudet, pipcet, xfq, . . . SylvainPasche, SpiderMonkey Internals, Mozilla | MDN, 2018. Saatavissa (viitattu 21.3.2019): https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey/Internals.

[24] J. Foote, Hijacking the control flow of a WebAssembly program, Fastly Blog, 2018. Saatavissa (viitattu 22.4.2019): https://www.fastly.com/blog/hijacking-control-flow-webassembly-program.

[25] C. Ge, K. Huang, Analyzing the Economies of Scale of Software as a Service Software Firms: A Stochastic Frontier Approach, IEEE Transactions on Engineering Management, Vol. 61, (4) , 2014, 610-622, 10.1109/TEM.2014.2359975.

[26] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, . . . J. F. Bastien, Bringing the web up to speed with WebAssembly, Proceedings of the 38th ACM SIGPLAN Conference on programming language design and implementation, Vol. 128414, 2017, 185-200, doi:10.1145/3062341.3062363.

[27] N. Hamilton, The A-Z of Programming Languages: JavaScript, 2008. Saatavissa (viitattu 18.2.2019): https://www.computerworld.com.au/article/255293/a-z_programming_languages_javascript/.

[28] C. Hammacher, Liftoff: a new baseline compiler for WebAssembly in V8, V8, 2018. Saatavissa (viitattu 16.4.2019): https://v8.dev/blog/liftoff.

[29] H. Heiskanen, Yhden sivun web-sovellukset tulevat, oletko valmis? 2013. Saatavissa (viitattu 8.3.2019): https://gofore.com/yhden-sivun-web-sovellukset-tulevat-oletko-valmis/.

[30] D. Herman, L. Wagner, A. Zakai, Working Draft, asm.js, 2014. Saatavissa (viitattu 25.3.2019): http://asmjs.org/spec/latest/.

[31] D. Herrera, E. Lavoie, H. Chen, L. Hendren, Numerical computing on the web: Benchmarking for the future, DLS 2018 - Proceedings of the 14th ACM SIGPLAN International Symposium on Dynamic Languages, co-located with SPLASH 2018, 2018, 88-100, doi:10.1145/3276945.3276968.

[32] S. Hykes, Solomon Hykes on Twitter, Twitter, 2019. Saatavissa (viitattu 28.4.2020): https://twitter.com/solomonstre/status/1111004913222324225.

[33] A. Jangda, B. Powers, A. Guha, E. Berger, Mind the Gap: Analyzing the Performance of WebAssembly vs. Native Code , 2019.

[34] KeyCDN, The Growth of Web Page Size, KeyCDN, 2018. Saatavissa (viitattu 18.2.2019): https://www.keycdn.com/support/the-growth-of-web-page-size.

[35] A. Kharraz, Z. Ma, P. Murley, C. Lever, J. Mason, A. Miller, . . . M. Bailey, Outguard: Detecting in-browser covert cryptocurrency mining in the wild , The World Wide Web Conference, 2019, 840-852.

[36] A. Link, Big browser comparison test: Internet Explorer vs. Firefox, Opera, Safari and Chrome - Update: Firefox 3.5 Final, PC Games Hardware, 2009. Saatavissa (viitattu 21.3.2019): http://www.pcgameshardware.de/Tools-Software-156186/Tests/Internet-browser-tested-with-Peacekeeper-Sunspider-and-Acid-687738/.

[37] A. Lonkar, S. Chandrayan, The dark side of WebAssembly, Virus Bulletin Conference October 2018 , 2018.

[38] D. Mandelin, an overview of TraceMonkey, Mozilla Hacks, 2009. Saatavissa (viitattu 21.3.2019): https://hacks.mozilla.org/2009/07/tracemonkey-overview/.

[39] J. K. Martinsen, H. Grahn, A. Isberg, Combining thread-level speculation and just-in-time compilation in Google's V8 JavaScript engine, Concurrency and Computation: Practice and Experience, Vol. 29, (1) , 2017, n/a, 10.1002/cpe.3826.

[40] B. McFadden, T. Lukasiewicz, J. Dileo, J. Engler, NCC Group Whitepaper: Security Chasms of WASM, (1.0) , 2018.

