TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Rando Mihkelsaar 153501IAPM

RÄSIFUNKTSIOONIL PÕHINEVA DIGITAALSE SIGNATUURISKEEMI REALISATSIOON RIISTVARALISEL TURVAMOODULIL

Magistritöö

Juhendaja: Andres Ojamaa
PhD

Kaasjuhendaja: Ahto Truu
MSc

Tallinn 2020

Autorideklaratsioon

Kinnitan, et olen koostanud antud lõputöö iseseisvalt ning seda ei ole kellegi teise poolt varem kaitsmisele esitatud. Kõik töö koostamisel kasutatud teiste autorite tööd, olulised seisukohad, kirjandusallikatest ja mujalt pärinevad andmed on töös viidatud.

Autor: Rando Mihkelsaar

13.05.2020

Annotatsioon

Käesolev magistritöö keskendub räsifunktsioonil põhineva BLT-signatuuriskeemi prototüübi loomisele kiipkaardi jaoks. Selleks antakse ülevaade olemasolevatest räsifunktsioonidel põhinevatest signatuuriskeemidest ja analüüsitakse loodava prototüübi funktsionaalseid ja mittefunktsionaalseid nõudeid kasutades agentorienteeritud modelleerimise vahendeid. UML-ist tuttavate skeemide abi kirjeldatakse prototüübi komponente, sõnumivahetusprotokolli ja andmemudeleid. Prototüüp realiseeritakse Java Card platvormil ning lisaks kirjeldatakse BLT-signatuuriskeemi realiseerimiseks kasutatud võtmete, Merkle'i puu ja autentimistee koostamise algoritme. Peale prototüübi arendamist võrreldakse loodud signatuuriskeemi teiste sigantuuriskeemidega.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 80 leheküljel, 7 peatükki, 21 joonist, 19 tabelit.

Abstract

Implementation of a digital signature scheme based on a hash function for a hardware security module

In order to ensure the integrity and authenticity of electronically exchanged messages, many information systems rely on RSA, DSA and ECDSA signature schemes. In 1994, Peter W. Shor demonstrated that these signature scheme algorithms are vulnerable to quantum computer attacks. Although there is no real quantum computer that is capable of using Shor's algorithm, such computers are being developed and improved with admirable speed. In view of this, there is a real need for signature schemes that are immune to known quantum computer hacks.

The main objective of this master's thesis is to create a prototype of a BLT signature scheme for a smart card. To this end, we first give an overview of existing signature schemes, hash functions, and Java Card platform architecture for the smart card. Next, we apply an agent-oriented modeling to analyze the functional and non-functional requirements of the prototype. We use UML to describe the components of the prototype, the messaging protocol between the components and data models. We employ the Java Card platform to implement the prototype. In addition, we give an overview of the algorithms that are used to generate BLT signature scheme keys, the Merkle tree, and authentication path. Last but not least, we compare the properties of the BLT signature scheme with other signature schemes.

The thesis is in Estonian and contains 80 pages of text, 7 chapters, 21 figures, 19 tables.

Lühendite ja mõistete sõnastik

Signatuuriskeem	Algoritmide kogum, mis võimaldab digitaalallkirjade kasutamist. Sisaldab algoritme võtmepaari genereerimiseks, allkirjastamiseks ja allkirja verifitseerimiseks.
eIDAS	*electronic IDentification, Authentication and trust Services;* Euroopa Parlamendi ja Euroopa Liidu Nõukogu määrus e-identimise ja e-tehingute jaoks vajalike usaldusteenuste kohta.
RSA	*Rivest-Shamir-Adleman*; asümmeetrilist võtit kasutav krüptograafiline süsteem, mille tugevus põhineb suurte arvude tegurdamise keerukusel.
DSA	*Digital Signature Algorithm*; digitaalsignatuuri algoritm; signatuuriskeem, mis põhineb diskreetlogaritmide arvutamise keerukusel.
ECDSA	*Elliptic Curve Digital Signature Algorithm;* signatuuriskeem, mis põhineb elliptkrüptograafial.
XMSS	*eXtended Merkle Signature Scheme;* räsifunktsioonil põhinev signatuuriskeem.
SPHINCS	Räsifunktsioonil põhinev signatuuriskeem, kus pole vaja hoida olekut kasutatud võtmetest.
BLT	*Buldas-Laanoja-Truu;* räsifunktsioonidel põhinev signatuuriskeem, mis kasutab allkirja koostamiseks serveri abi.
UK	Uurimisküsimus.
CI	*Continuous Integration*; pidev lõiming.
UML	*Unified Modeling Language*; ühtne modelleerimiskeel; üldotstarbeline visuaalne keel peamiselt tarkvaraprojektide kirjeldamiseks.
AOM	*Agent-Oriented Modeling*; agentorienteeritud modelleerimine.
Räsifunktsioon	Deterministlik ühesuunaline funktsioon, mis seab ükskõik kui suurele sisendandmele vastavusse fikseeritud suurusega väljundi nii, et iga väiksemgi muutus sisendandmetes põhjustab suuri muutusi väljundandmetes.
SHA	*Secure Hash Algorithm*; turvaline räsialgoritm.
SHA-2	Krüptograafiliste räsifunktsioonide perekond.
SHA-256	SHA-2 krüptograafiliste räsifunktsioonide perekonda kuuluv räsifunktsioon.
SHA-3	Krüptograafiliste räsifunktsioonide perekond.
Merkle'i puu	Kahendpuu, kus lehtedeks on andekogumite räsid ja iga sisetipu väärtus on sellele vahetult alluvate tippude väärtuste konkatenatsiooni räsi.

W-OTS	*Winternitz One-Time Signature;* räsipõhine signatuuriskeem, kus privaatvõtit saab allkirjastamiseks kasutada üks kord.
W-OTS^{+}	W-OTS sigantuuriskeemi edasiarendus.
XMSS^{MT}	*XMSS Multi-Tree;* XMSS signatuuriskeemi edasiarendus.
HORST	Räsipõhine signatuuriskeem, kus allkirjastamisvõtit saab allkirjastamiseks kasutada paar korda.
SPHINCS-256	SPHINCS signatuuriskeem.
SPHINCS+	SPHINCS signatuuriskeemi edasiarendus.
BLT-TB	Räsipõhine signatuuriskeem, kus igat allkirjastamisvõtit saab allkirjastamiseks kasutada kindlal ajahetkel. Allkirja koostamiseks kasutatakse serveri abi.
BLT-OT	BLT-signatuuriskeem, mis kasutab allkirjastamiseks ühekordseid võtmeid.
BLT-OT-N	BLT-signatuuriskeem, kus BLT-OT võtmetest on koostatud Merkle'i puu.
GSM	*Global System for Mobile communications*; ülemaailmne mobiilside süsteem.
Java Card platvorm	Java programmeerimiskeelel põhinev platvorm kiipkaardirakenduste kirjutamiseks.
JCRE	*Java Card Runtime Environment.*
JCVM	*Java Card Virtual Machine.*
API	*Application Programming Interface*; rakendusprogrammiliides.
ROM	*Read-Only Memory*; püsimälu.
RAM	*Random Access Memory*; muutmälu.
EEPROM	*Electrically Erasable Programmable Read-Only Memory;* elektriliselt kustutatav programmeeritav püsimälu.
APDU	*Application Protocol Data Unit*; rakenduse protokollo andmeüksus.
PIN	*Personal Identification Number;* lühike parool, mis koosneb kümnendnumbritest.
HTTP	*Hypertext Transfer Protocol; h*üperteksti edastuse protokoll.
MAC	*Message Authentication Code;s*õnumiautentimiskood.
HMAC	*Hash-based Message Authentication Cod*; räsipõhine sõnumiautentimiskood.
POSIX	*Portable Operating System Interface.*
AES	*Advanced Encryption Standard*; progressiivne krüpteerimisstandard.
ATR	*Answer To Reset.*
TLV	*Type-Length-Value*; vorming, mida kasutatakse BLT-allkirjade salvestamiseks.

Sisukord

Jooniste loetelu

Tabelite loetelu

1 Sissejuhatus

Antud magistritöö eesmärgiks on realiseerida räsifunktsioonil põhinev digitaalse signatuuriskeemi prototüüp riistvaralisel turvamoodulil, mis oleks immuunne teada olevatele kvantarvutite rünnetele. Selleks vaadeldakse olemasolevaid asümmeetrilisi ning räsifunktsioonil põhinevaid signatuuriskeeme, analüüsitakse loodava süsteemi funktsionaalseid ja mittefunktsionaalseid nõudeid, kirjeldatakse loodava süsteemi arhitektuuri, sõnumivahetuse protokolli ja andmestruktuure. Peale prototüübi realiseerimist võrreldakse tulemust olemasolevate lahendustega.

1.1 Taust ja probleem

Digitaalseid allkirju kasutatakse mitmetes infosüsteemides tagamaks elektrooniliselt vahetatavate sõnumite terviklust ja autentsust. Samuti on eIDAS (ingl *electronic IDentification, Authentication and trust Services*) määrus [1] loonud Euroopa Liidus eeldused personaalsete ja ettevõtete piiriüleste digitaalsete allkirjade levikuks. Mitmed riigid peale Eesti on hakanud välja andma kiipkaarti millega on võimalik anda digitaalseid allkirju, mis on samaväärsed käsitsi antud allkirjadega (näiteks Saksamaa, Portugal ja Slovakkia [2]). Antud personaliseeritud kiipkaardid võimaldavad inimestel ja ettevõtetel asendada käsitsi antavad allkirjad elektroonilistega, mis on kiire, mugav ning võimaldab säästa aega ja raha.

Asümmeetrilised krüptograafilised süsteemid (nt. RSA [3], DSA [4] ja ECDSA [5]) on levinumad viisid digitaalsete allkirjade andmiseks. Antud süsteemid koosnevad kasutaja matemaatiliselt seotud avalikust ja privaatsest võtmetest, kus avalikku võtit kasutatakse allkirjade valideerimiseks ning privaatset võtit allkirjastamiseks. Tihtipeale hoitakse asümmeetrilisi privaatseid võtmeid riistvaralises allkirjastamisseadmes (ingl *hardware security module*), kus toimub ka allkirjastamine. Kõige lihtsam ja odavaim riistvaraline allkirjastamisseade on kiipkaart.

Asümmeetriliste süsteemide turvalisus põhineb eeldusel, et avalikust võtmest privaatse võtme tuletamine on ründajale arvutuslikult ületamatult kulukas. Peter W. Shor näitas oma töös [6], et RSA, DSA ja ECDSA salajase võtme tuletamine avalikust võtmest on kvantarvutil efektiivselt lahendatav. Kuigi reaalset kvantarvutit, mis suudaks Shor'i algoritmi tõhusalt kasutada veel ei eksisteeri, siis on olemas reaalne vajadus allkirjastamise algoritmide järele, mis on immuunsed teada olevatele kvantarvuti rünnetele.

Praeguste teadmiste kohaselt on räsifunktsioonid ja sümmeetrilised algoritmid, mis kasutavad andmete krüpteerimiseks ja dekrüpteerimiseks ühte ja sama salajast võtit, kvantrünnete suhtes vastupidavamad. Kuigi räsifunktsioonidel põhinevaid signatuuriseekme on mitmeid (nt. XMSS (*eXtended Merkle Signature Scheme*) [7], SPHINCS [8]), siis on need võrreldes asümmeetriliste skeemidega tihtipeale ebaefektiivsed (allkirjastamine võtab liiga kaua aega või allkiri ise on asümmeetriliste algoritmidega antud allkirjadega võrreldes mahult suur).

Ahto Buldas, Denis Firsov, Risto Laanoja, Henri Lakk ja Ahto Truu tutvustasid oma töödes BLT nime kandvat signatuuriskeemi [8, 9], mis kasutab räsifunktsioone ning allkirjade efektiivsemaks andmiseks ja allkirja suuruse vähendamiseks serveri abi. Niinimetatud serveripõhises signatuuriskeemis on allkirja moodustamiseks vaja kasutaja, kes omab enda privaatset võtit, ning serveri koostööd. Erinevalt teistest asümmeetrilistest skeemidest kasutab BLT allkirjastamiseks ajaga seotud ühekordseid võtmeid. BLT tõestab iga võtme kasutamise ajatempliga ja takistab teisi osapooli võtmeid väärkasutamast, tühistades iga võtme kohe pärast kasutamist.

Kuigi esialgu võib tunduda, et serveri abi kasutamine allkirja koostamisel on miinus, siis tegelikult vajavad ka teised asümmeetrilisel krüptograafial põhinevad signatuuriskeemid serveri abi. Selleks, et tõestada allkirjastamise ajahetke (näiteks juriidiliste dokumentide allkirjastamisel) ja allkirjastamiseks kasutatud võtme kehtivust allkirjastamise hetkel, lisatakse allkirja juurde ajatempel ning võtme kehtivuskinnitus. Ajatempli ja kehtivuskinnituse saamiseks kasutatakse usaldusväärseid kolmandaid osapooli [11]–[13].

1.2 Ülesande püstitus

Antud magistritöö keskendub BLT-põhise signatuuriskeemi kliendipoolsele osale, mis kasutab allkirjastamiseks kasutatavate võtmete genereerimiseks ja hoidmiseks kiipkaarti. Kuna BLT kasutab allkirjastamiseks ühekordseid võtmeid, siis on nende genereerimine ning efektiivne kasutamine allkirjastamiseks seadmetel, millel on vähe arvutamisvõimust ja salvestusmahtu, raskendatud. Töö eesmärgiks on demonstreerida, et BLT-signatuuriskeemi jaoks on võimalik luua kiipkaardirakenduse prototüüp, mida saab efektiivselt kasutada sõnumite allkirjastamiseks.

Töö peamine uurimisküsimus (UK) on sõnastatud järgmiselt: „**Kuidas efektiivselt realiseerida BLT-põhiste võtmete genereerimine ja sõnumite allkirjastamise prototüüp kasutades kiipkaarti**”? Vastamaks peamisele uurimisküsimusele struktureeritult, on see jagatud järgmisteks alamküsimusteks:

- **UK-1: Millised on funktsionaalsed ja mittefunktsionaalsed nõuded BLT-kliendipoolse rakenduse realiseerimiseks?**
- **UK-2: Millist sõnumivahetuse protokolli ja andmestruktuure on vaja BLT-kiipkaardi rakenduse prototüübi realiseerimiseks?**
- **UK-3: Kuidas realiseerida BTL-signatuuriskeemi võtmehalduse ja allkirjastamise moodul?**

Esimesele alamküsimusele vastamiseks kasutatakse AOM (*Agent-Oriented Modeling*) mudelipõhist lähenemist [14]. Teisele alamküsimusele vastamiseks kasutatakse UML-i (*Unified Modeling Language*) [15] vahendeid. Viimasele alamküsimusele vastamiseks realiseeritakse *Java Card* [16] platvormil töötav kiipkaardirakendus, mis tegeleb BLT-võtmete haldamise ja BLT-allkirjade loomisega, ja serveriga suhtlemise moodulit sisaldav kliendirakenduse prototüüp. Lisaks võrreldakse loodud BLT signatuuriskeemi efektiivsust teiste olemasolevate signatuuriskeemide realisatsioonidega.

1.3 Metoodika

BLT-signatuuriskeemi prototüübi arendamisel lähtutakse väledast (ingl *agile*) arendusmeetoodikast, kus tarkvaraga seotud sammud (nõuete analüüs, süsteemi disainimine, tarkvara arendus, testimine) on jagatud iteratsioonideks. Tarkvara arenamisel lähtutakse tabelis 1 välja toodud ekstreemprogrammeerimise (ingl e*xtrme programming*) tarkvaarendusmetoodika praktikatest [17], [18].

