TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

ALTTI HIIRONEN

ÄLYSOPIMUKSET JA ETHEREUM-SOVELLUSALUSTA

Diplomityö

Tarkastajat: professori Jarmo Harju
ja DI Joona Kannisto
Tarkastajat ja aihe hyväksytty
7. joulukuuta 2016

TIIVISTELMÄ

ALTTI HIIRONEN: Älysopimukset ja Ethereum-sovellusalusta
Tampereen teknillinen yliopisto
Diplomityö, 55 sivua, 1 liitesivu
Joulukuu 2017
Tietotekniikan diplomi-insinöörin tutkinto-ohjelma
Pääaine: Tietoturvallisuus
Tarkastajat: professori Jarmo Harju ja diplomi-insinööri Joona Kannisto
Avainsanat: älysopimus, lohkoketju, ohjelmointikäytännöt, digitaalinen valuutta

Lohkoketjuteknologialla tarkoitetaan hajautettua vertaisverkossa toimivaa tietokantaa, jonka ylläpitoon jokainen vertaisverkossa oleva toimija osallistuu. Lohkoketjut mahdollistavat luottamuksen rakentamisen Internetissä eri toimijoiden välille ilman tarvetta kolmanteen osapuoleen, jonka tehtävänä olisi varmentaa toimijoiden välisten viestien oikeellisuus. Lohkoketjuteknologia on mahdollistanut uusien vaihtoehtoisten digitaalisten valuuttojen toteuttamisen. Ensimmäinen lohkoketjuteknologiaa hyödyntävä ja edelleen käytössä oleva järjestelmä oli digitaalinen valuutta nimeltään Bitcoin.

Bitcoinin ideaa eteenpäin vievä hajautettu järjestelmä Ethereum mahdollistaa ohjelmoitavan digitaalisen valuutan käyttäen älysopimuksia. Älysopimukset ovat itseään suorittavia tietokoneohjelmia, jotka kuvaavat sopimuspohjaista toimintaa. Ethereum mahdollistaa älysopimusten suorittamisen lohkoketjussa, jossa niihin ohjelmoidut sopimusehdot toteutuvat ilman kolmannen osapuolen varmennusta. Tämän työn tarkoituksena on selvittää, kuinka lohkoketjuteknologia on teknisesti toteutettu, käyttäen esimerkkinä Bitcoinia. Tämän pohjalta käydään läpi myös Ethereum-sovellusalustan toteutus. Työssä toteutetaan sovellus, joka hyödyntää Ethereum-sovellusalustalle ohjelmoituja älysopimuksia, ja arvioidaan millaisia tietoturvariskejä ja ohjelmointikäytäntöjä niiden toteuttamiseen liittyy.

Työssä toteutettu sovellus mahdollistaa työsopimusten solmimisen käyttäen Ethereum-sovellusalustalle ohjelmoituja älysopimuksia. Sovellus toteutettiin asiakas-palvelin-mallia hyödyntävänä web-sovelluksena, jossa palvelin on yhteydessä Ethereum-sovellusalustaan. Sovelluksessa hyödynnettiin myös verkkoselaimeen saatavilla olevaa MetaMask-laajennusta, jonka avulla asiakaskäyttöliittymässä voidaan olla myös suoraan yhteydessä älysopimusten kanssa käyttäen käyttäjän omaa Ethereum-tiliä.

Yleisesti älysopimusten toteuttamisessa on tärkeää ymmärtää niiden suorittamisen mahdolliseksi tekevän lohkoketjuteknologian toiminta ja ominaisuudet, jotta mahdollisiin haavoittuvuuksiin ja virhetilanteisiin voidaan varautua. Ohjelmointikäytännöt muokkaantuvat jatkuvasti, koska teknologia on vielä luonteeltaan uutta. Lisäksi Ethereum-sovellusalusta on saamassa seuraavien vuosien aikana merkittäviä päivityksiä, minkä vuoksi siihen liittyvän kehityksen ja uutisoinnin seuraamista suositellaan.

ABSTRACT

ALTTI HIIRONEN: Smart contracts and Ethereum application platform
Tampere University of Technology
Master of Science Thesis, 55 pages, 1 Appendix page
December 2017
Master's Degree Programme in Information Technology
Major: Data Security
Examiner: Professor Jarmo Harju and M.Sc. Joona Kannisto
Keywords: smart contract, blockchain, best practices, digital currency

Blockchains are decentralized databases which are distributed via peer-to-peer network. Every node in the peer-to-peer network is responsible for keeping the database up-to-date. Blockchains make it possible to build trust between actors via Internet without the need of a third party endorser. Therefore, new digital currencies such as Bitcoin have emerged using blockchain technology. Bitcoin was the first digital currency that used blockchain as a distributed ledger and is still in use.

Ethereum application platform is a distributed system which, compared to Bitcoin, supports smart contracts. In Ethereum they enable programmable digital currency. Smart contracts are independent computer programs and when executed in Ethereum blockchain they function as they were programmed. This thesis aims to describe how the blockchain technology is implemented in Bitcoin and Ethereum. Additionally, an application utilizing smart contracts was made and is used to evaluate what kind of data security risks and coding conventions there are.

The implemented application allows creating employment contracts which are described in smart contracts. The application uses a client-server model in which the server is connected to an Ethereum client. Additionally, the application client can directly communicate with the smart contracts using a web browser extension called MetaMask.

When working with smart contracts it is important to understand the properties and operation of blockchain technology that makes the execution of smart contracts possible. This helps preparing for possible vulnerabilities and exceptions in smart contract programming. The smart contract coding conventions are continuously developing as the technology is relatively new. Furthermore, the Ethereum application platform is getting major updates over the next years, which is why it is recommended to follow the development of the Ethereum and the news about the ecosystem.

ALKUSANAT

Haluan kiittää työni tarkastajia Jarmo Harjua ja Joona Kannistoa, joiden antamien ohjeiden ja palautteiden avulla työ valmistui. Työn aihe ja sisältö hahmottuivat myös heidän kanssaan käytyjen keskustelujen avulla.

Suurimmat kiitokset haluan osoittaa vanhemmilleni, jotka ovat aina tukeneet minua opinnoissani. Haluan kiittää myös ystäviäni ja työkavereitani, jotka tukivat minua diplomityön teossa ja auttoivat sen oikolukemisessa.

Kiitos myös avovaimolleni Kirsille jatkuvasta tuesta aina työn alusta sen loppuun.

Tampereella, 14.11.2017

Altti Hiironen

SISÄLLYS

1. Johdanto 1
2. Lohkoketjuteknologia 4
2.1 Bitcoinin käyttö 5
2.2 Lohkoketjun rakenne 6
2.3 Lohkoketjun toiminta vertaisverkossa 9
2.4 Transaktioiden rakenne 10
2.5 Lohkoketjun konsensus 13
2.6 Lohkoketjun päivittäminen 15
3. Ethereum-sovellusalusta 19
3.1 Historia 19
3.2 Toimintaperiaate 21
3.2.1 Ethereum-tilit ja transaktiot 21
3.2.2 Ether-kryptovaluutta 23
3.2.3 Ethereum-virtuaalikone 24
3.2.4 Ethereum-lohkoketju 25
3.2.5 Älysopimukset 28
3.3 Solidity-ohjelmointikieli 28
4. Älysopimuksia hyödyntävän sovelluksen toteuttaminen 32
4.1 Toteutettava sovellus 32
4.2 Palvelin 34
4.3 Asiakas 38
5. Älysopimusten ohjelmointikäytännöt ja toteutetun sovelluksen arviointi . . 41
5.1 Ohjelmointikäytännöt ja haavoittuvuudet 41
5.2 Huomioita toteutetusta sovelluksesta 45
5.3 Kehitystyökalujen arviointi 46
6. Yhteenveto 48

Lähteet 50

LIITE A. ÄLYTYÖSOPIMUKSEN SOPIMUSKOODI 56

TERMIT

ABI	Application Binary Interface
Bitcoin	lohkoketjuteknologiaa hyödyntävä järjestelmä
Bloom-filtteri	probabilistinen tietorakenne (eng. *bloom filter*)
BTC	Bitcoin-valuutan valuuttayksikkö
Digitaalinen allekirjoitus	bittivirrasta yksityisellä avaimella muodostettu tieto, jolla bittivirran alkuperä voidaan varmentaa
Digitaalinen valuutta	sähköinen raha
Enolohko	lohko, jolla tiedetään olevan sama vanhempi kuin käsiteltävän lohkon vanhemmalla (eng. *uncle block*)
Epäsymmetrinen salaus	salausmenetelmä, jossa hyödynnetään julkista ja yksityistä avainta
ether	Ethereumissa oleva kryptovaluutta ja sen valuuttatunnus
Ethereum	lohkoketjuteknologiaa hyödyntävä järjestelmä
Ethereum-tili	ulkoisesti omistetun tilin ja sopimustilin abstraktio
Ethereum-virtuaalikone	Ethereum-transaktioiden suoritusympäristö (*EVM*)
Fiat-raha	vaihdannan väline, jonka arvo perustuu valtiovallan määrittämiin ominaisuuksiin
Genesis block	lohkoketjun ensimmmäinen lohko

geth	Ethereum-asiakasohjelma
Hajautettu sovellus	älysopimuksista koostuva kokonaisuus (eng. *decentralized application*)
Hard fork	lohkoketjun päivitystapa, joka ei ole yhteensopiva aiempien sääntöjen kanssa
JavaScript	tulkattava ohjelmointikieli
JSON	JavaScript Object Notation -tiedostomuoto
JSON-RPC	JSON-Remote Procedure Call -protokolla
Julkinen avain	salaukseen ja allekirjoitusten varmentamiseen käytettävä avain
Konsensus	tila, jossa kaikki lohkoketjun käyttäjät ovat samaa mieltä siitä, mikä on uusin lohkoketjuun lisätty lohko
Konsensusalgoritmi	tapa, jolla lohkoketjun uusin lohko päätetään
Käyttämätön ulostulo	Bitcoin-transaktioista muodostuva tietorakenne, jota käytetään uusien transaktioiden sisääntulona (eng. *Unspent Transaction Output, UTXO*)
Lohko	otsikkotietueesta ja tarvittavasta datasta muodostuva kokonaisuus (eng. *block*)
Lohkoaika	aika, joka kuluu edellisen lohkon lisäämisestä uuden lohkon lisäämiseen (eng. *block time*)
Lohkoketju	lohkoista muodostuva linkitetty lista (eng. *blockchain*)
Lohkoketjuteknologia	hajautettu vertaisverkossa toimiva tietokanta

Lompakko	asiakassovellus lohkoketjupohjaisen järjestelmän kanssa toimimiseen (eng. *wallet*)
Louhija	lohkoketjun toimija, joka muodostaa uusia lohkoja noudattaen käytössä olevaa konsensusalgoritmia (eng. *miner*)
Louhinta	tapa, jolla uusia lohkoketjun lohkoja muodostetaan (eng. *mining*)
Merkle-Patricia-puu	kryptografisesti todennettava tietorakenne avain–arvo-parien käsittelyyn
Merkle-puu	kryptografisista tiivisteistä koostuva binääripuun kaltainen tietorakenne
Node.js	suoritusalusta JavaScript-ohjelmakoodille
Operaatiokoodi	koodi, jolla ohjataan virtuaalikoneen toimintaa
Otsikkotietue	sisältää lohkolle tärkeää tietoa kuten viittauksen edellisen lohkon otsikkotietueeseen
Pino	abstrakti tietotyyppi
Polttoaine	Ethereum-transaktioihin sisällytettävä tieto, kuinka paljon transaktion suoritukseen voidaan käyttää laskentaoperaatioita (eng. *gas*)
Proof of stake	konsensusalgoritmi, jossa tietyn kokoisella sijoituksella saadaan oikeus valita seuraava lohko
Proof of work	konsensusalgoritmi, jossa tavoitteena on laskea tiivistefunktiota ja tuottaa tarvittava määrä osittaisia törmäyksiä
QR-koodi	kaksiulotteinen kuviokoodi

satoshi	Bitcoin-valuutan pienin yksikkö, yhden bitcoinin miljoonasosan sadasosa
Soft fork	lohkoketjun päivitystapa, joka on yhteensopiva jo ennestään olevien sääntöjen kanssa
Solidity	sopimuspohjainen ohjelmointikieli sopimuskoodin kirjoittamiseen
Sopimuskoodi	sopimustilille asetettava ohjelmakoodi, joka ohjaa sopimustilin toimintaa
Sopimustili	Ethereum-tili, jonka toimintaa ohjaa sen sopimuskoodi
Testrpc	Ethereum-asiakasohjelma, joka simuloi Ethereumin toimintaa
The DAO	hajautettu sovellus, joka noudattaa hajautetun autonomisen organisaation ideaa
Tiiviste	tiivistefunktion tulos, josta ei pysty tehokkaasti päättelemään, mikä on ollut tiivistefunktion sisääntulo
Tiivistefunktio	tuottaa vaihtelevan mittaisesta bittivirrasta kiinteämittaisen tuloksen
Truffle	Ethereum-kehitykseen tarkoitettu sovelluskehys
Ulkoisesti omistettu tili	Ethereum-tili, jonka toimintaa ohjaa käyttäjä yksityisen avaimen kautta
Vaikeusaste	proof of work -algoritmissa tarvittavien törmäysten vaadittu määrä (eng. *difficulty*)
Vue.js	JavaScript-sovelluskehys

Yksityinen avain	salauksen purkamiseen ja allekirjoitusten tekoon käytettävä avain
Älysopimus	itseään suorittava tietokoneohjelma, joka kuvaa sopimuspohjaista toimintaa (eng. *smart contract*)

1. JOHDANTO

Nykyistä yhteiskuntaa ja sosiaalista järjestystä ohjaavat instituutiot, joiden toiminnan edellytyksenä on luottamus. Tapa, jolla ihmiset toimivat keskenään, pohjautuu instituutioiden asettamiin käytäntöihin ja lakeihin. Rahainstituutiot, kuten pankit, toimivat itsenäisinä kokonaisuuksina, joiden toiminta ei ole täysin läpinäkyvää. Pankkien tavoite on tehdä voittoa keräämällä korkoa asiakkaille myönnetyistä luotoista, missä suurimpana riskinä on usein pankin ajautuminen maksukyvyttömyyteen. Riskin toteutuessa vastuu pankilla olleiden varojen pelastamisesta siirtyy muille instituutioille, kuten valtiolle. Yhdessä eri instituutiot ovat niin sanotussa kilpailutilanteessa, jossa jokainen yrittää käsitellä tietoa oman etunsa mukaisesti, esimerkiksi yhteiskunnan järjestyksen tai liiketoiminnan hyödyksi.

Jotta rahan käyttö vaihdon välineenä on mahdollista, rahainstituutiot pitävät kirjaa asiakkaidensa varallisuudesta ja rahaliikenteestä: kuka maksaa ja kenelle, ja onko maksusuoritus oikeellinen. Pankeilla on siis auktoriteetti määritellä asiakkaiden varallisuus. Ongelmana rahainstituutiorakenteessa on, että kaikki eivät ole oikeutettuja omaan pankkitiliin, mikä luo epätasa-arvoisuutta ympäri maailmaa [20]. Pankit toimivat luotettavana kolmantena osapuolena eri toimijoiden välisissä rahansiirroissa, mistä ne veloittavat asiakkaitaan.

Teknologian kehittyessä Internet on luonut mahdollisuuden toteuttaa vaihtoehtoisia digitaalisia valuuttoja nykyisten fiat-valuuttojen, kuten Yhdysvaltain dollarin ja euron, rinnalle. Kaikki nyky-yhteiskunnassa käytettävä raha on fiat-rahaa, jonka arvo pohjautuu valtiovallan toimesta määriteltyihin ominaisuuksiin [55]. Fiat-rahalla ei ole itsessään arvoa, elleivät rahainstituutiot tunnista sitä käyväksi vaihdannan välineeksi. Ongelmana vaihtoehtoisten digitaalisten valuuttojen toteuttamisessa on ollut tilanne, jossa samaa digitaalista rahaa käytettäisiin useammin kuin kerran [58]. Ongelman yksi ratkaisu on ollut tuoda järjestelmään keskitetty toimija, joka on toiminut rahan varmentajana, aivan kuten pankit nykyisessä rahajärjestelmässä. Vuonna 2009 julkaistu Bitcoin ratkaisi ongelman ilman keskitettyjä toimijoita käyttäen lohkoketjuteknologiaa rahaliikenteen varmentamiseen [58]. Bitcoin oli siis ensimmäinen digitaalinen valuutta, joka oli täysin hajautettu ja kaikille käyttäjille vapaa rahajärjestelmä.

Lohkoketjuteknologia mahdollistaa luottamuksen rakentamisen eri toimijoiden välille ilman kolmatta osapuolta, käyttäen kryptografiaa viestien oikeellisuuden määrittämiseen. Teknologian ensimmäinen luonnollinen sovellusalue olikin Bitcoinin toteuttama rahajärjestelmä ilman keskitettyä toimijaa. Bitcoin on myös luonteeltaan fiat-rahaa, mutta sen ominaisuudet perustuvat tietoteknisiin algoritmeihin, joita kaikki Bitcoin-asiakassovellukset noudattavat. Bitcoinissa lohkoketjulla tarkoitetaan yhteisesti jaettavaa tietokantaa, jossa pidetään kirjaa rahasta ja niiden omistajista, ja jota on hyvin vaikea väärentää tai muuttaa jälkikäteen. Näiden ominaisuuksien myötä lohkoketjuteknologia on siis tapa toteuttaa hyvin vikasietoinen hajautettu järjestelmä. Bitcoin on edelleen käytössä oleva järjestelmä, mutta se herätti myös mielenkiinnon toteuttaa muita lohkoketjupohjaisia järjestelmiä. Näistä lupaavin on Ethereum, johon tämä työ keskittyy.

Ethereum on vuonna 2013 esitelty hajautettu järjestelmä, joka vie Bitcoinin ideaa eteenpäin mahdollistamalla täysin ohjelmoitavan digitaalisen valuutan käyttäen älysopimuksia [47]. Älysopimuksella tarkoitetaan sopimuspohjaista toimintaa, joka toimii Internetin välityksellä eri toimijoiden välillä ilman tarvetta kolmansiin osapuoliin, jotka varmentaisivat sopimukseen määriteltyjen ehtojen noudattamista [67]. Ethereum mahdollistaa älysopimusten toteuttamisen käyttäen lohkoketjussa suoritettavaa ohjelmointikieltä, mikä tekee sopimuksista muuttumattomia ja turvallisia. Ethereumin ensimmäinen versio julkaistiin vuonna 2015 [38].

Ethereumin suosio on vuonna 2017 kasvanut jyrkästi ja sen markkina-arvo on yli 4 miljardia euroa [24], mikä tekee siitä Bitcoinin jälkeen toiseksi suosituimman vaihtoehtoisen digitaalisen valuutan. Digitaalisen valuutan lisäksi lohkoketjussa suoritettavien älysopimusten avulla voidaan toteuttaa hajautettuja sovelluksia, joissa hyödykkeiden vaihto on toteutettu hajautetusti ilman keskitettyjä toimijaa. Ethereum toimii hajautettujen sovellusten sovellusalustana, jossa jokainen voi hyödyntää Ethereumin lohkoketjua osana omaa sovellustaan. Ethereumin avulla jokainen voi esimerkiksi toteuttaa oman joukkorahoituskampanjan, jossa säännöt rahoituksen toteutumisesta on määritelty älysopimukseen, ja jonka ehdot ovat kaikkien osapuolten varmennettavissa [36].

Työn tarkoituksena on aluksi selvittää, kuinka lohkoketjuteknologia toimii, käyttäen esimerkkinä Bitcoinia, joka on kyseistä teknologiaa ensimmäistä kertaa hyödyntävä järjestelmä. Tämän pohjalta käydään läpi myös Ethereumin toteutus ja kuinka se eroaa Bitcoinista lohkoketjupohjaisena järjestelmänä. Työssä keskitytään myös Ethereumin mahdollistamiin älysopimuksiin ja niitä hyödyntäviin sovelluksiin, ja selvitetään mitä niillä tarkoitetaan ja millaisia tietoturvariskejä sekä ohjelmointikäytäntöjä niiden toteutuksiin liittyy. Lopuksi työssä toteutetaan älysopimuksia

hyödyntävä sovellus, jonka avulla arvioidaan kuinka älysopimuksia hyödyntävän sovelluksen toteuttaminen onnistuu käyttämällä Ethereum-sovellusalustaa.

2. LOHKOKETJUTEKNOLOGIA

Lohkoketjuteknologia voidaan katsoa alkaneeksi vuonna 2008, kun Satoshi Nakamoto pseudonyymillä esiintynyt ryhmä tai henkilö julkaisi artikkelin ”Bitcoin: A Peer-to-Peer Electronic Cash System” [58]. Artikkelissa kuvataan, kuinka digitaalinen valuutta voidaan toteuttaa ilman keskitettyä tahoa, jonka tehtävänä on normaalisti varmentaa maksujen oikeellisuus. Ratkaisuna Nakamoto esitti kokonaan vertaisverkossa toimivan hajautetun tietokannan, jota pidetään yllä käyttäen erilaisia konsensusalgoritmeja. Neljä kuukautta myöhemmin samaa pseudonyymiä käyttäen julkaistiin Bitcoin-protokollan ensimmäinen versio, mikä aloitti tällä hetkellä vanhimman käytössä olevan lohkoketjun.

Bitcoin -sanalla on useampi merkitys. Se pitää sisällään käsityksen itse valuutasta ja teknologiasta. Se tarkoittaa samalla myös Bitcoin-valuutan yksikköä, jolloin se tässä työssä kirjoitetaan pienellä alkukirjaimella ja jonka valuuttatunnus on BTC. Muita yleisesti käytössä olevia yksiköitä bitcoinin lisäksi ovat bitit ja satoshit [12]. Bitit ovat yhden bitcoinin miljoonasosia ja niiden sadasosat ovat puolestaan satosheja. Satoshit ovat Bitcoinin pienin käytettävä yksikkö.

