Tampereen yliopisto

Johannes Haapakoski

SUUNNITTELUMALLIT ETHEREUM-ÄLYSOPIMUKSISSA

Tieto- ja sähkötekniikan tiedekunta
Kandidaatintyö
Syyskuu 2019

TIIVISTELMÄ

Johannes Haapakoski: Suunnittelumallit Ethereum-älysopimuksissa
Design patterns in Ethereum smart contracts
Kandidaatintyö
Tampereen yliopisto
Tietotekniikan tutkinto-ohjelma
Syyskuu 2019

Ethereum on hajautettu järjestelmä, joka tarjoaa laskenta-infrastruktuurin, joka suorittaa ohjelmia, joita kutsutaan älysopimuksiksi. Älysopimukset ovat muuttumattomia tietokoneohjelmia, joiden suorittaminen on determinististä. Älysopimukset toimivat siis aina sillä tavalla kuin niiden logiikkaan on määritelty. Älysopimusten luonteen takia käyttäjät pystyvät luomaan erilaisia transaktioita toistensa välillä ilman, että heidän täytyy luottaa kehenkään kolmanteen osapuoleen tai toisiinsa. Ethereumille luontaisen kryptovaluutan Etherin avulla käyttäjät pystyvät välittämään arvoa toisillensa. Tämän työn tarkoitus on tutkia, millaisia suunnittelumalleja Ethereum-älysopimuksissa käytetään sekä mitä ongelmia ne ratkaisevat.

Älysopimusten muuttumattomuus luo haasteita niitä suunniteltaessa. Koska älysopimukset ovat muuttumattomia, niiden päivittäminen ei onnistu, mikäli tätä varten ei ole suunniteltu jotakin järjestelmää. Tässä työssä esitellään suunnittelumallit, joilla älysopimukseen pystytään luomaan sen logiikan päivittämiseen tarvittava toiminnalisuus. Tällaisia suunnittelumalleja kutsutaan ylläpitomalleiksi.

Mikäli älysopimuksen toiminta halutaan jossakin kohtaa pysäyttää, tulee suunnitteluvaiheessa myös tälle luoda jokin järjestelmä. Tässä työssä esitellyt elinkaarimallit tarjoavat käytäntöjä, miten älysopimuksen toiminta voidaan pysäyttää.

Älysopimukset usein sisältävät rahanarvoisia kryptovaluuttoja. Tämän takia pahanaikeisilla toimijoilla on suuri insentiivi etsiä haavoittuvaisuuksia älysopimuksista. Tässä työssä esitellään joitakin suunnittelumalleja, joita kannattaa hyödyntää, kun suoritetaan joitakin tavallisia toimintoja. Tällaisia suunnittelumalleja kutsutaan toimintomalleiksi, ja niiden avulla voidaan vähentää haavoittuvaisuuksia älysopimuksissa.

Älysopimuksiin tulee usein määritellä ehtoja, joiden perusteella älysopimus voi estää jonkin toiminnon suorittamisen. Tässä työssä esitellyt valtuutusmallit tarjoavat käytäntöjä, miten tällainen toiminnalisuus voidaan määritellä.

Ethereum mahdollistaa uusien kryptovaluuttojen helpon luomisen alustansa päälle. Tällaisten kryptovaluuttojen, joita kutsutaan tokeneiksi, kannattaa noudattaa Ethereumin tarjoamia standardeja, jotta ne voidaan helposti integroida valmiiksi rakennettuihin applikaatioihin, kuten lompakkoihin. Tässä työssä esitellään käytetyimmät token-standardit.

Avainsanat: Ethereum, älysopimus, suunnittelumalli, älysopimusten suunnittelumallit, lohkoketju

SISÄLLYSLUETTELO

1 JOHDANTO

Kryptovaluutat ovat yleisesti ottaen digitaalisia hyödykkeitä, joiden tarkoitus on toimia vaihdannan välineenä. Ensimmäinen kryptovaluutta Bitcoin julkaistiin vuonna 2009. Bitcoinin käyttötarkoitus on tarjota rahajärjestelmä, jonka avulla kaksi osapuolta pystyvät tekemään transaktioita ilman luotettua kolmatta osapuolta. [15]

Vuoden 2013 loppupuolella Bitcoin-intoilija Vitalik Buterin alkoi miettimään keinoja, miten Bitcoiniin voisi lisätä toiminnallisuutta. Tästä syntyi idea Ethereumiin. Ethreumin ensisijainen käyttötarkoitus ei ole toimia rahajärjestelmänä (vaikkakin siitä löytyy kaikki toiminnot mitä rahajärjestelmä tarvitsee), vaan tarjota maailmanlaajuisesti hajautettu laskenta-infrastruktuuri, joka suorittaa ohjelmia, joita kutsutaan älysopimuksiksi[1]. Tämän kandidaatintyön tarkoituksena on tutkia, millaisia suunnittelumalleja Ethereum-älysopimuksissa käytetään sekä mitä ongelmia ne ratkaisevat.

Luvussa 2 käydään läpi Ethereum-alustan toimintaperiaate, ja luvussa 3 käsitellään älysopimusten suunnittelumalleja. Suunnittelumalleista esitetään havainnolistavat esimerkkitoteutukset Solidity-ohjelmointikielellä.

2 ETHEREUM-ALUSTA

Ethereum kokonaisuutena on hajautettu transaktiopohjainen tilakone[26]. Ethereumissa on muisti, johon voi tallentaa sekä dataa, että koodia. Muutokset muistissa tallennetaan Ethereumin lohkoketjuun. Kuten yleiskäyttöinen tietokone, myös Ethereum voi ladata koodia sen tilakoneeseen ja ajaa sen. Kaksi suurinta eroa Ethereumin ja yleiskäyttöisen tietokoneen välillä ovat, että Ethereumin tilan muutoksia hallitsee konsensussäännöt ja että Ethereumin tila on hajautettu globaalisti usealle tietokoneelle.[1]

2.1 Tilit

Ethereumin tila koostuu objekteista, joita kutsutaan tileiksi. Tilejä on kahdenlaisia: ulkoisesti omistettuja tilejä ja sopimustilejä. Ulkoisesti omistettuja tilejä hallinoi se entiteetti, joka omistaa kyseisen tilin salatun avaimen ja sopimustilejä tilin sopimuskoodiin ohjelmoitu logiikka.[2] Ulkoisesti omistetut tilit ovat siis Ethereumin käyttäjien tilejä, ja sopimustilit ovat älysopimusten tilejä.

Tilit sisältävät neljä kenttää[2]:

1. Nonce-arvo
2. Tilin Ether-saldo
3. Tilin sopimuskoodi
4. Tilin muisti

Nonce-arvo on laskuri, jota kasvatetaan aina yhdellä, kun tililtä lähetetään transaktio. Tämän avulla voidaan pitää huolta siitä, että jokainen transaktio prosessoidaan vain kerran. Tilin Ether-saldo kertoo, kuinka paljon tilillä on Etheriä. Tilin sopimuskoodi ja muisti sisältyvät tiliin vain siinä tapauksessa, että tili on sopimustili.[2] Tilin sopimuskoodissa on määritelty tilin käyttämä logiikka. Sopimuskoodi sisältää käytännössä funktioita, joita voidaan kutsua transaktioiden avulla. Tilin muistiin on tallennettu arvot, joista älysopimus pitää kirjaa.

Ethereumissa käytetään julkisen avaimen salausmenetelmää ulkoisesti omistettujen tilien luomisessa. Ensiksi käyttäjä valitsee satunnaisen luvun väliltä 1 - 2^{256}. Tämä numero on käyttäjän salainen avain. Tästä salaisesta avaimesta lasketaan käyttäjän julkinen avain käyttämällä elliptisen käyrän kertolaskua. Elliptisen käyrän kertolasku on niin kutsuttu yhdensuuntainen funktio. Se on helppo tehdä yhteen suuntaan, mutta mahdoton

toiseen. Toisin sanoen käyttäjä pystyy helposti laskemaan julkisen avaimensa salatusta avaimestaan, mutta julkisen avaimen avulla on mahdoton päätellä käyttäjän salattu avain. Julkinen avain toimii tilin tunnisteena, ja salattua avainta käytetään transaktioiden allekirjoittamiseen.[1]

Sopimustileillä ei ole salattuja avaimia, vaan tiliä kontrolloi täysin sen sisältämä sopimuskoodi. Sopimustilit eivät itsenäisesti tee mitään, vaan ne toimivat vain silloin, kun ulkoisesti omistetulta tililtä tulee transaktio sopimustilille. Silloin ne toimivat juuri sillä tavalla kuin niiden sopimuskoodiin on määritelty.[1]

Jokaisella tilillä on osoite. Ulkoisesti omistetuilla tileillä osoite on tilin julkisen avaimen Keccak-256-tiivisteen 20 viimeistä tavua. Sopimustilien osoite määritetään deterministisesti sopimustilin tekijän osoitteen ja nonce-arvon perusteella silloin, kun sopimustili luodaan. Osoitteet ovat 40 merkkiä pitkiä, ja yleensä niissä on 0x-etuliite, joka kertoo, että osoite on esitetty heksadesimaalina.[1]