Tabel 1. Tarkvaraarendusmetoodika praktikad.

Praktika	**Seletus**
Töö planeerimine (ingl *planning game*)	Praktika eesmärk on enne arendust kirjutada, hinnata ning prioritiseerida tööülesanded.
Lihtne disain (ingl *simple design*)	Praktika lähtub sellest, et tarkvara disain peaks olema võimalikult lihte algusest peale. Antud lähenemine tagab, et ei tehta üleliigseid töid ning tarkvara valmib suurema tõetäosusega õigeks ajaks.
Testimine	Testimisel lähtutakse testjuhitud arenduse (ingl *test driven development*) praktikast [19], kus tarkvaraarendaja kirjutab enne koodi ühiktesti.
Kollektiivne omand (ingl *collective code ownership*)	Kuigi käesolevas projektis on üks arendaja, siis kogu koodi hoitakse *Github* [20] koodirepositooriumis.
Pidev lõiming (ingl *continuous integration (*CI*)*)	Praktika lähtub sellest, et peale igat koodimuudatust tarkvara integreeritakse ning testitakse. Projektis kasutatakse *CI* tööriistana *Github Actions* [21] nimelist töökeskonda, mis on integreeritud *Github* koodirepositooriumiga. Peale uue koodi avalikustamist käivitatakse kõik projekti koodiga kaasas olevad testid.
Disaini täiustamine (ingl *refactoring*)	Disaini täiustamise käigus täiustatakse tarkvara koodi (nt korduste eemaldamine) nii, et säiliks koodi funktsionaalsus. Pidev disaini täiustamine tagab parema arusaadavuse tarkvara disainist ning lihtsustab tuleviku arendusi.
Ühtne kodeerimisstandard (ingl *coding standard*)	Praktika lähtub sellest, et kõik projektis osalevad tarkvaarendajad järgivad ühtse koodi kirjutamise standardit/juhiseid.
Sobiv tempo (ingl *sustainable pace*)	Praktika lähtub sellest, et programmeerijad teevad paremat tööd siis, kui nad on puhanud.

Ekstreemprogrammeerimise arendusmetoodikas on veel mitmeid teisi praktikaid, mida käesoleva töö iseloomu tõttu on raske rakendada. Paarisprogrammeerimise (ingl *pair programming*) praktikat on antud projekti puhul võimatu rakendada, kuna selles projektis tegeleb koodi kirjutamisega ainult üks inimene. Kuna projektil puudub reaalne klient, siis on raske järgida kliendi kohaloleku (ingl *on-site customer*) ja väikeste redaktsioonide (ingl *small releases*) ekstreemprogrammeerimise praktikaid.

Prototüübi ja magistritöö haldamiseks kasutatakse *Kanban* projektihaldusmeetodit [22]. *Kanban* on levinud projektihaldusmetoodika, mida kasutatakse peale tarkvaraarendus mitmetest teistes tegevusvaldkondades. *Kanban* keskendub peamiselt töövoo visualiseermisele ning piirab hetkel töös olevate ülesannete hulka. Töö koostamise käigus kasutatakse tööülesannete haldamiseks *Github Issues* [23] ja *Kanban* tahvlina *Github Projects* [23] keskkonda. *Kanban* tahvlil on tööülesanded jaotatud nelja erinevasse kategooriasse: „*teha*", „*töös*", „*ootab tagasisidet*" ning „*tehtud*". Tööülesande liikumine ühest kategooriast teise on saab toimuda eeldefineeritud tingimustel. Näiteks kategooriast „töös" kategooriasse „ootab tagasisidet" saab tööülesannet liikuda siis, kui see on teostatud ning kategooriast „teha" kategooriasse „töös" saab tööülesanne liikuda ainult siis, kui tööülesande täitmiseks on piisavalt sisendit ja kategoorias „töös" pole rohkem kui kaks tööülesannet.

Projektiga seotud nõuete haldamiseks kasutatakse AOM mudelipõhist lähenemist, millest antakse ülevaade jaotises 3.2. Prototüübi disaini kirjeldamiseks kasutatakse UML-i vahendeid, millest antakse täpsem ülevaade jaotises 4.2.

1.4 Ülevaade tööst

Ülejäänud magistritöö struktuur on järgmine. Teises peatükis antakse põhjalik ülevaate seotud kirjandusest ja teoreetilisest taustast. Kolmas peatükk keskendub BLT-signatuuriskeemi funktsionaalsetele ja mittefunktsionaalsetele nõuetele. Neljandas peatükis kirjeldatakse BLT allkirjastamise protsesse ja andmestruktuure. Viiendas peatükis käsitletakse BLT-kiipkaardirakenduse ja kliendi arvutis oleva tarkvara loomist. Kuuendas peatükis analüüsitakse loodud BLT-kiipkaardirakendust, võrreldes seda teiste kiipkaardil töötavate allkirjastamisrakendustega. Viimane peatükk võtab kokku tehtud töö.

2 Teoreetiline taust ja kirjanduse ülevaade

Antud peatükk annab ülevaate seotud kirjandusest ja teoreetilisest taustast. Jaotis 2.1 tutvustab krüptograafilisi räsifunktsioone, jaotis 2.2 kirjeldab Merkle'i puud, jaotis 2.3 keskendub asümmeetrilistele signatuuriskeemidele, jaotis 2.4 vaatleb krüptograafilisi ajatempleid, jaotis 2.5 tutvustab enamlevinumaid räsifunktsioonidel põhinevaid signatuuriskeeme ning viimases peatükis antakse ülevaade kiipkaardist.

2.1 Krüptograafilised räsifunktsioonid

Krüptograafiline räsifunktsioon on deterministlik ühesuunaline funktsioon, mis seab ükskõik kui suurele sisendandmele vastavusse fikseeritud suurusega väljundi nii, et iga väiksemgi muutus sisendandmetes põhjustab suuri muutusi väljundandmetes. Räsifunktsiooni sisendit kutsutakse *sõnumiks* ning väljundit *räsiks* (ingl *hash*) või *sõnumilühendiks* (ingl *message digest*). Heal krüptograafilisel räsifunktsioonil H on järgmised omadused [24] :

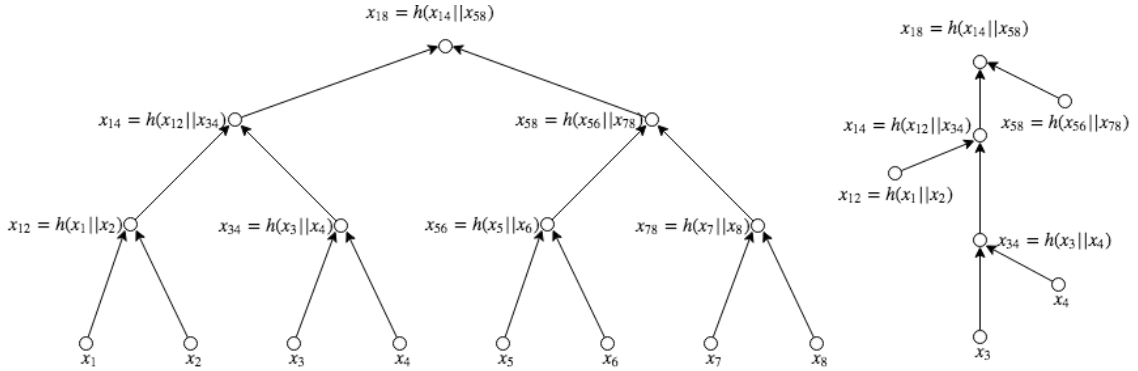
- iga sõnumi m korral peab räsi $d = H(m)$ olema efektiivselt arvutatav;
- ükskõik millise etteantud räsi d korral ei tohi praktikas olla võimalik leida sellega sobivat sõnumit m , nii et $H(m) = d$; räsifunktsiooni, millel on antud omadus, kutsutakse *originaalikindlaks* (ingl *preimage resistant*);
- mistahes etteantud sõnumi m_1 korral ei tohi praktikas olla võimalik leida sõnumit m_2 , nii et $m_1 \neq m_2$ ja $H(m_1) = H(m_2)$; räsifunktsioon, millel on antud omadus, kutsutakse *lisaoriginaalikindlaks* (ingl *second preimage resistant*);
- ei tohi olla leitav kahte sõnumit m_1 ja $m2$, nii et $m_1 \neq m_2$ ja $H(m_1) = H(m_2)$; räsifunktsiooni, millel on antud omadus, kutsutakse *kollisioonivabaks* (ingl *collision resistant*).

Krüptograafilised räsifunktsioonid on tänapäeval laialdaselt levinud. Näiteks kasutatakse neid asümmeetriliste signatuuriskeemide juures sisendandmete räsimiseks ja paroolide salvestamiseks andmebaasi. SHA-2 [25] ja SHA-3 [26] perekonda kuuluvad räsifunktsioonid on head näited praktikas kasutatavatest krüptograafilistest räsifunktsioonidest.

2.2 Merkle'i puu

Merkle'i puu ehk *räsipuu* on kahendpuu, kus lehtedeks on andekogumite räsid ja iga sisetipu väärtus on sellele vahetult alluvate tippude väärtuste konkatenatsiooni räsi. Antud andmestruktuuri tutvustas esimest korda Ralph Merkle 1979. aastal, et efektiivselt allkirjastada mitut dokumenti, mis on kaetud sama digitaalse allkirjaga [27], [28].

Merkle'i puu võimaldab genereerida tõestuse ehk *autentimistee (räsiahel)* pikkusega $\log n$ iga sisendräsi $x_i (1 \le i \le n)$ kohta, et näidata, et räsi x_i oli üks sisendräside hulgast $\{x_1, \ldots, x_n\}$ ja osales puu tipuräsi arvutamisel. Joonisel 1 on vasakul räsipuu lehtedeks sisendandmete räsid $x_1, \ldots, x_8$ ning vasakul on lehe x_3 autentimistee. Autentimistee võimaldab tõestada, et lehele x_3 vastav andekogum on osalenud juurräsi x_{18} arvutamisel.



Joonis 1. Merkle'i puu ja autentimistee näide.

Merkle'i puu on tänapäeval väga levinud andmestruktuur. Näiteks kasutatakse seda andmebaasisüsteemides ja failivahetusvõrgu protokollis tervikluse tagamiseks [29].

2.3 Asümmeetrilised signatuuriskeemid

Asümmeetrilised signatuuriskeemid, mis on tuntud ka kui avaliku võtme krüptograafial põhinevad signatuuriskeemid, koosnevad omavahel matemaatiliselt seotud avalikust ja privaatsest võtmest, kus privaatset võtit kasutatakse sõnumi allkirjastamiseks ning avallikku võtit allkirja kontrollimiseks [24]. Kuna tavaliselt on asümmeetrilised signatuuriskeemid suurte sisendite puhul aeglased, siis enne allkirjastamist sõnum räsitakse kasutades krüptograafilist räsifunktsiooni. Järgmisena rakendatakse räsi väärtusele signeerimisfunktsiooni, mis RSA ja ECDSA puhul on krüpteerimisfunktsioon, ning lisatakse allkiri esialgsete andmete kõrvale. Allkirja kontrollimiseks räsitakse sõnum kasutades sama krüptograafilist räsifunktsiooni, mis allkirjastamisel. Järgmisena kasutatakse allkirjastaja avalikku võtit, et allkiri dekrüpteerida. Kui dekrüpteeritud allkiri on võrdne esialgsest sõnumit arvutatud räsiga, siis on allkiri kehtiv.

1978. aastal Ron Rivest'i, Adi Shamir'i ja Leonard Adleman'i poolt tutvustatud RSA algoritm [3] on üks vanim ja enimlevinuim asümmeetrilisel krüptograafial põhinev signatuuriskeem. RSA turvalisus põhineb suurte kordarvude tegurdamise raskusel. Teine laialdaselt levinud asümmeetrilisel krüptograafial põhinev signatuuriskeem on ECDSA [5], mis põhineb elliptkõveratel. Võrreldes RSA-ga on ECDSA võtmed ja allkirjad lühemad.

Peter W. Shor näitas oma töös [6], et RSA, DSA ja ECDSA salajase võtme tuletamine avalikust võtmest on kvantarvutil efektiivselt lahendatav. Kuna kvantarvuteid arendatakse ja täiendatakse pidevalt ning uute signatuuriskeemide evitamine võtab kaua aega, siis on olemas reaalne vajadus signatuuriskeemide järele, mis ei põhine eelpool tutvustatud asümmeetrilistel signatuuriskeemidel.

2.4 Krüptograafilised ajatemplid

Krüptograafilisi ajatempleid kasutatakse tõestamaks, et digitaalsed andmed eksisteerisid mingil teatud ajahetkel ning ajatempli võtjal oli nendele admetele ligipääs. Tavaliselt kasutatakse selleks usaldusväärset kolmandat osapoolt, mis lisab andmetele (või andmete räsile) õige aja ning allkirjastab saadud tulemuse. Kui usaldusväärne kolmas

osapool on andmetele lisanud ajatempli, siis ei saa antud andmeid muuta ka andmete looja ilma, et ajatempel muutuks kehtetuks.

2.4.1 Asümmeetrilistel võtmetel põhinevad ajatemplid

Asümmeetrilistel võtmetel põhinevates ajatemplisüsteemides kinnitab ajatempliteenuse pakkuja andmete eksisteerimist teatud ajahetkel, allkirjastades andmete räsi ja praeguse ajahetke oma privaatvõtmega. Kuna nendes süsteemides on ajatempel kaitstud ainult digitaalse allkirjaga, siis töötavad need ajatemplid eeldusel, et teenusepakkuja ei väljasta võltsitud ajatempleid. Asümmeetrilistel võtmetel põhinevate ajatemplite jaoks on loodud mitmeid standardeid: IETF RFC 3161 [11], ISO/IEC 18014 [30] ja ANSI ANS X9.95 [31].

2.4.2 Räside linkimisel põhinevad ajatemplid

Räside linkimisel põhinevad ajatemplid toetuvad erinevate andmete räside omavahelisel sidumisele. Haber ja Stornetta [32] kirjeldasid lahendust, kus iga ajatempel sisaldas viidet eelmisele ja järgnevale andmehulgale. Benaloh ning de Mare [33] ja Bayer, Haber ning Stornetta [34] soovitasid räsipõhisel linkimisel efektiivsuse tõstmiseks kasutada Merkle'i puud, mis võimaldas kiiremini kontrollida ajatempli õigsust. Räside linkimisel põhinevaid ajatempleid saab muuta usaldusväärsemaks, avaldades teatud ajahetkel (näiteks korra kuus) Merkle'i puu tipuräsi laialtlevinud meediakanalis (nt ajalehes). Räside linkimisel põhinevate ajatemplite skeemide turvalisust, kus ajatempliteenuse pakkujat ei pea usaldama, on tõestatud mitmetes erinevates töödes [35]–[37]. Tänapäeval on Merkle'i puul põhinevaid ajatempliteenusepakkujaid mitmeid: Surety [38], Guardtime KSI [39] ja Opentimestamps.

2.5 Räsifunktsioonidel põhinevad signatuuriskeemid

Lamport ja Merkle olid ühed esimestest, kes 1970-ndatel aastatel hakkasid tegelema räsifunktsioonidel põhinevate signatuuriskeemidega. Põhjus, miks räsifunktsioonidel põhinevad signatuuriskeemid pole võrreldes asümmeetriliste signatuuriskeemidega levinud, on selles, et tavaliselt saab igat privaatset võtit kasutada allkirjastamiseks ainult üks kord, allkirjastamiseks vajalikud võtmed ja/või allkirjad on mahult suured ning

allkirjastamiseks kuluv aeg on tunduvalt pikem võrreldes asümmeetriliste signatuuriskeemidega.