Lohkoketjujen yksi tärkeimmistä ominaisuuksista on niiden muuttumattomuus. Lohkoketjuun lisätty tieto on siellä pysyvästi ja sitä on hyvin vaikea muuttaa jälkikäteen. Tämä ominaisuus mahdollistaa digitaalisen valuutan olemassaolon [58]. Bitcoinissa lohkoketjulla pidetään yllä julkista kirjanpitoa transaktioista ja tileistä. Kuka tahansa voi tietyllä ajanhetkellä kerätä lohkoketjusta tiedon siitä kenellä on bitcoineja ja kuinka paljon. Transaktiot ovat tietorakenteita, jotka yksinkertaistetusti sisältävät tiedon siirrettävästä valuutasta kahden toimijan kesken [21].

Bitcoinin kehitykselle ei ole tarkkoja prosesseja, vaan oikea ja haluttu toiminnallisuus määräytyy Bitcoin Core -asiakasohjelman toteutuksesta, jonka kehitykseen kaikki voivat osallistua [7, 21]. Tämän vuoksi Bitcoinia ei voi kutsua täydellisesti hajautetuksi järjestelmäksi, koska Bitcoiniin ja sen mahdollistavaan lohkoketjun toteutukseen tulevat muutokset johdetaan hallitusti Bitcoin Core -projektin kautta [21]. Bitcoin Coren avulla saavutetaan yksimielisyys Bitcoinin teknisestä toteutuksesta ja käytössä olevista säännöistä, joilla järjestelmä toimii.

Lohkoketjuteknologia tarjoaa ratkaisun niin sanottuun *double-spending* -ongelmaan [58]. Digitaalisissa valuutoissa haasteena on ollut tunnistaa tilanne, jossa samaa digitaalista valuuttaa käytetään useammin kuin kerran. Ennen Bitcoinia tämä on ratkaistu käyttäen luotettua ja keskitettyä tahoa, jonka tehtävä on ollut varmentaa transaktioiden oikeellisuus. Tässä ratkaisussa ongelmana on Nakamoton [58] mukaan se, että koko rahajärjestelmä on yhden toimijan vastuulla. Lohkoketjuteknologiassa vastuu on kaikilla järjestelmän osallistujilla eikä vain yhdellä keskitetyllä taholla. Tästä syystä Bitcoinissa kaikki transaktiot ovat julkisesti esillä lohkoketjussa [58], mikä tekee lohkoketjusta julkisen kirjanpitokirjan [21].

Bitcoinin sisältämää lohkoketjua voidaan kuvata avoimeksi ja julkiseksi [59]. Julkisella lohkoketjulla tarkoitetaan lohkoketjua, johon kuka tahansa voi ilman lupaa lähettää transaktioita ja lukea niitä [59]. Samalla, jos käyttäjä haluaa, hän voi osallistua transaktioiden ja lohkojen varmentamiseen sekä lohkoketjun konsensuksen ylläpitoon osallistumalla uusien lohkojen louhintaan. Tällöin Bitcoinin lohkoketjua voidaan kuvata myös avoimeksi. Bitcoin on vain yksi lohkoketjuteknologian sovellusalue, minkä vuoksi lohkoketjuja voi olla olemassa myös luvanvaraisina ja yksityisinä [59]. Tällaiset lohkoketjut ovat yleensä yritysten sisäisessä käytössä olevia ja tiettyyn tarkoitukseen rakennettuja järjestelmiä, jotka voidaan integroida osaksi jo valmiita järjestelmiä. Vain yrityksellä on pääsy lohkoketjun sisältämään tietoon ja sen ylläpitoon voidaan valita tiettyjä toimijoita, jolloin konsensuksen ylläpito muuttuu luvanvaraiseksi [59]. Nämä eivät ole ainoita tapoja, joilla lohkoketjuteknologiaa on mahdollista soveltaa, vaan näiden välistä voi löytyä myös esimerkiksi julkisia, mutta luvanvaraisia lohkoketjuja.

2.1 Bitcoinin käyttö

Bitcoinin käytön voi aloittaa ottamalla käyttöön asiakaspääteohjelman, jota kutsutaan lompakoksi (*wallet*). Bitcoin-lompakolla tarkoitetaan bitcoinien käytön mahdollistavaa ohjelmaa, joka yhdistyy Bitcoin-verkkoon ja hallitsee käyttäjän omistamia bitcoineja. Lompakoita on olemassa sekä työpöytä- että mobiilikäyttöön ja niiden ominaisuudet vaihtelevat riippuen käyttäjäryhmästä, jolle ne on suunnattu.

Jokainen lompakko pystyy luomaan käyttäjälle oman Bitcoin-osoitteen, johon bitcoineja voidaan vastaanottaa. Tämä vastaa perinteistä tilinumeroa, mutta Bitcoin-osoitteesta ei voida päätellä kuka sen omistaa ja se koostuu sarjasta numeroita ja kirjaimia. Lompakko muodostaa myös käyttäjälle yksityisen avaimen, jonka avulla käyttäjä voi hallita Bitcoin-osoitteessa olevia bitcoineja. Jos käyttäjä jostain syystä menettää yksityisen avaimen, hän menettää myös osoitteessa olevat bitcoinit. Tällöin osoitteessa olevat bitcoinit jäävät jumiin. Tästä syystä lompakot keskittyvät toiminallisuuksissaan varmistamaan, että käyttäjän avaimet ovat turvassa.

Yksi tapa saada bitcoineja Suomessa on niiden ostaminen. Esimerkiksi Bittiraha.fi tarjoaa palvelun, jossa bitcoineja voi ostaa käyttäen pankkisiirtoa [18]. Maksun jälkeen bitcoinit siirtyvät ostajan määrittelemään Bitcoin-osoitteeseen. Bitcoineja voi ostaa myös käteisellä käyttäen Bittimaattia [17].

Bitcoinien hankinnan jälkeen niiden käyttäminen on mahdollista lompakko-sovelluksella. Ainoat transaktioon tarvittavat tiedot ovat bitcoinien määrä ja kenelle bitcoinit maksetaan eli toisen osapuolen Bitcoin-osoite. Mobiililompakot mahdollistavat myös QR-koodien käytön, minkä ansiosta Bitcoin-osoitteesta muodostettu QR-koodi voidaan helposti lukea maksun saajaksi käyttäen mobiililaitteen kameraa.

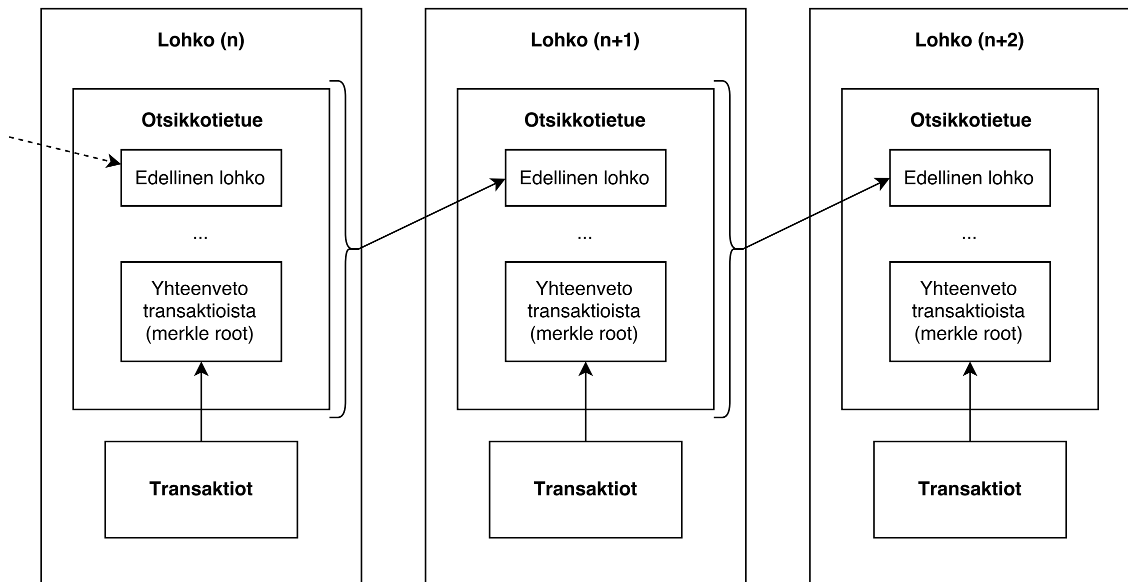
Kun transaktio hyväksytään, lompakko lähettää sen Bitcoin-verkkoon. Bitcoin-verkossa eri toimijat varmentavat transaktion ja lähettävät sitä edelleen muille verkon toimijoille. Maksun vastaanottajan lompakko voi ilmoittaa saapuvasta maksusta lompakon käyttäjälle kuuntelemalla Bitcoin-verkossa välitettäviä viestejä. Maksu näkyy saajalla vahvistamaton-tilassa, ennen kuin se on sisällytetty lohkoketjussa uuteen lohkoon. Kun maksu on vahvistettu eli lisätty lohkoketjussa uuteen lohkoon, maksu on suoritettu, eikä sitä voi enää perua.

2.2 Lohkoketjun rakenne

Lohkoketjujen rakenteelle ei ole olemassa yhtä tarkkaa määritelmää tai standardia, koska teknologia on vielä nuori ja sen kaikki sovellusalueet eivät ole vielä tarkalleen tiedossa. Rakenteeseen vaikuttaa myös tietokannan haluttu käyttötarkoitus. Hyvänä esimerkkinä voidaan kuitenkin käyttää Bitcoinissa käytössä olevaa lohkoketjua ja sen rakennetta, koska se on ensimmäinen ja edelleen käytössä oleva lohkoketju.

Lohkoketju koostuu nimensä mukaisesti lohkoista, jotka on ketjutettu toisiinsa linkitetyn listan tavoin [3]. Lohkon rakenne voidaan erottaa kahteen osaan: otsikkotietueeseen ja transaktioista koostuvaan listaan. Otsikkotietue sisältää lohkolle ja lohkoketjulle tärkeää metadataa, kuten käytössä olevan protokollaversion, viitteen edelliseen lohkoon ja yhteenvedon lohkon sisältämistä transaktioista. Louhinnassa käytettävät muuttujat on myös sisällytetty lohkon otsikkotietueeseen. Kuvassa 2.1 on esitetty lohkoketjun rakenne yleisellä tasolla.

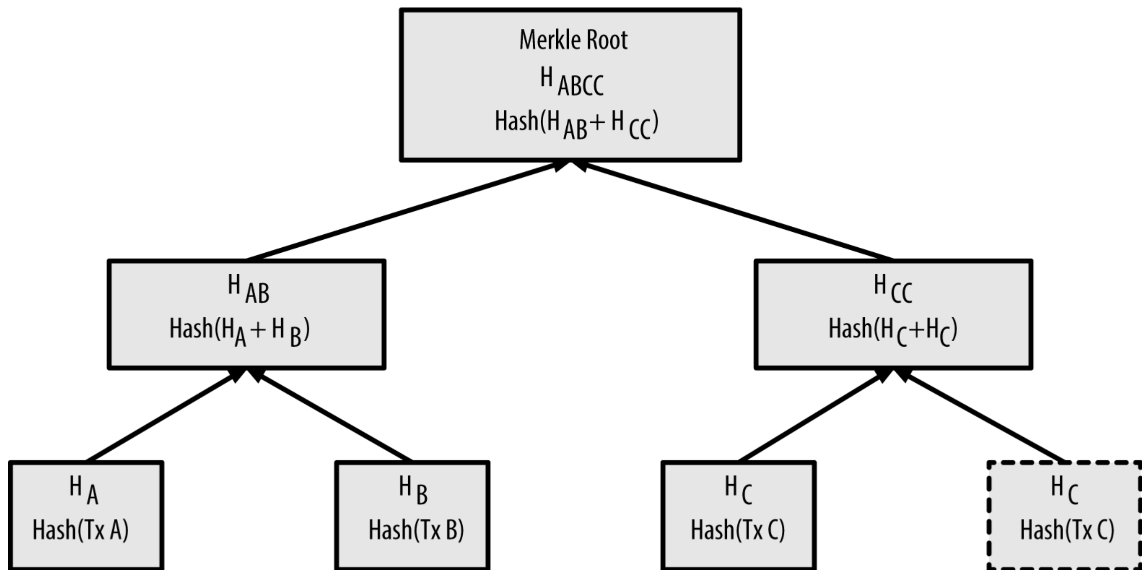
Otsikkotietueen sisältämä viite edelliseen lohkoon on itseasiassa vain edellisen lohkon otsikkotietueesta laskettu digitaalinen sormenjälki. Tästä seuraa, että lohkoilla on aina tietty järjestys ketjussa ja jokainen ketjussa oleva lohko viittaa korkeintaan yhteen vanhempaan lohkoon. Digitaalinen sormenjälki on kryptografisesta tiivistefunktiosta saatava tulos.



Kuva 2.1 *Lohkoketjun yleinen rakenne.*

Kryptografialla on lohkoketjujen toiminnassa tärkeä rooli, jotta tarvittava tiedon luotettavuus ja muuttumattomuus voidaan saavuttaa. Kryptografinen tiivistefunktio on laskennallisesti tehokas funktio, joka tuottaa vaihtelevan mittaisesta bittivirrasta aina kiinteämittaisen tuloksen [56, 53]. Jotta tiivistefunktiota voidaan hyödyntää kryptografisesti, täytyy sen täyttää seuraavat ominaisuudet: 1) Tiivistefunktion h pitää tuottaa sellaisia tuloksia, joista ei pysty tehokkaasti päättelemään, mikä on ollut sisään tuleva bittivirta. Toisin sanoen tiivistefunktiossa $h(x) = y$ tulee olla laskennallisesti vaikeaa päätellä, mikä on ollut sisääntulo x. Funktion pitää olla siis yksisuuntainen. 2) Sen, että tiivistefunktio tuottaa törmäyksiä $h(x') = h(x)$, tulee olla erittäin epätodennäköistä. Törmäyksellä tarkoitetaan tilannetta, jossa kaksi eri sisääntuloa tuottavat täysin saman tiivisteen. Tällöin tuloksesta käytettävä termi ”digitaalinen sormenjälki” ei ole enää mieluisa, koska sormenjälki ei ole uniikki tietylle bittivirralle.

Lohkon identiteetti on siis sen otsikkotietueesta laskettu kryptografinen tiiviste. Lohko ei itse tallenna omaa identiteettiään, vaan jokainen voi halutessaan identifioida lohkon laskemalla tiivisteen itse [3]. Lohkot on kuitenkin linkitetty käyttäen lohkojen identiteettejä. Tämä takaa sen, että lohkojen sisällön pitää pysyä muuttumattomana, jotta lohkoketjun rakenne pysyisi eheänä. Yhdenkin bitin muutos lohkon sisällössä muuttaa sen identiteettiä, rikkoen samalla viitteen uudemmasta lohkosta siihen. Viitteen korjaaminen vastaamaan muuttuneen lohkon uutta identiteettiä ei ole myöskään mielekästä, koska silloin korjauksia pitäisi suorittaa jokaiseen lohkoon, jotka on sen jälkeen lisätty.



Kuva* *2.2 *Merkle-puun rakenne kolmen alkion tietojoukolle [3].*

Lohkoketjun ensimmäinen lohko on erikoistapaus, koska se ei sisällä viittausta muihin lohkoihin [3]. Sille on vakiintunut englanninkielinen termi ”genesis block”, mikä kuvaa hyvin uuden lohkoketjun alkua. Ensimmäinen lohko toteutetaan tietokantaa käyttävään protokollaan staattisesti, jotta kaikilla lohkoketjun käyttäjillä olisi sama käsitys siitä, mistä ketju alkaa. Esimerkiksi Bitcoinin ensimmäisen lohkon rakenne ja metadata on ohjelmoitu Bitcoin-protokollaan [6].

Lohkoihin sisällytetyt transaktiot eivät suoraan vaikuta lohkon identiteettiin. Bitcoinissa transaktioista lasketaan erikseen kryptografinen tiiviste, joka sisällytetään lohkon otsikkotietueeseen [58]. Tiiviste lasketaan käyttämällä Merkle-puuta (*Merkle tree*), joka mahdollistaa nopean ja tehokkaan tavan varmentaa tietyn transaktion olemassaolo tietyssä lohkossa [58, 3].

Merkle-puu on binääripuun kaltainen tietorakenne, joka sisältää kryptografisia tiivisteitä [3]. Binääripuu koostuu solmuista, joilla voi olla kaksi lapsisolmua. Samalla etäisyydellä puun juuresta olevat solmut muodostavat tasoja. Merkle-puun ideana on ensiksi laskea yksitellen tietojoukon jokaisesta alkiosta tiiviste, jotka muodostavat puun ainoat lehtisolmut syvimmälle puun juuresta katsottuna. Seuraavan tason solmut muodostetaan yhdistämällä kahden lehtisolmun sisältämät tiivisteet ja laskemalla tästä uusi tiiviste. Solmujen ja tasojen muodostamista jatketaan, kunnes Merkle-puun juuri on saatu laskettua. Merkle-puun täytyy olla kokonainen, jotta jokainen tietojoukon alkio voidaan sisällyttää siihen. Tämä tarkoittaa esimerkiksi Bitcoinissa sitä, että jos transaktioiden määrä ei ole jaollinen kahdella, kopioidaan viimeinen transaktio täydentämään puu kokonaiseksi [3]. Kuvassa 2.2 on esitetty, kuinka Merkle-puu muodostetaan kolmen alkion tietojoukolle.

Alkuperäisessä Bitcoin-artikkelissa [58] transaktioiden tiivistäminen, käyttäen Merkle-puuta ja sisällyttämällä ainoastaan juuressa oleva tiiviste lohkon otsikkotietueeseen, esitetään tapana pienentää lohkoketjun kokoa. Lohkoketjun kasvaessa vanhojen lohkojen sisältämät transaktiot voidaan jättää huomioimatta ja hakea tarvittaessa Bitcoin-verkosta muilta käyttäjiltä. Lohkossa olevan transaktion varmentaminen onnistuu käymällä Merkle-puuta lävitse vain tarvittavilta osin [3]. Käyttäjä voi laskea kahdesta lehtisolmusta, joista toinen sisältää halutun transaktion tiivisteen, puuta ylöspäin päätyen juuritiivisteeseen. Jos juuressa oleva tiiviste vastaa lohkon otsikkotietueessa olevaa, on transaktion olemassaolo varmennettu. Varmennuksessa ei tarvitse käyttää muita lehtisolmuja, vaan riittää, että tiedetään jokaisella tasolla oleva sisarsolmu. Käytännössä tätä hyödyntävää mekanismia ei artikkelissa esitetä, vaan se toteutettiin Bitcoin-protokollaan myöhemmin, vuonna 2013 [58, 10].

2.3 Lohkoketjun toiminta vertaisverkossa

Lohkoketju on luonteeltaan hajautettu tietokanta. Se toimii vertaisverkossa (*peer-to-peer network*), toisin kuin tavanomaiset keskitetyt tietokannat, jotka pohjautuvat tiedon jakelussa asiakas-palvelin-malliin [58]. Lohkoketjua ei itsessään ole sidottu yhteen tiettyyn palvelimeen, vaan kaikki sen käyttäjät voivat kopioida lohkoketjun itselleen [3]. Julkisen lohkoketjun olemassaolo vaatiikin, että käyttäjät osallistuvat sen ylläpitoon ja jakamiseen. Vastakohtana asiakas-palvelin-mallissa keskitetty tietokanta sijaitsee fyysisesti yhdessä kohteessa, johon kaikki käyttäjät ottavat yhteyttä [4]. Keskitetyn tietokannan ylläpitäjän vastuulla on esimerkiksi käyttäjäoikeuksien hallinta. Lohkoketjussa puolestaan kaikki käyttäjät ovat yhdenvertaisessa asemassa, eikä tietokannan ylläpitoon ole määritelty erillisiä luvanvaraisia rooleja [3].

Vertaisverkossa kaikki verkon osallistujat toimivat palvelimina ja asiakkaina toisilleen. Tämä parantaa verkon suorituskykyä, koska tiedonsiirrossa ei luoteta yhteen tai useampaan keskitettyyn palvelimeen. Julkiset vertaisverkot ovat myös hyvin hajautettuja, koska käyttäjät voivat liittyä verkkoon mistä tahansa käyttäen Internetiä. Näistä syistä on luontevaa, että desentralisaatiota ja kestävyyttä vaativa lohkoketju toimii vertaisverkossa [58]. Avoin vertaisverkko tuo mukanaan myös yhden lohkoketjujen suurimmista haasteista: kuinka tiedon oikeellisuus varmistetaan ja miten oikea tieto määritellään, jos jokainen verkon osallistuja pystyy halutessaan lähettämään haluamaansa tietoa lohkoketjun tilasta?

Voimassa oleva lohkoketju määritellään niin sanotulla pisimmän ketjun säännöllä [3, 58]. Koska lohkoketjua ylläpitävä vertaisverkko voi ulottua maantieteellisesti laajalle alueelle, vaikuttavat viestien etenemisajat siihen, kuinka lohkoketjun tila

näkyy eri käyttäjille. Pisimmän ketjun sääntö määrittää, että oikea ja voimassa oleva lohkoketjun tila on se lohkoketjun versio, jossa on eniten lisättyjä lohkoja. Tällä käsitellään siis tilannetta, jossa louhijat julkaisevat yhtäaikaisesti uuden lohkon verkkoon. Näistä voimaan jäävä lohko on se, jonka päälle seuraava lohko louhitaan.

Lohkoketju toimii käyttäen vertaisverkkoprotokollia, jotka määrittelevät, kuinka eri järjestelmien tulee kommunikoida keskenään [3]. Protokollat määrittyvät lohkoketjun käyttötarkoituksen mukaan ja esimerkiksi Bitcoinissa on käytössä useita vertaisverkkoprotokollia. Tärkein näistä on kuitenkin alkuperäinen koko Bitcoin-verkon toiminnan mahdollistava protokolla, joka toteutettiin ensimmäiseen Bitcoin-asiakaspääteohjelmaan. Protokollalle ei ole tarkkaa määritelmää, vaan se johdetaan kyseisen asiakaspääteohjelman toteutuksesta [15]. Muut Bitcoinissa käytössä olevat protokollat laajentavat itse Bitcoin-verkkoa lisätoiminnallisuuksilla, joita ei ole toteutettu alkuperäiseen protokollaan.