[41] R. McIlroy, Firing up the Ignition interpreter, V8, 2016. Saatavissa (viitattu 21.3.2019): https://v8.dev/blog/ignition-interpreter.

[42] Microsoft Edge Team, A break from the past, part 2: Saying goodbye to ActiveX, VBScript, attachEvent…, Microsoft Edge Blog, 2015a. Saatavissa (viitattu 19.3.2019): https://blogs.windows.com/msedgedev/2015/05/06/a-break-from-the-past-part-2-saying-goodbye-to-activex-vbscript-attachevent/.

[43] Microsoft Edge Team, Bringing Asm.js to Chakra and Microsoft Edge, Microsoft Edge Blog, 2015b. Saatavissa (viitattu 25.3.2019): https://blogs.windows.com/msedgedev/2015/05/07/bringing-asm-js-to-chakra-microsoft-edge/.

[44] A. Møller, WebAssembly: a quiet revolution of the web: technical perspective, Communications of the ACM, Vol. 61, (12) , 2018, 106, 10.1145/3282508.

[45] M. Musch, C. Wressnegger, M. Johns, K. Rieck, New kid on the web: A study on the prevalence of webassembly in the wild, Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), Vol. 11543, 2019, 23-42, doi:10.1007/978-3-030-22038-9_2.

[46] Native Client, Chrome Developers. Saatavissa (viitattu 20.3.2019): https://developer.chrome.com/native-client.

[47] B. Nelson, Goodbye PNaCl, Hello WebAssembly! Chromium Blog, 2017. Saatavissa (viitattu 21.3.2019): https://blog.chromium.org/2017/05/goodbye-pnacl-hello-webassembly.html.

[48] Node.js contributors, Node.js v14.0.0 Documentation, Node.js, 2020. Saatavissa (viitattu 28.4.2020): https://nodejs.org/api/wasi.html.

[49] A. Osmani, 10 years of Speed in Chrome Chromium Blog, 2018. Saatavissa (viitattu 21.3.2019): https://blog.chromium.org/2018/09/10-years-of-speed-in-chrome_11.html.

[50] S. Padmanabhan, P. Jha, WebAssembly at eBay: A Real-World Use Case, eBay, 2019. Saatavissa (viitattu 10.3.2020): https://tech.ebayinc.com/engineering/webassembly-at-ebay-a-real-world-use-case/.

[51] R. Paul, Firefox to get massive JavaScript performance boost, Ars Technica, 2008. Saatavissa (viitattu 21.3.2019): https://arstechnica.com/information-technology/2008/08/firefox-to-get-massive-javascript-performance-boost/.

[52] R. Pereira, M. Couto, F. Ribeiro, R. Rua, J. Cunha, J. P. Fernandes, J. Saraiva, Energy efficiency across programming languages: how do energy, time, and memory relate?, Proceedings of the 10th ACM SIGPLAN International Conference on software language engineering, 2017, 256-267, doi:10.1145/3136014.3136031.

[53] S. Pieters, What’s new in Chromium 57 and Opera 44, Dev.Opera, 2017. Saatavissa (viitattu 27.3.2019): https://dev.opera.com/blog/opera-44/.

[54] K. Purdy, Lifehacker Speed Tests: Safari 4, Chrome 2, and More, Lifehacker, 2009. Saatavissa (viitattu 21.3.2019): https://lifehacker.com/lifehacker-speed-tests-safari-4-chrome-2-and-more-5286869.

[55] React - A JavaScript library for building user interfaces, Saatavissa (viitattu 8.3.2019): https://reactjs.org/.

[56] M. Selakovic, M. Pradel, Performance issues and optimizations in JavaScript: an empirical study, Proceedings of the 38th International Conference on software engineering, 2016, 61-72, doi:10.1145/2884781.2884829.

[57] Y. Shi, K. Casey, M. Ertl, D. Gregg, Virtual machine showdown: Stack versus registers, ACM Transactions on Architecture and Code Optimization (TACO), Vol. 4, (4) , 2007, 1-36, 10.1145/1328195.1328197.