Antud jaotis annab ülevaate erinevatest enamlevinud räsifunktsioonidel põhinevatest signatuuriskeemidest.

2.5.1 Lamporti signatuuriskeem

Lamporti poolt 1979. aastal tutvustatud signatuuriskeem [40] oli üks esimesi räsifunktsioonidel põhinevaid allkirjaskeeme. Antud signatuuriskeem kasutab ühe biti allkirjastamiseks kahest n-bitisest juhuarvust koosnevat privaatset võtit k_0 ja k_1, mida saab allkirjastamiseks kasutada ainult üks kord. Avalikuks võtmeks, mis on teada kõigile osapooltele, on paar $(p_0=H(k_0), p_1=H(k_1))$, kus H on turvaline räsifunktsioon. Allkirjastamisel vaadatakse allkirjastatava biti väärtust. Kui see on 0, siis avaldatakse privaatsest võtmest väärtus $s=k_0$ ning kui biti väärtus on 1, siis avaldatakse privaatsest võtmest väärtus $s=k_1$. Peale allkirjastamist peab allkirjastaja hävitama kasutamata privaatse võtme. Allkirja s kontrollimiseks arvutatakse kõigepealt $H(s)$ ning seejärel vaadatakse allkirjastatud biti väärtust. Kui see on 1, siis peab $H(s)$ olema korrektse allkirja puhul võrdne avaliku võtmega p_0, vastasel juhul võtmega p_1 [41].

Pikema sõnumi allkirjastamiseks tuleb iga biti jaoks genereerida privaatvõti. Kui allkirjastatav sõnum s koosneb bittidest $s_{1,}s_{,2}...,s_n$ (n on bittide arv sõnumis s), siis privaatseks võtmeks on järjend juhuarvu paaridest $(k_{0,}^1 k_1^1),...,(k_{0,}^n k_1^n)$. Avalikuks võtmeks on järjend paaridest $(H(k_0^1), H(k_1^1)),...,(H(k_0^n), H(k_1^n))$. Sõnumi s allkirjastamisel signeeritakse igat sõnumi bitti s_i võtmega $(k_{0,}^i k_1^i)$ eelnevas lõigus kirjeldatud viisil. Piiramatu pikkusega sõnumite puhul võib sõnumi kõigepealt räsida ning seejärel allkirjastada sõnumi räsi.

Kui allkirjastatakse 256 bitist sõnumit, siis on allkirja suurus Lamporti allkirjaskeemis kasutades SHA-256 räsifunktsiooni 256*256 = 65536 bitti (8 kilobaiti) ja avaliku võtme suuruseks 256 * 2 * 256 = 131072 bitti (16 kilobaiti). Privaatvõtme pikkust saab vähendada genereerides kõik privaatvõtmed kasutades pseudojuharvude generaatorit.

2.5.2 Winternitzi signatuuriskeem

Winternitzi signatuuriskeemis (W-OTS) [42], [43] on võrreldes Lamporti skeemiga väiksemad võtmed ja allkirjad. Selleks, et Winternitzi signatuuriskeemis tuletada juhuslikult genereeritud privaatvõtmest s avalik võti p, tuleb räsifunktsiooni H sisendile s rakendada w korda ehk $p = H^w(s)$. Parameetrist w oleneb mitme bitist sõnumit võtmega x allkirjastada saab. Näiteks tuleks 4-bitise sõnumi allkirjastamiseks w väärtuseks valida $w = 2^4 = 16$. Allkirjastamisel tuleb privaatvõtmele x rakendada räsifunktsiooni $w - n$ korda, kus n on sõnumi kui w-bitise arvu väärtus. Näiteks allkirjastades neljabitist sõnumit 1010 (kümnendsüsteemis 10), tuleb privaatvõtmele x rakendada räsifunktsiooni 10 korda. Allkirja kontrollimisel tuleb allkirjale rakendada räsifunktsiooni n korda ning kontrollida, et tulemus oleks võrdne avaliku võtmega p. Iga allkiri peab sisaldama ka kontrollkoodi, et poleks võimalik tuletada allkirjast $H^m(s)$ allkirja $H^{m'}(s)$, kus $m \leq m' < w$. Kontrollkoodi arvutamist on detailselt kirjeldatud Buchmann, Dahmen, Ereth, Hülsing ja Rückert [44]. W-OTS ning selle edasiarendus W-OTS$^+$ [45] on laialdaselt kasutusel mitmetes teistes signatuuriskeemides. Sarnaselt Lamporti signatuuriskeemile saab ka Winternitzi signatuuriskeemis igat võtit kasutada allkirjastamiseks üks kord.

2.5.3 Merkle'i signatuuriskeem

Kuna Lamporti ja Winternitzi signatuuriskeemis saab igat privaatvõtit allkirjastamiseks kasutada ainult üks kord, siis mitme sõnumi allkirjastamiseks läheb iga kord vaja uut avalikku ja privaatset võtit, mille haldamine on ebaefektiivne. Ralph Merkle poolt tutvustatud signatuuriskeem [27] on edasiarendus Lamporti allkirjaskeemist, mis kasutab avaliku võtme mahu vähendamiseks räsipuud, kus lehtedeks on Lamporti allkirjaskeemi avalikud võtmed. Merkle'i signatuuriskeemi avalikuks võtmeks on räsipuu tipu räsi.

Kui Merkle'i allkirjaskeemis koosneb räsipuu m lehest, siis saab avalikku võtit kasutada allkirjastamiseks m korda (igat Lamporti allkirjaskeemi võtit üks kord). Merkle'i signatuuriskeemi allkiri sisaldab lisaks Lamporti allkirjale veel autentimisteed kasutatud Lamporti avalikust võtmest räsipuu juurtipuni. Seega on Merkle'i

signatuuriskeemis allkirja suuruseks $n^2+n\log_2 m$ bitti, kus m on Lamporti võtmete arv räsipuus ning n Lamporti signatuuriskeemi räsiväärtuse suurus.

2.5.4 XMSS signatuuriskeem

XMSS (*eXtended Merkle Signature Scheme*) [7] on Merkle'i allkirjaskeemi edasiarendus, kus enne puu sisendtipu räsi arvutamist rakendatakse mõlemale alamtipu räsile maskimist. Lisaks viitab iga XMSS puu leht L-puu juurräsile. L-puu on puu, kus lehtedeks on W-OTS$^+$ avalikud võtmed ning iga puu sisendtipu arvutamisel rakendatakse samasugust maskimist nagu XMSS puus. Võrreldes Merkle'i allkirjastamisskeemiga on XMSS'i allkirjad tunduvalt väiksemad (n=256 korral umbes 1,8 kilobaiti). XMSS allkirjasüsteemis tuleb hoida olekut kasutatud võtmetest, mis tähendab, et neid on raske kasutada olukordades, kus ühte ja sama võtit on vaja jagada.

Lisaks XMSS signatuuriskeemile on olemas ka XMSS^{MT} (ingl *XMSS Multi-Tree*) signatuuriskeem [46] , mis võimaldab teoreetiliselt allkirjastata lõpmata arv sõnumeid. Saranaselt XMSS signatuuriskeemile peab ka XMSS^{MT} signatuuriskeemis hoidma olekut kasutatud võtmetest.

2.5.5 Goldreich'i signatuuriskeem

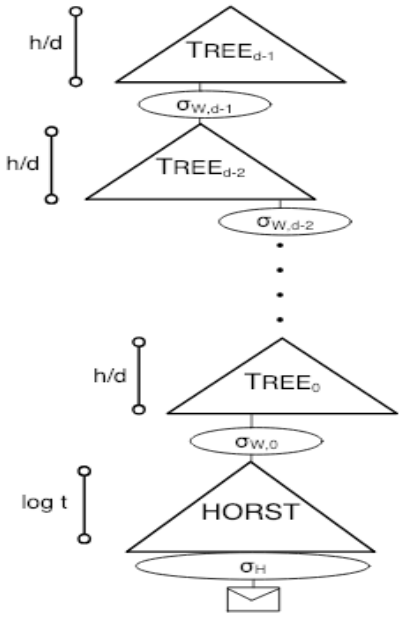
Goldreich'i signatuuriskeem [47] koosneb kahendpuust, mille kõrgus on võrdne sõnumi räsimiseks kasutatava räsifunktsiooni väljundi pikkusega (nt SHA-256 räsifunktsiooni korral on puu kõrgus 256). Goldreich'i signatuuriskeemi puud ei arvutata kunagi välja. Allkirjastamise hetkel kasutatakse allkirja jaoks vajalike puu tippude genereerimiseks deterministliku funktsiooni, mis genereerib iga allkirja koostamiseks vajaliku puu tipu jaoks ühekordse räsipõhise signatuuriskeemi võtmepaari. Antud võtmepaari kasutatakse tipu järglaste avalike võtmete allkirjastamiseks.

Goldreich'i allkiri koosneb ühekordsete räsipõhiste signatuuriskeemide allkirjadest, kus allkirjastatava sõnumi räsi kasutatakse allkirjastamisvõtme (kahendpuu lehe) valimiseks. Goldreich'i allkiri sisaldab sõnumi allkirja, mis on loodud kahendpuu lehes oleva ühekordse räsipõhise privaatvõtmega, ja autentimisteed koos iga tipu avaliku võtme ja allkirjaga puu lehest kahendpuu juurtippu. Kahendpuu juurtipp on Goldreich'i signatuuriskeemi avalik võti.

Erinevalt Merkle'i sigantuuriskeemist ei ole vaja Goldreich'i signatuuriskeemis hoida olekut kasutatud võtmetest ning allkirjade arvutamine ei nõua palju arvutusvõimust. Kuna aga Goldreich'i allkirja suurus võib küündida mitme megabaidini, siis kirjeldatud signatuuriskeemi on praktikas keeruline kasutada.

2.5.6 SPHINCS signatuuriskeem

2015. aastal tutvustatud SPHINCS [8] signatuuriskeemis ei ole vaja hoida olekut kasutatud võtmetest. SPHINCS signatuuriskeem põhineb Goldreich'i ja HORST [8] signatuuriskeemidel. HORST on signatuuriskeem, mille privaatvõtit saab allkirjastamiseks kasutada mõned korrad. SPHINCS allkirjaskeem koosneb „hüperpuust" (puu, mis koosneb mitmest teisest puust) kõrgusega h , kus h on d kordne. Hüperpuu ise koosneb d kihist, kus igas kihis on puu kõrguseks h/d . Iga hüperpuus oleva puu lehtedeks on W-OTS$^+$ võtmepaar, mida kasutatakse järgmises kihis oleva puu juurräsi allkirjastamiseks. Viimases kihis on WOTS$^+$ võtmetega allkirjastatud HORST avalik võti. Joonisel 2 on kujutatud SPHINCS'i signatuuriskeemi virtuaalset struktuuri.



Joonis 2. SPHINCS allkirja struktuur [8].

Kogu SPHINCS'i hüperpuud ei arvutata kunagi lõplikult välja, vaid allkirja jaoks vajalikud osad koostatakse allkirjastamise hetkel. Allkirjastatavatest andmetest sõltub,

millised puud allkirjastamise hetkel välja arvutatakse. SPHINCS signatuuriskeemist on edasi arendatud ka optimeeritud variant nimega SPHINCS$^+$ [48].

SPHINCS signatuuriskeem on umbes 30 korda aeglasem, kui XMSS. SPHINCS-256 signatuuriskeemis on avaliku ja privaatse võtme suuruseks 1KB ning allkirja suuruseks 41KB. SPHINCS$^+$ signatuuriskeemis on avaliku võtme suuruseks 48 baiti, privaatse võtme suuruseks 96 baiti ning allkirja suuruseks 30KB.

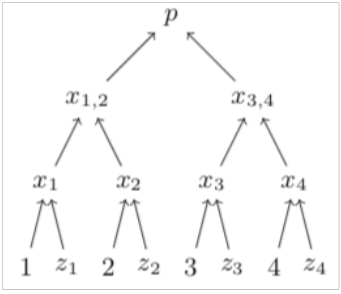
2.5.7 BLT-TB-signatuuriskeem

Buldas, Laanoja ja Truu tutvustasid 2017. aastal BLT nimelist räsipõhist signatuuriskeemi [10], kus privaatvõti koosneb ühest räsist ning allkirjade efektiivsemaks andmiseks ja allkirja suuruse vähendamiseks kasutatakse serveri abi. Käesolevas töös tähistatakse sellist allkirjaskeemi edaspidi nimega BLT-TB. Antud signatuuriskeemis on allkirjastajal loend järjestatud allkirjastamisvõtmetest, kus igat võtit saab allkirjastamiseks kasutada ainult kindlal ajahetkel. Peale sõnumi allkirjastamist ja ajahetke möödumist saab allkirjastamisvõtit kasutada sõnumi allkirja kontrollimiseks.

BLT-TB võtmete genereerimine ajahetkede $1,\dots,N$ jaoks toimub järgmiselt:

- Genereeritakse N k-bitist juhuslikku allkirjastamisvõtit: $(z_1,\dots,z_n)$
- Iga võti seotakse mingi kindla ajahetkega, arvutades $x_t = h(t, z_t)$, iga $t \in \{1,\dots,N\}$ jaoks
- Koostatakse Merkle'i puu, kus lehtedeks on eelmises punktis arvutatud räsid. Merkle'i puu tipp p on BLT-TB signatuuriskeemi avalik võti.

Joonisel 3 on näidatud BLT-TB avaliku võtme arvutamine N=4 korral.



Joonis 3. BLT-TB avaliku võtme arvutamine [10].

BLT-TB-signatuuriskeemis toimub sõnumi m allkirjastamine ajahetkel t järgmiselt:

- Arvutatakse sõnumi räsi: $x=h(m)$
- Moodustatakse allkirjastamispäring $y=h(x,z_t)$, mis saadetakse ajatempliteenust pakkuvale serverile.
- Server arvutab ja tagastab ajatempli ts_t.
- Kui ajatempli ts_t aeg on võrdne võtme z_t ajaga, siis allkirjastaja väljastab sõnumi allkirja $\langle t, z_t, ts_t, c_t \rangle$, kus c_t on autentimistee ajahetke t ja võtme z_t lehest avaliku võtmeni p.

Kuigi BLT-TB allkirjaskeem ei hoia olekut kasutatud võtmetest eeldatakse, et signeerija ja ajatempliteenust pakkuva serveri ajad on sünkroniseeritud.

BLT-TB-allkirja verifitseerimisel tuleb esimesena kontrollida, et allkirjastamiseks kasutatud võti z_t on osa avalikust võtmest p. Selleks arvutatakse $x_t=h(t,z_t)$ ja kontrollitakse autentimisteed lehest x_t avaliku võtmeni p. Järgmisena tuleb verifitseerida, et sõnumi m ja võtme z_t räsi on allkirjastatud ajatempliteenuse-pakkuja poolt ajahetkel t.

2.5.8 BLT-OT- ja BLT-OT-N-signatuuriskeem

BLT-TB-signatuuriskeem töötab hästi seadmetel, millel on usaldusväärne kell ning mis töötab pidevalt. BLT-TB signatuuriskeemi on keeruline kasutada kiipkaartidel, sest kiipkaardil pole ligipääsu kellaajale ning kiipkaardil ei ole enamus ajast ühendatud

arvutiga. Et kirjeldatud probleemi lahendada tutvustasid Ahto Buldas, Denis Firsov, Risto Laanoja, Ahto Truu ning Henri Lakk 2019. aastal BLT-OT nimelist signatuuriskeemi, mis on spetsiaalselt mõeldud kiipkaardi laadsetele seadmetele [9].