2.2 Ether ja Gas

Ether on Ethereum-verkolle luontainen valuutta. Sen tarkoitus on toimia arvonvälittäjänä verkon eri osapuolten välillä. Yksi Ether on jaollinen 10^{18} osaan, ja tällaista Etherin pienintä osaa kutsutaan Weiksi.[26]

Gas on Ethereumissa käytettävä yksikkö, jolla mitataan Ethereumin laskenta- ja muistiresurssien käyttöä. Kaikki transaktioiden ja älysopimusten suorittamat toiminnot maksavat kiinteän määrän Gasiä. Esimerkiksi kahden numeron yhteenlaskeminen maksaa 3 Gasiä ja transaktion lähettäminen 21 000 Gasiä.[1]

Gasillä maksetaan verkkoa ylläpitäville palvelimille niiden suorittamasta työstä. Jokaisen transaktion yhteydessä käyttäjä, joka luo transaktion, määrittelee, kuinka paljon Etheriä hän maksaa yhtä Gas-yksikköä kohden. Mitä enemmän käyttäjä maksaa, sitä nopeammin verkkoa ylläpitävät palvelimet todennäköisesti prosessoivat käyttäjän luoman transaktion.[1]

Gasin toinen käyttötarkoitus Ethereumin toiminnassa on palvelunestohyökkäysten estäminen. Vahingossa tehtyjen tai paha-aikeisesti luotujen loputtomien silmukoiden sekä muun resurssienhaaskauksen estämiseksi, käyttäjän tulee jokaisen transaktion yhteydessä määritellä, kuinka monta toimintoa transaktio pystyy maksimissaan suorittamaan. Mitä enemmän käyttäjä käyttää verkon resursseja, sitä enemmän hänen täytyy maksaa Gasiä.[2]

2.3 Transaktiot ja viestit

Transaktiot ovat ulkoisesti omistetuilta tileiltä lähtöisin olevia kryptograafisesti allekirjoitettuja viestejä, jotka kulkevat Ethereum-verkossa ja joiden tapahtuminen tallennetaan Ethereumin lohkoketjuun. Transaktiot laukaisevat tilanmuutoksen Ethereumissa; Ethereum ei itsestään suorita mitään, eivätkä älysopimukset aja mitään koodia itsestään. Kaikki suoritus alkaa transaktiosta.[1]

Transaktiot sisältävät seuraavat kentät[26]:

1. nonce
2. gasPrice
3. gasLimit
4. transaktion vastaanottajan tilin osoite
5. transaktiossa siirrettävän Etherin määrä
6. data-kenttä
7. v,r,s, ECDSA -digitaalisen allekirjoituksen (Elliptic Curve Digital Signature Algorithm) kolme komponenttia.

Nonce on lukuarvo, joka on transaktion lähettäjän tililtä lähteneiden transaktioiden lukumäärä. Nonce on tärkeä osa Ethereumin transaktioita. Sillä on kaksi käyttötarkoitusta: sen avulla transaktiot pystytään prosessoimaan siinä järjestyksessä, jossa ne on lähetetty, ja sen avulla pystytään myös estämään toistohyökkäykset (engl. replay attack).[1] Esimerkiksi, jos noncea ei olisi ja lähetettäisiin kaksi transaktioita lyhyin väliajoin, ei olisi mitään takuuta siitä, että ensiksi lähetetty transaktio prosessoitaisiin ensimmäisenä. Koska nonce on olemassa, pienemmän arvon omaava transaktio täytyy prosessoida ennen kuin suuremman arvon omaava transaktio voidaan prosessoida. Ilman noncea saman transaktion toistaminen monta kertaa olisi mahdollista. Koska nonce sisältyy transaktioon, jokainen suoritettu transaktio on uniikki. Koska jokaisen suoritettavan transaktion tulee olla uniikki, pahanaikeinen toimija ei pysty toistamaan samaa transaktiota monta kertaa.

GasPrice-kenttään käyttäjä spesifioi, kuinka paljon Weitä hän on valmis maksamaan yhtä Gas-yksikköä kohden. GasLimit-kenttään käyttäjä asettaa maksimimäärän transaktion Gasin kulutukselle. Jos transaktion vastaanottaja on ulkoisesti omistettu tili; eli siirretään vain etheriä tililtä toiselle, vaadittava Gasin määrä on aina 21 000. Etherin siirtohinnan pystyy siis helposti laskemaan kertomalla gasPrice-kenttään syötetyn arvon 21 000:nlla. Jos vastaanottava tili on sopimustili, pystytään transaktion suorittamiseen vaadittavaa Gasin määrää arviomaan, mutta ei tarkasti määrittämään. Sopimus tilastaan riippuen saattaa käyttää eri määrän Gasiä, vaikka sopimuksen samaa funktiota kutsuttaisiinkin samalla syötteellä.[1]

Transaktion vastaanottaja voi olla joko ulkoisesti omistettu tili tai sopimustili. Vastaanottaja kenttään tulee 20 tavua pitkä Ethereum-osoite. Käytännössä mikä tahansa 20 tavua pitkä merkkijono on validi vastaanottaja; Ethereum-protokolla ei mitenkään varmista, että

kyseiselle osoitteelle löytyy omistaja, jolla on hallussa kyseisen osoitteen salainen avaimen.[1] Jos vastaanottajan merkkaamisessa tapahtuu virhe, on todennäköstä, että transaktiossa siirrettävät Etherit eivät ole kenenkään saavutettavissa, sillä Ethereumin osoiteavaruus on erittäin suuri (2^{160} mahdollista osoitetta). Ne siis periaatteessa tuhoutuvat.

Transaktion pääasiallinen tietosisältö on transaktiossa siirrettävässä Etherin määrässä ja data-kentässä. Transaktion ei tarvitse sisältää kumpaakaan arvoa, se voi sisältää joko molemmat tai vain jommankumman. Ulkoisesti omistettujen tilien välillä tapahtuvien transaktioiden yhteydessä data-kentällä Ethereum-protokolla ei tee mitään. Jos transaktion vastaanottaja on sopimustili, niin data-kentässä on määritelty, mitä sopismustilin funktiota kutsutaan ja millä syötteellä.[1]

Digitaalisella allekirjoituksella on kolme käyttötarkoitusta Ethereumissa. Ensinnäkin, se todistaa, että transaktion luoman Ethereum-tilin omistaja on valtuuttanut transaktion. Toiseksi, se todistaa, että valtuutus on kiistämätön ja kolmanneksi, se todistaa, että transaktion sisältämää dataa ei ole muutettu allekirjoittamisen jälkeen.[1]

Sopismustilit pystyvät lähettämään transaktioita toisille sopimustileille. Tällaisia transaktioita kutsutaan viesteiksi. Viestejä ei tallenneta lohkoketjuun, koska jos sopimustili lähettää viestin, on sen aina alunperin laukaissut jokin ulkoisesti omistetulta tililtä tullut transaktio.[2]

2.4 Lohkoketju ja konsensus

Transaktiot kerätään lohkoihin. Peräkkäiset lohkot linkitetään toisiinsa kryptograafisen tiivisteen avulla siten, että seuraavasta lohkosta löytyy aina edellisen lohkon tiiviste. Lohkot toimivat rekisterinä siitä, miten Ethereumin tila on muuttunut ajan myötä. Tätä rakennetta kutsutaan lohkoketjuksi ja se muodostaa Ethereumin rungon.[26]

Koska Ethereum on hajautettu järjestelmä, jolla ei ole keskitettyä hallitsijaa, täytyy Ethereum-verkkoon osallistuvilla palvelimilla olla jokin tapa päättää yksimielisesti, mikä verkon nykyinen tila on. Tätä yhtenevään tilaan päätymistä kutsutaan "konsensukseen päätymiseksi."[1]

Ethereum, kuten kaikki lohkoketjujärjestelmät, käyttää kannustinvetoista turvallisuusmallia. Ethereum-verkossa olevat "louhijat" varmentavat verkkoa keräämällä verkkoon lähetettyjä transaktioita lohkoihin ja etsivät lohkolle proof-of-workia. Se louhija, joka ensimmäisenä löytää proof-of-workin louhimalleen lohkolle, pääsee lisäämään lohkonsa Ethereumin lohkoketjuun. Palkkioksi louhija saa 3.0 Etheriä ja kaikki Gas-palkkiot lohkon sisältämistä transaktioista.[5] Louhintaa ei siis tehdä ainoastaan verkon varmentamiseksi, vaan se toimii myös varallisuudenjakelumekanismina, sillä louhinta on ainut tapa, jolla järjestelmään syntyy uutta Etheriä[26].