Jokainen lohkoketjun käyttäjä osallistuu vertaisverkon ylläpitoon. Tämän vuoksi käyttäjiä voidaan kutsua lohkoketjun solmuiksi. Jokainen solmu osallistuu lohkoketjussa olevan tiedon välittämiseen sekä uusien transaktioiden ja lohkojen varmentamiseen [3]. Solmut vastaanottavat uusia transaktioita ja varmentavat niiden oikeellisuuden ennen kuin transaktiota välitetään muille verkon solmuille. Transaktion varmentaminen tehdään jokaisessa solmussa uudestaan, jotta voidaan varmistaa, ettei verkko kuormitu haitallisesta liikenteestä. Tiedon välittämisen lisäksi solmut voivat ylläpitää täydellistä kopiota lohkoketjun tilasta, mikä on välttämätöntä, jotta uusien transaktioiden varmentaminen on mahdollista täysin riippumatta muista verkon solmuista [3]. Solmut voivat myös osallistua uusien lohkojen muodostamiseen, jota kutsutaan lohkojen louhinnaksi. Louhintaan tarvitaan myös täydellinen kopio lohkoketjusta.

2.4 Transaktioiden rakenne

Bitcoinissa lohkoketjuun sisällytettävät transaktiot ovat tietorakenteita, joiden avulla voimassa oleva Bitcoin-verkon tila määräytyy. Transaktioita lukemalla voidaan päätellä missä Bitcoin-osoitteissa bitcoineja on ja kuinka paljon, ja ne mahdollistavat bitcoinien siirron eri toimijoiden välillä. Tietorakenteena transaktio muodostuu sisään- ja ulostuloista, joita voi olla yksi tai useampi [21]. Sisääntulot kuvaavat yleensä transaktion muodostavan tahon omistamia bitcoineja ja ulostulot transaktion jälkeistä tilaa, eli kenelle bitcoineja on siirretty.

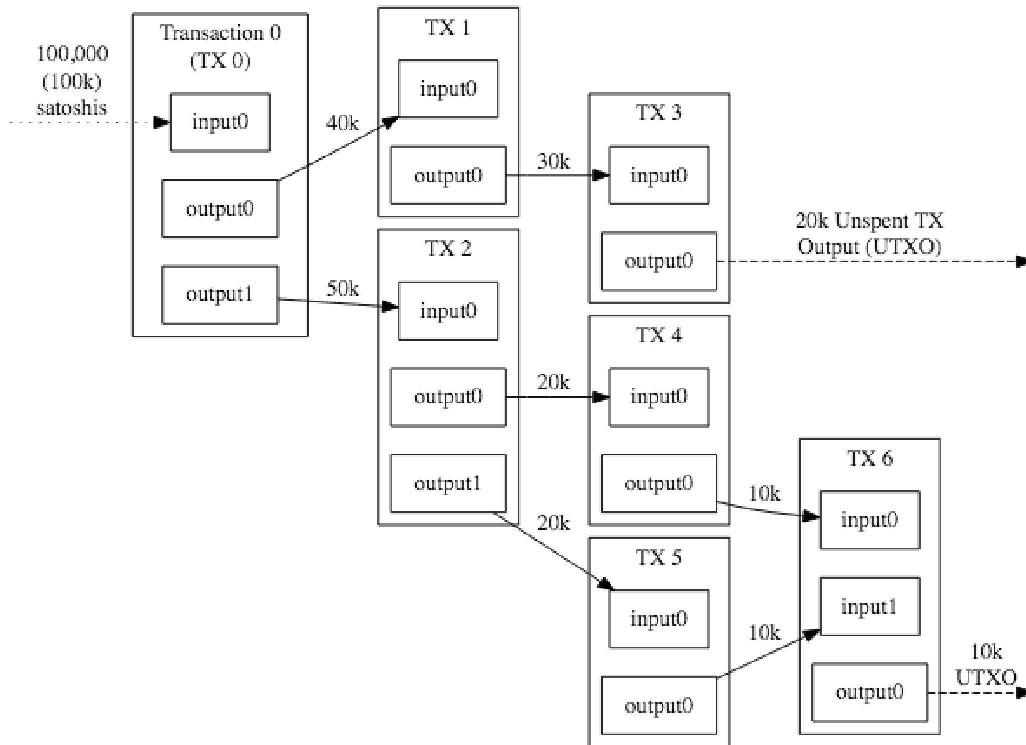
Bitcoin-valuutan omistajuus tietylle käyttäjälle mahdollistetaan hyödyntäen epäsymmetristä salausta [3]. Siinä salaus toteutetaan muodostamalla kaksi avainta:

julkinen ja yksityinen avain [56]. Julkista avainta käyttämällä viesti voidaan salata niin, että salauksen saa purettua vain käyttämällä siihen liittyvää yksityistä avainta. Viesti voidaan myös allekirjoittaa käyttäen yksityistä avainta, jolloin kaikki vastaavan julkisen avaimen haltijat voivat varmistaa viestin alkuperän käyttämällä julkista avainta. Näiden ominaisuuksien vuoksi epäsymmetristä salausta kutsutaan myös julkisen avaimen menetelmäksi. Salauksen turvallisuus perustuu siihen, että yksityistä avainta ei ole mahdollista laskennallisesti muodostaa julkisesta avaimesta. Bitcoinissa transaktioiden sisältöä ei ole salattu, vaan paino on epäsymmetrisen salauksen mahdollistavassa allekirjoituksessa [3].

Bitcoinien käyttöön tarvittava lompakko-sovellus on vastuussa avaimien hallinnasta [3]. Lompakko muodostaa yksityisen avaimen k, joka on 256-bittinen satunnaisesti muodostettu numerosarja. Yksityistä avainta ei ole tarkoitus jakaa kenellekään, vaan siitä johdetaan julkinen avain K käyttäen hyväksi elliptistä käyrää [3]. Elliptiset käyrät ovat yksi tapa muodostaa yksisuuntaisia funktioita, joista on hyvin vaikea päätellä, mikä on ollut funktion sisääntulona käytetty yksityinen avain k, vaikka funktion tulos, julkinen avain K, on tiedossa. Lompakon muodostama Bitcoin-osoite on lopulta julkisesta avaimesta K laskettu tiivistefunktio, joka on muunnettu lyhempään ja helpommin luettavaan muotoon.

Lohkoketjun ylläpitäjät pitävät kirjaa transaktioista ja sisällyttävät niitä uusiin lohkoihin osaksi lohkoketjua. Transaktioissa olevia ulostuloja, joita voi käyttää uusien transaktioiden sisääntuloissa, kutsutaan käyttämättömiksi ulostuloiksi (*Unspent Transaction Output, UTXO*) [3, 21]. Käyttämättömät ulostulot sisältävät tiedot käytettävissä olevasta Bitcoin-valuutasta käyttäen satosheja, jotka ovat bitcoinin pienin käytettävissä oleva yksikkö. Kuvassa 2.3 on esitetty, kuinka transaktiot muodostuvat käyttämättömistä ulostuloista. Bitcoin-verkon osallistujat pitävät kirjaa käyttämättömistä ulostuloista, ja esimerkiksi lompakko-sovellus muodostaa käyttäjän omistamien bitcoinien määrän laskemalla käyttäjälle kuuluvat käyttämättömät ulostulot yhteen. Jotta ulostuloja ei voi kuka tahansa käyttää, sisältävät ne satoshien lisäksi kentän, johon voidaan määritellä millä ehdoilla kyseisen ulostulon bitcoineja voi käyttää. Tätä kenttää voidaan kutsua kryptografiseksi arvoitukseksi, koska se muodostetaan käyttäen Bitcoiniin sisäänrakennettua skriptikieltä nimeltään *Script* [3].

Script on yksinkertainen ja tehokas skriptikieli, joka on suunniteltu mahdollistamaan Bitcoin-transaktioiden validointi [3, 21]. Se suoritetaan vasemmalta oikealle käyttäen hyväksi pinoa, mikä estää silmukoiden ja monimutkaisten suoritusrakenteiden suorituksen. Tämä on tarkoituksellista, koska se vähentää huomattavasti mahdollisuutta väärinkäytöksiin, mikä voisi vaikuttaa haitallisesti Bitcoin-verkon suorituskykyyn.



Kuva 2.3 *Käyttämättömien ulostulojen hyödyntäminen uusien transaktioiden sisääntuloissa [9].*

Yleisin käyttämättömiin ulostuloihin sisällytettävä kryptografinen arvoitus on niin sanottu *Pay-to-PubkeyHash* (*P2PKH*) [21]. Nimensä mukaisesti ulostuloon sisällytetään skripti, joka sisältää tiedon siitä, kenelle maksu suoritetaan. Kryptografinen arvoitus sisältää tässä tapauksessa sen käyttäjän Bitcoin-osoitteen, jolla on oikeus käyttää transaktion ulostuloon sisällytettyjä bitcoineja edelleen.

Samalla mekanismilla, kun käyttämättömiä ulostuloja halutaan käyttää uuden transaktion sisääntuloina, koostuvat sisääntulot viitteestä ulostuloon, sekä skriptistä, joka ratkaisee ulostulossa olevan kryptografisen arvoituksen [21]. *P2PKH*:n tapauksessa oikea ratkaisu on sisällyttää skriptiin digitaalinen allekirjoitus ja käyttäjän tiivistämätön julkinen avain. Digitaalinen allekirjoitus lasketaan transaktiosta tehdystä muokatusta kopiosta [16, 13, 11]. Kopiossa olevan vastaavan sisääntulon skripti korvataan siinä viitatun käyttämättömän ulostulon kryptografisella arvoituksella. Jos transaktiossa on useampia sisääntuloja, niiden skriptit jätetään tyhjiksi. Tästä kopiosta laskettu tiiviste allekirjoitetaan käyttäjän yksityisellä avaimella, josta muodostuu alkuperäisen transaktion sisääntulon skriptiin lisättävä digitaalinen allekirjoitus. Allekirjoitus muodostetaan käyttäen muokattua kopiota alkuperäisestä transaktiosta, koska se mahdollistaa transaktion sisällön varmentamisen. Jos allekirjoitus muodostettaisiin käyttäen alkuperäistä transaktiota, allekirjoitus ei enää

vastaisi transaktiota, koska myös se lisättäisiin osaksi transaktiota. Tällöin transaktiosta muodostettu tiiviste tuottaisi eri tuloksen kuin ennen.

Kun transaktio lähetetään Bitcoin-verkkoon, se varmennetaan tarkistamalla vastaako ratkaisussa annettu julkinen avain tiivistefunktion jälkeen käyttämättömässä ulostulossa olevaa Bitcoin-osoitetta [3]. Tämän lisäksi ratkaisussa annettu digitaalinen allekirjoitus takaa, että transaktiossa käytettävien bitcoinien oikea omistaja valtuuttaa niiden käytön, koska digitaalinen allekirjoitus on muodostettu käyttäen käyttäjän yksityistä avainta. Bitcoin-verkko voi myös todentaa, ettei transaktiossa oleva tieto ole muuttunut sen muodostamisen jälkeen, käyttäen ratkaisussa annettua allekirjoitusta ja julkista avainta [3].

Muita Bitcoinissa käytössä olevia yleisiä skriptejä ovat *P2PKH*:n lisäksi *multi-signature* ja *Pay-to-ScriptHash (P2SH)* [3]. *Multi signature* -skripti mahdollistaa nimensä mukaisesti käyttämättömien ulostulojen lukitsemisen useilla eri julkisilla avaimilla [21]. Skriptiin määritellään, kuinka moneen julkiseen avaimeen tarvitsee antaa vastaava digitaalinen allekirjoitus, jotta kyseisen ulostulon voi käyttää transaktiossa. Tällä saadaan luotua lisäturvaa esimerkiksi siten, että transaktion allekirjoittamiseen tarvitaan fyysisesti kaksi eri laitetta, joiden omat lompakko-sovellukset ovat vastuussa eri avaimista. *P2SH* puolestaan helpottaa monimutkaisten skriptien käyttöä transaktioissa siten, että maksu osoitetaan halutusta skriptistä laskettuun tiivisteeseen [14]. Tällöin näitä ulostuloja käytettäessä kryptografisen arvoituksen ratkaisu on määritellä kyseessä ollut skripti ja mahdollinen data, jolla ratkaisu saadaan. Esimerkiksi usean allekirjoituksen vaativa ulostulo voi sisältää vain tiivisteen *multi-signature* -skriptistä, joka määrittelee, mikä on tarvittava määrä allekirjoituksia ja mitkä ovat vastaavat julkiset avaimet. *P2SH* salaa siis transaktion ulostulosta tiedon, kuinka kyseisiä bitcoineja voi käyttää.

2.5 Lohkoketjun konsensus

Lohkoketjun konsensuksella tarkoitetaan tilaa, jossa kaikki lohkoketjun toimijat ovat samaa mieltä siitä, mikä on uusin lohkoketjuun lisätty lohko [21]. Julkisessa lohkoketjussa, kuten Bitcoinissa, toimijoiden välillä ei ole luottamusta, joten konsensus täytyy saavuttaa säännöillä, joita kaikkien tulisi noudattaa. Yleensä sääntöjen noudattamisesta voi seurata palkinto, mikä toimii kannustimena ylläpitää lohkoketjun oikeaa tilaa. Bitcoinissa uusien lohkojen muodostamista kutsutaan louhinnaksi [21]. Louhijat ovat Bitcoin-verkon käyttäjiä, jotka keräävät, varmentavat ja uudelleenlähettävät verkkoon lähetettyjä transaktioita sekä osallistuvat uusien lohkojen muodostamiseen.

Bitcoinissa lohkotason konsensus saavutetaan käyttäen *proof of work* -algoritmia [58]. Algoritmin ideana on, että louhijoiden täytyy tuottaa jotain vaikeasti saatavaa dataa, mutta joka on samalla helposti varmistettavissa muiden toimesta. Bitcoinissa louhijat laskevat kryptografista tiivistettä uuden lohkon otsikkotietueesta käyttäen SHA256-funktiota [3, 21]. Tiivisteen on kuitenkin tuotettava tietty määrä osittaisia törmäyksiä, joiden määrä määräytyy dynaamisesti Bitcoin-verkossa niin, että törmäyksien määrän täyttävä tiiviste löytyy 10 minuutin välein. Toisin sanoen lohkoketjuun lisätään uusi lohko noin 10 minuutin välein. Tarvittavien törmäysten määrää kutsutaan vaikeusasteeksi (*difficulty*) [21]. Koska tiivistefunktiot tuottavat aina täysin eri kiinteämittaisen tuloksen yhdenkin bitin muuttuessa laskettavassa bittivirrassa, ei tulevien törmäysten määrää voida ennustaa. Tämän vuoksi louhijoiden todennäköisyys löytää seuraava lohko on verrannollinen heidän hallitsemaansa laskentatehoon Bitcoin-verkossa, mutta on myös mahdollista, että törmäykset täyttävä tiiviste voi löytyä huomattavan lyhyessä ajassa [21]. Jos *proof of work* -algoritmi ei sisältäisi tiivistefunktioiden tuomaa satunnaisuutta, olisi odotettavaa, että louhija, joka hallitsisi suurinta laskentatehoa, löytäisi uusimman lohkon aina ensimmäisenä [21].

Proof of work -algoritmia varten otsikkotietueessa on tietokenttä laskurille, jota kasvattamalla tietueesta laskettavaa tiivistettä saadaan helposti muutettua [3]. Tarvittavien osittaisten törmäysten määrä on myös määritelty otsikkotietueessa [3], jonka arvo lasketaan aina uudelleen noin kahden viikon välein vastaamaan määriteltyä 10 minuutin lohkoaikaa [21]. Osittaiset törmäykset on yksinkertaisesti määritelty tiivisteeseen muodostuvien ensimmäisten nollien lukumääränä [21]. Esimerkiksi otsikkotietueesta lasketun SHA256-tiivisteen täytyy alkaa n määrällä nollabittejä, jotta louhija saavuttaa onnistuneesti *proof of work* -algoritmin mukaisen todistuksen. Tämän jälkeen louhija lähettää löytämänsä lohkon Bitcoin-verkkoon, jossa muut louhijat varmistavat sen oikeudellisuuden, tarkistamalla lohkon sisältämät transaktiot ja varmentamalla, että lohkon otsikkotietueesta laskettu tiiviste vastaa sillä hetkellä määriteltyä vaikeusastetta [21]. Jos uusi lohko on hyväksyttävä, louhija ottaa sen osaksi lohkoketjuaan, lähettää sen Bitcoin-verkossa eteenpäin ja aloittaa uuden lohkon muodostamisen tämän päälle. Jos kaksi tai useampi louhijaa löytävät uuden lohkon samanaikaisesti ja lähettävät ne verkkoon, pitävät louhijat näistä kaikkia lohkoketjun versioita muistissa [3]. Voimaan jäävä lohkoketju muodostuu pisimmän ketjun säännöllä kuitenkin niin, että pisimmässä ketjussa lohkojen yhteenlaskettu vaikeusaste on suurempi kuin muiden ketjujen [21]. Tällä varmistetaan, että ajan myötä koko Bitcoin-verkko päätyy samaan lohkoketjun tilaan [3].

Jotta lohkoketjun konsensus saavutetaan, tarvitaan *proof of work* -algoritmin laskemiseen oikeita resursseja, joista merkittävin on sähkö [21]. Algoritmissa louhijoiden

vastuulla on kerätä tarpeeksi laskentaresursseja, joiden avulla he pystyvät onnistuneesti löytämään uuden lohkon. Vaihtoehtoinen tapa saavuttaa lohkoketjun konsensus on ottaa käyttöön virtuaalinen louhinta, jossa lohkon löytäminen vaatisi ainoastaan tietyn varallisuuden vaihtamisen kykyyn valita seuraava lohkoketjun lohko [21]. Tällöin *proof of work* korvattaisiin eräänlaisella riskinotolla (*proof of stake*), jossa tietyn kokoisella sijoituksella saatu oikeus valita seuraava lohko määräisi voimassa olevan lohkoketjun tilan [3]. Sijoituksen aikana kyseistä varallisuutta ei voisi käyttää, vaan se keräisi korkoa ja palautuisi tietyn ajan kuluttua takaisin sijoittajalleen.

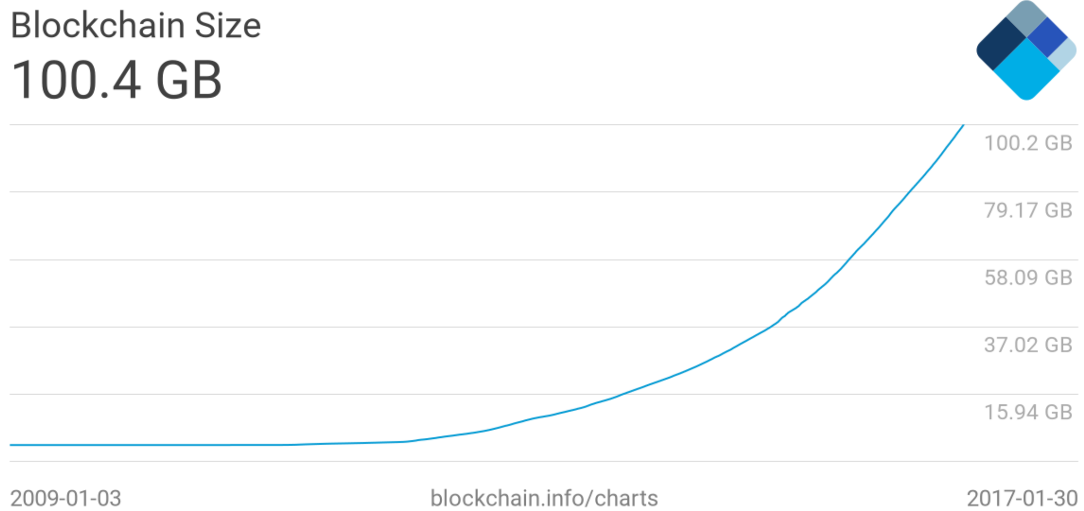
Bitcoinissa louhijat kilpailevat keskenään löytääkseen uuden lohkon ensimmäisenä. Uuden lohkon louhija saa palkkioksi tehdä erityisen transaktion, jossa louhija siirtää täysin uusia luotuja bitcoineja omalle tililleen [21]. Tämä palkinto toimii kannustimena louhijoille, jotta lohkoketjun tila pysyy oikeana, eikä sisällä haitallista tai väärää tietoa. Jos louhijan lähettämä lohko ei noudata sääntöjä, joita suurin osa lohkoketjussa noudattaa, se ei tule osaksi voimassa olevaan lohkoketjua eikä louhija saa palkintoaan [21]. Tällöin on louhijan kannalta sääntöjen noudattaminen hyödyllisempää kuin niiden rikkominen. Louhinta on ainoa tapa, jolla uusia bitcoineja muodostetaan Bitcoin-verkkoon [3, 21]. Lohkopalkinnon sisältämien bitcoinien määrä puolittuu joka neljäs vuosi, kunnes bitcoineja on luotu yhteensä 21 miljoonaa vuoteen 2140 mennessä. Tämän jälkeen uusia bitcoineja ei enää luoda.

Lohkopalkinto ei ole ainoa tulonlähde louhijoille, vaan he saavat kerätä lohkoon sisällytetyistä transaktioista niin sanottua palvelumaksua [21]. Palvelumaksun määrä transaktiolle määräytyy transaktioiden ulos- ja sisääntulojen erotuksena [21]. Palvelumaksu on havainnollistettu kuvassa 2.3, jossa jokaisen transaktion palvelumaksu on 10 000 satoshia. Louhijat saavat sisällyttää nämä erotukset lohkopalkinnon sisältävään transaktioon, jolloin ne siirtyvät louhijan käyttöön [51]. Louhijat saavat myös itse päättää mitkä transaktiot he lohkoon sisällyttävät, esimerkiksi priorisoimalla palvelumaksua sisältävät transaktiot lohkoihin [3]. Tästä voi seurata tilanne, että ilman palvelumaksua olevat transaktiot eivät koskaan tule vahvistetuiksi, eli osaksi voimassa olevaa lohkoketjua.