[58] N. Silvanovich, The Problems and Promise of WebAssembly, Project Zero, 2018. Saatavissa (viitattu 22.4.2019): https://googleprojectzero.blogspot.com/2018/08/the-problems-and-promise-of-webassembly.html.

[59] C. Smith, A Brief History of Google's V8 JavaScript Engine, 2017. Saatavissa (viitattu 8.3.2019): https://www.mediacurrent.com/blog/brief-history-googles-v8-javascript-engine/.

[60] J. Staromiejska, Why and When Should You Consider Node.js for Your Next Project? Monterail, 2018. Saatavissa (viitattu 18.3.2019): https://www.monterail.com/blog/when-use-nodejs-development.

[61] A. Turner, WebAssembly Is Fast: A Real-World Benchmark of WebAssembly vs. ES6, Medium, 2018. Saatavissa (viitattu 3.5.2019): https://medium.com/@torch2424/webassembly-is-fast-a-real-world-benchmark-of-webassembly-vs-es6-d85a23f8e193.

[62] L. Wagner, asm.js in Firefox Nightly, Luke Wagner's Blog, 2013. Saatavissa (viitattu 25.3.2019): https://blog.mozilla.org/luke/2013/03/21/asm-js-in-firefox-nightly/.

[63] E. Wallace, WebAssembly cut Figma's load time by 3x, Medium, 2017. Saatavissa (viitattu 3.5.2019): https://medium.com/figma-design/webassembly-cut-figmas-load-time-by-3x-76f3f2395164.

[64] D. Wallach, Technical perspective Native Client: a clever alternative, Communications of the ACM, Vol. 53, (1) , 2010, 90, 10.1145/1629175.1629202.

[65] Wasmer contributors, Wasmer Docs, 2020. Saatavissa (viitattu 28.4.2020): https://docs.wasmer.io/.

[66] Wasmtime contributors, Wasmtime, 2020. Saatavissa (viitattu 28.4.2020): https://bytecodealliance.github.io/wasmtime/.

[67] C. Watt, Mechanising and verifying the WebAssembly specification, Proceedings of the 7th ACM SIGPLAN International Conference on certified programs and proofs, Vol. 2018-, 2018, 53-65, doi:10.1145/3167082.

[68] WebAssembly. Saatavissa (viitattu 27.3.2019): https://webassembly.org/.

[69] WebAssembly Community Group, WebAssembly Specification, (Release 1.0) , 2019.

[70] WebAssembly contributors, WebAssembly, Github, 2020. Saatavissa (viitattu 30.4.2020): https://github.com/WebAssembly.

[71] B. Yee, D. Sehr, G. Dardyk, J. Chen, R. Muth, T. Ormandy, . . . N. Fullagar, Native Client: a sandbox for portable, untrusted x86 native code, Communications of the ACM, Vol. 53, (1) , 2010, 99, 10.1145/1629175.1629203.

[72] A. Zakai, Emscripten: an LLVM-to-JavaScript compiler, Proceedings of the ACM international conference companion on object oriented programming systems languages and applications companion, 2011, 301-312, doi:10.1145/2048147.2048224.

[73] A. Zakai, Compiling to WebAssembly: It's Happening! Mozilla Hacks, 2015. Saatavissa (viitattu 19.4.2019): https://hacks.mozilla.org/2015/12/compiling-to-webassembly-its-happening/.

[74] A. Zakai, Why WebAssembly is Faster Than asm.js, Mozilla Hacks, 2017. Saatavissa (viitattu 25.3.2019): https://hacks.mozilla.org/2017/03/why-webassembly-is-faster-than-asm-js/.

[75] A. Zakai, Small WebAssembly Binaries with Rust + Emscripten, Alon Zakai's Blog, 2018. Saatavissa (viitattu 19.4.2019): https://kripken.github.io/blog/binaryen/2018/04/18/rust-emscripten.html.

[76] A. Zakai, R. Nyman, Gap between asm.js and native performance gets even narrower with float32 optimizations, Mozilla Hacks, 2013. Saatavissa (viitattu 29.3.2019): https://hacks.mozilla.org/2013/12/gap-between-asm-js-and-native-performance-gets-even-narrower-with-float32-optimizations/.