BLT-OT kasutab allkirjastamiseks ühekordseid võtmeid ja sarnaselt BLT-TB skeemile kasutatakse BLT-OT-signatuuriskeemis allkirja andmiseks serveri abi. Erinevalt BLT-TB-signatuuriskeemist allkirjastatakse BLT-OT-signatuuriskeemis kellaaega, mis on saadud serverilt. Kui allkirjastada l -bitist kellaaega (näiteks 32-bitist POSIX kellaaega), siis BLT-OT privaatvõti koosneb l juhuslikult genereeritud arvust: $sk=(z_{0,}\ldots,z_{l-1})$. Avalikuks võtmeks on $pk=H(X)$, kus $X=(x_{0,}\ldots,x_{l-1})$, $x_i=H(z_i)$ iga $(0\le i<l)$ korral ning H on räsifunktsioon.

Sõnumi m allkirjastamiseks BLT-OT-signatuuriskeemis saadetakse andmed $(H(m,X),pk)$ ajatempliteenust pakkuvale serverile, mille peale server tagastab ajatempli S_t . Klient võtab ajatemplist t l-bitise aja ning koostab järjendi $W=(w_o,w_{1,}\ldots,w_{l-1})$, kus w_i (iga $0\le i<l$) on arvutatud valemi 1 järgi. Allkirjaks on paar (W,S_t) .

$$w_i=\begin{cases} z_i & \text{,kui ajas } t \text{ on biti } i \text{ väärtus 1} \\ x_i=H(z_i) & \text{,vastasel korral} \end{cases} \quad (1)$$

Vältimaks privaatvõtme sk taaskasutamist, peab allkirjastaja peale allkirjastamist võtme hävitama.

Sõnumi m ja allkirja (W,S_t) kontrollimiseks peab verifitseerija ajatemplist S_t välja võtma aja t ning arvutama $(x_0,x_1,\ldots,x_{l-1})$ kasutades valemit 2.

$$x_i=\begin{cases} H(w_i) & \text{, kui ajas } t \text{ on biti } i \text{ väärtus 1} \\ w_i=H(z_i) & \text{, vastasel korral} \end{cases} \quad (2)$$

Allkiri on õige, kui $H(x_0,x_1,\ldots,x_{l-1})=pk$ ja S_t on valiidne ajatempel, mis on väljastatud ajal t allkirjastamisel kasutatud ajatempliteenusepakkuja poolt ning ajatempel on võetud andmetele (m,X,pk) .

Koostades N erinevast BLT-OT võtmetest Merkle'i puu, kus lehtedeks on BLT-OT-võtmed, saab sellist skeemi kasutada allkirjastamiseks N korda. Käesolevas töös nimetatakse sellist allkirjaskeemi edaspidi tähisega BLT-OT-N. Sarnaselt Merkle'i allkirjaskeemile on BLT-OT-N avalikuks võtmeks räsipuu tipuräsi.

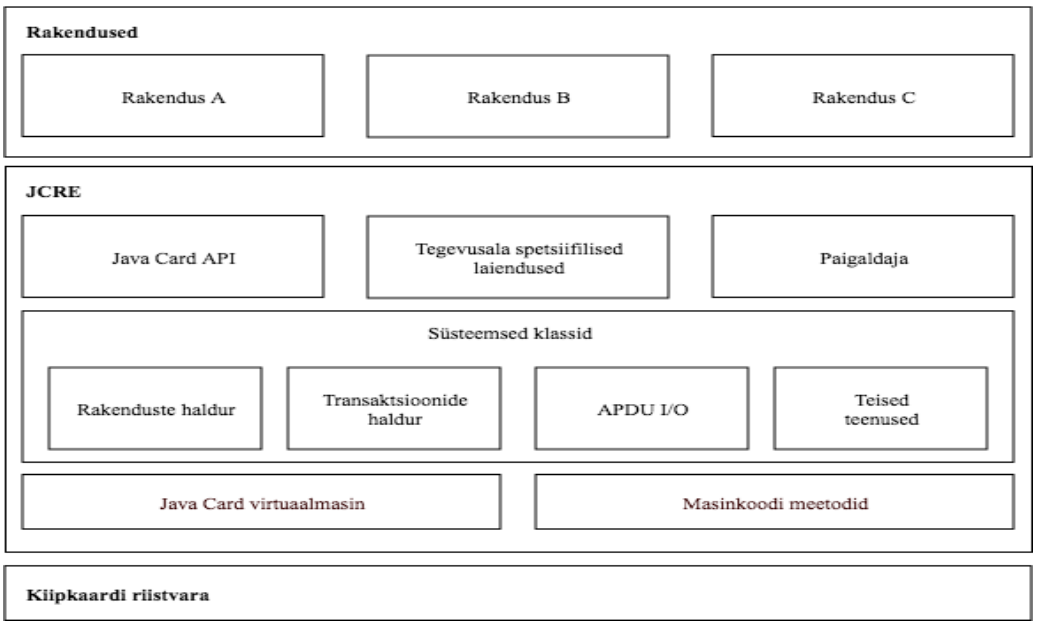
2.6 Kiipkaart

Kiipkaart on mikrokiipi sisaldav plastikaart, mis suudab turvaliselt töödelda ja talletada erinevat informatsiooni. Sakslased Jürgen Dethloff ja Helmut Gröttrup olid esimesed, kes 1968. aastal käisid välja idee kiipkaardist. Alates 1990-ndatest aastatest on kiipkaart tänu *GSM* (*Global System for Mobile communications*) ja pangakaartidele laialdaselt levinud väga erinevates valdkondades. 1996. aastal tutvustati *Java Card* platvormi, mis võimaldab luua rakendusi kiipkaartidele, mis võivad töötada erinevate tootjate riistvaral. Antud platvorm põhineb populaarsel programmeerimiskeelel Java ning on laialdaselt levinud [16].

Käesolevas jaotises antakse ülevaade *Java Card* platvormi arhitektuurist, tutvustatakse sõnumivahetusprotokolli kiipkaardiga suhtlemiseks ning kirjeldatakse *Java Card* tehnoloogia eeliseid ning puudusi.

2.6.1 Java kiipkaardi arhitektuur

Joonisel 4 on välja toodud Java kiipkaardi arhitektuuri, mis koosneb rakenduste, *JCRE* (ingl *Java Card Runtime Environment*) ja riistvara kihist. *JCRE* eraldab erinevate tootjate riistvara kaardi rakenduste kihist, võimaldades tarkvaraarendajatel kirjutada rakendusi, mis ühilduvad erinevate tootjate riistvaraga. *JCRE* vastutab kaardi ressursside majandamise, võrgusuhtluse, rakenduste käivitamise ja rakenduste turvalisuse eest [16], [49].



Joonis 4. Java kiipkaardi arhitektuur.

JCRE koosneb *Java Card* virtuaalmasinast (*JCVM*), masinkoodi meetoditest ning süsteemsetest, Java Card raamistiku ja tootja poolt eelpaigaldatud laiendustest. *JCVM* käivitab baitkoodi, vastutab mälu ning mälus olevate objektide halduse ja turvalisuse eest. Kuna *JCVM* ise on suhteliselt aeglane, siis on võimalik teatud juhtudel, näiteks krüptograafiliste funktsioonide puhul, käivitada otse masinkoodi. Süsteemsed klassid vastutavad rakenduste paigaldamise, aktiveerimise ja deaktiveerimise, transaktsioonide halduse ning kaardilugeja ja rakenduse vahelise suhtluse eest. *Java Card API* sisaldab klasse ja liideseid rakenduste loomiseks. Lisaks võib kaardil olla eelpaigaldatud mingile tegevusalale spetsiifilisi laiendusi ja rakendusi. Paigaldaja vastutab rakenduste paigaldamise eest kaardile pärast kaardi tootmist. Java kiipkaardile võib paigaldada mitmeid rakendusi ning need on üksteisest ning süsteemsetest klassidest eraldatud tulemüüriga, mis tähendab seda, rakendused ei pääse ligi teiste rakenduste mälule [16], [49] .

Java kiipkaart sisaldab kolme erinevat tüüpi mälu: püsimälu ehk ROM (ingl *Read-Only Memory*), muutmälu ehk RAM (ingl *Random-Access Memory*) ja elektriliselt kustutatav programmeeritav püsimälu ehk EEPROM (ingl *Electrically Erasable Programmable Read-Only Memory*). ROM mällu on kaardi tootmise ajal paigaldatud *JCVM* ja kaardi operatsioonisüsteem ning tavakasutaja ROM mällu andmeid salvestada ei saa. RAM ja

EEPROM mällu on võimalik rakendustel andmeid kirjutada ja lugeda. RAM mällu kirjutamine on umbes 1000 korda kiirem, kuid erinevalt EEPROM mälust ei säili andmed RAM mälus peale kaardi eemaldamist kaardilugejast. Java kiipkaardi rakendusi hoitakse EEPROM mälus.

2.6.2 Java kiipkaardi sõnumivahetusprotokoll

Alates Java kiipkaardi 3.0 spetsifikatsioonist [49] on võimalik kiipkaardiga suhelda kahel erineval viisil. Nendest esimene on tagasiühilduv eelmiste versioonidega ning kasutab suhtluseks ISO/IEC 7816 standardit [50], millele vastavalt käib suhtlus kiipkaardi ja kaardilugeja vahel kasutades APDU (ingl *Application Protocol Data Unit*) nimelisi andmeüksusi. Java kiipkaardi puhul tehakse vahet sisend ja väljund APDU-l. Tabelis 2 on välja toodud sisend APDU vorming.

Tabel 2. Sisend APDU vorming.

Välja nimi	**Välja pikkus baitides**	**Kirjeldus**
CLA	1	Instruktsioon klass. Tehakse vahet kahel klassi tüübil: • üldised klassitüübid, mis on defineeritud ISO/IEC 7816 standardi poolt • rakenduse spetsiifilised klassid
INS	1	Viitab reaalsele tegevusele, mida tehakse. Näiteks „vali fail”.
P1, P2	2	Lisaparameetrid reaalsele tegevusele. Näiteks saab failist lugemise puhul antud väljadel määrata, mis tüüpi faili loetakse (nt juurkataloog, kataloog, fail).
L_c	0, 1 või 3	Väljal CD olevate andmete pikkus baitides. Väärtus vahemikust 0 - 65535.
CD	c	Käsu andmed
L_e	0, 1, 2 või 3	Viitab mitu baiti andmeid oodatakse vastuseks.

Sisend APDU-s on väljad CLA, INS ja P1, P2 kohustuslikud. Tabel 3 kirjeldab väljund APDU vormingut, kus on kohustuslikud väljad SW1,SW2.

Tabel 3. Väljund APDU vorming.

Välja nimi	**Välja pikkus baitides**	**Kirjeldus**
RD	Maksimaalselt L_e baiti	Vastuse andmed
SW1-SW2	2	Käsu töötlemise staatus. Näiteks väärtus 90 00 (kuueteiskümnendsüsteemis) tähendab, et päring õnnestus.

Uuem suhtlusprotokoll võimaldab kasutada Java Servlet liideseid [51], kus kiipkaart käitub veebirakendusena. Antud suhtlusprotokoll pole levinud ning kaarditootjat, kes toetaks antud suhtlusprotokolli, on väga raske leida.

2.6.3 Java kiipkaardi piirangud ja eelised

Tänapäeval on Java kiipkaardil keskmiselt 80-144 kilobaiti EEPROM ja paar kilobaiti RAM mälu, mis seab rakendustele märkimisväärsed piirangud. Näiteks Merkle'i puu salvestamiseks, millel on 3560 256-bitist lehte, kuluks ainuüksi räsipuu tippude salvestamiseks 234 kilobaiti mälu. Ressursside puuduse tõttu on *Java Card* platvormil kasutusel minimaalne alamosa Java programmeerimiskeelest. Näiteks on toetatud ainult 8-bitine „*byte"* ja 16-bitine „*short"* primitiivne andmetüüp. Samas on toetatud Javast tuntud klassid, kuid klasside initsialiseerimine on ressursimahukas ning seda tuleks vältida. See tähendab ka seda, et enamasti ei saa kasutada enamlevinuid disainimustreid ning *Java Card* platvormil põhinev kood võib olla raskesti muudetav ning loetav. Samuti tuleb arvestada, et *Java Card* platvormil puudub automaatne mälukoristus ning rakenduse jaoks tuleks EEPROM mälu reserveerida rakenduse initsialiseerimise käigus.

Java Card platvorm ühildub erinevate standarditega ning võimaldab arendajatel toota rakendusi erinevate tootjate poolt toodetud kaartidele. *Java Card* pakub häid liideseid krüptograafiliste funktsioonide jaoks, mis teeb sellest ideaalse platvormi erinevate krüptograafiaga seotud rakenduste loomiseks. Samuti võimaldab *Java Card* platvorm paigaldada ühele kaardile mitu rakendust, mis ei pääse ligi üksteise andmetele.

3 Funktsionaalsed ja mittefunktsionaalsed nõuded

Käesolev peatükk keskendub BLT-signatuuriskeemi prototüübi funktsionaalsetele ja mittefunktsionaalsetele nõuetele. Selleks koostatakse ja kirjeldatakse agentorienteeritud modelleerimisest (AOM) tuttavad rolli- ja eemärgimudelid [14], mille põhjal saab luua süsteemi põhjalikuma disaini.

3.1 Sissejuhatus

Antud peatükk vastab jaotises 1.2 välja toodud uurimisküsimusele **UK-1: Millised on funktsionaalsed ja mittefunktsionaalsed nõuded BLT-kliendipoolse rakenduse realiseerimiseks?** Selleks, et vastata uurimisküsimusele struktureeritult, on see jagatud järgmisteks alamküsimusteks:

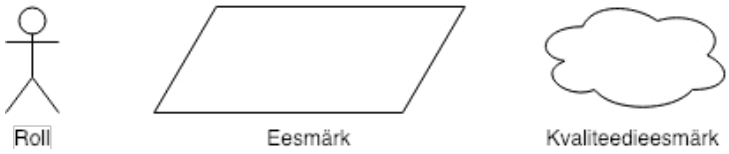
- **UK-1.1: Millised on loodavas süsteemis erinevate osapoolte rollid?**
- **UK-1.2: Millised on loodava süsteemis funktsionaalsed nõuded?**
- **UK-1.3: Millised on loodava süsteemi mittefunktsionaalsed nõuded?**

Jaotis 3.2 annab ülevaate analüüsiks kasutatavatest AOM rolli- ja eesmärgimudelitest. Jaotis 3.3 vastab alamküsimusele UK-1.1 ning keskendub AOM rollimudelitele. Jaotises 3.4 kirjeldatakse funktsionaalseid nõudeid AOM eesmärgimudelite abil ning jaotis 3.5 keskendub mittefunktsionaalsetele nõuetele kirjeldades täpsemalt eesmärgimudeli kvaliteedieesmärke. Jaotis 3.6 keskendub arutelule ning jaotis 3.7 sisaldab peatükki kokkuvõtet.

3.2 Kasutatavad agentorienteeritud modelleerimise elemendid

BLT prototüübi nõuete analüüsiks kasutatakse agentorienteeritud modelleerimisest tuntud rolli- ja eesmärgimudeleid [14]. Rollimudelit kasutatakse agendi, mis võib olla

inimene või seade, kohustuste kirjeldamiseks ning see koosneb järgmistest elementidest: rolli nimi, kirjeldus, kohustused ja piirangud. AOM eesmärgimudelit kasutatakse funktsionaalsete ja mittefunktsionaalsete nõuete ning nendega seotud rollide kirjeldamiseks. Eesmärgimudel koosneb eemärkidest (ingl *goal*), mis kirjeldavad funktsionaalseid nõudeid, kvaliteedieesmärkidest (ingl *quality goal*), mis kirjeldavad mittefunktsionaalseid nõudeid, ja eesmärkide saavutamiseks vajalikest rollidest. Eesmärkide kirjeldamiseks kasutatakse puulaadset struktuuri, kus iga eesmärk võib jaotuda alameesmärkideks. Kvaliteedieesmärgid ja rollid on omakorda ühendatud eesmärkidega. Joonisel 5 on välja toodud eesmärgimudeli elemendid, mida käesolevas töös kasutatakse.