2.6 Lohkoketjun päivittäminen

Bitcoinin toiminta perustuu lohkoketjuun, johon kaikki transaktiot tallennetaan. Tästä syystä lohkoketjun koko on Bitcoinin käytön lisääntyessä kasvanut pisteeseen, jossa lohkoketjun ylläpito vaatii kirjoitushetkellä[1] yli 100 gigatavua levytilaa [19]. Kuvassa 2.4 on esitetty lohkoketjun koon kasvu gigatavuina Bitcoinin alusta

[1] 31.1.2017



Kuva 2.4 *Bitcoin-lohkoketjun koko [19].*

vuoden 2017 alkuun. Lohkoketjun suuren koon lisäksi Bitcoinin skaalautuvuus on tuonut uusia haasteita sen käyttöön. Skaalautuvuudella tarkoitetaan Bitcoin-verkon kykyä palvella yhä suurempaa käyttäjäkuntaa ilman, että sen toiminta huomattavasti heikkenisi [51]. Bitcoin-verkon suorituskyvyn transaktioiden käsittelyssä on mitattu olevan noin seitsemän transaktiota sekunnissa, mikä on huomattavasti vähemmän kuin esimerkiksi Visa-verkon, jonka on ilmoitettu pystyvän käsittelemään normaalitilanteessa kaksituhatta transaktiota sekunnissa [51]. Jotta Bitcoinin skaalautuvuusongelma voidaan ratkaista, täytyy tarvittavat muutokset saada voimaan niin, että Bitcoinissa oleva hajautettu konsensus säilyy eheänä. Tämän lisäksi myös mahdolliset tietoturvallisuuteen liittyvät kysymykset täytyy ottaa huomioon.

Lohkoketjupohjaisten järjestelmien päivitystavat voidaan jaotella kahteen eri kategoriaan [51, 21]: päivityksiin, jotka ovat yhteensopivia jo ennestään olevien sääntöjen kanssa (*soft fork*) ja päivityksiin, jotka eivät ole yhteensopivia ennestään olevien sääntöjen kanssa (*hard fork*). *Soft fork* -päivityksen jälkeen lohkoketjussa olevia sääntöjä tiukennetaan niin, että uusien sääntöjen mukaisesti muodostetut lohkot ovat hyväksyttäviä myös niiden toimijoiden mukaan, jotka eivät ole päivitystä tehneet. Tämä on yleisesti turvallisin tapa muokata lohkoketjun toimintaa, koska se pakottaa ajan myötä kaikki päivittämään toimintansa vastaamaan uusia sääntöjä [21]. Muuten vanhoilla säännöillä muodostettuja lohkoja ei lopulta enää hyväksytä voimassa olevaan lohkoketjuun, jos lohkoketjussa suurempi osa laskentatehosta noudattaa uusia sääntöjä. Tällöin vanhoja sääntöjä noudattanut louhija on käyttänyt laskentatehoaan turhaan, samalla menettäen lohkopalkkionsa [21]. Vastakohtaisesti *hard fork* -päivityksen tuomilla säännöillä louhitut lohkot eivät päädy osaksi voi-

massa olevaa lohkoketjua, ellei suurin osa lohkoketjussa olevasta laskentatehosta ole jo valmiiksi päivittänyt omia sääntöjään.

Suurin Bitcoinin skaalautuvuutta rajoittava tekijä on kiinteäksi määritelty lohkojen enimmäiskoko, joka on asetettu yhteen megatavuun [51]. Tästä johtuen Bitcoin-verkko ei pysty varmentamaan kuin tietyn määrän transaktioita kymmenen minuutin välein, jolloin louhijat pystyvät priorisoimaan yhä suuremmilla palvelumaksuilla lähetettyjä transaktioita uusiin lohkoihin. Skaalautuvuutta voitaisiin parantaa muuttamalla lohkojen enimmäiskokoa, mutta se vaatisi lohkoketjuun *hard fork* -päivityksen, mikä ei olisi Bitcoin-yhteisön mukaan toivottava ratkaisu [51]. Sen sijaan *soft fork* -päivitys nimeltään *segregated witness* on sisällytetty Bitcoin Core -toteutukseen [8].

Segregated witness muuttaa Bitcoinin toimintaa niin, että koko transaktiosta pystytään laskemaan digitaalinen allekirjoitus heti sitä muodostettaessa [51]. Tämän muutoksen jälkeen transaktio pystytään identifioimaan jo ennen kuin se on lisätty lohkoketjuun. Tähän asti transaktioiden sisältämät allekirjoitukset ovat kattaneet vain osia transaktion sisällöstä, jolloin koko transaktion identifioiva tiiviste on ollut lopullinen vasta kun se on sisällytetty lohkoketjuun. Tämä on mahdollistanut Bitcoin-verkkoon lähetetyn transaktion muokkaamisen ylimääräisellä datalla niin, että alkuperäisen tekijän laskema tiiviste ei ole enää vastannut kyseistä transaktiota lohkoketjussa [51]. Transaktioiden muokattavuus on ollut esteenä niin sanotuille lohkoketjun ohittaville maksukanaville (*off-chain payment channel*) [51]. *Segregated witness* parantaa Bitcoinin skaalautuvuutta mahdollistamalla maksukanavat toimijoiden välillä niin, että keskinäisiä transaktioita ei tarvitsisi sisällyttää lohkoketjuun. Lohkoketjuun sisällytettäisiin ainoastaan maksukanavan identifioiva transaktio ja transaktio, joka lopuksi sulkisi maksukanavan. Muutoksen on arvioitu parantavan Bitcoinin suorituskykyä nelinkertaisesti [51]. *Segregated witness* aktivoituu vasta, kun kahden viikon aikaikkunassa 95 prosentista louhituista lohkoista löytyy tieto, että louhija on valmis siirtymään uusiin sääntöihin [8]. Jos 95 prosentin kynnys ei täyty, aloitetaan kahden viikon ajanjakso uudestaan. Louhija pystyy lohkopalkkion sisältämässä transaktiossa kertomaan kantansa käyttämällä siinä olevaa skriptikenttää, koska se on lohkopalkkion sisältämässä sisääntulossa tyhjä [3].

Bitcoin-yhteisössä eriävät mielipiteet skaalautuvuuden parantamiseen ovat aiheuttaneet useiden eri Bitcoin-asiakasohjelmien synnyn, joista alkuperäinen Bitcoin Core on kuitenkin edelleen suosituin [2]. Lohkon maksimikoon kasvattamista kannattavat ovat toteuttaneet Bitcoin Unlimited -asiakasohjelman, joka laajentaa Bitcoin Core -toteutusta muuttamalla lohkon maksimikoon määrittelyn vastaamaan yleistä konsensusta [1]. Se ei suoraan kasvata lohkon maksimikokoa vaan mahdollistaa

lohkoketjun toimijoiden välisen kommunikaation, jonka avulla mahdollinen muutos suoritetaan dynaamisesti säilyttäen lohkoketjun konsensus. *Segregated witnessin* sisältämä Bitcoin Core ja Bitcoin Unlimited kilpailevat keskenään siitä, kumpi toteutus määrittää Bitcoinin konsensuksen ja niin sanotun oikean lohkoketjun. Kirjoitushetkellä[2] Bitcoin-verkosta noin 86 prosenttia käyttää alkuperäistä Core-toteutusta ja noin 8 prosenttia Unlimited-toteutusta [2].

[2] 31.1.2017

3. ETHEREUM-SOVELLUSALUSTA

Ethereum on lohkoketjupohjainen järjestelmä, joka mahdollistaa hajautettujen sovellusten toteuttamisen ja suorittamisen. Hajautetut sovellukset koostuvat älysopimuksista, joiden suoritus tapahtuu Ethereum-virtuaalikoneessa ja joiden suorituksesta maksetaan käyttäen ether-kryptovaluuttaa. Ethereum toimii sovellusalustana hajautetuille sovelluksille, joiden suoritus tapahtuu luotettavasti lohkoketjussa.

3.1 Historia

Ethereum syntyi tarpeesta toteuttaa lohkoketjupohjainen järjestelmä, joka olisi abstraktiotasolla korkeammalla kuin tiettyyn käyttötarkoitukseen kehitetyt lohkoketjujärjestelmät [69]. Korkea abstraktiotaso mahdollistaa useiden erilaisten sovellusten toteuttamisen samaan lohkoketjujärjestelmään ilman, että alla olevaa teknologiaa täytyisi muuttaa vastaamaan sovelluksen tarpeita.

Ethereum-projektin virallinen aloitus tapahtui Yhdysvalloissa järjestetyssä Bitcoin-konferenssissa tammikuussa 2014, jossa Ethereumin keksijä Vitalik Buterin esitteli teknisen ratkaisunsa, kuinka lohkoketjupohjainen järjestelmä toimisi alustana hajautetuille sovelluksille [38, 32]. Myöhemmin huhtikuussa Ethereum-projektin toinen perustajajäsen Gavin Wood julkaisi spesifikaation *Ethereum Yellow Paper*, jossa kuvataan tarkasti [69], kuinka järjestelmä tulee toteuttaa, ja se toimiikin teknisenä erittelynä Ethereum-asiakaspääteohjelmille [38].

Projektin rahoitus toteutettiin perustamalla Ethereum-säätiö [38], jonka tehtävä on tukea ja edistää avoimien ja hajautettujen ohjelmistoarkkitehtuurien kehitystä [39]. Vuoden 2014 kesällä säätiö aloitti Ethereumissa olevan kryptovaluutan Etherin ennakkomyynnin, mikä keräsi säätiölle bitcoineja silloisen kurssin mukaisesti yli 18 miljoonan dollarin edestä. Se mahdollisti projektin olemassaolon ja kehityksen jatkamisen [38].

Alustava kehitys jatkui aina vuoden 2015 kesään, jolloin ensimmäisen version *Frontierin* julkaiseminen yleiseen käyttöön aloitti Ethereumissa olevan julkisen lohkoketjun toiminnan [38]. Julkaisu mahdollisti myös louhinnan aloittamisen Ethereum-

verkossa. *Frontier* oli kehittäjille tarkoitettu Ethereumin beta-versio, jota ei oltu tarkoitettu vielä hajautettujen sovelluksien tuotantokäyttöön. Ennen *Frontierin* julkaisua Ethereumia oli testattu ja auditoitu niin kehittäjien kuin kolmansien osapuolten puolesta, mikä paljasti toteutuksessa olevia ongelmia ja tietoturvallisuusriskejä [38].

Maaliskuussa 2016 julkaistiin tuotantokäyttöön tarkoitettu ja kirjoitushetkellä[1] voimassa oleva versio *Homestead*, joka sisälsi muutoksia, jotka eivät olleet yhteensopivia edellisen julkaisun kanssa [38]. *Homestead* oli *hard fork* -päivitys, mikä edellytti kaikkien asiakaspääteohjelmien päivittämistä yhteensopivaan versioon uusien muutosten kanssa. *Homestead* ei kuitenkaan ollut ainoa vuoden 2016 aikana tehty *hard fork* -päivitys Ethereum-verkkoon.

Heinäkuussa 2016 Ethereumiin tehty *hard fork* toimi pelastussuunnitelmana The DAO -nimiseen hajautettuun sovellukseen kohdistuneessa hakkeroinnissa [23]. The DAO noudattaa hajautetun autonomisen organisaation ideaa, jossa tavoitteena on muodostaa sovellus, joka vastaa toiminallisuudeltaan perinteisiä organisaatioita, kuitenkin niin, että ihmisten toimintaa ohjaa sovellus eivätkä itse ihmiset [54]. Ongelmaksi osoittautui The DAO:n koodissa oleva virhe, jota hyödyntäen huhtikuussa joukkorahoituksella kerättyjä ethereitä alettiin siirtää The DAO:lta hyökkääjän perustamaan autonomiseen hajautettuun organisaatioon, joka oli rakenteeltaan samanlainen kuin The DAO [54]. Hyökkääjän saamien ethereiden arvo oli silloin noin 60 miljoonaa dollaria [31]. Ethereumiin tehty *hard fork* muokkasi lohkoketjun transaktiohistoriaa niin, että hyökkääjän hallussa olleet etherit siirrettiin älysopimukseen, jonka kautta niiden alkuperäiset omistajat saivat lunastettua ne takaisin itselleen [23]. *Hard fork* suoritettiin Ethereum-säätiön ja Ethereum-käyttäjäyhteisön enemmistön tukemana. Lohkoketjun transaktiohistorian muokkaaminen ei kuitenkaan ollut kaikkien mielestä oikea tapa toimia, mistä syntyi Ethereum Classic -versio, joka jatkoi Ethereum-lohkoketjua ilman transaktiohistoriaa muokkavaa *hard fork* -päivitystä [54].

Syksyllä 2016 Ethereum-verkkoon kohdistui myös palvelunestohyökkäyksiä, joissa hyökkääjä lähetti transaktioita, jotka sisälsivät laskennallisesti raskaita sopimuksia, mutta jotka olivat hyökkääjälle kuitenkin edullisia lähettää [52]. Tästä seurasi, että hyökkääjän tekemät transaktiot hidastivat verkon toimintaa kohtuuttomasti. Näiden ongelmien korjaamiseksi Ethereum sai kaksi *hard fork* -päivitystä loka- ja marraskuussa.

Seuraava suuri Ethereumiin tuleva päivitys on nimeltään *Metropolis* [22]. Version tavoitteena on tuoda Ethereum lähemmäksi loppukäyttäjiä, ja siinä on panostet-

[1] 13.2.2017

tu täysin toimivaan graafiseen käyttöliittymään, jonka avulla Ethereumissa olevia hajautettuja sovelluksia on mahdollista käyttää [49, 32]. *Metropolis* sisältää myös pieniä parannuksia, jotka ovat osa pitkän aikavälin suunnitelmaa vähentää järjestelmän kompleksisuutta [22]. *Metropolista* seuraava versio on *Serenity*, jonka lopullisina tavoitteina on tuoda parannuksia skaalautuvuuteen sekä siirtyä käyttämään *proof of stake* -konsensusalgoritmia [32].

3.2 Toimintaperiaate

Ethereumin toimintaperiaatetta voidaan kuvata tilakoneella [69]. Lohkoketjun ensimmäinen lohko kuvaa Ethereumin ensimmäistä hyväksyttävää tilaa, jonka jälkeen tila on muuttunut ainoastaan hyväksyttävien transaktioiden toimesta. Hyväksyttävät transaktiot noudattavat protokollaan asetettuja reunaehtoja ja ne vievät aina hyväksyttävään tilaan. Tieto Ethereumin voimassa olevasta tilasta sisällytetään aina lohkoketjun uusimpaan lohkoon [69].

3.2.1 Ethereum-tilit ja transaktiot

Bitcoinissa tilalla voidaan tarkoittaa tietoa siitä, kenellä on tiettynä ajanhetkenä bitcoineja ja kuinka paljon. Kyseistä tilaa kuvaavat verkossa olevat käyttämättömät ulostulot, joihin käyttäjillä on oikeus omistamiensa yksityisten avainten kautta. Bitcoin-verkon tila ei ole siis selkeä tietyllä ajanhetkellä, vaan se on muodostettava tarkastelemalla kyseisiä ulostuloja. Ethereumin tila on yksinkertaistettu koostumaan Ethereum-tileistä, jotka sisältävät suoraan tiedon kyseisellä tilillä olevasta valuutasta [47].

Ethereumissa käyttäjät muodostavat itselleen niin sanottuja ulkoisesti omistettuja tilejä (*externally owned account, EOA*), jotka pystyvät lähettämään transaktioita Ethereum-verkkoon [37, 47]. Tilin identifioi tilin osoite, joka johdetaan käyttäjän omistamasta julkisesta avaimesta [69]. Julkinen avain on puolestaan johdettu käyttäjän satunnaisesti muodostamasta yksityisestä avaimesta, jolla tilin omistajuus voidaan todentaa. Tilejä kutsutaan ulkoisesti omistetuiksi, koska niiden toimintaa ohjaavat käyttäjät, jotka omistavat tilejä vastaavat yksityiset avaimet.

Ulkoisesti omistettujen tilien lisäksi Ethereumissa on tilejä, joiden toimintaa ohjaavat tileille asetetut sopimuskoodit [37, 47]. Sopimustilit ovat itsenäisiä toimijoita, joiden sopimuskoodia ei voi tilin muodostamisen jälkeen muuttaa. Tilillä oleva sopimuskoodi suoritetaan ainoastaan silloin, kun sopimustili vastaanottaa sille kohdistetun transaktion tai viestin. Sopimustilillä on myös tietovarasto, johon sopimuskoodin käyttämä pysyvä data voidaan tallentaa.

Ethereumissa kaikki tapahtumat alkavat ulkoisesti omistettujen tilien lähettämistä transaktioista [37]. Transaktiot ovat allekirjoitettuja tietorakenteita, joilla voidaan siirtää valuuttaa tileiltä toisille ja kutsua sopimustileillä olevaa sopimuskoodia, jolloin sopimustilin suoritus aktivoituu [37, 47]. Transaktiot sisältävät tiedon lähettäjän ja vastaanottajan tilien osoitteista sekä lähettäjältä vastaanottajalle siirrettävän valuutan määrästä [69]. Jos vastaanottaja on sopimustili, transaktiossa olevaan vapaaehtoiseen datakenttään voidaan määritellä sopimukselle välitettävä viesti, joka mahdollistaa tiedon välityksen sopimuskoodille. Transaktion allekirjoitus muodostetaan käyttämällä elliptisten käyrien allekirjoitusalgoritmia (*ECDSA*) [69]. Transaktiosta lasketaan tiiviste ilman allekirjoitukseen käytettäviä tietokenttiä, ja se allekirjoitetaan käyttäen *SECP-256k1* käyrää. Allekirjoitus sisällytetään transaktioon, jotta transaktio voidaan myöhemmin varmentaa Ethereum-verkossa.

Kuten Bitcoinissa, Ethereumissa transaktioihin liittyy myös ajatus palvelumaksusta. Transaktiot sisältävät kaksi tietokenttää *STARTGAS* ja *GASPRICE*, joihin täytyy määritellä, kuinka paljon lähettäjä on valmis maksamaan transaktion suorittamisesta, jotta transaktio sisällytetään lohkoketjuun [69]. Ensimmäinen kenttä kuvaa kuinka monta laskentaoperaatiota transaktio saa suorittaa, ja jälkimmäinen määrittää kuinka paljon lähettäjä on valmis maksamaan yhdestä laskentaoperaatiosta. Voidaan ajatella, että transaktion lähettäjä ostaa louhijoilta polttoainetta (*gas*), jota käyttämällä transaktio suoritetaan [37]. Tämä toimii myös turvallisuusmekanismina haitallista käyttöä vastaan [47]. Määrittelemällä laskentaoperaatioille tietty yläraja taataan se, että transaktioiden käynnistämä koodin suoritus loppuu viimeistään silloin, kun transaktiolla ollut polttoaine loppuu. Tällaisessa tapauksessa, jossa polttoaine loppuu kesken sopimuskoodin suorituksen, kaikki Ethereumin tilaan tulleet muutokset transaktion aloituksesta kumotaan. Transaktio on kuitenkin hyväksyttävä, jolloin louhija saa itselleen suorituksessa käytetyn maksun. Tästä seuraa, että jos hyökkääjä haluaa toteuttaa esimerkiksi palvelunestohyökkäyksen Ethereum-verkkoon, täytyy hyökkääjän maksaa siitä suhteessa käytettävään laskentatehoon [47]. Jos transaktion käynnistämä sopimuskoodi suoritetaan tarjotun polttoaineen rajoissa, palautuu ylimääräisestä polttoaineesta maksettu summa takaisin lähettäjän tilille.

Sopimustilit syntyvät myös transaktioiden toimesta. Transaktioiden erikoistapaus on sopimustilin muodostaminen, jolloin transaktio lähetetään ilman vastaanottajaa [69]. Transaktio luo Ethereumiin sopimustilin, jonka osoite muodostetaan käyttäen lähettäjän osoitetta ja lähettäjän tilillä olevaa laskuria, jonka tehtävänä on ulkoisesti omistetulla tilillä laskea, kuinka monta transaktiota kyseiseltä tililtä on lähetetty [69]. Laskurin arvo sisällytetään myös uusiin transaktioihin, jota vertaamalla voidaan varmentaa, että samaa transaktiota käsitellään vain kerran. Sopimustili aluste-

taan transaktioon sisällytetyllä sopimuskoodilla, joka korvaa normaalin transaktion tietorakenteessa olevan datakentän [69].

Sopimustilit voivat myös kutsua toisia Ethereum-tilejä käyttäen viestejä (*message call*) [44, 69]. Viestit ovat rakenteeltaan kuin transaktioita, mutta ne muodostetaan vain jos sopimuskoodista kutsutaan toisia tilejä. Jos viestin vastaanottaja on sopimustili, voidaan viestiin määritellä transaktion tavoin *STARTGAS*-kenttä, joka määrittää kutsuvan sopimuskoodin suoritukseen annettavan polttoaineen määrän. Käytettävä polttoaine otetaan alkuperäisestä suoritusketjun aloittavasta transaktiosta, jolloin suoritusketju suoritetaan transaktioon määritellyn polttoaineen rajoissa. Viestit siis mahdollistavat useiden sopimustilien käytön yhdellä transaktiolla, jos ensimmäinen sopimuskoodi käyttää suorituksessaan muita sopimustilejä.

Ethereumin tilaa voidaan myös lukea suorittamalla kutsuja (*call*) [44]. Kutsut eroavat transaktioista siten, että ne eivät voi muuttaa Ethereumin tilaa, koska niitä ei louhita osaksi lohkoketjua kuten transaktioita. Tästä syystä kutsujen tekeminen on myös ilmaista, koska ne suoritetaan ainoastaan paikallisessa Ethereum-asiakaspääteohjelmassa. Kutsut mahdollistavat sopimuskoodissa määriteltyjen tilamuuttujien arvojen lukemisen.