Proof-of-work on todistus siitä, että lohkon luomiseen on käytetty tietty määrä prosessointitehoa. Uuden lohkon löytämiseen vaadittavaa prosessointitehoa kutsutaan lohkon

vaikeusasteeksi. Vaikeusaste muuttuu jokaisen uuden lohkon löytämisen jälkeen sen perusteella, kuinka kauan aikaa edellinen lohko kesti löytää siten, että Ethereum-verkko löytää uuden lohkon keskimäärin 12 sekunnin välein. Verkon palvelimet päätyvät konsensukseen verkon tilasta valitsemalla aina sen lohkoketjun, johon on käytetty eniten prosessointitehoa, verkon viralliseksi tilaksi. [5]

Ethereumissa käytetty proof-of-work algoritmi on nimeltään Ethash. Ethereum louhinnassa tulee löytää nonce-syöte Ethash-algoritmiin siten, että tulos on lohkon vaikeusasteesta riippuvan raja-arvon alapuolella. Todennäköisyys, että louhija löytää ensimmäisenä uuden lohkon voidaan laskea jakamalla louhijan kontrolloima prosessointiteho verkon prosessointitehon kokonaismäärällä.[5]

2.5 Ethereumin virtuaalikone

Kun transaktion suorittaminen johtaa sopimuskoodin ajamiseen tai uuden sopimustilin luomiseen, tarvitaan Ethereumin virtuaalikonetta laskemaan, millainen tilanmuutos näistä operaatioista syntyy. Arvonvälitys transaktiot yhdeltä ulkoisesti omistetulta tililtä toiselle eivät sitä tarvitse, mutta käytännössä kaikki muut transaktiot tarvitsevat Ethereumin virtuaalikoneen laskeman tilanmuutoksen.[1]

Ethereumin virtuaalikone on melkein Turing-täydellinen tilakone. Täysin Turing-täydellisessä virtuaalikoneessa olisi pysähtymisongelma; ei olisi mitään tapaa määrittää, pysähtyykö koodin suorittaminen jollakin syötteellä koskaan.[1] Tämän takia transaktioita prosessoivat palvelimet ennen pitkää joutuisivat ikuiseen silmukkaan, joka johtaisi Ethereum-verkon toiminnan pysähtymiseen. Koska kaikkia suoritettavia transaktioita rajoittaa transaktiolle saatavilla olevan Gasin määrä, jokaisen transaktion suorittaminen tulee lopulta pysähtymään.[1]

Virtuaalikone termiä käytetään usein viittaamaan todellisten tietokoneiden virtuaalisointiin. Tällaisten virtuaalikoneiden tulee tarjota ohjelmalliset abstraktiot fyysisestä laitteistosta, sekä järjestelmäkutsuista ja muusta käyttöjärjestelmäytimen toiminnalisuudesta.[1]

Ethereumin virtuaalikoneen toimialue on paljon suppeampi. Se tarjoaa abstraktiot vain laskennasta ja muistista. Ethereumin virtuaalikone on siis samankaltainen kuin esimerkiksi Javan virtuaalikone. Javan virtuaalikone on suunniteltu tarjoamaan ajonaikainen ympäristö, joka ei välitä alla olevasta käyttöjärjestelmästä tai laitteistosta. Korkean tason ohjelmointikielet kuten Java käännettään tavukoodiksi, jonka Javan virtuaalikone pystyy suorittamaan. Samalla tavalla Ethereumin virtuaalikoneella on oma käskykanta, joksi korkean tason älysopimusten ohjelmointikielet kuten Solidity käännettään.[1]

Siinä tilanteessa, kun transaktio johtaa sopimuskoodin suorittamiseen, virtuaalikoneesta luodaan ilmentymä. Kutsuttavan sopimustilin koodi ladataan virtuaalikoneen lukumuistiin, ohjelmalaskuri asetetaan nollaan, tilin muistista ladataan arvot sekä asetetaan kaikki lohko- ja ympäristömuuttujat. Tärkeä muuttuja on transaktiossa määritelty gasLimit-arvo.

Kun koodin suorittaminen etenee gasLimit-arvosta vähennetään suoritettavien toimintojen Gas-kulu. Jos jossakin kohtaa suorittamista saatavilla oleva Gasin määrä menee nollaan, syntyy “Out of Gas”-poikkeus. Tässä tapauksessa transaktion suorittaminen keskeytetään. Ethereumin tilaan ei tehdä muita muutoksia kuin että transaktion lähettäjän Ether-saldosta vähennetään poikkeustilaan päätymiseen asti käytettyjen resurssien hinta sekä transaktion lähettäjän nonce-arvoa kasvatetaan yhdellä.[1]

2.6 Älysopimukset ja tokenit

Älysopimus termin määritteli alunperin Nick Szabo vuonna 1994. Szabon mukaan älysopimus on tietokoneistettu transaktio-protokolla, joka suorittaa lainalaisen sopimuksen ehdot[22]. Tämä ajatus on Ethereum-älysopimusten pohjalla, vaikkakaan Ethereumin älysopimukset eivät ole lainalaisia sopimuksia.

Ethereumin kontekstissa älysopimukset ovat muuttumattomia tietokoneohjelmia, joiden suorittaminen on determinististä. Muuttumattomuudella tarkoitetaan sitä, että sen jälkeen, kun älysopimus on lähetetty Ethereum-verkkoon, sen sisältämää logiikkaa ei pystytä mitenkään muokkaamaan. Ainut tapa muokata älysopimusta on luoda siitä uusi instanssi. Deterministisyydellä tarkoitetaan sitä, että älysopimuksen suorittamisen lopputulos on aina sama, mikäli Ethereumin lohkoketjun tila on sama, ja transaktio, joka sitä kutsui on sama.[1]

Älysopimukset ohjelmoidaan yleensä jollakin korkean tason ohjelmointikielellä ja käännetään sitten Ethereum virtuaalikoneen tavukoodiksi. Älysopimusten ohjelmointiin on tarjolla useita ohjelmointikieliä. Ethereum älysopimuksia on mahdollista ohjelmoida Solidityllä, joka muistuttaa C:tä ja JavaScriptiä; Serpentillä, joka muistuttaa Pythonia; LLL:llä, joka muistuttaa Lispiä; tai Vyperillä, joka myös muistuttaa Pythonia. Näistä suosituin on Solidity.[6]

Eräs suuri käyttökohde älysopimuksille on viime vuosina ollut joukkorahoitus. ICO:t (initial coin offering) ovat kryptovaluutta-alan vastine IPO:ille (initial public offering). IPO:lla tarkoitetaan sitä, kun yksityinen yritys rahoittaa toimintaansa julkisesti myymällä osakkeitaan silloin, kun uusi osake liikkeellelasketaan[11]. Ethereumin päälle rakennettut ICO:t toimivat siten, että yritys, joka haluaa luoda uuden ICO:n, luo älysopimuksen, johon voi lähettää Etheriä. Käyttäjät, jotka haluavat osallistua ICO:on, lähettävät Etheriä älysopimukseen ja saavat vastineeksi projektin tokeneita suhteessa lähettämäänsä Etherin määrään. ICO-älysopimuksissa on yleensä myös määritelty jokin tavoitemäärä Etheriä, joka pitää vähintään saada kerättyä. Mikäli tähän tavoitteeseen ei päästä, Etherit luovutetaan älysopimuksesta takaisin ICO:on sijoittaneille käyttäjille[9]. ICO:t keräsivät vuonna 2018 yhteensä lähes 8 miljardia Yhdysvaltain dollaria[13].

Tokenit ovat Ethereumin päälle älysopimuksilla rakennettuja ali-valuuttoja. Tokeneilla on useita käyttötarkoituksia. Ne voivat edustaa esimerkiksi todellisen maailman hyödykkeitä, kuten yritysten osakkeita, kultaa tai Yhdysvaltain dollareita.[2] Esimerkiksi TrueUSD on

```
def send(to, value):
    # Mikäli transaktion lähettäjän token-saldo on enemmän
    # tai samanverran kuin lähetettävä tokenin määrä,
    # vähennetään lähettäjän token-saldosta lähetettävien
    # tokenien määrä, ja siirretään se vastaanottajan token-saldoon
    if self.storage[msg.sender] >= value:
        self.storage[msg.sender] = self.storage[msg.sender] - value
        self.storage[to] = self.storage[to] + value
```

Ohjelma 2.1. *Tokenien siirto-funktio, perustuu lähteeseen [2]*

Ethereumin päälle rakennettu kryptovaluutta tokeni, joka vastaa Yhdysvaltain dollaria. Jokainen TrueUSD-tokeni on arvoltaan aina yhden dollarin[24].

Perustuvanlaatuinen funktio token-älysopimuksen luomiseen Serpent-ohjelmointikielellä on esitetty Ohjelmassa 2.1. Msg.sender viittaa transaktion lähettäjän Ethereum-osoitteeseen, ja self.storage viittaa älysopimuksen pitkäaikaismuistiin. Älysopimuksen pitkäaikaismuistiin on tässä tapauksessa talletettu Ethereum-osoitteita vastaavat token-saldot.