Joonis 5. AOM eesmärgimudeli elemendid.

Joonisel vastab rollile kriipsujuku, eesmärgile rööpkülik ja kvaliteedieesmärgile pilve kujutis. Eesmärk ja alameesmärk on omavahel seotud pideva joonega ning kvaliteedieesmärk ja roll on seotud eesmärgiga kasutades punktiirjoont.

3.3 Rollimudelid

Tabelis 4 on välja toodud kiipkaardi kasutaja rollimudel oma vastutuste ja piirangutega. Olenevalt olukorrast ja reaalsematest turvanõuetest võib selle rolli omakorda jagada kaheks: BLT-rakenduse paigaldaja ja sõnumi allkirjastaja. Antud magistritöös on lihtsuse mõttes eeldatud, et BLT-rakenduse paigaldaja ja selle tulevane kasutaja on üks ja sama isik.

Tabel 4. Kiipkaardi kasutaja rollimudel.

Rolli nimi	Kiipkaardi kasutaja
Kirjeldus	Reaalne isik, kes omab kiipkaarti, mille peale on paigaldatud või tahetakse paigaldada BLT-kiipkaardirakendus.
Vastutused	BLT-rakenduse paigaldamise algatamine. Sõnumi allkirjastamise algatamine.

Piirangud	Kaarti saab algväärtustada ainult siis, kui see on algväärtustamata. Sõnumit saab allkirjastada, kui kaart on algväärtustatud ning kasutaja teab õiget PIN-koodi. Ühte BLT-privaatvõtit saab allkirjastamiseks kasutada ainult üks kord. Maksimaalne antavate allkirjade arv on võrdne BLT-privaatvõtmete arvuga.

Tabelis 5 on välja toodud BLT-kliendirakenduse rollimudel. Sellelt mudelilt on näha, et BLT-kliendirakendus suhtleb BLT-serveri ja BLT-kiipkaardirakendusega.

Tabel 5. BLT-kliendirakenduse rollimudel.

Rolli nimi	BLT-kliendirakendus
Kirjeldus	BLT-kiipkaardi kasutaja arvutisse paigaldatud programm, mis oskab suhelda BLT-serveri ja BLT-kiipkaardiga.
Vastutused	Suhtlus BLT- kiipkaardirakendusega. Suhtlus BLT-serveriga.
Piirangud	Suhtlus BLT-kiipkaardiga saab toimuda, kui kiipkaart on kiipkaardilugejaga arvutiga ühendatud. Suhtlus BLT-serveriga saab toimuda, kui olemas on internetiühendus.

Tabelis 6 on välja toodud BLT-kiipkaardirakenduse rollimudel, mille vastutusalasse kuulub privaatvõtmete ning muu allkirjastamiseks vajaliku info hoidmine.

Tabel 6. BLT-kiipkaardirakenduse rollimudel.

Rolli nimi	BLT-kiipkaardirakendus
Kirjeldus	Kiipkaardile paigaldatud Java Card platvormil töötav rakendus.
Vastutused	BLT-võtmete genereerimine ja turvaline hoidmine. BLT-serveri avaliku võtme hoidmine. Sõnumi räsi allkirjastamine.
Piirangud	BLT-võtmeid saab genereerida ainult siis, kui kaardile ei eksisteeri võtmeid. Sõnumit saab allkirjastada juhul, kui kaart on algväärtustatud ja kaardil eksisteerib vähemalt üks kasutamata BLT-privaatvõti. Sõnumit saab allkirjastada juhul, kui kasutaja on sisestanud õige PIN-koodi.

Tabelis 7 on välja toodud verifitseerija rollimudel, mis tegeleb BLT-allkirjade verifitseerimisega.

Tabel 7. Verifitseerija rollimudel.

Rolli nimi	Verifitseerija
Kirjeldus	Isik või rakendus, kes verifitseerib BLT-allkirja.
Vastutused	Vastutab BLT-allkirja verifitseerimise eest.
Piirangud	Puuduvad

Tabelis 8 on välja toodud BLT-serveri rollimudel, mis tegeleb BLT-kiipkaardirakenduse avaliku võtme registreerimisega ja BLT-TB-allkirja koostamisega.

Tabel 8. BLT-serveri rollimudel.

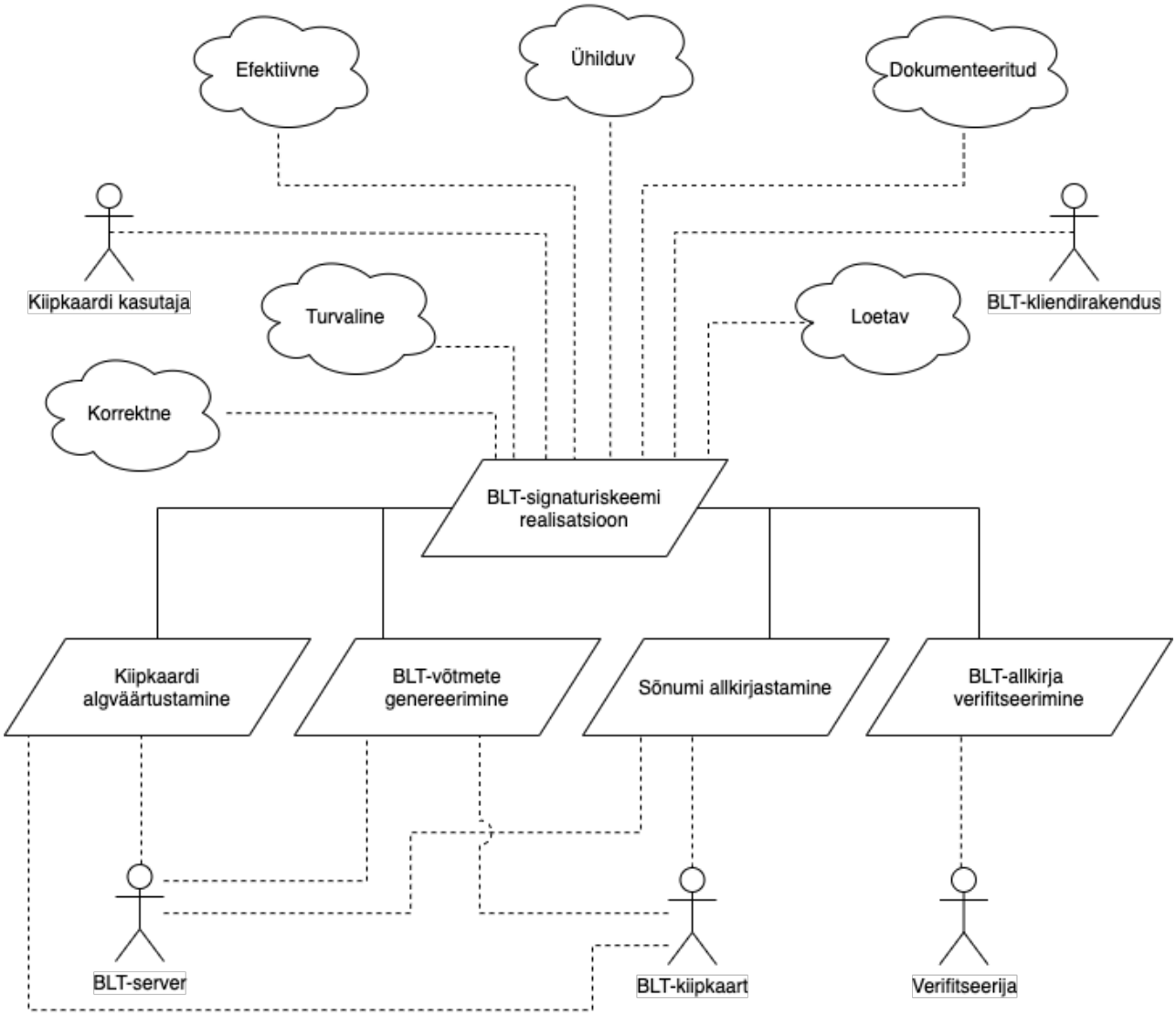
Rolli nimi	BLT-server
Kirjeldus	BLT-server on usaldusväärne osapool.
Vastutused	BLT-kiipkaardi avaliku võtme registreerimine. BLT-allkirjade koostamine.
Piirangud	Puuduvad

BLT-serveri rolli puhul tuleb arvestada, et BLT-TB-allkirja koostamiseks on vaja ka ajatempliteenuse pakkuja rolli. Kuna käesolev magistritöö keskendub BLT-signatuuriskeemi kliendipoolsele osale, siis lihtsuse mõttes on ajatempliteenusepakkuja roll jäetud defineerimata. Eeldatud on, et BLT-kiipkaardirakenduse poolt allkirjastatav kellaaeg võetakse BLT-TB-allkirjast.

3.4 Funktsionaalsed nõuded

Joonisel 6 on välja toodud loodava süsteemi peamine eesmärgimudel. Eesmärgimudelis on peaeesmärk realiseerida BLT-signatuuriskeem, mis omakorda jaguneb neljaks

alameesmärgiks: kiipkaardi algväärtustamine, BLT-võtmete genereerimine, sõnumi allkirjastamine ja BLT-allkirja verifitseerimine.



Joonis 6. BLT-signatuuriskeemi eesmärgimudel.

Kiipkaardi algväärtustamise käigus paigaldatakse kiipkaardile BLT-kiipkaardirakendus, reserveeritakse kaardi tööks vajalik mälu ja salvestatakse kaardile BLT-serveri avalik võti. BLT-võtmete genereerimise käigus genereeritakse allkirjastamiseks vajalikud privaatvõtmed ja avalik võti ning registreeritakse kiipkaardil genereeritud BLT-võti BLT-serveris. Jooniselt on näha, et kiipkaardi algväärtustamisega on seotud *kiipkaardi kasutaja*, *BLT-kliendirakendus*, *BLT-server* ja *BLT-kiipkaart* rollid. Sõnumi allkirjastamisel kõigepealt sõnum räsitakse ning sõnumi räsi allkirjastatakse kasutades BLT-OT-N-signatuuriskeemi. Sõnumi allkirjastamisega on seotud samad rollid, mis

kiipkaardi algväärtustamise eesmärgiga. BLT-allkirja verifitseerimise eesmärk võimaldab kolmandatel osapooltel ning samuti kiipkaardi kasutajal kontrollida BLT-allkirja õigsust kasutades BLT-kliendirakendust. Joonisel 6 välja toodud kvaliteedieesmärgid on täpsemalt kirjeldatud järgmises jaotises.

3.5 Mittefunktsionaalsed nõuded

Tabel 9 sisaldab joonisel 6 välja toodud kvaliteedieesmärke, mis kirjeldavad loodava süsteemid mittefunktsionaalseid nõudeid. Tabeli esimeses veerus on välja toodud kvaliteedieesmärk/mittefunktsionaalne nõue ning teine veerg sisaldab nõude kirjeldust.

Tabel 9. Mittefunktsionaalsete nõuete kirjeldused.

Mittefunktsionaalne nõue/ kvaliteedieesmärk	**Kirjeldus**
Efektiivne	Efektiivne tähendab loodava rakenduse juures järgmist: **1.** kiipkaardi võtmete genereerimine ei tohiks 3650 (10 võtit iga päeva kohta aastas) privaatvõtme korral võtta rohkem aega rohkem kui 30 minutit, **2.** allkirjastamine ei tohiks keskmiselt aega võtta rohkem kui 10 sekundit ja **3.** BLT-allkirja verifitseerimise aeg peab jääma alla 0.5 millisekundi .
Turvaline	Turvaline tähendab antud rakenduse puhul järgmist: **1.** BLT-OT-privaatvõtmeid tuleb hoida turvaliselt kiipkaardil, **2.** BLT-OT-privaatvõtmed peab genereerima kiipkaart, **3.** ei tohi olla võimalust kiipkaardilt kätte saada kasutamata BLT-OT privaatvõtmeid ja **4.** allkirjastamiseks kasutatav PIN-kood tuleb hoida turvaliselt kiipkaardil.
Ühilduv	Loodav BLT-kliendirakendus peab töötama Java 1.8 või uuema Java virtuaalmasinaga. Loodav BLT-kiipkaardirakendus peab töötama Java Card 2.2.2 või uuema Java Card platvormiga.
Korrektne	Võtmete genereerimine, allkirjastamine ning allkirja verifitseerimine peab vastama jaotises 2.5.8 välja toodud nõuetele.

Mittefunktsionaalne nõue/ kvaliteedieesmärk	**Kirjeldus**
Dokumenteeritud	BLT-kiipkaardi ja BLT-kliendirakenduse liidesed ning kood peab olema dokumenteeritud nii, et seda oleks võimalikult lihtne edasi arendada teistel osapooltel.
Loetav	BLT-kiipkaardi ja BLT-kliendirakenduse kood peab olema arusaadav ning loetav inimestele, kes on tuttavad Java ja Java Card platvormidega.

3.6 Lihtsustused

Kuna käesoleva töö peamiseks eesmärgiks on demonstreerida BLT-signatuuriskeemi efektiivset kasutamist kiipkaardil, siis on hoitud rollide hulk minimaalsena. Teatud olukordades on reaalne, et *kiipkaardi kasutaja* roll tuleb jagada kaheks erinevaks rolliks, millest esimene tegeleb BLT-kiipkaardirakenduse paigaldamise ja algväärtustamisega ning teine allkirjastamisega. Samal põhjusel pole funktsionaalsete nõuete juures kirjeldatud nõudeid, mis tegelevad PIN-koodi vahetamise ja privaatvõtmete uuesti genereerimisega. Eeldatud on, et kiipkaardi kasutaja saab alati antud olukordades BLT-rakenduse kiipkaardilt eemaldada ning uuesti paigaldada. Reaalses maailmas võib rakenduse uuesti paigaldamine olla keerukas või võimatu ning selliste probleemide tekkimisel tuleks loodud eesmärgimudelit täiendada uute funktsionaalsete ja mittefunktsionaalsete nõuetega.

3.7 Kokkuvõte

Käesolevas peatükis kasutati agentorienteeritud modelleerimisest tuttavaid rolli- ja eesmärgimudeleid, et kirjeldada loodava BLT prototüübi funktsionaalseid ja mittefunktsionaalseid nõudeid. Koostatud eesmärgimudeli põhjal koosneb prototüüp neljast peamisest eesmärgist: kiipkaardi algväärtustamine, BLT-võtmete genereerimine, sõnumi allkirjastamine ja BLT-allkirja verifitseerimine. Välja toodud eesmärkidega on seotud viis rolli: kiipkaardi kasutaja (vastutab kiipkaardi algväärtustamise ja allkirja koostamise eest), BLT-allkirja verifitseerija, BLT-kiipkaardirakendus, BLT-server ja BLT-kliendirakendus. Eesmärgimudelil toodi välja kuus mittefunktsionaalset nõuet:

efektiivne, ühilduv, dokumenteeritud, loetav, turvaline ja korrektne. Kirjeldatud nõuded saab võtta aluseks BLT-signatuuriskeemi täpsemaks projekteerimiseks.