3.2.2 Ether-kryptovaluutta

Ethereum sisältää Bitcoinin tavoin oman kryptovaluutan nimeltään ether [69]. Sen tärkein rooli on toimia maksuvälineenä loppukäyttäjiltä louhijoille, jotka ylläpitävät Ethereum-verkkoa. Vastoin yleistä käsitystä ethereitä ei ole tarkoitus käyttää suoraan valuuttana kuten bitcoineja, vaan niiden tarkoitus on toimia Ethereumissa maksuvälineenä, jolla älysopimusten suorittamiseen tarvittavat resurssit ostetaan [35]. Käyttäjä tarvitsee siis ethereitä, jotta voi olla vuorovaikutuksessa älysopimusten kanssa. Älysopimusten kehittäjiä kannustetaan tekemään laadukasta sovelluskoodia, koska huonosti suunnitellun koodin suorittaminen maksaa enemmän sen käyttäjille. Todellisuudessa ethereitä käytetään myös arvonsiirron välineenä suoraan käyttäjältä käyttäjälle ilman älysopimusten hyödyntämistä.

Ethereitä oli kesällä 2017 liikkeellä noin 90 miljoonaa kappaletta [25], josta 60 miljoonaa oli syntynyt vuonna 2014 järjestetyn ennakkomyynnin seurauksena [35]. Bitcoinien tavoin Ethereitä luodaan lisää louhinnan yhteydessä. Onnistuneesti uuden lohkon muodostanut taho saa lohkopalkkion, joka on 5 etheriä [35]. Ethereumissa tällä hetkellä käytössä oleva *proof of work* -algoritmi palkitsee myös louhijat, jotka löytävät uuden lohkon, mutta joka ei kuitenkaan päädy voimassa olevaan lohkoketjuun [69]. Nämä niin sanotun enolohkon (*uncle block*) löytäjät saavat myös lohkopalkkion,

joka on 2-3 etheriä [35]. Kokonaisuudessaan ethereiden luonti on rajoitettu 18 miljoonaan vuodessa, jonka on arvioitu ajan kuluessa olevan määrä, joka ethereitä päätyy käyttökelvottomaksi esimerkiksi yksityisten avaimien häviämisen seurauksena [35]. Myöhemmin Ethereumin siirtyessä käyttämään *proof of stake* -algoritmia lohkoketjun ylläpitoon tarvittava sähkönkulutus pienenee huomattavasti, jolloin uusien ethereiden luomisen uusien lohkojen muodostamisen yhteydessä odotetaan pienenevän huomattavasti [35].

3.2.3 Ethereum-virtuaalikone

Ethereumin toiminta perustuu sen sisältämän Ethereum-virtuaalikoneen (*EVM*) toimintaan [69]. Kaikki Ethereumissa olevat transaktiot suoritetaan Ethereum-virtuaalikoneessa, joka mahdollistaa sopimustilien ja niiden sopimuskoodien olemassaolon. Ethereum-virtuaalikoneen voidaan ajatella olevan yksi suuri järjestelmä, joka koostuu kaikista Ethereum-asiakaspääteohjelmaa suorittavista tietokoneista.

EVM on yksinkertainen pinoarkkitehtuuriin perustuva kone, jonka sananpituus on 32 tavua [69]. Sen suunnitteluperiaatteisiin kuuluvat muun muassa yksinkertaisuus, deterministisyys ja erikoistuminen lohkoketjukontekstissa suoritettaviin ohjelmiin [46]. Yksinkertaisuudella tarkoitetaan, että EVM:n tulee sisältää mahdollisimman vähän ja mahdollisimman matalan tason operaatiokoodeja, jotka ohjaavat virtuaalikoneen toimintaa. EVM:n sisältämät operaatiokoodit kattavat aritmeettiset laskutoimitukset, vertailu- ja bittitason loogiset operaatiot, ympäristö- ja lohkotiedot, pino-, muisti-, tallennus- ja tietovuonohjausoperaatiot, lokitusoperaatiot ja järjestelmäoperaatiot [30]. Näistä erityisesti lohkotiedot erottavat Ethereum-virtuaalikoneen toiminnan muista tyypillisistä virtuaalikoneista, mitä tukee suunnitteluperiaatteissa mainittu erikoistuminen. EVM pystyy siis yhdellä operaatiokoodilla selvittämään esimerkiksi halutun lohkon järjestysnumeron [30]. Deterministisyydellä tarkoitetaan, että EVM:n toiminnassa ei saa olla tulkinnanvaraa, vaan jokaisella Ethereum-asiakaspääteohjelman suorittajalla virtuaalikoneen on tuotettava samoja tuloksia transaktioiden varmentamisessa ja sopimuskoodien suorittamisessa. Tämä ominaisuus on Ethereumissa olevan konsensuksen kannalta välttämätöntä.

Ethereum-virtuaalikone on Turing-täydellinen [69]. EVM pystyy suorittamaan sille ohjelmoituja ohjelmia ja ratkaisemaan niissä kuvatut ongelmat. Tässä tapauksessa ohjelmilla tarkoitetaan sopimustileillä olevia sopimuskoodeja. Ohjelmat voivat sisältää monimutkaisia ohjausrakenteita kuten silmukoita, jotka voivat periaatteessa olla päättymättömiä. Ohjelmien suoritusta rajoitetaan kuitenkin Ethereum-virtuaalikoneessa polttoaineparametrilla, joka kuvaa kuinka monta laskentaoperaatiota ohjelma saa suorituksessaan käyttää [69]. Jokaiselle operaatiokoodille on määritelty sen

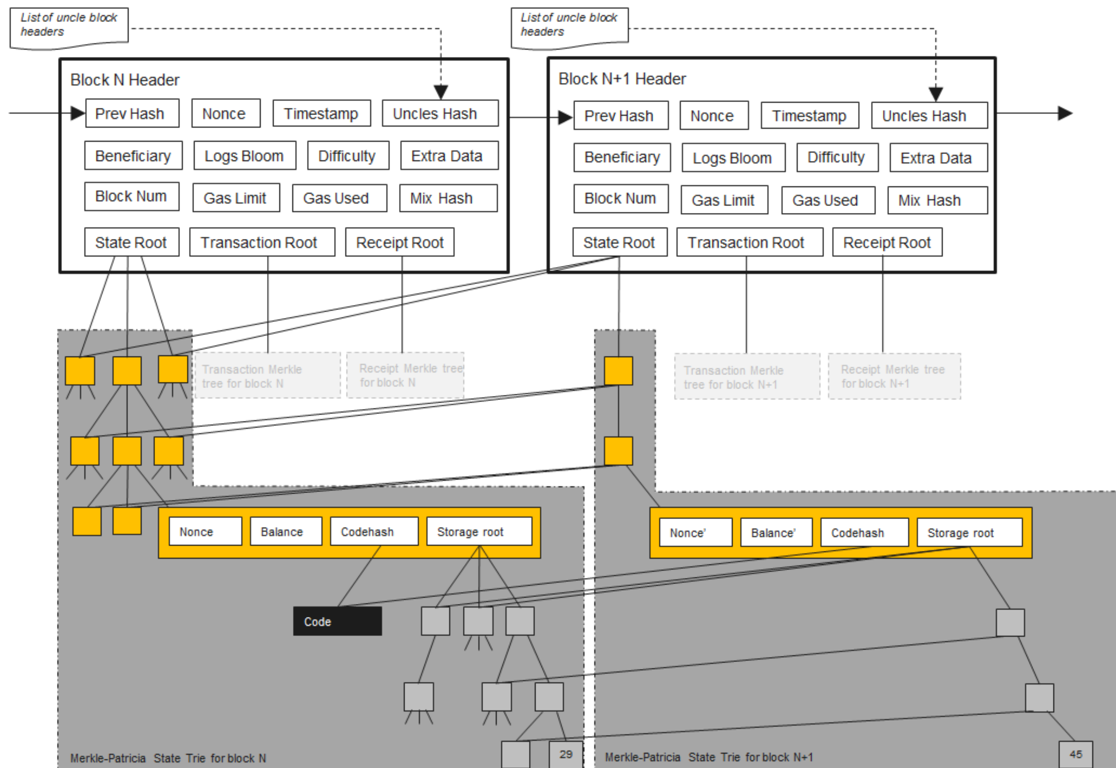
suoritukseen tarvittavan polttoaineen määrä. Tämä toimii yksinkertaisena turvallisuusmallina virtuaalikoneen väärinkäyttöä vastaan.

Ohjelmilla on käytössään kolme eri tapaa tallentaa tietoja: pysyvä tallennusmuisti, tilapäinen muisti ja ajonaikainen pino [65]. Jokaisella ohjelmalla on pysyvä tallennusmuisti, johon tallennetaan ohjelman tilamuuttujien arvot. Pysyvää tallennusmuistia ylläpidetään Ethereumin tilassa, mikä tarkoittaa, että tallennusmuistin sisältö tallennetaan osaksi lohkoketjua. Tämä mahdollistaa tilamuuttujien arvojen säilymisen sopimuskoodille kohdistettujen funktiokutsujen välillä. Tilapäisille tiedoille on toteutettu muisti, jota hyödynnetään esimerkiksi funktioparametrien säilytykseen funktiokutsuissa. Kolmas tapa tietojen tallentamiseen on ajonaikainen pino, jota hyödynnetään funktioissa olevien paikallisten muuttujien toiminnassa.

3.2.4 Ethereum-lohkoketju

Ethereum-lohkoketjun toteutus vastaa suurelta osin Bitcoinissa olevaa lohkoketjua [47]. Suurin ero näiden välillä on se, että Ethereumin lohkoketju sisällyttää myös lohkoon kyseisessä lohkossa käsitellyt transaktiot. Lohkon rakenne muodostuu otsikkotietueesta, transaktiolistasta ja listasta otsikkotietueita, jotka kuvaavat kyseisen lohkon enolohkoja [69]. Enolohkot ovat lohkoja, joilla tiedetään olevan sama vanhempi kuin käsiteltävän lohkon vanhemmalla. Näiden sisällyttäminen osaksi lohkoketjua mahdollistaa Ethereumissa pienemmän lohkoajan verrattuna Bitcoiniin. Ethereumissa voimassa oleva lohkoketju määritellään sen mukaan, mikä lohkoketjun reitti ensimmäisestä lohkosta viimeisimpään lohkoon vaatii eniten laskentaa [69].

Lohkon otsikkotietueen rakenne on esitetty kuvassa 3.1. Lohkojen ketjuttaminen tapahtuu samoin kuin Bitcoinissa, jossa lohkon otsikkotietueeseen sisällytetään edellisen lohkon otsikkotietueesta laskettu tiiviste (*prev hash*) [69]. Otsikkotietue sisältää myös lohkon löytäjän määrittelemän Ethereum-tilin osoitteen (*beneficiary*), johon lohkosta muodostuneet palkkiot maksetaan, lohkon vaikeusasteen (*difficulty*), lohkon järjestysnumeron (*block num*), lohkon muodostamisen aikaleiman (*timestamp*) ja vapaaehtoisen datakentän (*extra data*), jonka lohkon löytäjä voi halutessaan täyttää [69]. Lohkojen kokoa hallitaan määrittelemällä lohkoon polttoaineen maksimimäärä (*gas limit*), jonka rajoissa lohkon sisällyttämät transaktiot tulee suorittaa, sekä kuinka paljon polttoainetta todellisuudessa käytettiin (*gas used*) [69]. Polttoaineen maksimäärä rajoittaa siis lohkoon sisällytettävien transaktioiden määrää. Lohkon enolohkojen otsikkotietuelistasta laskettu tiiviste sisällytetään myös osaksi lohkon otsikkotietuetta (*uncles hash*) [69].



Kuva 3.1 *Ethereum-lohkon otsikkotietueen rakenne [41].*

Ethereumin tilaa käsitellään hyödyntäen muokattua Merkle-Patricia-puuta, joka on pohjimmiltaan kryptografisesti todennettava tietorakenne avain–arvo-parien käsittelyyn [69]. Ethereumin tilalla tarkoitetaan tietoa kaikista Ethereum-tileistä ja niihin liittyvistä tilin tilatiedoista, jotka sisältävät esimerkiksi tiedon siitä kuinka paljon tilillä on ethereitä [69]. Merkle-Patricia-puun lehtisolmuista voidaan laskea Merkle-puun tavoin tiivisteitä, joita yhdistämällä saadaan muodostettua juurisolmu, joka sisältää yhden tiivisteen, jolla koko tietorakenteen sisältö voidaan varmentaa. Kun kaikki lohkon sisältämät Ethereumin tilaa muokkaavat transaktiot on suoritettu, sisällytetään Merkle-Patricia-puun juurisolmun tiiviste osaksi lohkon otsikkotietuetta (*state root*) [69]. Tämä mahdollistaa sekä voimassa olevan Ethereumin tilan varmentamisen että aikaisempien tilojen palauttamisen. Merkle-Patricia-puu mahdollistaa juurisolmun uudelleen laskemisen vain sen muuttuneiden polkujen osalta. Tämä ominaisuus on Ethereumin tilan ylläpidon kannalta tehokas, koska jokainen transaktio muokkaa Ethereumin tilaa ja vaikuttaa puun rakenteeseen ja sisältöön. Kuvassa 3.1 on esitetty kuinka Ethereumin tilan sisältävä Merkle-Patricia-puu muokkaantuu, kun sopimustilillä olevan tilamuuttujan arvo muuttuu uudessa lohkossa. Itse puuta ei siis sisällytetä osaksi lohkoketjua vaan sen ylläpito tapahtuu Ethereum-asiakassovelluksessa.

Lohkon otsikkotietueeseen sisällytetään myös lohkon sisältämistä transaktioista lasketun Merkle-puun juurisolmu (*transaction root*), jonka avulla transaktioiden olemassaolo kyseissä lohkossa voidaan varmentaa [69]. Tämä vastaa toteutukseltaan Bitcoinissa tehtävää Merkle-puun muodostamista. Kun Ethereum-verkkoon lähetettyjä transaktioita suoritetaan, muodostuu niistä niin sanottuja transaktiokuitteja (*transaction receipt*), jotka kuvaavat tietyn transaktion suorittamisen jälkeistä tilaa [69]. Transaktiokuitti sisältää transaktion suorittamisen jälkeisen Ethereumin tilan sekä tiedon kuinka paljon transaktion suorittaminen kulutti polttoainetta ja mitä lokitietoja transaktio muodosti. Transaktiokuiteista lasketun Merkle-puun juurisolmu sisällytetään myös lohkon otsikkotietueen kenttään *receipt root* [69]. Jotta transaktioista muodostuneita lokiviestejä voidaan hakea Ethereum-lohkoketjusta tehokkaasti, lasketaan transaktiokuiteista olevista lokitiedoista niin sanottu Bloom-filtteri (*bloom filter*), joka sisällytetään lohkon otsikkotietueen kenttään *logs bloom* [69]. Lokitiedoista muodostettu Bloom-filtteri sisältää tiedon siitä, mikä Ethereum-tili on kirjoittanut lokiviestin ja mitkä ovat olleet lokiviestin tietokenttien nimet [42]. Esimerkiksi kaikki tietyn Ethereum-tilin tekemät lokiviestit tietyssä lohkossa saadaan hakemalla ensin lohkon otsikkotietueesta olevasta Bloom-filtteristä, sisältääkö se halutun tilin osoitetta. Jos tili löytyy, voi Ethereum-asiakasohjelma suorittaa kaikki lohkossa olevat transaktiot uudestaan ja palauttaa kutsujalle halutun Ethereum-tilin luomat lokiviestit transaktiokuitteja käyttäen. Tätä toiminnallisuutta kutsutaan myös nimellä transaktioloki.

Ethereumissa uusien lohkojen muodostaminen ja lohkoketjun konsensuksen ylläpitäminen vastaavat idealtaan Bitcoinissa olevaa toteutusta [69]. Ethereumissa lohkoketjun konsensus saavutetaan käyttäen *proof of work* -algoritmia nimeltään *Ethash* [45]. Algoritmi on suunniteltu siten, että sen tehokkaaseen suorittamiseen oleva laitteisto on jo valmiiksi olemassa. Tämä saavutetaan siten, että algoritmin suoritus pohjautuu käytettävissä olevaan muistin nopeuteen. Tällä suunnitteluratkaisulla pyritään estämään louhinnan keskittymistä tietyille ryhmille, jotka pystyvät rakentamaan laitteistoja, joilla algoritmin suoritus olisi huomattavasti tehokkaampaa verrattuna muilla louhijoilla käytössä olevaan laitteistoon. Käytännössä *Ethash*-algoritmin suorittamiseen käytetään kuluttajille tarkoitettuja näytönohjaimia, mikä on vaikuttanut merkittävästi niiden saatavuuteen ja hinnoitteluun [68].

Ethereum-lohkoketjuun lisättävät uudet lohkot muodostuvat louhinnan kautta keskimäärin joka 16. sekunti [40]. Lohkoaika on siis huomattavasti lyhyempi verrattuna Bitcoinin kymmeneen minuuttiin. Ethereum käyttää yksinkertaistettua *GHOST*-protokollaa (*Greedy Heaviest Observed Subtree*), joka mahdollistaa aiemmin mainittujen enolohkojen huomioimisen voimassa olevassa lohkoketjussa [69].

3.2.5 Älysopimukset

Älysopimukset (*smart contract*) ovat ohjelmakoodilla rakennettuja tietokoneohjelmia, joiden sisältämän sopimuspohjaisen toiminnan suoritus tapahtuu ilman kolmansien osapuolten tekemää varmennusta [54]. Ajatus älysopimuksista on esitetty jo vuonna 1996, jolloin kryptografi Nick Szabo kuvasi artikkelissaan [66] kuinka toisilleen tuntemattomat tahot voisivat toteuttaa sopimuspohjaista toimintaa tietoverkkojen välityksellä, käyttäen verkkokaupankäyntiin tarkoitettuja protokollia. Szabo esitti, että eri osapuolten määrittelemät sopimusehdot suoritettaisiin käyttäen transaktioprotokollia, joiden vastuulla sopimusehtojen suorittaminen olisi. Edellytykset älysopimusten toteuttamiseen tulivat myöhemmin mahdollisiksi lohkoketjujen keksimisen myötä.

Ethereumissa sopimustileille asetettava sopimuskoodi täyttää älysopimusten määritelmän. Sopimuskoodilla voidaan kuvata sopimuspohjaisen toiminnan ehdot, joiden täyttyminen tarkastetaan transaktion suorituksen yhteydessä. Ethereumin yhteydessä puhutaan myös niin sanotuista hajautetuista sovelluksista (*decentralized application*), joilla tarkoitetaan älysopimuksista koostuvia kokonaisuuksia, joiden suoritus tapahtuu hajautetusti Ethereum-sovellusalustalla [43]. Verrattuna tyypilliseen web-sovellukseen, jossa palvelimella sijaitseva ohjelmakoodi suoritetaan keskitetysti, hajautetun sovelluksen ohjelmakoodi rakentuu kokonaan Ethereumissa suoritettavista älysopimuksista. Käytännössä älysopimuksia hyödyntävän sovelluksen arkkitehtuurissa voidaan hyödyntää näiden yhdistelmää, joka on esitetty luvussa 4 toteutetussa sovelluksessa.

3.3 Solidity-ohjelmointikieli

Solidity on korkean tason ohjelmointikieli, jonka avulla Ethereumissa olevien sopimustilien sopimuskoodi voidaan kirjoittaa [62]. Solidityn syntaksi muistuttaa JavaScriptiä, minkä vuoksi sen oppimiskynnys on sekä kokeneille että aloitteleville ohjelmoijille matala. Se on ohjelmointiparadigmaltaan sopimuspohjainen (*contract-oriented*) ohjelmointikieli, joka muistuttaa olio-ohjelmointia niin, että olioiden sijaan toiminta jäsennetään sopimuksiin.

Solidityllä kirjoitettu lähdekooditiedosto 3.2 alkaa käytettävän kieliversion määrittelyllä. Rivillä 1 oleva avainsana *pragma* ohjeistaa kääntäjää, millä Solidity-versiolla ohjelmakoodi tulee kääntää Ethereum-virtuaalikoneen tavukoodiksi [64]. Kielen kehityksessä käytetään semanttista versiointia, minkä avulla taaksepäin yhteensopimattomat versiot voidaan nopeasti tunnistaa. Esimerkiksi lähdekooditiedostoa 3.2

```
1  pragma solidity ^0.4.0;
2
3  contract VirtualCoffeeMachine {
4      address owner;
5      uint public price;
6
7      function VirtualCoffeeMachine(uint initialPrice) {
8          owner = msg.sender;
9          price = initialPrice;
10     }
11
12     modifier onlyOwner() {
13         if(msg.sender != owner) throw;
14         _;
15     }
16
17     modifier costs(uint quantity) {
18         if(quantity == 0) throw;
19         var totalPrice = calcTotalPrice(quantity);
20         if(msg.value < totalPrice) throw;
21         _;
22     }
23
24     function collectMoney() onlyOwner() {
25        owner.transfer(this.balance);
26     }
27
28     function buyDrink(uint quantity) public payable
29         costs(quantity) returns (string)
30     {
31         return "Here is your coffee";
32     }
33
34     function calcTotalPrice(uint quantity) private
35         constant returns (uint)
36     {
37         return price*quantity;
38     }
39 }
```

Ohjelma 3.2 *Esimerkki Soliditylla ohjelmoidusta sopimuksesta.*

ei käsitellä kääntäjällä, jonka kieliversio on pienempi kuin 0.4.0 tai suurempi kuin 0.4.x, jossa x tarkoittaa symanttisessa versioinnissa korjausten määrää. Kirjoitushetkellä[2] Solidityn uusin versio 0.4.10 täyttää edellä määritellyt ehdot.

Sopimukset koostuvat tilamuuttujista sekä funktioista, jotka voivat muuttaa tilamuuttujien tilaa [60]. Sopimuksilla oleva pysyvä data säilötään tilamuuttujiin, joiden avulla sopimuksilla on käsitys niiden voimassaolevasta tilasta. Julkisesti määriteltyjä funktioita ja tilamuuttujia voidaan kutsua käyttämällä transaktioita. Sopimuksel-

[2] 6.3.2017

le voi määritellä rakentajan, joka alustaa sopimuksen tiettyyn tilaan. Rakentajaa kutsutaan ainoastaan silloin, kun sopimustili muodostetaan.