Tämän lisäksi tulee lisätä jokin järjestelmä, jolla tokenit aluksi jaetaan[2]. Tämä voidaan toteuttaa esimerkiksi ICO:n yhteydessä siten, että ICO:on osallistujat saavat tokeneita siinä suhteessa, kuinka paljon Etheriä he sijoittavat ICO:on. Lisäksi olisi hyvä olla funktio, jolla pystyy kyselemään token-saldoja Ethereum-osoitteiden perusteella[2]. Tällä tavalla pystyy luomaan uusia kryptovaluuttoja ilman, että täytyy itse luoda uusi lohkoketju ja konsensus-mekanismi. On huomattavasti helpompaa käyttää Ethereum-alustaa, johon nämä ovat totetutettu valmiiksi.

Älysopimusten avulla tokeneille voidaan luoda muitakin käyttötarkoituksia kuin vain niiden siirteleminen käyttäjältä toiselle. Tällaisia projekteja rakennetaan Ethereumin päälle useita. Esimerkiksi Augur on Ethereum-alustan päälle rakennettava projekti, jolla on oma tokeninsa nimeltään Reputation. Augur tarjoaa käyttäjilleen hajautetut ennustusmarkkinat. Käyttäjät voivat luoda tapahtumia, joiden lopputuloksesta käyttäjät äänestävät Reputation-tokeneilla. Yksinkertaistettuna Augur toimii siten, että joku käyttäjä luo tapahtuman, jonka lopputuloksesta muut äänestävät. Sanotaan, että tapahtumalla on kaksi mahdollista lopputulosta A ja B. Mikäli B kerää valtaosan äänistä, B:stä tulee tapahtuman lopputulos ja A:ta äänestäneiden käyttäjien sijoittamat Reputation-tokenit jaetaan B:tä äänestäneille. Tämä kannustaa käyttäjiä äänestämään kaikista todennäköisimmän lopputuloksen puolesta.[20]

3 ÄLYSOPIMUSTEN SUUNNITTELUMALLIT

Suunnittelumalli abstraktoi ja nimeää yleisen suunnittelurakenteen tärkeimmät aspektit. Suunnittelumallilla on nimi, joka kuvaa sen ratkaisemaa suunnitteluongelmaa. Sillä on jokin ongelma, jonka ratkaisemiseen se on luotu. Suunnittelumalli tarjoaa ratkaisun tähän ongelmaan. Suunnittelumallit eivät ole niinkän konkreetteja ratkaisuja, vaan abstrakteja kuvauksia siitä, miten ongelma voidaan ratkaista.[10]

Tässä luvussa käydään läpi Etherum-ekosysteemissä tyypillisesti esiintyvien ongelmien ratkaisuun luotuja suunnittelumalleja. Suunnittelumalleista esitetään esimerkkitoteutukset Solidity-ohjelmointikielellä.

3.1 ERC-token standardit

ERC:t (Ethereum Request for Comments) ovat Ethereumin standardeja. ERC-token -standardit määrittelevät Ethereum-tokeneille rajapinnat, jotka token-älysopimusten tulee tarjota.

Standardoidut rajapinnat ovat hyödyllisiä siksi, että ne helpottavat huomattavasti applikaatioiden luomista tokeneiden ympärille. Esimerkiksi lompakko-applikaatiot hyötyvät tokeneiden standardoiduista rajapinnoista. Lompakko-applikaatiot toimivat pääsääntöisenä käyttöliittymänä Ethereumiin. Lompakko-applikaation avulla käyttäjä voi esimerkiksi tarkastella Ether- ja token-saldojansa, lähettää transaktioita ja hallita osoitteita.[1] ERC-token standardeja on useita erilaisia. Käytetyin on ERC-20 -standardi[1].

3.1.1 ERC-20

Suurin osa tokeneista noudattaa ERC-20 -standardia. ERC-20 -standardi tarjoaa perustuvanlaatuisen rajapinnan tokeneille, jotka ovat fungiibelejä.[1] Fungiibeli tarkoittaa, että yksi tokeni on aina samanarvoinen kuin toinen. Esimerkiksi eurot ovat fungiibelejä; ei ole väliä millä eurolla maksat kaupassa.

Jotta token noudattaa ERC-20 -standardia, tulee sen toteuttaa ohjelmassa 3.1 esitetyt metodit. Name-, symbol- ja decimals-metodit ovat vapaaehtoisia, mutta ne voidaan toteuttaa käytettävyyden lisäämiseksi.[4]

Transfer-metodilla käyttäjä pystyy siirtämään omistamiansa tokeneita. Transfer-metodin kutsun tulee epäonnistua, mikäli kutsujan token-saldo on pienempi kuin _value-argumentissa määritelty arvo. Transfer-metodin tulee laukaista Transfer-tapahtuma.[4]

Tapahtumat ovat Solidity-ohjelmointikielen abstraktio Ethereumin kirjaustoiminnalisuudesta. Applikaatiot voivat tarkkailla Ethereumin lohkoketjuun kirjattuja tapahtumia Ethereum-asiakasohjelman RPC-rajapinnan kautta.[21]

Rajapinnasta löytyy myös metodit, joilla käyttäjä voi antaa jollekin toiselle tilille oikeuden siirrellä hänen omistamiaan tokeneita. Käytännössä tämä toiminnalisuus löytyy, jotta käyttäjät voivat antaa tokeninsa älysopimuksen käytettäväksi. Approve-metodin tulee laukaista Approval-tapahtuma. Mikäli metodia kutsutaan uudestaan, se korvaa tällä hetkellä käytettävissä olevien tokeneiden määrän _value-argumentin arvolla. TransferFrom-metodin kutsun tulee epäonnistua, mikäli sitä kutsuvalla tilillä ei ole oikeuksia siirtää _from-argumentissa määriteltyä tililtä _value-argumentin verran tokeneita. TransferFrom-metodin tulee laukaista Transfer-tapahtuma.[4]

3.1.2 ERC-721

ERC-721 määrittelee rajapinnan tokeneille, jotka eivät ole fungiibelejä. Jokainen tokeni on siis uniikki. Tällaisilla tokeneilla voidaan kuvata digitaalisten tai fyysisten varojen omistamista. Esimerkiksi talojen tai uniikkien taideteosten omistamisen voisi kuvata lohkoketjuun ERC-721 -tokeneilla.[3]

CryptoKitties-projekti on tähän mennessä suurin ERC-721 -standardia käyttävä projekti. CryptoKitties on Ethereumin päälle rakennettu peli, jossa keräillään digitaalisia kissoja. Jokaista olemassaolevaa kissaa on olemassa sitä vastaava tokeni. Kissoja voi myydä toisille pelaajille ja niitä voi risteyttää toistensa kanssa luoden uusia kissoja.[23] Kun CryptoKitties julkaistiin vuonna 2017, siitä tuli niin suosittu, että se käytännössä ylikuormitti Ethereum-verkon[12].

ERC-721 yhteensopivan tokenin tulee toteuttaa ERC-721- ja ERC-165 -rajapinnat. ERC-165 määrittelee rajapinnan, jonka avulla älysopimukselta voi kysyä, mitä rajapintoja se tukee. ERC-165 -rajapinta on esitetty Ohjelmassa 3.2. Jokaisella standardoidulla rajapinnalla on oma tunniste. Esimerkiksi ERC-721 -rajapinnan tunniste on 0x80ac58cd.[3]

ERC-721 -rajapinta on esitetty Ohjelmassa 3.4. Jokaisella tokenilla tulee olla uniikki uint-256-tunniste ERC-721 -älysopimuksen sisällä. ERC-721 -standardissa on turvallinen siirtofunktio safeTransferFrom, sekä epäturvallinen siirtofunktio transferFrom. Erona näiden kahden välillä on se, että safeTransferFrom-metodia käytettäessä tarkistetaan tokenin vastaanottavalta tililtä, mikäli se on sopimustili, pystyykö se vastaanottamaan tokenin. Jotta sopimustili pystyy vastaaanottamaan ERC-721 -tokenin, tulee sen toteuttaa Ohjelmassa 3.3 esitetty ERC721TokenReceiver -rajapinta. Kun safeTransferFrom-metodia kutsutaan, se tarkistaa, onko _to-argumentissa määritelty tili sopimustili. Mikäli vastaanot-

```
contract IERC20 {
    /// @return Tokenin nimi
    function name() public view returns (string)
    /// @return Tokenin symboli. Esimerkiksi Ether->ETH
    function symbol() public view returns (string)
    /// @return Moneenko osaan token on jaollinen. Esimerkiksi
    /// 10^18 osaan -> funktio palauttaa 18
    function decimals() public view returns (uint8)
    /// @return Olemassaolevien tokeneiden kokonaismäärä
    function totalSupply() public view returns (uint256)
    /// @param _owner Tili, minkä saldo halutaan tietää
    /// @returns _owner tilillä olevien tokeneiden määrä
    function balanceOf(address _owner)
        public view returns (uint256 balance)
    /// @param _to Osoite, johon siirretään
    /// @param _value Paljonko tokeneita siirretään
    /// @return Onnistuiko siirto vai ei
    function transfer(address _to, uint256 _value)
        public returns (bool success)
    /// @param _from Osoite, josta siirretään
    /// @param _to Osoite, johon siirretään
    /// @param _value Paljonko tokeneita siirretään
    /// @return Onnistuiko siirto vai ei
    function transferFrom(address _from, address _to, uint256 _value)
        public returns (bool success)
    /// @param _spender Osoite, jolle annetaan oikeudet käyttää tokeneita
    /// @param _value Paljonko tokeneita _spender saa käyttää
    /// @return Onnistuiko oikeuksien antaminen vai ei
    function approve(address _spender, uint256 _value)
        public returns (bool success)
    /// @param _owner Tokenien omistaja
    /// @param _spender Tili, jonka käyttäoikeuksia tiedustellaan
    /// @return paljonko _spender saa käyttää _ownerin tokeneita
    function allowance(address _owner, address _spender)
        public view returns (uint256 remaining)


    event Transfer(address indexed _from,
        address indexed _to, uint256 _value)

    event Approval(address indexed _owner,
        address indexed _spender, uint256 _value)
}
```