4 Sõnumivahetusprotokoll ja andmestruktuurid

Käesolev peatükk keskendub BLT-signatuuriskeemi prototüübi disainile. Antud peatükis kasutatakse UML-ist tuntud komponendiskeemi (ingl *component diagram*), et anda ülevaade loodava süsteemi komponentidest. UML-järgnevusskeeme (ingl *sequence diagram*) kasutatakse süsteemi sõnumivahetusprotokolli kirjeldamiseks ning andmemudelist ülevaate andmiseks kasutatakse klassiskeeme (ingl *class diagram*).

4.1 Sissejuhatus

Antud peatükk vastab jaotises 1.2 välja toodud uurimisküsimusele **UK-2: Millist sõnumivahetuse protokolli ja andmestruktuure on vaja BLT-kiipkaardi rakenduse prototüübi realiseerimiseks?** Selleks, et vastata uurimisküsimusele struktureeritult, on see jagatud järgmisteks alamküsimusteks:

- **UK-2.1: Millistest komponentidest loodav süsteem koosneb?**
- **UK-2.2: Milline on loodava süsteemi andmemudel?**
- **UK-2.3: Kuidas toimub komponentidevaheline suhtlus?**

Jaotis 4.2 annab ülevaate UML-i komponendi-, klassi-, ja järgnevusskeemidest. Jaotis 4.3 kirjeldab loodava süsteemi komponente ning vastab alaküsimusele UK-2.1. Jaotis 4.4 keskendub alamküsimusele UK-2.2 ning kirjeldab BLT-kliendirakenduse ja BLT-kiipkaardirakenduse andmemudelit kasutades UML-klassiskeeme. Jaotises 4.5 antakse ülevaade sõnumivahetusprotokollist erinevate süsteemsete komponentide vahel kasutades UML-järgnevusskeeme. Jaotis 4.6 sisaldab peatüki kokkuvõtet.

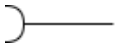
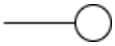
4.2 Kasutatavad UML-skeemid

Selleks, et paremini mõista peatükkides 4.3, 4.4 ja 4.5 välja toodud skeeme, tutvustame selles jaotises lühidalt kasutatavaid UML-skeemide tüüpe. Jaotis 4.5.2 annab ülevaate komponendiskeemist, jaotis 4.2.2 tutvustab klassiskeemi ning jaotis 4.2.3 keskendub järgnevusskeemile.

4.2.1 Komponendiskeem

UML-komponendiskeem kirjeldab süsteemi komponente ja nende vahelisi seoseid [15]. Tabelis 10 on välja toodud lõputöös kasutatavad komponendiskeemi tähistused.

Tabel 10. Komponendiskeemi tähistused.

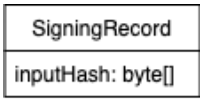
Tähistus	**Kirjeldus**
	Kasutatakse komponendi tähistamiseks.
	Komponendi poolt nõutav/kasutatav liides.
	Komponendi poolt pakutav liides.

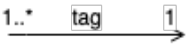
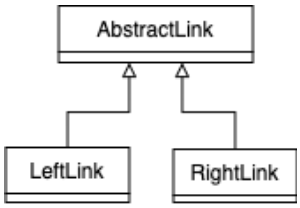
Komponenti kujutatakse ristkülikuna millele on lisatud juurde komponendi ikoon, komponendi poolt pakutavat liidest kujutatakse kuulina ning komponendi poolt kasutavat liidest kujutatakse pesana.

4.2.2 Klassiskeem

Klassiskeem kirjeldab süsteemi objektide klasse, seoseid objektide vahel, objekti operatsioone ja atribuute. Tabelis 11 on välja toodud kasutatavad klassiskeemi tähistused ja seletused.

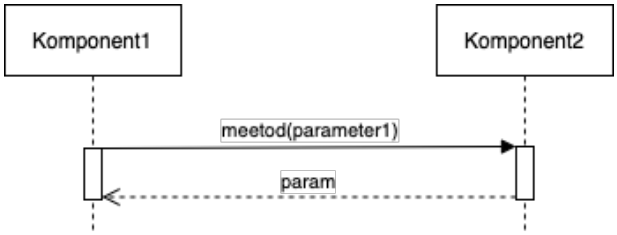
Tabel 11. Klassiskeemi tähistused.

Tähistus	**Kirjeldus**
	Kasutakase klassi tähistamiseks, kus ristküliku ülemine osa sisaldab klassi nime ning alumine osa klassi objektide atribuute (omadusi).

Tähistus	Kirjeldus
	Seost kahe klassi vahel tähistatakse pideva joonena, kus noole ots näitab suunda lähteklassilt sihtklassile. Tekst noole peal tähistab atribuudi nime.
	Selleks, et näidata, et kui mitut objekti võib mingi klassi atribuut (omadus) hõlmata, kasutataks võimsustikku. Võimsustikud määratletakse alumise ja ülemise piirgiga, kus alumiseks piiriks võib olla suvaline naturaalarv ning ülapiiriks on suvaline positiivne arv või tähis * (piiramatu arvu korral) [15].
	Kasutataks kompositsiooni tähistamiseks.
	Näitel on kujutatud abstrakset klassi ning seda klassi laiendavad klassid. Abstraksete klasside puhul hakkab klassi nimi eesliidesega „Abstract”. Kui klass A laiendab klassi B, siis seda tähistatakse pideva joonena, kus suunda näitab tühi noolepea.

4.2.3 Järgnevusskeem

Süsteemi klasside ja komponentide koostöö kirjeldamiseks kasutataks UML-järgnevusskeeme. Joonisel 7 on kujutatud järgnevusskeemi näide, kus ristkülikud tähistavad erinevaid süsteemi komponente, vertikaalne punktiirjoon kujutab komponendi ajajoont, valge kast ajajoone peal näitab hetke, millal komponent on aktiivne, pidev joon näitab kommunikatsiooni (funktsiooni väljakutset koos parameetritega) ja punktiirjoon tagastust kahe komponendi vahel [15].



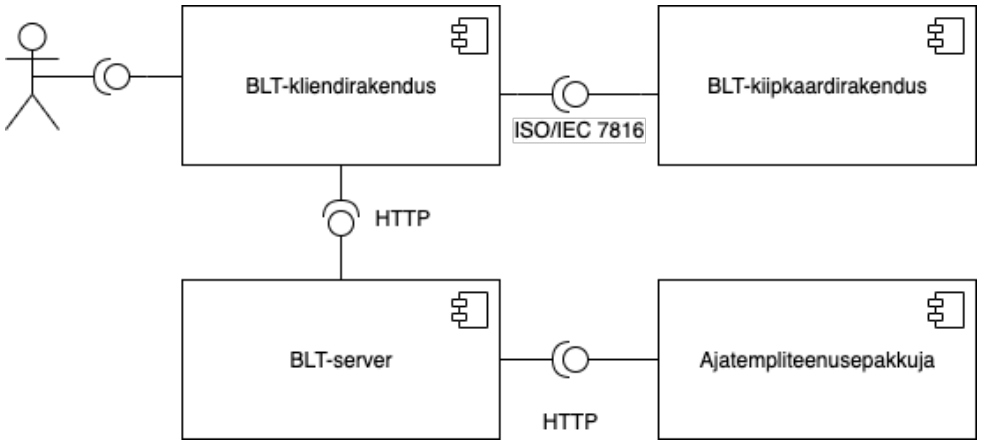
Joonis 7. Järgnevusskeemi näide.

Käesoleva töö järgnevusskeemidel ei kujutata silmuseid ja tingimusliku käitumist

(näiteks väljakutse ebaõnnestumine), sest kirjeldatud konstruktsioonid muudavad järgnevusskeemid raskesti loetavaks.

4.3 Süsteemi komponendid

Loodava BLT-signatuuriskeemi prototüübi komponendiskeem on välja toodud joonisel 8. Jooniselt on näha, et BLT-signatuuriskeemi prototüüp koosneb neljast komponendist: BLT-kliendirakendusest, BLT-kiipkaardirakendusest, BLT-serverist ja ajatempliteenusepakkujast. Lõppkasutaja kasutab BLT-kliendirakendust kiipkaardi algväärtustamiseks, võtmete genereerimiseks, allkirjastamiseks ning allkirjade verifitseerimiseks. BLT-kliendirakendus suhtleb BLT-allkirja koostamiseks HTTP (*Hypertext Transfer Protocol*) protokolli kasutades BLT-serveriga ning ISO/IEC 7816 protokolli [50] kasutades BLT-kiipkaardiga. BLT-server suhtleb ajatemplite saamiseks üle HTTP protokolli ajatempliteenust pakkuva komponendiga.



Joonis 8. BLT-signatuuriskeemi komponendiskeem.

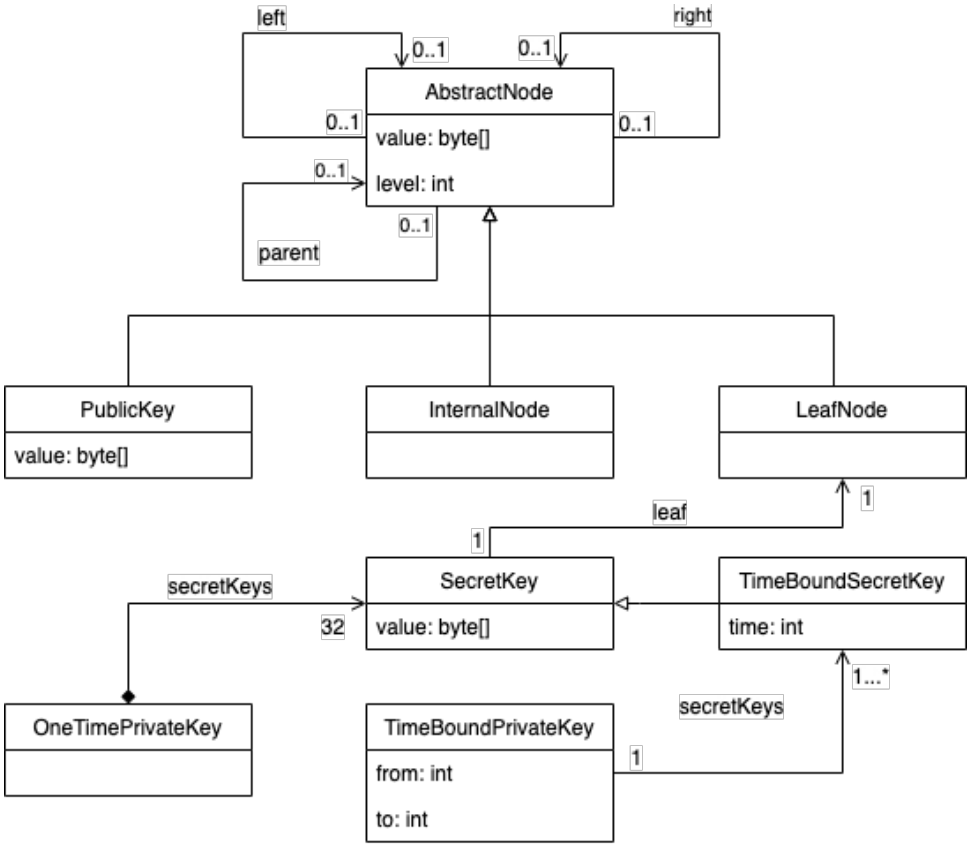
BLT-kiipkaart kasutab allkirjastamiseks BLT-OT-N-signatuuriskeemi ja BLT-server BLT-TB-signatuuriskeemi. Kirjeldatud lahendus eeldab, et BLT-kiipkaart peab usaldama BLT-serverit ning verifitseerima ainult BLT-serverilt saadud BLT-TB-allkirja. Kuna kiipkaardi poolt allkirjastatav kellaaeg tuleb BLT-TB-allkirjast, siis võimaldab lahendus vajadusel vahetada ajatempliteenusepakkujat või ajatempli standardit.

4.4 BLT andmemudelid

BLT-võtmete ja allkirja andmemudeli koostamisel on arvestatud, et BLT-signatuuriskeemist eksisteerib mitu erinevat versiooni ning andmemudelid peaks toetama BLT-TB- ja BLT-OT-N-signatuuriskeeme. Samuti on arvestatud, et andmemudeleid oleks võimalikult lihtne laiendada (näiteks uue BLT-signatuuriskeemi lisandumisel). Jaotis 4.4.1 kirjeldab BLT võtmete ja jaotises 4.4.2 BLT allkirja andmemudelit.

4.4.1 BLT-võtmete klassiskeem

Joonisel 9 on kujutatud BLT-võtmete klassiskeemi, kus Merkle'i puu koosneb tippudest (*AbstractNode* klass), millest on kolm erinevat realisatsiooni. *PublicKey* klass esindab Merkle puu juurtippu, mis on BLT-signatuuriskeemi avalik võti, *InternalNode* klass esindab Merkle puu vahetippe ning *LeafNode* klass esindab Merkle'i puu lehti. Igal Merkle'i puu vahetipul võib olla üks vasak ja parem alluv ning üks ülemus. Merkle'i puu juurtipul puudub ülemus ning lehtedel puuduvad vasak ja parem alluv. *SecretKey* klass sisaldab allkirjastamiseks vajalikku võtit ning viidet Merkle'i puu lehele. Antud viidet kasutataks autentimistee koostamiseks. *OneTimePrivateKey* ja *TimeBoundPrivateKey* klassid on vastavalt BLT-OT-N- ja BLT-TB-signatuuriskeemide privaatsed võtmed.

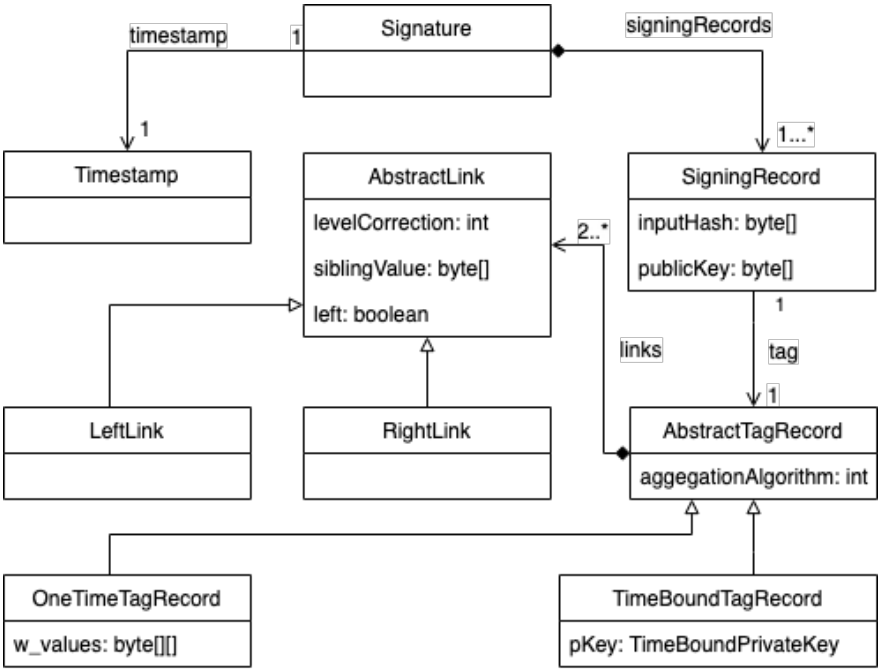


Joonis 9. BLT-võtmete klassiskeem.

TimeBoundPrivateKey klass sisaldab ajavahemikku, millal privaatset võtit kasutada saab, ning viidet BLT-TB allkirjastamisvõtmetele (*TimeBoundSecretKey* klass). Igat *TimeBoundSecretKey* objekti, mis laiendab *SecretKey* klassi, saab allkirjastamiseks kasutada kindlal ajahetkel (*time* atribuut). *OneTimePrivateKey* klass koosneb täpselt 32 salajasest võtmest (eeldusel, et allkirjastatakse 32-bitist POSIX aega).