Lähkekooditiedoston 3.2 riviltä 3 alkaa sopimuksen määrittely, joka pitää sisällään sopimukselle määritellyt tilamuuttujat ja funktiot. Ensimmäinen funktio rivillä 7 on sopimuksen rakentaja. Rakentajalle voidaan antaa parametrina muuttujan *initialPrice* arvo, joka asetetaan myöhemmin rakentajassa julkisen tilamuuttujan *price* arvoksi. Rakentajassa, rivillä 8, asetetaan myös sopimuksen omistajuus vastaamaan tiliä, joka on lähettänyt kyseisen sopimustilin luontitransaktion. Transaktion lähettäjä saadaan Solidityssä tietoon käyttämällä valmiiksi asetettua globaalia muuttujaa *msg.sender*, joka palauttaa transaktion lähettäjän osoitteen [63]. Muita globaalien muuttujien kautta saatavia tietoja ovat muun muassa nykyiseen lohkoon liittyvät tiedot, kuten lohkonumero ja aikaleima, sekä transaktiossa lähetettyjen ethereiden määrä [63].

Funktioille ja tilamuuttujille voidaan muiden oliopohjaisten kielten tavoin määritellä suojausmääre määrittämään, mistä kyseistä funktiota tai tilamuuttujaa voidaan kutsua [60]. Solidityssä funktioiden oletussuojausmääre on *public*, mikä tarkoittaa, että funktioita voidaan kutsua niin sopimuksen sisältä kuin ulkopuoleltakin. Suojausmääre *private* puolestaa rajaa funktion kutsumisen sopimuksen sisälle. Suojausmääreet eivät kuitenkaan rajaa tiedon saatavuutta, vaan kaikki sopimuskoodi on sopimuksen ulkopuoliselle tarkastelijalle saatavilla olevaa tietoa [60]. Tilamuuttujien oletussuojausmääre on *internal*, mikä rajaa tilamuuttujien kutsumisen kyseiseen sopimukseen ja kaikkiin siitä periytettyihin sopimuksiin. Solidity tukee moniperintää eli sopimus voi periytyä useammasta kuin yhdestä sopimuksesta [60]. Perinnän tavoitteena on helpottaa ja selkeyttää Solidityllä määriteltyjä sopimuksia. Periytettyä sopimusta käännettäessä kaikki sopimukset, joista käännettävä sopimus riippuu, kopioidaan osaksi käännettävää sopimusta, ja lopputuloksena on yksi sopimus, joka julkaistaan sellaisenaan lohkoketjuun. Periytyminen ei siis sellaisenaan mahdollista sopimuskoodin uudelleenkäyttöä lohkoketjussa.

Lähdekooditiedoston 3.2 rivillä 5 on määritelty virtuaalisen kahviautomaatin tarjoaman kahvin hinta. Tilamuuttuja on suojausmääreeltään *public*, mikä käännettäessä luo sille automaattisesti saantifunktion (*getter function*), joka palauttaa tilamuuttujan arvon [60]. Koska funktio palauttaa vain tilamuuttujaan asetetun arvon, eikä suorita muita operaatiota, jotka muuttaisivat sopimuksen tilaa, voidaan saantifunktiota kutsua ilman lohkoketjuun lähetettävää transaktiota. Tällöin funktioon voidaan kohdistaa kutsu, joka suoritetaan pelkästään paikallisella olevalla Ethereum-asiakaspääteohjelmalla. Kutsun suoritus ei siis maksa mitään, koska se suoritetaan ainoastaan paikallisesti eikä muuta lohkoketjun tilaa. Solidityssä kaikki avainsanalla *constant* merkityt funktiot lupaavat olla muokkaamatta sopimuksen tilaa, joten

niitä voidaan kutsua paikallisesti [60]. Rivillä 34 määriteltyä funktiota ei kuitenkaan voida kutsua sopimuksen ulkopuolelta, *constant* avainsanasta huolimatta, koska sen suojausmääre on *private*.

Funktiolle voidaan määritellä modifioijia, jotka suoritetaan ennen varsinaista, kutsuttua funktiota [60]. Lähdekooditiedoston 3.2 riveillä 12 ja 17 on määritelty kaksi modifioijaa avainsanalla *modifier*. Esimerkiksi *onlyOwner*-modifioijassa oleva ohjelmakoodi suoritetaan vain, jos *collectMoney*-funktiota kutsutaan. Jotta suoritus palautuu kutsuttuun funktioon, täytyy modifioijan ohjelmakoodissa suorituksen jatkua riville 14, jossa alaviivalla määritellään suorituksen paluu kutsuttuun funktioon. Modifioijilla voidaan asettaa funktioille esiehtoja, joiden täytyy täyttyä ennen funktion suorittamista. Niiden avulla samoja esiehtoja ei tarvitse määritellä jokaiseen funktioon erikseen. Jos *collectMoney*-funktiota kutsuu joku muu kuin sopimuksen omistaja, rivillä 13 määritelty esiehto ei täyty ja sopimus heittää poikkeuksen. Tällöin transaktion suoritus keskeytetään ja kaikki tilaan kohdistuneet muutokset perutaan transaktion alkamista vastaavaan tilaan [60]. Poikkeuksen käsittelyssä transaktiossa annettu polttoaine (*gas*) kulutetaan loppuun, mutta transaktiossa mahdollisesti olevat etherit palautetaan käyttäjälle.

Rivillä 28 määritellyssä *buyDrink*-funktiossa oleva *payable* avainsana erottaa Solidityn normaaleista oliopohjaisista kielistä. Avainsana mahdollistaa funktion vastaanottaa ethereitä, jotka on määritelty transaktioon, jolloin ne onnistuneen funktiosuorituksen jälkeen siirtyvät sopimustilille [60]. Ohjelmoijan vastuulle jää kuitenkin tarkastaa, onko transaktiossa lähetetty tarpeeksi ethereitä laukaisemaan tietty toiminallisuus. Esimeriksi rivillä 17 määritelty *costs*-modifioija tarkastaa, riittääkö transaktiossa olevat etherit tiettyyn määrään virtuaalikahvia.

4. ÄLYSOPIMUKSIA HYÖDYNTÄVÄN SOVELLUKSEN TOTEUTTAMINEN

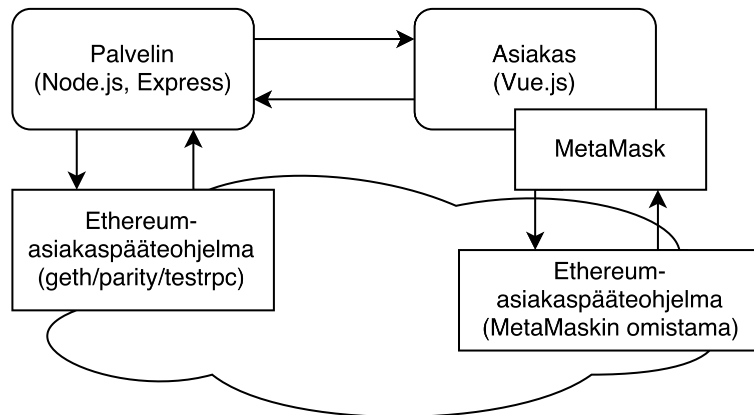
Tässä luvussa käydään läpi Ethereumia ja älysopimuksia hyödyntävän esimerkkisovelluksen toiminta ja suunnitteluperiaate. Sovelluksen on tarkoitus havainnollistaa, kuinka Ethereumia voidaan hyödyntää web-sovelluksessa ja kuvata, kuinka sovellus voi kommunikoida Ethereum-sovellusalustan kanssa.

4.1 Toteutettava sovellus

Toteutettava sovellus mahdollistaa työnantajan ja työntekijän välisen työsopimuksen viemisen Ethereum-lohkoketjuun älysopimuksen muodossa.[1] Älysopimukseen kuvatut sopimusehdot ovat yksiselitteisiä, mikä tekee niistä teoriassa turvallisempia verrattuna kirjallisiin sopimuksiin, joissa on mahdollisuus tulkinnanvaraisuuteen. Älysopimukset mahdollistavat myös tavanomaisten sopimusten viemisen digitaaliseen muotoon vähentäen niiden solmimisesta ja käsittelystä aiheutuvia kustannuksia. Sovelluksessa luotavat älytyösopimukset tallennetaan lohkoketjuun, jolloin niiden arkistoiminen toimii hajautetusti ja saatavuus paranee. Sovellus tarjoaa käyttöliittymän, jolla uuden työsopimuksen luominen on mahdollista syöttämällä työsopimukseen liittyvät tiedot. Olennaista sovelluksessa on, että työntekijän täytyy vielä erikseen hyväksyä älytyösopimus käyttäen omaa Ethereum-tiliään, jonka jälkeen työsopimus tallennetaan hyväksyttynä lohkoketjuun. Sopimukseen määritellään aika, johon mennessä työntekijän on viimeistään hyväksyttävä työsopimus. Älysopimukseen mallinnettu työsopimus ei tämän työn kontekstissa ole täydellinen ottaen huomioon työsopimuslaissa määritellyn työsopimuksen sisällön.

Kuvassa 4.1 on esitetty sovelluksen yleinen arkkitehtuuri. Sovellus noudattaa asiakas-palvelin-mallia, jossa palvelin hallitsee lohkoketjuun lähetettyä älysopimusta, jonka vastuulla on uusien älytyösopimusten luominen. Palvelin tarjoaa myös rajapinnan, jota käyttäen uusien työsopimuksen luominen on mahdollista. Jotta pal-

[1] Toteutetun sovelluksen lähdekoodit ovat saatavilla osoitteissa
`https://github.com/salatti/employment-contract-backend`
`https://github.com/salatti/employment-contract-frontend`



Kuva 4.1 *Sovelluksen arkkitehtuuri yleisellä tasolla.*

velin voi kommunikoida Ethereum-lohkoketjun kanssa, täytyy palvelimen suorittaa myös Ethereum-asiakasohjelmaa. Kuvassa 4.1 esitetty pilvi kuvaa Ethereum-sovellusalustan vertaisverkkoa, jossa lohkoketjuun lähetetyt älysopimukset sijaitsevat. Sovelluksen kehityksessä käytettiin testaukseen ja kehitykseen tarkoitettua Ethereum-asiakasohjelmaa nimeltään *testrpc*.

Testrpc on Node.js-alustaa hyödyntävä Ethereum-asiakasohjelma, joka simuloi toiminnallisuudeltaan kokonaista asiakasohjelmaa kuten *geth*, mutta se ei kommunikoi muiden Ethereum-asiakasohjelmien kanssa vertaisverkon välityksellä [48]. *Testrpc* toimii hiekkalaatikkotilassa ja luo käynnistyessään uuden Ethereum-lohkoketjun sekä tarvittavan määrän ulkoisesti omistettuja tilejä, joille kullekin on asetettu käytettävissä olevia ethereitä. Ulkoisesti omistettujen tilien hallitsemiseen tarvittavat yksityiset avaimet tulostuvat ohjelmaa suoritettaessa komentoriville, mikä mahdollistaa tilien käyttämisen kehityksessä ja testauksessa. *Testrpc* tarjoaa muiden Ethereum-asiakasohjelmien tavoin JSON-RPC-protokollan mukaisen rajapinnan viestintään Ethereum-asiakasohjelman kanssa. Kommunikointi asiakasohjelman kanssa rajapintaa käyttäen mahdollistaa sovelluksessa hyödynnettävän Ethereum-asiakasohjelman vaihtamisen ilman, että sovellukseen tarvitsisi tehdä muutoksia.

Asiakaskäyttöliittymässä hyödynnetään Google Chrome -selaimeen saatavilla olevaa MetaMask-laajennosta, joka mahdollistaa kommunikoinnin Ethereumissa olevien älysopimusten kanssa ilman käyttäjällä olevaa Ethereum-asiakassovellusta [57]. MetaMask sisältää Ethereum-lompakon ja se pystyy hallitsemaan ulkoisesti omistettuja tilejä sekä allekirjoittamaan Ethreumiin lähetettäviä transaktioita. MetaMask pystyy kommunikoimaan itse Ethereum-lohkoketjun kanssa käyttäen MetaMaskin ylläpitämiä palvelimia, joissa suoritetaan Ethereum-asiakassovellusta.

4.2 Palvelin

Sovelluksen palvelin on toteutettu Node.js-alustaa käyttävällä Express-sovelluskehyksellä. Lisäksi palvelimella hyödynnetään Ethereum-kehitykseen tarkoitettua sovelluskehystä nimeltään Truffle [28]. Truffle helpottaa Solidityllä ohjelmoitujen älysopimusten hallintaa niiden kääntämisestä käyttöönottoon ja mahdollistaa älysopimusten automaattisen testauksen [29].

Truffle muodostaa käännetystä sopimuksesta niin sanotun artifact-tiedoston, jota Truffle käyttää sisäisesti eri toiminnallisuuksissa, kuten sopimustilin muodostamisessa [29]. Artifact-tiedosto sisältää JSON-muodossa sopimuksen ABI-rajapintakuvauksen sekä sopimuksen käännetyn tavukoodin. ABI-rajapintakuvaus kuvaa, kuinka sopimuksen funktioita voidaan kutsua ja missä muodossa kutsun parametrit ja paluuarvo tulee käsitellä. Truffle sisältää myös toiminnallisuuden sopimuksen lähettämiseen Ethereum-lohkoketjuun migraatioiden avulla [29]. Onnistuneen migraation jälkeen sopimuksen artifact-tiedostoon päivitetään tieto siitä, mihin Ethereum-verkkoon kyseinen sopimus on lähetetty ja mikä kyseisen sopimustilin osoite on. Lähdekooditiedostossa 4.2 on esitetty Trufflen muodostama artifact-tiedosto, joka sisältää esimerkkinä yhden funktion ABI-rajapintakuvauksen. Sopimuksen käännetty tavukoodi on esitetty rivillä 18 ja sopimustilin osoite, jossa kyseinen sopimuskoodi sijaitsee, on esitetty rivillä 24. Ilman ABI-rajapintakuvausta sopimuksen sisältämiä funktioita olisi vaikea tunnistaa suoraan käännetystä tavukoodista.

Palvelimelle on toteutettu rajapinta, jota käyttämällä uusien sopimusten luominen on mahdollista. Rajapinta pystyy myös palauttamaan sovelluksen kautta lisättyjen älytyösopimusten osoitteet liitettynä työntekijän Ethereum-tilin osoitteeseen. Sovelluksessa ei ole käytössä tyypilliseen web-sovellukseen kuuluvaa tietokantaa, vaan kaikki työsopimuksiin liittyvät toiminnot tehdään hallitusti *ContractCreator*-sopimuksen kautta. *ContractCreator*-sopimuksen toteutus on esitetty lähdekooditiedostossa 4.3. Sopimuksen käyttöä on rajoitettu määrittelemällä sille omistajuus, mikä tehdään sopimuksen rakentajassa rivillä 10. Rakentajaa kutsutaan, kun sopimus lähetetään Ethereum-verkkoon Trufflen tekemässä migraatiossa. Truffle käyttää migraatioissa oletusarvoisesti Ethereum-asiakasohjelmalta ensimmäistä saatavilla olevaa Ethereum-tiliä, jota käyttämällä lähetetään sopimustilien muodostamiseen tarvittavat transaktiot. Sopimuksen omistajaksi asetetaan siis palvelimen hallussa oleva Ethereum-tili, jonka voidaan ajatella olevan sovelluksessa työnantajan virallinen Ethereum-tili. *ContractCreator*-sopimuksen rivillä 23 on määritelty funktio, jota kutsumalla uusia älytyösopimuksia voidaan lisätä. Funktiossa uusi sopimus luodaan avainsanalla *new*, mikä vaatii, että luotavan sopimuksen sopimuskoo-

```
1  {
2    "contract_name": "ContractCreator",
3    "abi": [
4      {
5        "constant": false,
6        "inputs": [
7          {"name": "addrOfEmployee", "type": "address"},
8          {"name": "employee", "type": "bytes32"},
9          {"name": "lastAccTime", "type": "uint256"}
10       ],
11       "name": "createEmplyoymentContract",
12       "outputs": [],
13       "payable": false,
14       "type": "function"
15     },
16     ...
17   ],
18   "unlinked_binary": "0x6060604052341561000f57600080fd5...",
19   "networks": {
20     "1503311726535": {
21       "events": {...}
22       },
23       "links": {},
24       "address": "0xc89ce4735882c9f0f0fe26686c53074e09b0d550",
25       "updated_at": 1503311762381
26     }
27   },
28   "schema_version": "0.0.5",
29   "updated_at": 1503311762381
30 }
```

Ohjelma 4.2 *Truffle-sovelluskehyksen muodostama artifact-tiedosto.*

di on oltava funktion suoritushetkellä tiedossa. *ContractCreator*-sopimuksella onkin riippuvuus itse älytyösopimuksen Solidity-ohjelmakoodiin ja se luetaan osaksi sopimusta rivillä 2, jotta niiden luominen on mahdollista sopimuksen sisältä. Älytyösopimuksen sopimuskoodi on esitetty liitteessä 1. Jotta uuden luodun sopimustilin osoite voidaan välittää funktion kutsujalle, tapahtuma lokitetaan käyttäen Solidityn *event*-toiminnallisuutta, joka mahdollistaa tietojen kirjoittamisen Ethereum-virtuaalikoneen transaktiolokiin. Transaktiolokia voidaan myös käyttää selvittämään kaikki *ContractCreator*-sopimuksen kautta luodut älytyösopimukset, minkä vuoksi palvelimella ei ole tarvetta ylläpitää erillistä tietokantaa.

Palvelimella olevan rajapinnan POST-metodireitin toteutus on esitetty lähdekooditiedostossa 4.4. Toteutuksessa hyödynnetään *Web3*- ja *truffle-contract*-JavaScript-kirjastoja. *Web3*-kirjasto tarjoaa helppokäyttöisen JavaScript-rajapinnan, jonka avulla Ethereum-asiakassovellukseen tehtäviä JSON-RPC-protokollan mukaisia kutsuja tehdään. Rajapinta tarjoaa myös joukon apumetodeja, jotka tekevät tarvitta-

```
1  pragma solidity ^0.4.15;
2  import "./EmploymentContract.sol";
3
4  contract ContractCreator {
5      address public owner;
6      address public latest;
7      uint numContracts;
8
9      function ContractCreator() {
10         owner = msg.sender;
11     }
12
13     modifier onlyOwner() {
14         require(msg.sender == owner);
15         _;
16     }
17
18     event NewContract(
19         address indexed addrOfEmployee,
20         address indexed addrOfContract
21     );
22
23     function createEmplyoymentContract(address addrOfEmployee,
24         bytes32 employee, uint lastAccTime) onlyOwner
25     {
26         EmploymentContract newCont =
27             new EmploymentContract(addrOfEmployee, employee,
28                 lastAccTime);
29
30         latest = newCont;
31         numContracts++;
32         NewContract(addrOfEmployee, newCont);
33     }
34 }
```

Ohjelma 4.3 *ContractCreator-sopimus, joka hallitsee älytyösopimuksia.*

vat tietotyyppimuutokset helpoiksi. Toteutuksessa *Web3*-kirjastoa käytetään apumetodien käyttöön – esimerkiksi rivillä 13 käytetään Web3-kirjaston metodia, joka palauttaa annetun merkkijonon heksadesimaalimuodossa.

Palvelin kommunikoi *ContractCreator*-sopimuksen kanssa käyttäen *truffle-contract*-JavaScript-kirjastoa. *Truffle-contract* abstrahoi rivillä 7 sopimuksen toiminnallisuuden *ContractCreator*-objektiin käyttäen Truffle-sovelluskehyksen muodostamaa artifact-tiedostoa. *ContractCreator*-objekti alustetaan rivillä 8 käyttämään paikallisen Ethereum-asiakassovelluksen JSON-RPC-rajapintaa. Jotta POST-metodireitissä voidaan kutsua Truffle-migraatiossa lähetetyn *ContractCreator*-sopimuksen funktioita, täytyy se ensiksi luoda *ContractCreator*-objektin sisältämällä *deployed*-metodilla. *Deployed*-metodi palauttaa uuden instanssin kyseisestä so-

```
1  const Web3 = require('web3');
2  const contract = require('truffle-contract');
3  const web3 = new Web3(
4    new Web3.providers.HttpProvider('http://localhost:8545'));
5  const contractCreatorArtifacts =
6    require('../truffle/build/contracts/ContractCreator.json');
7  const ContractCreator = contract(contractCreatorArtifacts);
8  ContractCreator.setProvider(
9    new Web3.providers.HttpProvider('http://localhost:8545'));
10
11 router.post('/', (req, res) => {
12   let contractCreatorInstance;
13   const employeeName = web3.fromAscii(req.body.employeeName);
14   const employeeAddr = req.body.employeeAddr;
15   const lastAccTime = moment(req.body.lastAccTime).unix();
16
17   ContractCreator.deployed()
18     .then((instance) => {
19       contractCreatorInstance = instance;
20       Promise.all([
21         contractCreatorInstance
22           .createEmplyoymentContract(employeeAddr, employeeName,
23             web3.toBigNumber(lastAccTime))
24       ])
25         .then(([result]) => {
26           let addrOfNewContract;
27           for (let i = 0; i < result.logs.length; i += 1) {
28             const log = result.logs[i];
29             if (log.event === 'NewContract') {
30               addrOfNewContract = log.args.addrOfContract;
31               break;
32             }
33           }
34           res.json({ address: addrOfNewContract });
35         });
36     }).catch((err) => {
37       console.log(err);
38     });
39 });
```

Ohjelma 4.4 *Rajapinnan POST-metodireitti.*

pimuksesta käyttäen *artifact*-tiedostoon määriteltyä sopimustilin osoitetta. Uutta muodostettua instanssia voidaan käyttää sopimustilillä olevien funktioiden kutsumiseen. Rivillä 22 kutsutaan *ContractCreator*-sopimuksen *createEmploymentContract*-funktiota käyttäen POST-reitille tulleita kyselyparametreja. Funktiokutsusta muodostuu transaktio, joka lähetetään sovelluksen työnantajan Ethereum-tililtä Ethereum-verkossa olevalle *ContractCreator*-sopimustilille. Funktiokutsussa olevien parametrien tulee olla *artifact*-tiedostossa olevan ABI-rajapintakuvauksen määrittelemässä muodossa. Esimerkiksi työntekijän nimi pitää antaa funktiolle hek-

Network

Current network: An unknown network.