Ohjelma 3.1. *ERC-20 -rajapinta, lähteestä [4]*

```
interface IERC165 {
    /// @param _interfaceId rajapinnan tunniste
    /// @return Tukeeko sopimus _interfaceId rajapintaa
    function supportsInterface(bytes4 _interfaceId) external view returns (
        bool);
}
```

Ohjelma 3.2. *ERC-165 -rajapinta, lähteestä [3]*

```
interface ERC721TokenReceiver {
    /// @param _operator Osoite, joka kutsui safeTransferFrom-metodia
    /// @param _from Osoite, joka omisti tokenin
    /// @param _tokenId Tokenin tunniste
    /// @param _data Ylimääräinen data-kenttä
    /// @return `bytes4(keccak256("onERC721Received(address,address,uint256,bytes)"))`
    /// paitsi poikkeustilanteessa

    function onERC721Received(address _operator, address _from, uint256
        _tokenId, bytes _data) external returns(bytes4);
}
```

Ohjelma 3.3. *ERC-721TokenReceiver -rajapinta, lähteestä [3]*

tava tili on sopimustili, metodi kutsuu vastaanottavan tilin onERC721Received-metodia. Siirtofunktion suoritus peruutetaan, mikäli funktiokutsu palauttaa jotakin muuta kuin maagisen numeron (engl. magic number)
‘bytes4(keccak256("onERC721Received(address,address,uint256,bytes)"))‘.[3]

Siirron voi suorittaa tokenin omistaja, tili, jolle tokenin omistaja on antanut oikeuden approve-metodilla tai tili, jolle tokenin omistaja on antanut oikeuden setApprovalForAll-metodilla. Tili, jolle on annettu oikeudet setApprovalForAll-metodilla, voi myös käyttää approve-metodia tokeneille. Kaikkien siirtofunktioden tulee laukaista Transfer-tapahtuma ja oikeutusmetodien tulee laukaista niitä vastaavat Approval-tapahtumat.

3.2 Valtuutusmallit

Valtuutusmalleilla voidaan rajoittaa oikeutta kutsua funktioita asettamalla niiden kutsuille joitakin rajoitteita[25]. Esimerkiksi oikeutta kutsua funktioita voidaan rajoittaa tilien osoitteiden tai kuluneen ajan perusteella.

3.2.1 Ownable

On tyypillistä, että vain sopimuksen omistaja saisi kutsua joitakin sopimuksen funktioita. Tämä voidaan saavuttaa siten, että tallennetaan sopimuksen luomisen yhteydessä sen luoneen tilin osoite sopimukseen, ja rajataan oikeutta kutsua funktioita kutsujan osoitteen perusteella.[25]

Yksinkertainen esimerkkitoteutus tällaisesta sopimuksesta on esitetty Ohjelmassa 3.5 OnlyOwner-määreen (engl. modifier) avulla voidaan estää muita kuin sopimuksen omistajaa kutsumasta tiettyjä funktioita.

Määreillä voidaan muuttaa funktion käyttäytymistä. Esimerkiksi niillä voidaan automaattisesti tarkistaa jokin ehto ennen funktion kutsumista.[21] OnlyOwner-määre tarkistaa ennen funktion kutsumista, että funktion kutsuja on tilille merkitty omistaja.

```
contract IERC721 is IERC165 {
    /// @param owner _Osoite, jonka saldo halutaan selvittää
    /// @return '_owner' osoitteen omistamien tokeneiden määrä
    function balanceOf(address _owner)
        public view returns (uint256 balance);
    /// @param _tokenId Tokenin tunniste
    /// @return '_tokenId' Tunnisteen omaavan tokenin omistaja
    function ownerOf(uint256 _tokenId) public view returns(address owner);
    /// @param _from Tokenin tämänhetkinen omistaja
    /// @param _to Tokenin uusi omistaja
    /// @param _tokenId Siirretävän tokenin tunniste
    function safeTransferFrom(address _from, address _to, uint256 _tokenId)
        public;
    /// @param _from Tokenin tämänhetkinen omistaja
    /// @param _to Tokenin uusi omistaja
    /// @param _tokenId Siirretävän tokenin tunniste
    function transferFrom(address _from, address, _to, uint256 _tokenId)
        public;
    /// @param _to Osoite, jolle annetaan oikeudet
    /// @param _tokenId Tokeni, jolle annetaan oikeudet
    function approve(address _to, uint256 _tokenId) public;
    /// @param _tokenId Token, minkä oikeuksien haltija halutaan tietää
    /// @return Tokenin oikeuksien haltija
    function getApproved(uint256 _tokenId) public view returns (address
        operator);
    /// @param _operator Osoite, jonka oikeudet asetetaan
    /// @param _approved poistetaanko vai annetaanko oikeudet
    function setApprovalForAll(address _operator, bool _approved) public;
    /// @param _owner Tokenin omistaja
    /// @param _operator Tokenin oikeuksien haltija
    /// @return Onko '_operator' osoitteella oikeudet
    function isApprovedForAll(address _owner, address _operator) public
        view returns (bool);
    /// @param _from Tokenin tämänhetkinen omistaja
    /// @param _to Tokenin uusi omistaja
    /// @param _tokenId Siirretävän tokenin tunniste
    /// @param _data ylimääräinen data
    function safeTransferFrom(address _from, address _to, uint256 _tokenId,
        bytes _data) public;

    event Transfer(address indexed _from, address indexed _to, uint256
        indexed _tokenId);

    event Approval(address indexed _owner, address indexed _approved,
        uint256 indexed _tokenId);

    event ApprovalForAll(address indexed _owner, address indexed _operator,
        bool _approved);
}
```

Ohjelma 3.4. *ERC-721 -rajapinta, lähteestä [16]*

```
contract Ownable {
    address private _owner;

    event OwnershipTransferred(address indexed previousOwner, address
        indexed newOwner);

    // Alustetaan sopimuksen omistajaksi se tili, joka luo sopimuksen
    constructor () internal {
        _owner = msg.sender;
        emit OwnershipTransferred(address(0), _owner);
    }

    /// @return Sopimuksen tämänhetkinen omistaja
    function owner() public view returns (address) {
        return _owner;
    }

    /// Tarkistetaan, onko transaktion lähettäjä sopimuksen omistaja
    modifier onlyOwner() {
        require(msg.sender == _owner, "Ownable: caller is not the owner");
        _;
    }

    /// @param newOwner sopimuksen uusi omistaja
    function transferOwnership(address newOwner) public onlyOwner {
        require(newOwner != address(0), "Ownable: new owner is the zero
            address");
        emit OwnershipTransferred(_owner, newOwner);
        _owner = newOwner;
    }
}
```

Ohjelma 3.5. *Ownable-suunnittelumalli, perustuu lähteeseen [17]*

Sopimuksen omistaja voi siirtää sopimuksen omistajuuden jollekin toiselle tilille transferOwnership-funktiolla. Se laukaisee OwnershipTransferred-tapahtuman, jotta applikaatiot voivat huomata, mikäli sopimuksen omistaja vaihtuu.