4.4.2 BLT-allkirja klassiskeem

Joonisel 10 on kujutatud BLT-allkirja klassiskeemi. BLT-allkiri koosneb *Signature* klassist, mis sisaldab ühte ajatemplit (*Timestamp* klass) ning ühte kuni mitut *SigningRecord* klassi.



Joonis 10. BLT-allkirja klassiskeem.

SigningRecord klass ühendab omavahel ära sisendräsi (*inputHash* atribuut), avaliku võtme (*publicKey* atribuut), mida kasutati allkirjastamiseks, ning andmed, mis tõestavad, et allkirjastamiseks kasutatud võti on osa avalikust võtmest (*AbstractTagRecord* klass). *AbstractTagRecord* klassi laiendab *OneTimeTagRecord* klass, mis sisaldab allkirjastamiseks kasutatud BLT-OT salajase võtme väärtusi, mis on arvutatud jaotises 2.5.8 tutvustatud valemi 1 järgi. Sarnaselt sisaldab *TimeBoundTagRecord* klass viidet BLT-TB salajasele võtmele. *AbstractTagRecord* klassiga on seotud ka autentimistee klassid (*LeftLink* ja *RightLink*), mis tõestab, et allkirjastamiseks kasutatud võti on osa avalikust võtmest. Kirjeldatud klassiskeem põhineb Henri Lakk'i tööl [52].

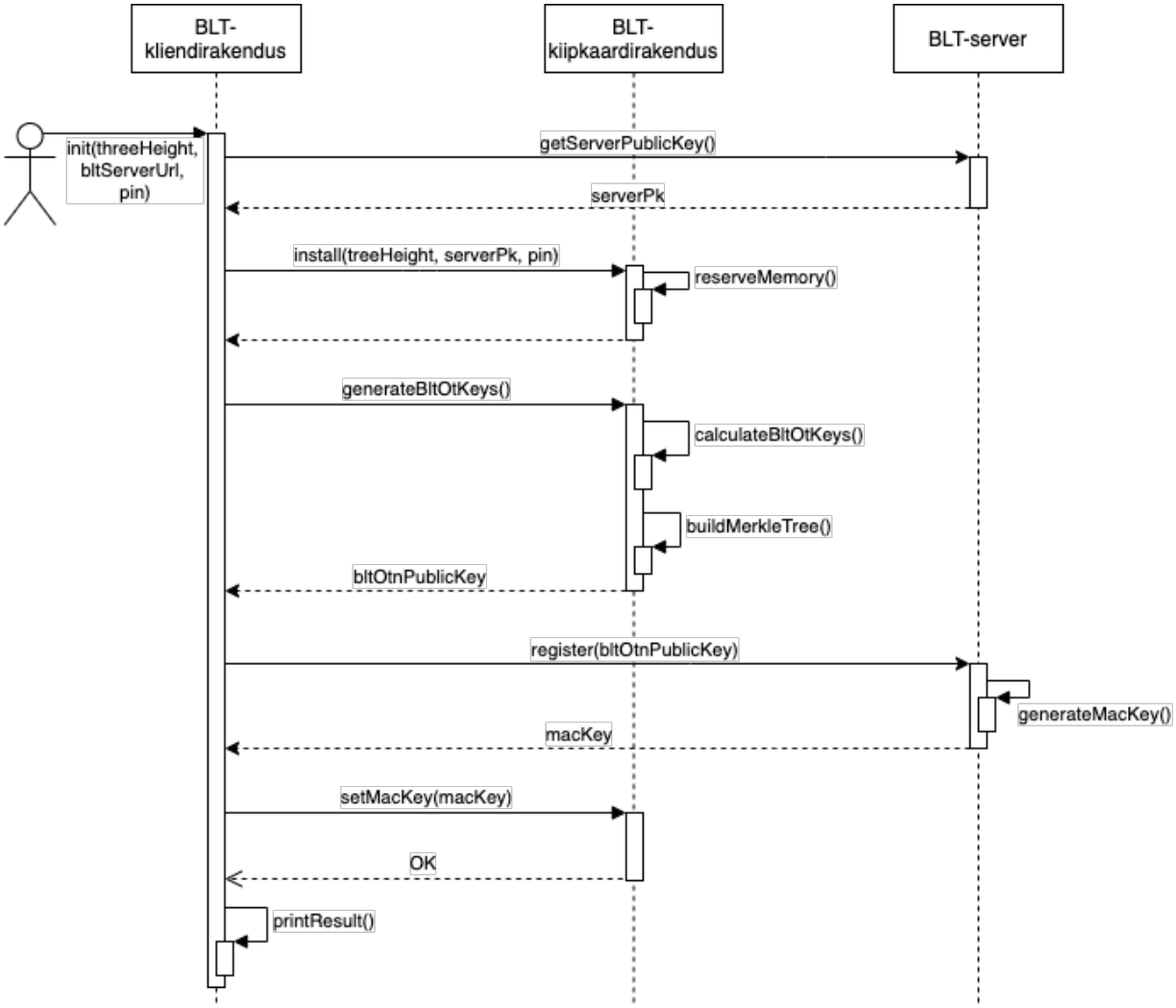
4.5 Sõnumivahetusprotokolli ülevaade

Selleks, et saada parem ülevaade loodava BLT-signatuuriskeemi prototüübist on käesolevas jaotises kirjeldatud detailsemalt peatükis 3.4 välja toodud funktsionaalseid nõudeid. Selleks on koostatud iga funktsionaalse nõude kohta järgnevusskeem ning kirjeldatud sisendeid ja väljundeid, mis liiguvad jaotises 4.3 tutvustatud erinevate prototüübi komponentide vahel. Jaotis 4.5.1 annab ülevaate BLT-OT-N-

signatuuriskeemi kiipkaardirakenduse algväärtustamisest ja võtmete genereerimist, jaotis 4.5.2 kirjeldab allkirjastamist ning jaotis 4.5.3 keskendub BLT-allkirja verifitseerimisele.

4.5.1 Kiipkaardi rakenduse paigaldamine ja võtmete genereerimine

Joonisel 11 on välja toodud BLT-kiipkaardirakenduse algväärtustamise ja BLT-võtmete genereerimise järgnevusskeem. Kiipkaardirakenduse algväärtustamise päring käivitatakse lõppkasutaja poolt ning sisaldab genereeritava Merkle'i puu kõrgust, BLT-serveri aadressi ning kiipkaardi PIN-koodi. Kõigepealt küsitakse BLT-serveri käest serveri avalik võti ning saadetakse see koos puu kõrguse ja PIN-koodi parameetritega kaardile (*install* funktsiooni väljakutse). Väljakutse tulemusena reserveerib kaart kiipkaardil mälu, mis on vajalik kaardi tööks. Järgmisena saadab BLT-kliendirakenduse komponent kiipkaardile käsu BLT-OT-N võtmete arvutamiseks (*generateBltOtKeys* funktsiooni väljakutse), mille tulemusena arvutab kaart BLT-OT-võtmed ning ehitab genereeritud võtmetest Merkle'i puu. Funktsiooni *generateBltOtKeys* tulemusena tagastab kaart BLT-kliendirakenduse komponendile arvutatud Merkle'i puu juurräsi (BLT-OT-N avalik võti), mis saadetakse edasi BLT-serverile registreerimiseks (*register* funktsioon). BLT-server tagastab võtme registreerimispäringu peale MAC võtme, mis saadetakse edasi kaardile (*setMacKey* funktsioon).



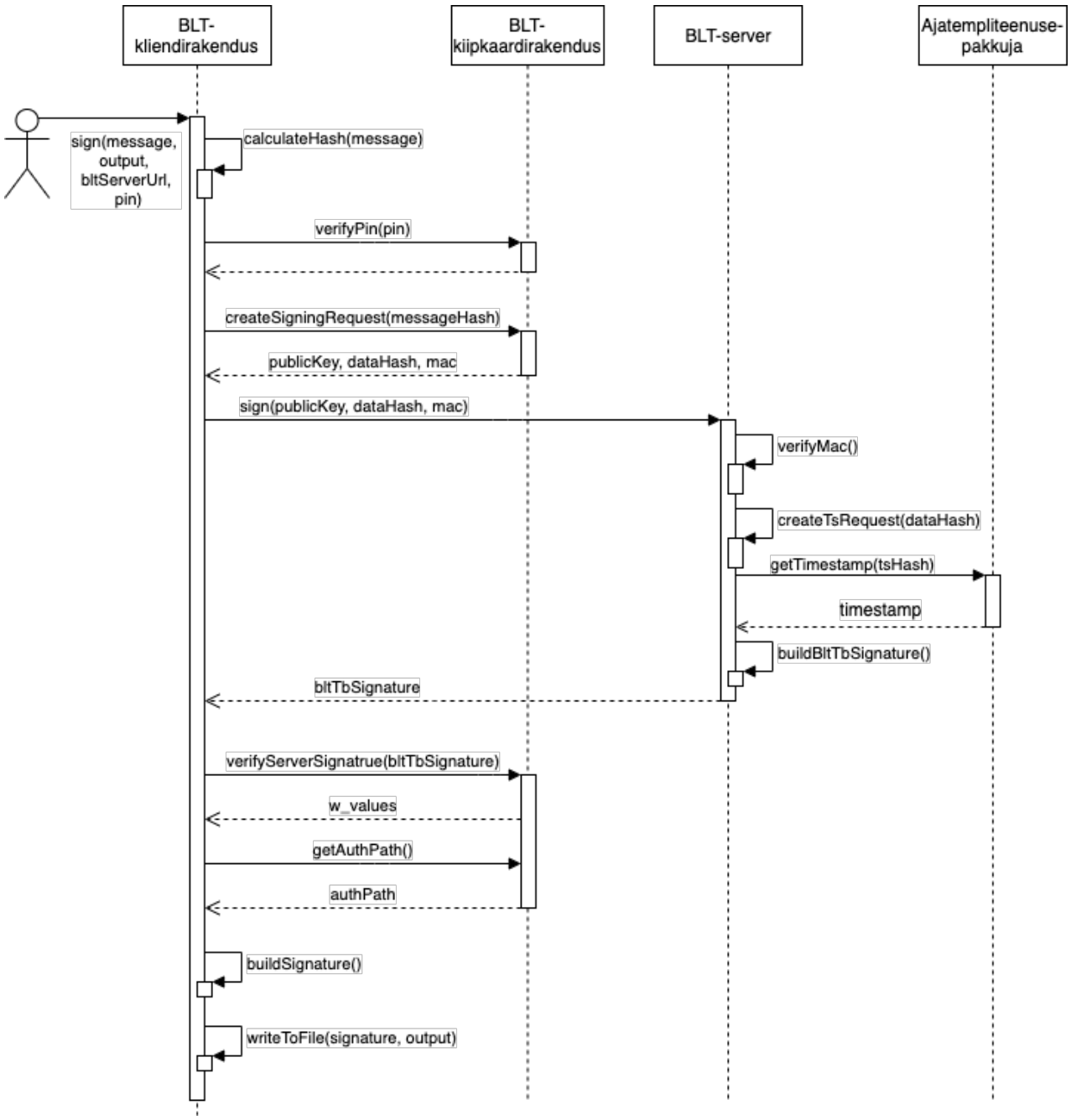
Joonis 11. BLT-kiipkaardirakenduse algväärtustamine ja võtmete genereerimine.

MAC võtit kasutab kiipkaart allkirjastamispäringute sõnumiautentimiskoodi arvutamiseks. Arvutatud sõnumiautentimiskood pannakse kaasa iga allkirjastamispäringuga ning BLT-server saab sõnumiautentimiskoodi verifitseerides veenduda, et allkirjastamispäring tuli õigelt kaardilt. Lisaks võimaldab kiipkaardi avaliku võtme registreerimine tulevikus lihtsasti realiseerida võtme tühistamise funktsionaalsust.

4.5.2 Sõnumi allkirjastamine

Selleks, et kasutaja saaks allkirjastada sõnumit, peab ta BLT-kliendirakendusele saatma allkirjastatava sõnumi, kiipkaardi PIN-koodi, BLT-serveri aadressi ja failinime, kuhu

BLT allkiri salvestatakse. Sõnumi allkirjastamise järgnevusskeem on välja toodud joonisel 12.



Joonis 12. BLT-signatuuriskeemi allkirjastamise järgnevusskeem.

Esimesena sõnum räsitakse kasutades SHA-256 räsifunktsiooni ning seejärel saadetakse kiipkaardile kontrollimiseks PIN-kood (*verifyPin* funkstiooni väljakutse). Õige PIN-koodi korral saadab BLT-kliendirakendus BLT-kiipkaardile allkirja koostamise

päringu (*createSigningRequest* funktsiooni väljakutse), mis tagastab BLT-kliendirakendusele

- kaardil genereeritud BLT-OT-N avaliku võtme (*publicKey*),
- BLT-serverile allkirjastamiseks mineva räsi (*dataHash*), mis on arvutatud valemi $dataHash = H(messageHash, X)$ järgi, kus H on turvaline räsifunktsioon, *messageHash* allkirjastatava sõnumi räsi ning X BLT-OT-N salajase võtmete räsid ja
- sõnumiautentimiskoodi (*mac*), mille arvutamiseks kasutatakse valemit $mac = HMAC(publickKey \| dataHash, macKey)$, kus $HMAC$ on räsipõhine sõnumiautentimiskoodi algoritm ning *macKey* kiipkaardile salvestatud sõnumiautentimiskoodi võti.

Järgmisena edastab BLT-kliendirakendus saadud tulemused BLT-serverile (*sign* funktsiooni väljakutse), mis valideerib sõnumiautentimiskoodi õigsust ning genereerib ajatempliteenusele mineva sisendräsi kasutades valemit $tsHash = H(dataHash, secretKey_t)$, kus $secretKey_t$ on BLT-TB salajane võti ajahetkel t. Ajatemplipäringu räsi saadetakse ajatempliteenusepakkujale (funktsiooni *getTimestamp* väljakutse), mis tagastab ajatempli aja t kohta. Seejärel BLT-server koostab ja tagastab BLT-kliendirakendusele BLT-allkirja, mis sisaldab ajatemplit, allkirjastatud räsi, BLT-TB salajast ja avalikku võtit ning autentimisteed BLT-TB slajasest võtmest BLT-TB avaliku võtmeni. Seejärel edastab BLT-kliendirakendus BLT-TB-allkirja BLT-kiipkaardirakendusele (*verifyServerSignature* funktsiooni väljakutse), mis kontrollib, et BLT-TB-allkirja sisendräsi on võrdne BLT-kiipkaardirakenduse poolt koostatud *dataHash* räsiga ning BLT-TB salajane võti vastab kaardile salvestatud BLT-TB avalikule võtmele. Kui kontrollid õnnestuvad, siis BLT-kiipkaart tagastab BLT-kliendirakendusele osad BLT-OT salajasest võtmest, mis on arvutatud peatükis 2.5.8 välja toodud algoritmi järgi. Järgmisena saab BLT-kliendirakendus küsida BLT-kiipkaardilt autentimistee (*getAuthPath* funktsiooni väljakutse), mis tõestab, et BLT-OT võti on osa BLT-OT-N avalikust võtmest. Seejärel BLT-kliendirakendus lisab BLT-OT-N allkirja BLT-serveri poolt saadud allkirja juurde

(*buildSignature* funktsiooni väljakutse) ning salvestab selle etteantud faili (*writeToFile* funktsiooni väljakutse).