Current provider: Metamask

Current account: 0x22d491bde2303f2f43325b2108d26f1eaba1e32b

Emplyoment Contract

Contract address: 0x5cb1848a868b67c6e8d2719647ffe6c092a64ebd

Data

Employee address: 0x22d491bde2303f2f43325b2108d26f1eaba1e32b

Employee name: John Doe

Contract created: 27/08/2017 12:05:45

Acceptance deadline: 10/09/2017 00:00:00

Contract accepted: false

Accept

Kuva 4.5 *Työsopimuksen hyväksymisnäkymä.*

sadesimaalikoodattuna merkkijonona. Kun funktiokutsussa muodostunut transaktio hyväksytään Ethereum-lohkoketjuun, se palauttaa *result*-objektin, joka sisältää funktiokutsusta muodostuneen transaktion lisäksi transaktiossa muodostuneet lokitapahtumat. Lokitapahtumista saadaan haettua juuri muodostuneen *EmploymentContract*-sopimuksen osoite, joka palautetaan POST-metodireitin kutsujalle rivillä 34.

4.3 Asiakas

Sovellukseen kuuluva asiakassovellus on toteutettu käyttäen JavaScript-sovelluskehystä nimeltään Vue.js. Sovelluskehys mahdollistaa samojen palvelimella käytössä olevien JavaScript-kirjastojen käyttämisen myös asiakassovelluksessa. Asiakassovellus tarjoaa käyttöliittymän, jolla uusia älytyösopimuksia voidaan luoda kutsumalla palvelimella olevaa rajapintaa. Edellisen lisäksi asiakassovelluksella on näkymä, jolla työntekijä pystyy hyväksymään hänelle luodun työsopimuksen suoraan käyttäen MetaMask-laajennusta. Kuvassa 4.5 on esitetty asiakassovelluksessa oleva työsopimuksen hyväksymisnäkymä. Hyväksymisnäkymä ei kommunikoi sovelluksen palvelimen kanssa, vaan on suoraan yhteydessä Ethereum-sovellusalustaan käyttäen MetaMask-laajennusta.

Kuva 4.6 MetaMask-laajennoksen muodostama transaktionhyväksymisnäkymä.

Halutun älytyösopimuksen Ethereum-osoite toimitetaan hyväksymisnäkymälle reittiparametrina. Hyväksymisnäkymän toteutuksessa käytetään palvelimella olevan POST-metodireitin tavoin *truffle-contract*-kirjastoa, jolla luodaan työsopimuksen toiminallisuutta kuvaava *EmploymentContract*-objekti. Jotta näkymästä voidaan suoraan kommunikoida halutun työsopimuksen kanssa, näkymässä luodaan *EmploymentContract*-sopimuksesta instanssi käyttäen *EmploymentContract*-objektin *at*-metodia, jolle annetaan parametriksi reittiparametrina saatu Ethereum-osoite. Tätä instanssia käyttäen sopimuksen tilamuuttujien tiedot saadaan luettua. Tiedot on esitetty kuvan 4.5 *Data*-otsikon alla.

Google Chrome -selaimeen saatavan MetaMask-laajennuksen toiminta perustuu sivulle injektoitavaan *web3*-objektiin. Tätä käyttäen hyväksymisnäkymässä luotava *EmploymentContract*-objekti alustetaan *setProvider*-metodilla, jolla objekti asetetaan kommunikoimaan MetaMask-laajennoksen kanssa. Jos sivulle ei injektoida *web3*-objektia, alustetaan *EmploymentContract*-objekti käyttämään oletusarvoisesti paikallista Ethereum-asiakassovellusta.

Kuvassa 4.6 on esitetty MetaMask-laajennuksen muodostama transaktionhyväksymisnäkymä, joka näytetään, kun työntekijä haluaa hyväksyä työsopimuksen. MetaMask muodostaa älytyösopimukselle lähetettävän transaktion, jossa kutsutaan sen *acceptContract*-funktiota. Koska kyseiselle funktiolle ei ole sopimuskoodissa määri-

telty avainsanaa *payable*, tunnistaa MetaMask transaktiossa lähetettävien etherien määräksi 0. Hyväksymällä transaktio *submit*-painikkeella MetaMask lähettää transaktion Ethereum-verkkoon käyttäen MetaMaskin omistuksessa olevia Ethereum-asiakaspääteohjelmia. Kun transaktio on hyväksytty osaksi lohkoketjua, päivittyy älytyösopimukselle tieto sen hyväksymisestä. Tiedon päivittyminen voidaan varmistaa käyttäen kuvassa 4.5 esitettyä työsopimuksen hyväksymisnäkymää, jossa esitetään hyväksymisen jälkeen tieto siitä, milloin sopimus on hyväksytty.

5. ÄLYSOPIMUSTEN OHJELMOINTIKÄYTÄNNÖT JA TOTEUTETUN SOVELLUKSEN ARVIOINTI

Tässä luvussa käsitellään älysopimusten ja niitä hyödyntävien sovellusten tietoturvallisuutta ja ohjelmointikäytäntöjä. Lisäksi luvussa arvioidaan tässä työssä toteutetun sovelluksen kehitysprosessia ja siinä käytettyjä työkaluja.

5.1 Ohjelmointikäytännöt ja haavoittuvuudet

Älysopimusten ohjelmointi on haastavaa, koska Ethereum-sovellusalustaa hyödyntävien sovelluksien toteuttaminen on ohjelmistokehityksen kannalta uutta. Tästä syystä on luonnollista, että ohjelmointikäytännöt muokkaantuvat vielä uusien parhaiden käytäntöjen myötä [26]. Yleisellä tasolla älysopimusten ohjelmointi on rinnastettavissa sulautettujen järjestelmien ohjelmointiin ja kriittisten finanssipalveluiden toteuttamiseen, joissa vikasietoisuus on tärkeässä roolissa [26]. Älysopimuksiin kohdistuvat muutokset ovat myös vaikeasti toteutettavissa ilman sitä mahdollistavaa arkkitehtuuria. Vikasietoisuus edellyttää, että älysopimusten toteuttamisessa ei varauduta ainoastaan tunnettuihin haavoittuvuuksiin, vaan myös vielä löytämättömiin haavoittuvuuksiin [26].

Älysopimukset voivat varautua odottamattomiin virheisiin ja vielä löytämättömiin haavoittuvuuksiin noudattamalla niin sanottua *circuit breaker* -suunnittelumallia [26]. Mallia noudattamalla sopimusten tulee sisältää toiminnallisuus, jonka avulla sopimuksen muuta toiminallisuutta voidaan rajoittaa, ettei virheen tai haavoittuvuuden sisältämää sopimuskoodia voida suorittaa. Esimerkiksi jos sopimuksen sisältämässä funktiossa löytyy virhe, missä ethereitä siirretään toiselle sopimukselle, voidaan kyseisen funktion suoritus estää tarkastelemalla sopimuksella olevaa tilamuuttujaa, joka kuvaa onko funktion suorittaminen turvallista. Kyseistä tilamuuttujaa voi esimerkiksi sopimuksen hallitsija muuttaa vastaamaan ajantasaista tietoa löydetyistä haavoittuvuuksista. Tämä vaatii kehittäjiltä jatkuvaa Ethereumiin liittyvien uutisten seuraamista, jotta reagointi uusien haavoittuvuuksien löytyessä on tarvitta-

van nopeaa. Sopimuskoodissa olevien virheiden määrää voi myös vähentää tekemällä älysopimusten logiikasta mahdollisimman yksinkertaista tunnistamalla sovelluksesta ne toiminnot, jotka ovat tarpeellisia suorittaa hajautetusti lohkoketjupohjaisissa sopimuksissa [26].

Sopimusten testaaminen ennen älysopimuksia hyödyntävän sovelluksen tuotantokäyttöön ottamista on yleisesti tärkeää [26]. Esimerkiksi Truffle-sovelluskehys mahdollistaa Solidityllä ohjelmoitujen sopimusten automaattisen testaamisen käyttäen sekä JavaScriptiä että Solidityä. Kaikki sopimuksiin ohjelmoitu toiminnallisuus on syytä testata käyttäen automaattisesti suoritettavia testejä, jotta mahdolliset virheet havaitaan kehityksen varhaisessa vaiheessa. Testauksessa kannattaa myös hyödyntää julkisia Ethereum-testiverkkoja, joissa sopimuksia voidaan suorittaa oikeissa lohkoketjuympäristöissä ilman, että käytettäisiin oikeita rahanarvoisia ethereitä [26]. Bug bounty -ohjelmat, joissa virheiden löytämisestä tarjotaan palkkio niiden löytäjille, ovat myös hyvä tapa saada ulkopuolisia tahoja osaksi testaamista [26].

Solidityssä on mahdollista käsitellä virhetilanteita heittämällä poikkeuksia (*throw exception*). Yleisesti Solidityllä kirjoitetuissa funktioissa on hyvä tehdä tarkistuksia niin funktioiden alussa kuin lopussakin mahdollisten virhetilanteiden tunnistamiseksi. Funktion alussa, ennen varsinaisen funktiologiikan suoritusta, on syytä tarkistaa, että funktiolle toimitetut parametrit ovat odotetussa muodossa, ja että funktion suorittamiseen tarvittavien tilamuuttujien arvot ovat hyväksyttäviä. Esimerkiksi jos funktion suorittaminen halutaan rajata vain sopimuksen omistajalle, on tarkistettava vastaako transaktion lähettäjä sopimuksen tilamuuttujaksi asetettua omistajaa. Vastaavasti funktion lopussa on tarpeellista varmentaa sopimuksen tila, vastaako se funktiossa tehtyjä muutoksia. Poikkeuksia voi syntyä myös esimerkiksi tilanteissa, joissa suoritettavalta sopimuskoodilta loppuu transaktiossa määritelty polttoaine (*gas*) tai maksimimäärä sallittuja ulkoisesti kutsuttavia funktioita ylittyy. Solidityssä ei ole mahdollista varautua mahdollisiin poikkeuksiin niitä sieppaamalla (*catch exception*), vaan poikkeuksen tullessa sopimuskoodin suorittaminen lopetetaan ja kaikki tilamuutokset kumotaan. Ennen Solidityn versiota 0.4.10 ainoa mahdollinen tapa poikkeuksen heittämiselle oli avainsana *throw*, jota käyttäen kaikki transaktiossa määritelty polttoaine kulutettiin loppuun, tilamuutokset kumottiin ja kutsujalle palautettiin yksinkertainen virheviesti, jota ei ollut mahdollista muuttaa kuvaamaan missä ja miksi virhe syntyi. Tästä syystä käyttäen *throw*-avainsanaa, mahdollisen virheviestin välitys kutsujalle tuli tehdä käyttäen muita tapoja kuten Solidityn *event*-toiminnallisuutta. Tämä teki sopimusten virhetilanteiden tunnistamisesta ja testaamisesta hankalaa, koska kaikki sopimuskoodissa heitetyt poikkeukset näkyivät sopimuskoodin kutsujalle samalla virhekoodilla.

Solidityn versiossa 0.4.10 poikkeuksien heittämiseen esiteltiin seuraavat funktiot: *assert()*, *require()* ja *revert()* [27]. Näiden funktioiden avulla Solidityllä kirjoitettusta sopimuskoodista on helpommin tunnistettavissa ja ymmärrettävissä, millaisiin virhetilanteisiin sopimuskoodi on varautunut. Funktiot myös parantavat itse virhetilanteiden käsittelyä Ethereumiin tulevan *Metropolis*-päivityksen myötä. Suurin muutos virheiden käsittelyssä tulee olemaan *revert()*-funktion toiminta. *Metropolis*-päivityksen jälkeen *revert()* tulee toimimaan kuten *throw*-avainsana, mutta se palauttaa käyttämättömäksi jääneen polttoaineen funktion kutsujalle. Päivityksen myötä funktioihin voidaan myös määritellä kutsujalle palautuva virheviesti, mikä parantaa virheiden tunnistamista ja testaamista [27].

Haavoittuvuuksilla tarkoitetaan sovelluksessa olevia puutteita, joita hyödyntämällä ulkopuolinen taho voi vaarantaa sovelluksessa olevan tiedon saatavuuden, luottamuksellisuuden ja eheyden. Ethereum-sovellusalustaan ja älysopimuksiin sovellettuna tiedon saatavuudella voidaan tarkoittaa, että Ethereumiin lähetetyt sopimukset ovat käyttäjien saatavilla. Saatavuuteen voivat vaikuttaa esimerkiksi Ethereum-verkkoon kohdistuvat hyökkäykset, jotka rajoittavat sen suorituskykyä. Luottamuksellisuus kuvaa tiedon rajoittamista vain sallituille käyttäjille. Ethereum-sovellusalustan kannalta luottamuksellisuutta on hankala arvioida, koska käytännössä kaikki sopimustileille asetettu tieto on julkisena tavukoodina kaikkien saatavilla. Älysopimuksiin asetettuja rajoitteita tiedon saatavuudesta voidaan kuitenkin arvioida esimerkiksi tarkastelemalla, onko sopimuskoodissa käytetty tiedon kannalta sopivia suojausmääreitä. Älysopimusten eheyttä vaarantavat haavoittuvuudet ovat yleisempiä verrattuna muihin haavoittuvuuksiin. Eheydellä tarkoitetaan tiedon luotettavuutta. Esimerkiksi älysopimuksessa voi olla sopimuskoodissa määriteltynä, kuinka kyseisen sopimustilin hallussa olevia ethereitä voidaan käyttää. Jos hyökkääjä pystyy ohittamaan tai muuttamaan tätä logiikkaa, on älysopimuksen eheys vaarantunut.

Älysopimuksiin liittyvät haavoittuvuudet voidaan jaotella kolmeen eri kategoriaan riippuen siitä, millä tasolla haavoittuvuus ilmenee [5]. Haavoittuvuudet voivat ilmentyä Solidityllä kirjoitetussa sopimuskoodissa, Ethereum-virtuaalikoneen tavukoodissa tai itse lohkoketjun toteutuksessa. Älysopimuksia hyödyntävän sovelluksen kehittämisen kannalta näistä on tärkeintä ymmärtää Solidityn sopimuskoodiin liittyvät haavoittuvuudet.

Älysopimuksista löytyvät haavoittuvuudet ovat yleensä seurausta ohjelmointivirheistä ja sopimusten suoritusympäristön puutteellisesta ymmärtämisestä [26]. Hyvänä esimerkkinä toimii haavoittuvuus [70], joka mahdollisti sen, että hyökkääjä olisi voinut saada haltuunsa yli 150 000 etheriä. Haavoittuvuus johtui ohjelmointi-

virheestä, joka olisi ollut mahdollista löytää tarvittavan laajalla testauksella. Kyseinen haavoittuvuus esiintyi Ethereum-asiakassovellus Parityn muodostamassa *multi-signature wallet* -sopimuksessa, joka mahdollisti etherien turvallisemman säilyttämisen verrattuna normaaliin yhden yksityisen avaimen hallinnassa olevaan tiliin tai sopimukseen. *Multi-signature wallet* -sopimus vaatii useamman kuin yhden Ethereum-tilin hyväksynnän ethereiden käyttöön, mikä estää niiden mahdollisen väärinkäytön, jos niiden omistajuus käsittää useita toimijoita.

Haavoittuvuuden sisältämä *multi-signature wallet* -sopimus hyödynsi Solidityn ominaisuutta, joka mahdollistaa sopimuskoodin jakamisen niin sanottuihin kirjastoihin (*library*) [70]. Kirjastot ovat itsenäisiä sopimustilejä, joiden sopimuskoodia on tarkoitus kutsua toisesta sopimuksesta käyttäen Solidityn *delegatecall*-kutsua [61]. Kyseinen kutsu mahdollistaa sopimuskoodin dynaamisen lataamisen toisesta sopimustilistä niin, että kyseinen sopimuskoodi suoritetaan käyttäen kutsujan kontekstia. Kontekstilla tarkoitetaan sopimustilin tietoja, joihin kuuluvat esimerkiksi tilin osoite ja ethereiden määrä. Lompakkosopimuksen alustusfunktio, jossa asetettiin lompakon omistajien Ethereum-tilien osoitteet, oli sijoitettu tällaiseen kirjastoon ja sitä kutsuttiin itse lompakkosopimuksen rakentajassa [70]. Lompakkosopimus sisälsi myös *fallback*-funktion, joka ohjasi kaikki sopimukselle kohdistetut tunnistamattomat funktiokutsut *delegatecall*-kutsulla sen käyttämään kirjastoon [70]. Tämä mahdollisti tilanteen, jossa hyökkääjä rakensi sopivan datan sisältämän transaktion, jolla se pystyi suorittamaan sopimuksen rakentajassa kutsutun alustusfunktion uudestaan [70]. Hyökkääjä pystyi siis asettamaan itsensä lompakkosopimuksen omistajaksi, mikä mahdollisti sopimuksella olevien ethereiden siirtämisen hyökkääjän määrittelemälle Ethereum-tilille.

Haavoittuvuus olisi ollut mahdollista estää sisällyttämällä lompakkosopimuksen alustusfunktioon tarkistus, joka estäisi lompakon omistajien uudelleen asettamisen. Toinen, ja myös luonnollisempi, tapa olisi ollut alustusfunktion sisällyttäminen suoraan lompakkosopimukseen sopivalla suojausmääreellä. Tunnistamattomien funktiokutsujen suora ohjaaminen kirjastona toimivalle sopimustilille on huono tapa hallita sopimuksen toimintaa [70]. Jos älysopimuksissa halutaan hyödyntää kirjastojen mahdollistamaa modulaarisuutta, sopimuksiin on hyvien käytäntöjen mukaisesti määriteltävä kaikki sallitut ulkoisesti kutsuttavat funktiot, joiden vastaavuus tehtäviin funktiokutsuihin tarkastetaan [70].

Älysopimuksissa olevien funktioiden suorittaminen alkaa aina Ethereumiin lähetettävästä transaktiosta. Jos sopimuksesta tehdään ulkoisia funktiokutsuja toisiin sopimuksiin, on mahdollista, että alkuperäistä funktiota kutsutaan uudestaan ulkoisesta sopimuksesta (*reentracy*) [5]. Tämä mahdollisti vuoden 2016 kesällä The DAO

-sovellukseen kohdistuneen hyökkäyksen [5]. The DAO -sopimus sisälsi funktion, joka mahdollisti kutsujalle kuuluvien ethereiden siirtämisen kutsujan omistamalle Ethereum-tilille. Funktiossa ollut haavoittuvuus oli funktion logiikassa, jossa vasta ethereiden siirron jälkeen asetettiin tieto siitä, että kutsujalle kuuluvat etherit oli siirretty. Tämä mahdollisti tilanteen, jossa ethereiden siirto kutsujan omistamalle Ethereum-tilille, joka tässä tapauksessa oli älysopimus, kutsuikin alkuperäistä siirtofunktiota uudestaan, jolloin siirtofunktiossa oletettiin, että kutsujalle kuuluvia ethereitä ei oltu vielä siirretty [5]. Näin hyökkääjä sai siirrettyä The DAO -sopimukselta itselleen enemmän ethereitä kuin oli tarkoitus. Haavoittuvuus olisi ollut mahdollista estää asettamalla kutsujalle kuuluvien ethereiden määrä jo ennen siirtoa vastaamaan tilannetta, jossa etherit olivat siirtyneet, jolloin siirto olisi ollut mahdollista tehdä vain kerran.

5.2 Huomioita toteutetusta sovelluksesta

Työsopimuksen vienti älysopimusten muotoon vaatii asiantuntijuutta työsopimuksiin liittyvistä sopimuskäytännöistä ja säädöksistä. Tässä työssä toteutetun sovelluksen on tarkoitus havainnollistaa, kuinka sopimusehdoiltaan yksinkertainen työsopimus mallinnetaan älysopimuksella. Tarkempien sopimusehtojen toteuttaminen ja mallintaminen älysopimuksilla mahdollistaisi älytyösopimuksen kokonaisvaltaisemman toimivuuden verrattuna kirjallisten sopimusten mahdollistamaan toimivuuteen. Esimerkiksi työsopimukseen tehtävät muutokset olisi mahdollista toteuttaa myös älysopimuksiin, toteuttamalla älytyösopimusten ympärille muita älysopimuksia, jotka sanelevat työsopimuksiin kohdistuvat sallitut muutokset. Nämä sopimukset, jotka sisältäisivät työsopimukseen tehtävien muutosten reunaehdot, voisivat muutoksesta riippuen vaatia niin työntekijän kuin työnantajan suostumuksen, mikä käytännössä vastaisi työsopimuksen hyväksymistä.

Toteutetussa sovelluksessa työsopimusten luominen tapahtuu käyttäen palvelimella olevaa Ethereum-tiliä. Tämä mahdollistaa sen, että jokainen, jolla on pääsy työsopimusten luomiseen tarkoitettuun käyttöliittymään, voi luoda tarvittaessa uuden työsopimuksen. Tätä voitaisiin rajoittaa muokkaamalla *ContractCreator*-sopimusta lisäämällä sille useampi kuin yksi omistaja. Kun uusi työsopimus luotaisiin, tarvitsisi kyseinen sopimus vielä muilta työnantajan henkilöiltä varmennuksen työsopimuksen lisäämiseen, ennen kuin se luotaisiin lohkoketjuun työntekijän hyväksyttäväksi. Tapa muistuttaisi siis edellä esitellyn *multi-signature wallet* -sopimuksen toimintaa. Jatkokehityksenä myös työntekijän palkanmaksu voitaisiin tuoda älytyösopimukseen, jolloin työntekijä saisi palkan ethereinä.

Toteutettu sovellus havainnollistaa myös kuinka Ethereum-sovellusalustaa hyödyntävän sovelluksen käyttöliittymä on mahdollista toteuttaa. Sovellus toteutettiin websovelluksena, jonka käyttämiseen tarvitaan vain verkkoselain. Toteutustekniikkaa tukevat web-sovellusten toteuttamiseen olemassa olevat sovelluskehykset, joita hyödyntämällä myös sovelluksen käyttöliittymä voidaan rakentaa tehokkaasti käyttäen jo toimiviksi havaittuja käytäntöjä.