3.2.2 Access restriction

Oletusarvoisesti sopimuksen metodi suoritetaan ilman, että tarkistetaan mitään esiehtoja. Mikäli halutaan, että suoritus sallitaan vain silloin, kun jotkin ehdot täyttyvät, voidaan tällaisille ehdoille luoda määritteet, jotka tarkistavat ehtojen täyttymisen ennen funktiokutsua.[25]

Esimerkki määritteitä hyödyntävästä sopimuksesta on esitetty Ohjelmassa 3.6. Esimerkissä tallennetaan sopimuksen luomisen yhteydessä sen luomisaika. Tätä voidaan esimerkiksi käyttää hyödyksi määritteissä tarkistamaan, että joitakin funktiota ei pysty kutsumaan sopimuksen luomisen jälkeen kuluneesta ajasta riippuen. Esimerkistä löytyy myös määrite, jolla voidaan tarkistaa, että transaktion yhteydessä siirretään vähintään tietty määrä Etheriä, sekä määrite, johon voi määritellä jonkin mukautetun ehdon.

```
contract AccessRestriction {
    uint public creationTime;
    /// asetetaan luomisaika rakentajassa myöhempää käyttöä varten
    constructor  ()  internal {
        creationTime = now;
    }

    modifier onlyBefore(uint _time) {
        require(now < _time); _;
    }

    modifier onlyAfter(uint _time) {
        require(now > _time); _;
    }

    modifier condition(bool _condition) {
        require(_condition); _;
    }

    modifier minAmount(uint _amount) {
        require(msg.value >= _amount); _;
    }
    /// Esimerkki funktio, joka käyttää määritteitä
    function example() payable onlyAfter(creationTime + 1 minutes)
        minAmount(2 ether) condition(msg.
        sender.balance >= 50 ether) public {
        // jotakin toiminnallisuutta
    }
}
```

Ohjelma 3.6. *Access restriction -suunnittelumalli, lähteestä [25]*

3.3 Toimintomallit

Toimintomallit tarjoavat käytäntöjä tyypillisten toimintojen suorittamiseen[25]. Varsinkin toiminnot, joiden yhteydessä saatetaan interaktoida toisten sopimusten kanssa, saattavat aiheuttaa ongelmia, jotka tulee ottaa huomioon[7].

3.3.1 Checks-Effects-Interactions

Kaikki interaktiot sopimukselta A sopimukselle B ja kaikki Ether siirrot antavat kontrollin suorituksesta sopimukselle B. Tämä mahdollistaa sen, että sopimus B voi kutsua sopimusta A uudelleen ennen kuin alkuperäinen interaktio sen kanssa on viety loppuun.[8] Tästä voi syntyä haavoittuvaisuuksia, jotka täytyy ottaa huomioon sopimusta suunniteltaessa.

Ohjelmassa 3.7 on esitetty virheellinen sopimus. Esimerkin sopimus on virheellinen, sillä Ether-siirrot voivat aina johtaa koodin ajamiseen vastaanottavassa päässä[7]. Sopimus, joka kutsuu withdraw-funktiota voi kutsua sitä uudelleen Ohjelman 3.7 rivillä seitsemän ennen kuin alkuperäiseen sopimukseen on päivitetty, että kutsuvan sopimuksen saldo on nolla. Täten sopimuksesta pystyy nostamaan enemmän varoja kuin mitä oli tarkoitettu.

```
contract Fund {
    /// Tietorakenne, jossa säilytetään osotteita vastaavat balanssit
    mapping(address => uint) shares;

    /// Funktio, jolla voi nostaa varat sopimuksesta
    function withdraw() public {
        (bool success,) = msg.sender.call.value(shares[msg.sender])("");
        if (success)
            shares[msg.sender] = 0;
    }
}
```

Ohjelma 3.7. *Virheellinen sopimus, lähteestä [8]*

```
contract Fund {
    /// Tietorakenne, jossa säilytetään osotteita vastaavat balanssit
    mapping(address => uint) shares;

     /// Funktio, jolla voi nostaa varat sopimuksesta
    function withdraw() public {
        uint share = shares[msg.sender];
        shares[msg.sender] = 0;
        msg.sender.transfer(share);
    }
}
```

Ohjelma 3.8. *Checks-Effects-Interactions -suunnittelumalli, lähteestä [8]*

Tällaisten tilanteiden välttämiseksi tulisi käyttää Checks-Effects-Interactions -suunnittelumallia aina, kun sopimuksen täytyy interaktoida toisen tuntemattoman sopimuksen kanssa jollakin tavalla. Mallin mukaan funktiossa tehtävät tarkistukset tulisi tehdä funktion alussa. Asioita joita esimerkiksi voisi tarkistaa on, paljonko Etheriä transaktion yhteydessä lähetettiin, kuka kutsuu funktiota ja ovatko argumentit oikeanlaisia. Seuraavaksi, mikäli tarkistukset menevät läpi, tulisi muuttaa sopimuksen tilamuuttujia. Interaktiot toisten sopimusten kanssa tulisi suorittaa aina funktion lopussa.[8]

Esimerkki samasta sopimuksesta, joka on toteutettu tätä mallia käyttäen on esitetty Ohjelmassa 3.8 Koska tässä sopimuksessa tilamuuttujaa muutetaan ennen kuin interaktoidaan toisen tilin kanssa, vaikka vastaanottava tili pystyisikin uudelleen kutsumaan withdraw-funktiota, sille ei lähetettäisi yhtään Etheriä. Tässä esimerkissä käytetään myöskin turvallisempaa tapaa siirtää varat; osoite.transfer(määrä) antaa vastaanottavalle tilille käyttöön vain 2300 Gasiä, joka ei riitä siihen, että vastaanottava sopimus pystyisi kutsumaan toista sopimusta. Ohjelmassa 3.7 käytetty osoite.call.value(määrä)(data) antaa oletusarvoisesti vastaanottavan tilin käyttöön kaiken jäljellä olevan Gasin.[7]

3.3.2 Pull payment

Kutsut ulkoisiin sopimuksiin voivat epäonnistua useista eri syistä ja niiden käyttämistä tulisi välttää, mikäli se vain on mahdollista. Tällaisia kutsuja ei tietenkään pystytä aina täysin välttämään. Mikäli sopimuksen täytyy siirtää varoja tai kutsua jotakin toista sopimusta,

```
1  contract auction {
2      public address highestBidder;
3      public uint highestBid;
4
5      /// Funktio, jolla käyttäjät voivat asettaa tarjouksia
6      function bid() external payable {
7          if (msg.value < highestBid) throw;
8
9          if (highestBidder != 0) {
10             if (!highestBidder.send(highestBid)) {
11                 throw;
12             }
13         }
14
15        highestBidder = msg.sender;
16        highestBid = msg.value;
17     }
18 }
```

Ohjelma 3.9. *Virheellinen esimerkki maksun suorittamisesta, perustuu lähteeseen [7]*

on paras eristää tällaiset toiminnot omiksi funktioikseen, joita kutsun vastaanottaja voi kutsua. Tämä on tärkeää varsinkin maksuille, joiden tapauksessa on paras antaa käyttäjien nostaa varat sopimuksesta, sen sijaan, että sopimus lähettäisi ne jonkin metodin yhteydessä.[7]

Ohjelmassa 3.9 on esitetty virheellinen sopimus, joka toteuttaa yksinkertaisen huutokaupan. Bid-funktiolla käyttäjät voivat asettaa tarjouksia. Sopimus pitää kirjaa siitä, kenellä on korkein tarjous ja kuinka suuri se on. Kun joku asettaa korkeamman tarjouksen, lähetetään edellisen korkeimman tarjouksen haltijalle hänen varansa takaisin.

Tässä sopimuksessa on kumminkin haavoittuvaisuus. Mikäli korkeimman tarjouksen haltijalle ei pysty lähettämään varoja, muut käyttäjät eivät pysty asettamaan tarjouksia, sillä funktio heittää poikkeuksen, mikäli varojen palautus epäonnistuu.[7] Olisi yksinkertaista luoda sopimus, joka asettaa tarjouksen, mutta jolta ei löydy tapaa vastaanottaa Etheriä. Mikäli sopimukseen siirretään Etheriä, ilman että kutsutaan samalla jotakin sen funktiota, jolla on payable-määre, niin kutsutaan sopimuksen fallback-funktiota[21]. Mikäli tällaista ei sopimukseen ole toteutettu, Ether-siirto epäonnistuu.

Ohjelmassa 3.10 on esitetty oikea tapa toteuttaa sama sopimus. Sen sijaan, että lähetettäisiin varat, pidetään kirjaa palautettavista varoista, ja annetaan käyttäjien nostaa ne älysopimuksesta takaisin erillisen metodin avulla. Näin vältytään siltä, että varojen siirtämisestä syntyisi haavoittuvaisuuksia sopimukseen.

3.4 Elinkaarimallit

Ethereum-verkkoon luotu sopimus tulee oletusarvoisesti olemaan siellä niin kauan kuin Ethereum-verkko on toiminnassa[25]. Mikäli sopimus halutaan jossakin kohtaa poistaa toiminnasta, tulee tällainen toiminnalisuus määritellä sopimukseen sen luontivaiheessa-

```
contract auction {
    public address highestBidder;
    public uint highestBid;
    mapping(address => uint) refunds;

    /// Funktio, jolla käyttäjät voivat asettaa tarjouksia
    function bid() external payable {
        if (msg.value < highestBid) throw;

        if (highestBidder != 0) {
            refunds[highestBidder] += highestBid;
        }

        highestBidder = msg.sender;
        highestBid = msg.value;
    }

    ///Funktio, jolla huutokaupan hävinneet käyttäjät voivat
    ///nostaa varansa takaisin
    function withdrawRefund() external {
        uint refund = refunds[msg.sender];
        refunds[msg.sender] = 0;
        if (!msg.sender.send(refund)) {
            refunds[msg.sender] = refund;
        }
    }
}
```

Ohjelma 3.10. *Esimerkki pull payment -suunnittelumallista, perustuu lähteeseen [7]*

```
contract Mortal is Ownable {
    function kill() onlyOwner {
        selfdestruct(_owner);
    }
}
```

Ohjelma 3.11. *Selfdestruct-suunnittelumalli, perustuu lähteeseen [21]*

sa[1]. Elinkaarimallit tarjoavat ratkaisuja sopimuksen toiminnan pysäyttämiseksi.

3.4.1 Selfdestruct

Ethereumin virtuaalikoneesta löytyy operaatio, jolla sopimuksen pystyy tuhoamaan[26], joten kaikista älysopimus-ohjelmointikielistä tulisi löytyä tätä vastaava funktio. Solidityssä tämä funktio on nimeltään selfdestruct.[21] Esimerkki sen käytöstä on esitetty ohjelmassa 3.11 Selfdestruct-funktio ottaa parametrina Ethereum-osoitteen, johon älysopimuksessa jäljellä olevat Etherit siirretään sen tuhoamisen yhteydessä.

Transaktiot, jotka lähetetään sellaiseen osoitteeseen, joka on tuhottu selfdestruct-kutsulla eivät enää johda minkäänlaiseen koodin suoritukseen, sillä osoitteesta pyyhitään kaikki tiedot. Jäljelle jää vain tyhjä tili, johon kenelläkään ei ole enää pääsyä.[1] Tästä syntyy ongelmia, mikäli kaikki sopimusta käyttävät tahot eivät ole tietoisia sopimuksen tuhoutumisesta.

```
contract Pausable is Ownable {

    bool private _paused;

    constructor () internal {
        _paused = false;
    }
    /// @return Onko sopimus tauolla vai ei
    function paused() public view returns (bool) {
        return _paused;
    }
    /// Määrite, jonka voi asettaa funktiolle, joita halutaan voida
    /// kutsua vain silloin kun sopimus ei ole tauolla
    modifier whenNotPaused() {
        require(!_paused, "Pausable: paused");
        _;
    }
    /// Määrite, jonka voi asettaa funktiolle, joita halutaan voida
    /// kutsua vain silloin kun sopimus on tauolla
    modifier whenPaused() {
        require(_paused, "Pausable: not paused");
        _;
    }
    /// Funktio, jolla voi asettaa sopimuksen tauolle
    function pause() public onlyOwner whenNotPaused {
        _paused = true;
    }
    /// Funktio, jolla voi ottaa sopimuksen pois tauolta
    function unpause() public onlyOwner whenPaused {
        _paused = false;
    }
}
```

Ohjelma 3.12. *Esimerkki pausable-suunnnittelumallista, perustuu lähteeseen [18]*

3.4.2 Pausable

Mikäli sopimus poistetaan käyttämällä selfdestruct-funktiota, niin kaikki Etherit jotka siihen siirretään sen jälkeen tuhoutuvat. Mikäli sopimuksen toiminta halutaan pysäyttää, mutta halutaan olla varmoja siitä, että sopimukseen vahingossa lähetetyt Etherit eivät tuhoudu, voidaan käyttää pausable-suunnittelumallia.

Esimerkki tästä mallista on esitetty ohjelmassa 3.12. Tässä esimerkissä sopimuksen omistaja voi laittaa sopimuksen tauolle. sopimuksen ollessa tauolla whenPaused-määritteen omaavia funktioita voidaan kutsua ja silloin, kun sopimus ei ole tauolla, whenNotPaused-määritteen omaavia funktioita voidaan kutsua.

3.5 Ylläpitomallit

Sopimusten päivittäminen uuteen versioon on mahdotonta, mikäli tällaista toiminnallisuutta ei ole etukäteen suunniteltu. Ylläpitomallit tarjoavat käytäntöjä, miten sopimuksia voidaan päivittää uuteen versioon.

```
contract ContractRegister is Ownable {
    address backendContract;
    address[] previousBackends;

    constructor () internal {
        _owner = msg.sender;
    }

    /// Funktio, jolla voidaan vaihtaa backend-sopimus
    /// @param newBackend Uuden backend-sopimuksen osoite
    function changeBackend(address newBackend) public
    onlyOwner()
    returns (bool)
    {
        if(newBackend != backendContract) {
            previousBackends.push(backendContract);
            backendContract = newBackend;
            return true;
        }

        return false;
    }
}
```

Ohjelma 3.13. *Esimerkki contract register -suunnittelumallista, lähteestä [25]*

3.5.1 Register

Sopimusrekisterin avulla voidaan pitää kirjaa sopimuksen edellisistä versioista ja päivittää sopimus uuteen versioon. Ideana sopimusrekisterissä on, että sopimusta käyttävät tahot kysyvät sopimusrekisteristä sopimuksen uusimman version osoitteen aina, kun he haluavat käyttää sopimusta[25].

Yksinkertainen esimerkki siitä, miten sopimuksen voi tallettaan rekisteri-sopimukseen on esitetty Ohjelmassa 3.13. Sopimuksen omistaja voi vaihtaa, mihin sopimukseen rekisteri viittaa ja sopimuksen edellisten versioiden osoitteet tallennetaan sopimukseen.

Sopimusrekisteriä käytettäessä tulee käyttäjien muistaa aina kysyä sopimusrekisteristä sopimuksen uusin versio. Kun sopimusrekisteriin halutaan lisätä uusi versio sopimuksesta, tulee miettiä, mitä tehdään vanhan sopimuksen sisältämän datan kanssa.[25]

3.5.2 Proxy

Vaikka ei olekaan mahdollista päivittää jo Ethereum-verkkoon julkaistun sopimuksen koodia, on mahdollista konfiguroida proxy-sopimus, jonka läpi kaikki kutsut sopimukseen tehdään. Tällä tavalla sopimuksen logiikka voidaan päivittää muuttamalla sopimus, johon proxy-sopimus viittaa. Proxy-sopimuksen avulla käyttäjät voivat aina lähettää transaktiot samaan osoitteeseen, koska proxy-sopimus ohjaa ne sopimukseen, jossa on päivitetty logiikka.[14]

```
contract Proxy {
  /// @return Osoite, jolle kutsu delegoidaan.
  function implementation() public view returns (address);

  // Fallback-funktio, joka delegoi sopimukseen saapuneen kutsun osoitteeseen,
  /// jonka implementation() palauttaa.
  function () payable public {
    address _impl = implementation();
    require(_impl != address(0));

    assembly {
      let ptr := mload(0x40)
      calldatacopy(ptr, 0, calldatasize)
      let result := delegatecall(gas, _impl, ptr, calldatasize, 0, 0)
      let size := returndatasize
      returndatacopy(ptr, 0, size)

      switch result
      case 0 { revert(ptr, size) }
      default { return(ptr, size) }
    }
  }
}
```

Ohjelma 3.14. *Esimerkki proxy-suunnittelumallista, lähteestä [19]*

Perustuvanlaatuinen toteutus proxy-mallille on esitetty Ohjelmassa 3.14. implementation-funktio palauttaa osoitteen, jossa sopimus, jolle transaktio delegoidaan sijaitsee. Tämän toteuttamisessa voisi käyttää esimerkiksi sopimusrekisteri-mallia. Implementation-funktio palauttaisi siis sopimusrekisteristä logiikka-sopimuksen viimeisimmän toteutuksen osoitteen.

Kun sopimukseen saapuu transaktio, jonka funktio-valitsin ei vastaa mitään kutsutun sopimuksen funktiosta, kutsutaan sopimuksen fallback-funktiota. Proxy-sopimuksessa tämä funktio ohjaa transaktion sopimukselle, johon varsinainen logiikka on toteutettu.[14]

Tässä fallback-funktiossa haetaan ensiksi logiikka-sopimuksen osoite ja tarkistetaan, että se on asetettu. Tämä proxy-sopimuksen toteutus perustuu matalan tason delegatecall-kutsuun. Delegatecall-kutsu mahdollistaa sen, että sopimus voi kutsua toisen sopimuksen funktioita ikään kuin ne olisivat sen omia. Soliditystä löytyy itsestään delegatecall-funktio, mutta sen avulla ei pysty palauttamaan kutsutun funktion palauttamaa arvoa, sillä se palauttaa vain totuuarvomuuttujan. Soliditystä löytyy kumminkin mahdollisuus käyttää assembly-kieltä, joka muistuttaa läheisesti Ethereumin virtuaalikoneen assemblyä. Tämän avulla saadaan proxy-sopimus palauttamaan logiikka-sopimuksen palauttama arvo.[14]