5.3 Kehitystyökalujen arviointi

Älysopimusten ohjelmointia ja toteuttamista Ethereum-sovellusalustalle helpottaa aiempi ohjelmointikokemus. Tämän lisäksi lohkoketjujen sekä niiden toiminnan ja ominaisuuksien tunteminen auttaa ymmärtämään, millaisia ongelmia älysopimuksilla voidaan ratkaista. Solidity-ohjelmointikieli muistuttaa oliopohjaisia ohjelmointikieliä, mutta sen suoritusympäristö Ethereum-virtuaalikoneessa tuo kieleen yksityiskohtia, jotka ovat ymmärrettävissä lohkoketjujen toiminnan kautta. Nämä yksityiskohdat on määritelty ja selitetty Solidityn dokumentaatiossa, joka auttaa ymmärtämään Solidityn suoritusympäristön toimintaa.

Ethereum on kirjoitushetkellä[1] suosituin lohkoketjupohjainen järjestelmä, joka mahdollistaa älysopimusten suorittamisen. Ethereumin kehitys on kuitenkin vielä kesken ja uusia ominaisuuksia tuodaan järjestelmään *hard fork* -päivitysten muodossa. Tämä mahdollistaa myös uusien operaatiokoodien lisäämisen Ethereum-virtuaalikoneeseen, mikä parantaa Solidity-ohjelmointikielen edellytyksiä sopimuskoodin toteuttamiseen. Solidity on saanut kritiikkiä sen ominaisuuksista ja suunnitteluratkaisuista. Soliditystä puuttuu esimerkiksi kokonaan muista oliopohjaisista ohjelmointikielistä tutut operaatiot *string*-tietotyypin käsittelyyn. Käytännössä kaikki Solidityn tietotyypit ovat 32 tavun mittaisia, mikä voi tietyissä tapauksissa aiheuttaa ongelmia niiden käsittelyssä [50]. Solidity on avoimen lähdekoodin projekti, jonka kehitykseen voi jokainen osallistua [33]. Aktiivinen kehitysyhteisö takaa, että Solidity ja sen ominaisuudet kehittyvät jatkuvasti. Soliditylle on myös kehitteillä Viper-niminen, vaihtoehtoinen, ohjelmointikieli, jolla sopimuskoodia voidaan kirjoittaa [34]. Sen suunnitteluratkaisuissa etusijalla ovat turvallisuus, ohjelmakoodin ja kääntäjän yksinkertaisuus sekä ohjelmakoodin luettavuus, jotta sopimuskoodi on mahdollisimman helposti auditoitavissa. Viper ei esimerkiksi tule sisältämään Solidityssä olevia ominaisuuksia kuten periytymistä tai ikuisia silmukoita, koska ne heikentävät sopimuskoodin luettavuutta ja mahdollistavat haavoittuvuuksien syntymisen.

[1] 23.10.2017

Truffle-sovelluskehys helpottaa älysopimuksia hyödyntävän sovelluksen kehittämistä huomattavasti ja mahdollistaa Soliditylla ohjelmoitujen sopimusten käyttämisen ohjelmakoodissa tehokkaasti sen tuomien abstraktioiden avulla. Se hyödyntää JavaScript-kirjastoille tarkoitettua pakettienhallintatyökalua nimeltään *npm*, mikä tekee sen käyttöönottamisesta helppoa. Solidity-ohjelmakoodin kääntäminen Ethereum-virtuaalikoneen tavukoodiksi vaatii sille tarkoitetun kääntäjän, jonka käyttö sellaisenaan on monimutkaista. Truffle sisältää Solidity-kääntäjän tehden sopimusten kääntämisestä ja lähettämisestä Ethereum-verkkoon yksinkertaista abstraktoimalla nämä tehtävät muutaman komennon taakse. Tämä nopeuttaa älysopimusten kehittämistä ja testaamista. Trufflen tekemät abstraktiot Solidityllä ohjelmoiduista sopimuksista helpottavat myös niihin liittyvien Ethereum-tilien osoitteenhallintaa, mikä tekee niitä hyödyntävän sovelluksen toteuttamisesta tehokasta ja yksinkertaisempaa.

Älysopimuksia hyödyntävän sovelluksen kehittämisessä on myös tarve suorittaa sovelluksen hyödyntämiä sopimuksia. *Testrpc* mahdollistaa älysopimusten suorittamisen ilman varsinaista Ethereum-asiakassovellusta. Se simuloi paikallisesti Ethereum-sovellusalustan toimintaa, mikä säästää kehitykseen kuluvaa aikaa ja resursseja verrattuna kehityksessä käytettävään asiakassovellukseen, joka on yhteydessä varsinaiseen Ethereum-verkkoon. Vaihtoehtoisesti kehitystä voisi myös tehdä suoraan Ethereum-verkossa olevissa testiverkoissa, mikä on kehityksen myöhemmissä vaiheissa suotavaa.

6. YHTEENVETO

Lohkoketjuteknologialla tarkoitetaan vertaisverkossa toimivaa hajautettua tietokantaa, jonka muuttumattomuus on turvattu kryptografisin keinoin. Se mahdollistaa digitaalisen valuutan olemassaolon ilman keskitettyä tahoa, jonka tehtävänä olisi varmentaa maksujen oikeellisuus. Lohkoketju muodostuu toisiinsa linkitetyistä lohkoista, jotka sisältävät lohkoketjuun tehtyjä muutoksia kuvaavia transaktioita. Lohkoketjun konsensusta pidetään yllä erilaisilla konsensusalgoritmeilla, jotka määrittävät säännöt, kuinka uusia lohkoja voidaan lohkoketjuun lisätä. Uusien lohkojen muodostajia kutsutaan louhijoiksi, joiden tehtävänä on kerätä ja varmentaa uuteen lohkoon sisällytettäviä transaktioita. Ensimmäinen lohkoketjuteknologiaa hyödyntävä järjestelmä oli Bitcoin, joka on edelleen käytössä oleva digitaalinen rahajärjestelmä. Bitcoiniin toteutettu lohkoketju toimi myös tässä työssä esimerkkinä siitä, kuinka lohkoketjuteknologia on teknisesti toteutettu.

Ethereum-sovellusalusta vie Bitcoinissa toteutetun lohkoketjuteknologian ideaa pidemmälle mahdollistamalla älysopimusten suorittamisen lohkoketjupohjaisessa ympäristössä. Älysopimukset ovat itseään suorittavia ohjelmia, jotka sisältävät ohjelmakoodilla kirjoitettuja sopimusehtoja. Lohkoketjuteknologian avulla älysopimuksiin kuvattu sopimuspohjainen toiminta ei vaadi kolmatta osapuolta varmentamaan sopimusehtojen noudattamista.

Tässä työssä toteutettiin myös sovellus, joka mahdollistaa työsopimusten solmimisen työnantajan ja työntekijän välillä käyttäen Ethereum-sovellusalustalla suoritettavia älysopimuksia. Sovellus toteutettiin web-sovelluksena hyödyntäen asiakas-palvelin-mallia, jonka lisäksi asiakas on suoraan yhteydessä Ethereum-sovellusalustaan käyttäen Google Chrome -verkkoselaimeen saatavilla olevaa MetaMask-laajennusta. Ethereum-sovellusalustan hyödyntäminen web-sovelluksessa on luontevaa, koska siihen tarkoitetut kehitystyökalut on toteutettu käyttäen web-teknologioita ja ne on suunnattu hyvin integroitaviksi osiksi erilaisia web-sovelluskehyksiä.

Älysopimusten kehittämisessä on hyödyllistä ymmärtää niiden suoritusympäristön, lohkoketjuteknologian, ominaisuudet ja toiminnallisuus mahdollisten haavoittuvuuksien ja erilaisten virhetilanteiden varalta. Lohkoketjuteknologia ja älyso-

pimuksiin tarkoitetut ohjelmointikielet ovat vielä luonteeltaan uutta teknologiaa, minkä vuoksi parhaat ohjelmointikäytännöt muokkaantuvat jatkuvasti. Yleisesti älysopimusten toteuttamisessa on syytä kiinnittää huomioita niiden vikasietoisuuteen ja kattavaan testaamiseen ennen tuotantokäyttöön ottamista. Myös Ethereum-sovellusalustaan liittyvän kehityksen ja uutisoinnin seuraaminen auttaa älysopimusten kehittämistä parantamalla ymmärrystä Ethereum-sovellusalustan uusista ominaisuuksista ja mahdollisesti löydetyistä haavoittuvuuksista.

Älysopimusten yleistyessä ratkaisemaan todellisia sopimuspohjaisen toiminnan ongelmia, on niiden teknisen toteutuksen kannalta mietittävä, onko lohkoketjuteknologia vielä valmis laajaan käyttöön. Ratkaistavia ongelmia on vielä muun muassa uusien konsensusalgoritmien käyttöönotossa ja lohkoketjujen skaalautuvuudessa, jotta teknologia saadaan laajemman käyttäjäkunnan tehokkaaseen käyttöön. Ethereum-sovellusalusta tulee saamaan vielä seuraavien vuosien aikana merkittäviä päivityksiä, joilla näitä ongelmia pyritään ratkaisemaan. Älysopimusten käyttöön liittyy myös lainsäädännöllisiä kysymyksiä, joihin on pyrittävä vastaamaan nopeasti lohkoketjuteknologian kehittyessä.

LÄHTEET

[1] “Bitcoin Unlimited,” Saatavissa (viitattu 31.1.2017): https://www.bitcoinunlimited.info/.

[2] “NodeCounter.com,” Saatavissa (viitattu 31.1.2017): http://nodecounter.com/.

[3] A. M. Antonopoulos, *Mastering Bitcoin: unlocking digital cryptocurrencies.* O’Reilly Media, Inc., 2014.

[4] Antony Lewis, “A Gentle Introduction To Blockchain Technology,” BraveNewCoin, Saatavissa (viitattu 7.12.2016): http://bravenewcoin.com/assets/Reference-Papers/A-Gentle-Introduction/A-Gentle-Introduction-To-Blockchain-Technology-WEB.pdf.

[5] N. Atzei, M. Bartoletti, and T. Cimoli, “A survey of attacks on ethereum smart contracts (sok),” in *International Conference on Principles of Security and Trust.* Springer, 2017, pp. 164–186.

[6] Bitcoin, “Bitcoin Core integration/staging tree,” Saatavissa (viitattu 14.11.2016): https://github.com/bitcoin/bitcoin/blob/master/src/chainparams.cpp.

[7] Bitcoin Project, “Bitcoin Core ,” Saatavissa (viitattu 23.1.2017): https://bitcoin.org/en/bitcoin-core/.

[8] ——, “Bitcoin Core version 0.13.1 released,” Saatavissa (viitattu 31.1.2017): https://bitcoin.org/en/release/v0.13.1.

[9] ——, “Bitcoin Developer Guide,” Saatavissa (viitattu 14.11.2016): https://bitcoin.org/en/developer-guide.

[10] ——, “Bitcoin-Qt version 0.8.0 released,” Saatavissa (viitattu 17.11.2016): https://bitcoin.org/en/release/v0.8.0.

[11] Bitcoin Stack Exchange, “How to redeem a basic Tx?” Saatavissa (viitattu 16.1.2017): http://bitcoin.stackexchange.com/questions/3374/how-to-redeem-a-basic-tx.

[12] Bitcoin Wiki, “FAQ,” Saatavissa (viitattu 14.11.2016): https://en.bitcoin.it/wiki/Help:FAQ.

[13] ——, “OP_CHECKSIG,” Saatavissa (viitattu 16.1.2017): https://en.bitcoin.it/wiki/OP_CHECKSIG.

[14] ——, “Pay to script hash,” Saatavissa (viitattu 30.1.2017): https://en.bitcoin.it/wiki/Pay_to_script_hash.

[15] ——, “Protocol documentation,” Saatavissa (viitattu 11.12.2016): https://en.bitcoin.it/wiki/Protocol_documentation.

[16] ——, “Transaction,” Saatavissa (viitattu 16.1.2017): https://en.bitcoin.it/wiki/Transaction.

[17] Bittiraha.fi, Saatavissa (viitattu 7.1.2017): https://bittimaatti.fi/.

[18] ——, “Bitcoinien osto,” Saatavissa (viitattu 7.1.2017): https://bittiraha.fi/osta.

[19] Blockchain Info, “Blockchain Size,” Saatavissa (viitattu 31.1.2017): https://blockchain.info/charts/blocks-size.

[20] Bloomberg L.P., “Why Half the World Doesn’t Have Bank Accounts,” Saatavissa (viitattu 22.4.2017): https://www.bloomberg.com/news/articles/2012-04-25/why-half-the-world-doesnt-have-bank-accounts.

[21] J. Bonneau, A. Miller, J. Clark, A. Narayanan, J. A. Kroll, and E. W. Felten, “Sok: Research perspectives and challenges for bitcoin and cryptocurrencies,” in *2015 IEEE Symposium on Security and Privacy*, 2015, pp. 104–121.

[22] V. Buterin, “Ethereum Research Update,” Saatavissa (viitattu 13.2.2017): https://blog.ethereum.org/2016/12/04/ethereum-research-update/.

[23] ——, “Hard Fork Completed,” Saatavissa (viitattu 13.2.2017): https://blog.ethereum.org/2016/07/20/hard-fork-completed/.

[24] CoinMarketCap, “CryptoCurrency Market Capitalizations,” Saatavissa (viitattu 22.4.2017): https://coinmarketcap.com/.

[25] ——, “Ethereum,” Saatavissa (viitattu 7.8.2017): https://coinmarketcap.com/currencies/ethereum/.

[26] ConsenSys, “Ethereum Contract Security Techniques and Tips,” Saatavissa (viitattu 10.9.2017): https://github.com/ConsenSys/smart-contract-best-practices.

[27] Consensys, “The use of revert(), assert(), and require() in Solidity, and the new REVERT opcode in the EVM,” Saatavissa (viitattu 21.9.2017): https://media.consensys.net/when-to-use-revert-assert-and-require-in-solidity-61fb2c0e5a57.

[28] ——, “Truffle,” Saatavissa (viitattu 20.8.2017): http://truffleframework.com/.

[29] ——, “Truffle Documentation,” Saatavissa (viitattu 20.8.2017): http://truffleframework.com/docs/.

[30] C. Dannen, *Introducing Ethereum and Solidity: Foundations of Cryptocurrency and Blockchain Programming for Beginners.* Apress, 2017.

[31] M. del Castillo, “The DAO Attacked: Code Issue Leads to $60 Million Ether Theft,” Saatavissa (viitattu 13.2.2017): http://www.coindesk.com/dao-attacked-code-issue-leads-60-million-ether-theft/.

[32] J. Dienelt, “Understanding Ethereum,” *New York, NY: CoinDesk*, 2016.

[33] Ethereum, “The Solidity Contract-Oriented Programming Language,” Saatavissa (viitattu 23.10.2017): https://github.com/ethereum/solidity.

[34] ——, “Viper,” Saatavissa (viitattu 23.10.2017): https://github.com/ethereum/viper.

[35] Ethereum Foundation, “Ether The crypto-fuel for the Ethereum network,” Saatavissa (viitattu 7.8.2017): https://ethereum.org/ether.

[36] ——, “Ethereum Project,” Saatavissa (viitattu 22.4.2017): https://www.ethereum.org/.

[37] Ethereum Homestead Documentation, “Account Types, Gas, and Transactions,” Saatavissa (viitattu 21.2.2017): http://ethdocs.org/en/latest/contracts-and-transactions/account-types-gas-and-transactions.html.

[38] ——, “History of Ethereum,” Saatavissa (viitattu 13.2.2017): http://ethdocs.org/en/latest/introduction/history-of-ethereum.html.

[39] ——, “The Ethereum Foundation,” Saatavissa (viitattu 13.2.2017): http://ethdocs.org/en/latest/introduction/foundation.html.

[40] Ethereum Network Status, Saatavissa (viitattu 21.10.2017): https://ethstats.net/.

[41] Ethereum Stack Exchange, “Ethereum block architecture,” Saatavissa (viitattu 14.8.2017): https://ethereum.stackexchange.com/questions/268/ethereum-block-architecture.

[42] ——, “How does Ethereum make use of bloom filters?” Saatavissa (viitattu 4.10.2017): https://ethereum.stackexchange.com/questions/3418/how-does-ethereum-make-use-of-bloom-filters.

[43] ——, “What is a DApp?” Saatavissa (viitattu 21.10.2017): https://ethereum.stackexchange.com/questions/383/what-is-a-dapp.

[44] ——, “What is the difference between a transaction and a call?” Saatavissa (viitattu 19.7.2017): https://ethereum.stackexchange.com/questions/765/what-is-the-difference-between-a-transaction-and-a-call.

[45] ——, “What proof of work function does Ethereum use?” Saatavissa (viitattu 21.10.2017): https://ethereum.stackexchange.com/questions/14/what-proof-of-work-function-does-ethereum-use.

[46] Ethereum Wiki, “Design Rationale,” Saatavissa (viitattu 14.8.2017): https://github.com/ethereum/wiki/wiki/Design-Rationale.

[47] ——, “White Paper,” Saatavissa (viitattu 21.2.2017): https://github.com/ethereum/wiki/wiki/White-Paper.

[48] Ethereumjs, “Testrpc,” Saatavissa (viitattu 20.8.2017): https://github.com/ethereumjs/testrpc.

[49] V. Gupta, “The Ethereum Launch Process,” Saatavissa (viitattu 13.2.2017): https://blog.ethereum.org/2015/03/03/ethereum-launch-process/.

[50] Hacker News, “Solidity has far worse problems than not being an advanced research language.” Saatavissa (viitattu 23.10.2017): https://news.ycombinator.com/item?id=14691212.

[51] J. Herrera-Joancomartí and C. Pérez-Solà, *Privacy in Bitcoin Transactions: New Challenges from Blockchain Scalability Solutions.* Springer International Publishing, 2016, pp. 26–44.

[52] H. Jameson, “FAQ: Upcoming Ethereum Hard Fork,” Saatavissa (viitattu 13.2.2017): https://blog.ethereum.org/2016/10/18/faq-upcoming-ethereum-hard-fork/.

[53] J. Katz and Y. Lindell, *Introduction to modern cryptography.* CRC press, 2008.

[54] K. Lauslahti, J. Mattila, and T. Seppälä, “Älykas sopimus – Miten blockchain muuttaa sopimuskäytäntöjä?” ETLA Raportit No 57.https://pub.etla.fi/ETLA-Raportit-Reports-57.pdf, 2016.

[55] N. G. Mankiw, *Principles of macroeconomics.* Cengage Learning, 2014.

[56] A. J. Menezes, P. C. Van Oorschot, and S. A. Vanstone, *Handbook of applied cryptography.* CRC press, 1996.

[57] Metamask, "Metamask," Saatavissa (viitattu 20.8.2017): https://metamask.io/.

[58] S. Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System," Saatavissa (viitattu 26.9.2016): https://bitcoin.org/bitcoin.pdf.

[59] G. W. Peters and E. Panayi, "Understanding modern banking ledgers through blockchain technologies: Future of transaction processing and smart contracts on the internet of money," *CoRR arXiv:1511.05740 [cs.CY]*, 2015.

[60] Solidity Documentation, "Contracts," Saatavissa (viitattu 19.3.2017): http://solidity.readthedocs.io/en/develop/contracts.html.

[61] ——, "Libraries," Saatavissa (viitattu 14.9.2017): https://solidity.readthedocs.io/en/develop/contracts.html#libraries.

[62] ——, "Solidity," Saatavissa (viitattu 6.3.2017): https://solidity.readthedocs.io/en/develop/.

[63] ——, "Special Variables and Functions," Saatavissa (viitattu 26.3.2017): http://solidity.readthedocs.io/en/develop/units-and-global-variables.html#special-variables-and-functions.

[64] ——, "Version Pragma," Saatavissa (viitattu 6.3.2017): https://solidity.readthedocs.io/en/develop/layout-of-source-files.html#version-pragma.

[65] ——, "What is the memory keyword? What does it do?" Saatavissa (viitattu 14.8.2017): http://solidity.readthedocs.io/en/develop/frequently-asked-questions.html#what-is-the-memory-keyword-what-does-it-do.

[66] N. Szabo, "Smart Contracts: Building Blocks for Digital Markets," Saatavissa (viitattu 21.10.2017): http://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart_contracts_2.html.

[67] D. Tapscott and A. Tapscott, *Blockchain revolution.* Portfolio Penguin, 2016.

[68] Tom's Hardware, "The Ethereum Effect: Graphics Card Price Watch," Saatavissa (viitattu 21.10.2017): http://www.tomshardware.com/news/ethereum-effect-graphics-card-prices,34928.html.

[69] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," Saatavissa (viitattu 5.2.2017): https://ethereum.github.io/yellowpaper/paper.pdf.

[70] Zeppelin Solutions, "The Parity Wallet Hack Explained," Saatavissa (viitattu 14.9.2017): https://blog.zeppelin.solutions/on-the-parity-wallet-multisig-hack-405a8c12e8f7.

LIITE A. ÄLYTYÖSOPIMUKSEN SOPIMUSKOODI

```
1  pragma solidity ^0.4.15;
2
3  contract EmploymentContract {
4      address public employerAddr;
5      address public employeeAddr;
6      bytes32 public employeeName;
7      uint public creationTime;
8      uint public acceptanceDeadline;
9      uint public acceptTime;
10
11     function EmploymentContract(address addrOfEmployee,
12         bytes32 employee, uint deadline)
13     {
14         employerAddr = msg.sender;
15         employeeAddr = addrOfEmployee;
16         creationTime = block.timestamp;
17         acceptanceDeadline = deadline;
18         employeeName = employee;
19     }
20
21     modifier onlyEmployee() {
22         require(msg.sender == employeeAddr);
23         _;
24     }
25
26     function acceptContract() onlyEmployee {
27         if (block.timestamp <= acceptanceDeadline) {
28             acceptTime = block.timestamp;
29         }
30     }
31
32     function isContractAccepted() constant returns(bool) {
33         if (acceptTime > 0) {
34             return true;
35         }
36         return false;
37     }
38 }
```