Muistiosoiteeseen 0x40 on solidityssä tallennettu seuraavan vapaan muistiosoittimen arvo. Tämä muistiosoitin tallennetaan ptr-muuttujaan. Kun seuraava vapaa muistiosoite on tiedossa, voidaan käyttää calldatacopy-komentoa. Tällä komennolla voidaan kopioida transaktion data-kentässä oleva informaatio muistiin. Calldatasize-komento palauttaa transaktion data-kentässä olevan informaation koon tavuissa. Calldatacopy kopioi tässä tapauksessa calldatasize-arvon verran tavuja ptr-muuttujan kertomaan sijantiin alkaen

transaktion data-kentän sijainnista 0.[14]

Seuraavaksi suoritetaan itse funktio-kutsu delegatecall-komennolla. Ensimmäinen argumentti kertoo, kuinka paljon Gasiä kutsulle annetaan käyttöön. Tässä käytetty gas-komento palauttaa suoritukseen jäljellä olevan Gasin määrän, eli kutsulle välitetään kaikki suorituksessa jäljelläoleva Gas. Seuraava argumentti on kutsuttavan sopimuksen osoite. Kaksi seuraavaa kenttää spesifioi kutsun syötteen. Syötteeksi asetetaan ptr-muuttujan kertomasta sijainnista alkaen calldatasize-komennon palauttaman arvon verran tavuja; eli kaikki alkuperäisen transaktion data. Kaksi viimeistä argumenttia kertoisivat, mihin sijaintiin muistissa kutsun palauttama arvo tallennetaan ja montako tavua palautetusta arvosta tallennetaan. Näitä ei käytetä, sillä ei ole vielä tiedossa, kuinka suuri palautettu arvo on kooltaan. Delegatecall palauttaa joko arvon 1, mikäli kutsu onnistui, tai arvon 0, mikäli se epäonnistui. Tämä arvo tallennetaan result-muuttujaan.[14]

Seuraavaksi tallennetaan palautetun datan koko tavuissa size-muuttujaan returndatasize-komennolla. Kun sen koko on tiedossa, se voidaan kopioida muistiin returndatacopy-komennolla. Lopuksi switch-lause joko palauttaa kutsun palauttaman datan, tai heittää poikkeuksen, mikäli jokin meni vikaan.[14]

4 YHTEENVETO

Tässä työssä tutkittiin Ethereum-älysopimusten suunnittelumalleja. Aluksi käytiin läpi Ethereum-alustan toimintaperiaate, jonka jälkeen käytiin läpi joitakin suunnittelumalleja, joita voidaan hyödyntää Ethereum-alustan päälle luotavissa projekteissa.

Älysopimusten muuttumattomuus luo useita haasteita, jotka tulee ottaa huomioon. Älysopimuksia suunniteltaessa tarvitsee olla erittäin kaukokatseinen, sillä edes niiden päivittäminen ei välttämättä onnistu, mikäli sitä varten ei ole etukäteen luotu jotakin järjestelmää. Suunnitteluvaiheessa tulee myös miettiä, halutaanko sopimuksen toiminta jossakin kohtaa pysäyttää.

Koska älysopimukset usein sisältävät rahanarvoista Etheriä tai tokeneita, pahanaikeisilla toimijoilla on suuri insentiivi yrittää löytää bugeja sopimuksista, joita he voivat hyväksikäyttää. Turvallisen koodin luominen älysopimuksiin on täten erittäin tärkeää. Suunnittelumallien hyödyntäminen ja parhaiden käytäntöjen noudattaminen vähentää merkittävästi riskiä, että älysopimuksesta löytyisi haavoittuvaisuus, jota voidaan hyväksikäyttää.

Älysopimuksiin tulee usein määritellä ehtoja, joiden perusteella älysopimus voi estää jonkin toiminnon suorittamisen. Valtuutusmalleja hyödyntämällä tällainen toiminnalisuus voidaan toteuttaa.

Mikäli halutaan luoda oma tokeni, kannattaa sille toteuttaa standardoitu rajapinta. Tämä mahdollistaa tokenin integraation lompakko-applikaatioihin ja helpottaa muiden kehittäjien työtä luoda tokenia hyödyntäviä applikaatioita.

Tässä työssä esitetyt esimerkkitoteutukset ovat suunnitelumallien havainnollistamista varten eikä niitä tule sellaisenaan käyttää ohjelmissa. Sen sijaan, että tekisi itse toteutukset yleisistä suunnittelumalleista, kannattaa käyttää esimerkiksi OpenZeppelin-kirjaston tarjoamia toteutuksia, mikäli se vain on mahdollista. OpenZeppelin-kirjaston tarjoamat sopimukset ovat turvallisuustarkastettuja ja täten todennäköisesti paljon turvallisempia käyttää kuin itse tehdyt toteutukset. Vaikka käyttäisikin turvallisuustarkastettuja toteutuksia suunnittelumalleista, täytyy silti olla hyvä käsitys siitä, miten ne toimivat.

LÄHTEET

[1] A. M. Antonopoulos ja G. Wood. *Mastering Ethereum*. O'reilly Media, 2018.

[2] V. Buterin. *A Next-Generation Smart Contract and Decentralized Application Platform*. 2014. URL: https://github.com/ethereum/wiki/wiki/White-Paper (viitattu 07. 07. 2019).

[3] W. Entriken, D. Shirley, J. Evans ja N. Sachs. *EIP 721: ERC-721 Non-Fungible Token Standard*. URL: https://eips.ethereum.org/EIPS/eip-721 (viitattu 25. 08. 2019).

[4] Ethereum Foundation. *EIP 20: ERC-20 Token Standard*. URL: https://eips.ethereum.org/EIPS/eip-20 (viitattu 22. 08. 2019).

[5] Ethereum Foundation. *Mining*. URL: https://github.com/ethereum/wiki/wiki/Mining (viitattu 20. 07. 2019).

[6] Ethereum Foundation. *Programming languages*. URL: https://github.com/ethereum/wiki/wiki/Programming-languages-intro (viitattu 11. 08. 2019).

[7] Ethereum Foundation. *Safety*. URL: https://github.com/ethereum/wiki/wiki/Safety (viitattu 28. 08. 2019).

[8] Ethereum Foundation. *Security Considerations*. URL: https://solidity.readthedocs.io/en/v0.5.11/security-considerations.html (viitattu 03. 09. 2019).

[9] J. Frankenfield. *Initial Coin Offering (ICO)*. URL: https://www.investopedia.com/terms/i/initial-coin-offering-ico.asp (viitattu 11. 08. 2019).

[10] E. Gamma, R. Helm, R. Johnson ja J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[11] A. Hayes. *Initial Public Offering (IPO)*. URL: https://www.investopedia.com/terms/i/ipo.asp (viitattu 11. 08. 2019).

[12] A. Hertig. *Loveable Digital Kittens Are Clogging Ethereum's Blockchain*. URL: https://www.coindesk.com/loveable-digital-kittens-clogging-ethereums-blockchain (viitattu 25. 08. 2019).

[13] icodata.io. *Funds raised in 2018*. URL: https://www.icodata.io/stats/2018 (viitattu 11. 08. 2019).

[14] E. Nadolinski ja F. Spagnuolo. *Proxy Patterns*. URL: https://blog.openzeppelin.com/proxy-patterns/ (viitattu 01. 09. 2019).

[15] S. Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System*. 2008. URL: https://bitcoin.org/bitcoin.pdf (viitattu 07. 07. 2019).

[16] OpenZeppelin. *IERC721*. URL: https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC721/IERC721.sol (viitattu 25. 08. 2019).

[17] OpenZeppelin. *Ownable*. URL: https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/ownership/Ownable.sol (viitattu 25. 08. 2019).

[18] OpenZeppelin. *Pausable*. URL: https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/lifecycle/Pausable.sol (viitattu 31. 08. 2019).

[19] OpenZeppelin. *Proxy*. URL: https://github.com/OpenZeppelin/openzeppelin-labs/blob/master/upgradeability_using_eternal_storage/contracts/Proxy.sol (viitattu 01. 09. 2019).

[20] J. Peterson, J. Krug, M. Zoltu, A. K. Williams ja S. Alexander. *Augur: a Decentralized Oracle and Prediction Market Platform*. 2018. URL: https://www.augur.net/whitepaper.pdf (viitattu 12. 08. 2019).

[21] Solidity team. *Solidity documentation: Contracts*. URL: https://solidity.readthedocs.io/en/v0.5.11/contracts.html (viitattu 30. 08. 2019).

[22] N. Szabo. *Smart Contracts*. 1994. URL: http://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart.contracts.html (viitattu 10. 08. 2019).

[23] The CryptoKitties Team. *CryptoKitties: Collectible and Breedable Cats Empowered by Blockchain Technology*. 2017. (Viitattu 25. 08. 2019).

[24] TrustToken. *TrueUSD*. URL: https://www.trusttoken.com/trueusd (viitattu 12. 08. 2019).

[25] M. Wöhrer ja U. Zdun. Design Patterns for Smart Contracts in the Ethereum Ecosystem. *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)* (2018), s. 1513-1520.

[26] G. Wood. *Ethereum: A secure decentralised generalised transaction ledger*. 2019. URL: https://ethereum.github.io/yellowpaper/paper.pdf (viitattu 07. 07. 2019).