4.5.3 BLT-allkirja verifitseerimine

BLT-allkirja verifitseerimine võtab sisendina ette allkirjastatud faili, BLT-allkirja ning järjendi usaldusväärsetest avalikest võtmetest. Allkirja verifitseermine toimub BLT-kliendirakenduse komponendis ning lähtub peatükkides 2.5.7 ja 2.5.8 tutvustatud BLT-TB- ning BLT-OT-allkirja verifitseerimise algoritmidest.

4.6 Kokkuvõte

Käesolevas peatükis kasutati UML-skeeme, et kirjeldada loodava BLT-signatuuriskeemi arhitektuuri. UML-komponendiskeemi abil kirjeldati loodava süsteemi komponente ja nende vahelisi seoseid ning liideseid. BLT-signatuuriskeemi prototüüp koosneb neljast peamisest komponendist: BLT-kliendirakendus, BLT-kiipkaardirakendus, BLT-server ja ajatempliteenusepakkuja. Klassiskeemide abil kirjeldati BLT-võtmete ja -allkirja andmestruktuure. Järgnevusskeemide abil anti detailne ülevaade kiipkaardi algväärtustamise, võtmete genereerimise ja BLT-allkirja koostamise ajal erinevate prototüübi komponentide vahel liikuvatest andmetest ning päringutest. Kirjeldatud skeemid saab võtta sisendiks süsteemi arendamiseks.

5 Realisatsioon

Käesolev peatükk keskendub BLT-signatuuriskeemi prototüübi realisatsioonile. Käesolev peatükis antakse ülevaade BLT-võtmete genereerimise, Merkle'i puu, autentimistee koostamise algoritmidest.

5.1 Sissejuhatus

Antud peatükk vastab jaotises 1.2 välja toodud uurimisküsimusele **UK-3: Kuidas realiseerida BTL-signatuuriskeemi võtmehalduse ja allkirjastamise moodul?** Selleks, et vastata uurimisküsimusele struktureeritult, on see jagatud järgmisteks alamküsimusteks:

1. **UK-3.1: Kuidas realiseerida BLT-OT-allkirjastamisvõtmete genereerimine kiipkaardi ning BLT-TB-allkirjastamisvõtmete genereerimine BLT-serveri komponendis?**

2. **UK-3.2: Kuidas realiseerida BLT-OT-N ja BLT-TB avaliku võtme arvutamine?**

3. **UK-3.3: Kuidas efektiivselt realiseerida autentimistee arvutamine?**

Jaotis 5.2 annab ülevaate BLT-OT- ja BLT-TB-privaatvõtmete genereerimise algoritmidest. Jaotis 5.3 vastab alamküsimusele UK-3.2 ning keskendub BLT-OT ja BLT-TB avalike võtmete väljaarvutamise algoritmidele ning jaotises 5.4 kirjeldatakse autentimistee arvutamist. Jaotises 5.5 antakse ülevaade realiseeritud prototüübi piirangutest ning jaotis 5.6 sisaldab peatükki kokkuvõtet. Lisa 1 sisaldab viidet prototüübi lähtekoodile.

5.2 BLT-privaatvõtmete genereerimine

Käesolev jaotis kirjeldab algoritme, mida kasutatakse BLT-privaatvõtmete genereerimiseks. Jaotises 5.2.1 antakse ülevaade BLT-OT-privaatvõtme koostamise algoritmist BLT-kiipkaardi komponendis ja jaotis 5.2.2 keskendub BLT-TB-privaatvõtme genereerimisele BLT-serveri komponendis.

5.2.1 BLT-OT-võtmete genereerimine

BLT-OT-võtme genereerimisel peab arvestama, et kiipkaardil on piiratud mälu ja arvutusvõimsus. On ilmselge, et kiipkaardile ei ole võimalik salvestada kõiki BLT-OT-võtmeid, sest üks BLT-OT-privaatvõti koosneb 32-st 32-baidisest salajasest võtmest, mis võtaks 4096 BLT-OT-võtme puhul vähemalt 32*32*4096=4194304 baiti=4096 KB mälu. Tavaliselt on kiipkaardil 40KB-160KB salvestusruumi.

BLT-OT-võtmete genereerimine põhineb ideel, et vajalik privaatvõti arvutatakse välja siis, kui seda on vaja kasutada. BLT-OT-privaatvõtme arvutamiseks kasutatakse deterministliku funktsiooni, mis põhineb sümmeetrilisel krüptograafial. Selleks genereeritakse kaardi algväärtustamise ajal AES (*Advanced Encryption Standard*) sümmeetriline võti ning 28-baidine juhuarv, mida kasutatakse BLT-OT-privaatvõtme arvutamisel. Joonis 13 kirjeldab algoritmi, mida kasutatakse BLT-OT-võtme genereerimiseks.

```
0: function byte[] calculatePublicKey(leafIndex) {
1:    hashes = new byte[]
2:    for i = 0 to 31 {
3:       secretKey = AES_crypt(seed || leafIndex || i)
4:       hashes = hashes || H(secretKey)
5:    }
6:    return H(hashes)
7: }
```

Joonis 13. BLT-OT-võtme genereerimine.

Joonisel 13 kirjeldatud algoritm võtab parameetritena ette võtme (Merkle'i puu lehe) järjekorranumbri. Iga salajase võtme arvutamiseks kasutatakse *AES_crypt* funktsiooni, mis krüpteerib kaardi algväärtustamise ajal genereeritud juhuarvu (28 baiti), võtme

järjekorranumbri (2 baiti) ning salajase võtme indeksi (2 baiti) kasutades kaardi algväärtustamisel genereeritud AES võtit (rida 3). Ülejäänud osa algoritmist on identne peatükis 2.5.8 tutvustatud BLT-OT võtme genereerimise algoritmiga.

Kirjeldatud algoritmi kasutades on kaardil vaja hoida ainult AES võtit ning ülejäänud võtmed on alati tuletatavad. BLT-OT-privaatvõtme genereerimise implementatsioon asub lähtekoodis blt-jc-applets/blt-javacard/src/main/java/com/github/mihkelsaar/blt kataloogis *BltOtAesKeyManager.java* klassis.

5.2.2 BLT-TB-privaatvõtme genereerimine

Kuna BLT-TB-signatuuriskeemi kasutatakse BLT-serveri komponendis, kus on piisavalt arvutusvõimsust ning salvestusruumi, siis BLT-TB-privaatvõtme genereerimine on identne peatükis 2.5.7 tutvustatud algoritmiga. BLT-TB-salajase võtme genereerimiseks kasutatakse Java programmeerimiskeele standardset juhuarvugeneraatorit.

BLT-TB-privaatvõtme genereerimise implementatsioon asub lähtekoodis *blt-signature/src/main/java/com/github/mihkelsaar/blt/key* kataloogis *KeyPairGenerator.java* klassis.

5.3 BLT-OT-N- ja BLT-TB-avaliku võtme arvutamine

Antud jaotis keskendub BLT-avaliku võtmete arvutamisele. Kuna BLT-OT-N- ja BLT-TB-avaliku võtme arvutamine taandub Merkle'i puu ehtimise algoritmile, siis antakse jaotises 5.3.1 ülevaade räsipuu ehitamise algoritmidest, mida kasutatakse BLT-kiipkaardi ja BLT-serveri komponentides. Jaotis 5.3.2 kirjeldab täpsemalt avaliku võtme genereerimist BLT-kiipkaardi komponendis ning jaotis 5.3.3 annab detailsema ülevaate avaliku võtme koostamisest BLT-serveri komponendis.

5.3.1 Merkle'i puu ehitamise algoritmid

Käesolev jaotis annab ülevaate kahest erinevast algoritmist, mida kasutatakse Merkle'i puu ehitamiseks. Nendest esimene põhineb Merkle'i poolt tutvustatud *TREEHASH* algoritmil [27] ning teine Merkle'i puu kihtide hoidmisel plokkides.

TREEHASH algoritm põhineb ideel, et puu ehitamisel hoitakse alles ainult neid puu tippe, mida on vaja tulevikus järgmiste puu tippude arvutamiseks. Selleks kasutatakse puu tippude hoidmiseks pinu (ingl *stack*) andmestruktuuri. Lehe lisamisel pinusse kontrollitakse, kas pinus viimase tipu kõrgus on võrdne lisatava tipu kõrgusega. Kui kõrgused on võrdsed, siis pinust eemaldatakse viimane tipp ning arvutatakse uus kahe tipu järglane. Kui kõrgused pole võrdsed, siis lisatakse leht pinu andmestruktuuri. Merkle'i puu tippude kõrguse kontrolli korratakse niikaua, kuni kahe viimase tipu kõrgus pole võrdne või kui kõik lehed on lisatud loodavasse Merkle'i puusse. Kirjeldatud algoritm on välja toodud joonisel 14.

```
 0: function update() {
 1:   while stack.size() > 1 {
 2:      if(!consolidate()) {
 3:         break
 4:      }
 5:   }
 6: }
 7:
 8: function consolidate() {
 9:    node1 = stack.pop()
10:    node2 = stack.pop()
11:    if (node1.level != node2.level) {
12:       stack.push(node2)
13:       stack.push(node1)
14:       return false
15:    }
16:    level = node1.level + 1
17:    node = Node(H(node2.value||node1.value||level), level)
18:    stack.push(node)
19:    return true
20: }
```

Joonis 14. BLT-avaliku võtme arvutamise TREEHASH algoritm.

Joonisel on muutuja *stack* pinu andmestruktuur, kus *stack.pop()* tagastab ja eemaldab pinus oleva viimase elemendi ning *stack.push()* lisab pinu andmestruktuuri etteantud elemendi.

Teine käesolevas magistritöös kasutatav Merkle'i puu juurtipu väljaarvutamise algoritm põhineb plokkidel. Selleks jagatakse puu *ülemiseks* ja *käesolevaks* plokiks. Kui meil on

Merkle'i puu kõrgusega N, siis *ülemises* plokis hoitakse lehti N/2 kihist. *Ülemisse* kihti minevad lehed arvutatakse välja kaustades *käesolevat* plokki. *Käesolevasse* plokki lisatakse Merkle'i puu lehti niikaua, kuni see selle autentimistee viib välja *ülemise* ploki leheni. Kui *käesoleva* ploki juurtipp on välja arvutatud, siis kopeeritakse see *ülemise* ploki lehte ning liigutakse edasi järgmiste lehtede arvutamisele. Kui kõik *ülemise* ploki lehed on välja arvutatud, siis arvutatakse nende lehtede põhjal välja kõik *ülemise* puu kihid kuni Merkle'i puu juurtipuni.

Plokkidel põhinev algoritm võtab võrreldes TREEHASH algoritmiga rohkem mälu, kuid võimaldab realiseerida kiirema autentimistee arvutamise algoritmi. Kui genereeritaval puul on 4096 lehte (puu kõrgusega 13), siis on *ülemise* ploki kõrgus 7 ning mälus hoitakse 2^{7-1}=64 lehte ning 2^{7-1}-1=63 sisetippu. *Käesolevas* plokis hoitakse mälus samuti 64 lehte ning 63 sisetippu. Seega võtavad mõlemad plokid SHA-256 räsifunktsiooni kasutades (64+63)*33=4191 baiti mälu (eeldusel, et lisaks räsile hoitakse mälus ka tipu kihi numbrit (üks bait)). TREEHASH algoritm salvestab 13 räsi, mis võtab kokku 13*33=429 baiti mälu.

5.3.2 Merkle'i puu ehitamine BLT-kiipkaardi komponendis

TREEHASH algoritm sobib hästi Merkle'i puu ehitamiseks BLT-kiipkaardil, millel on 40-60 KB EEPROM mälu. TREEHASH algoritmi puhul ei pea Merkle'i puu ehitamisel hoidma kõiki Merkle'i puu tippe kiipkaardi mälus. Kuna kiipkaardil on piiratud ressursid, siis ei ole Java Card SDK-s realiseeritud pinu andmestruktuuri. TREEHASH algoritmi kasutamiseks on kiipkaardi moodulis realiseeritud pinu andmestruktuur, mis põhineb baidimassiividel. Realiseeritud lähtekood asub git-i koodihoidlas „*treehash*" harus, *blt-jc-applets/blt-javacard/src/main/java/com/github/mihkelsaar/blt* kataloogis *Stack.java* ja *MerkleTree.java* klassides.

Plokkidel põhinev algoritm sobib kasutamiseks kiipkaartidel, millel on rohkem kui 60KB EEPROM mälu. Plokkidel põhinev Merkle'i puu koostamise lähtekood asub git-i koodihoidlas „*master*" harus *blt-jc-applets/blt-javacard/src/main/java/com/github/mihkelsaar/blt* kataloogis *MerkleTree.java* klassis.

5.3.3 Merkle'i puu ehitamine BLT-serveri komponendis

BLT-serveri realisatsioonis kasutatakse pinu andmestruktuurina Java SDK-st pärit pinu *java.util.ArrayDeque* klassi. TREEHASH algoritmi realisatsioon asub *blt-signature/src/main/java/com/github/mihkelsaar/blt/key* kataloogis *MerkleTreeBuilder.java* klassis. Erinevalt BLT-kiipkaardi realisatsioonist hoiab BLT-server kõiki Merkle'i puu tippe mälus.

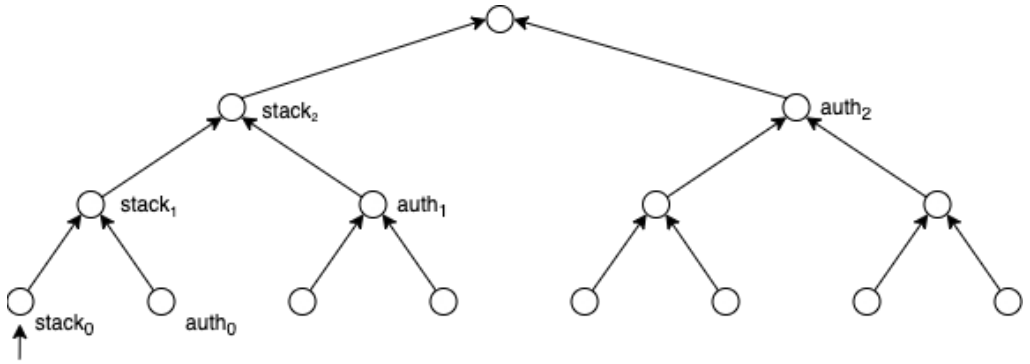
5.4 Autentimistee arvutamine BLT-allkirja jaoks

Käesolev jaotis annab ülevaate autentimistee arvutamistest BLT-kiipkaardi ja BLT-serveri komponentides. Autentimisteed kasutatakse tõestamaks, et Merkle'i puu lehte kasutati Merkle'i puu juurtipu arvutamisel. Iga BLT-TB- ja BLT-OT-N-allkiri sisaldab autentimisteed kasutatud võtmest (Merkle'i puu lehest) Merkle'i puu juurtipuni. Jaotised 5.4.1 ja 5.4.2 annavad ülevaate autentimistee arvutamise algoritmidest BLT-kiipkaardi komponendis ning jaotis 5.4.3 tutvustab autentimistee koostamist BLT-serveri komponendis.

5.4.1 TREEHASH algoritmil põhinev autentimistee arvutamine kiipkaardil

BLT-kiipkaardi autentimistee arvutamise algoritm põhineb Merkle'i *TREEHASH* algoritmil [27]. Selleks, et autentimisteed efektiivsemalt arvutada kasutatakse mitut pinu andmestruktuuri. Kui loodava puu tipu kõrgus on *H*, siis luuakse Merkle'i puu arvutamise ajal *H-1* pinu andmestruktuuri. Olgu stack_h pinu maksimaalse kõrgusega h ning meetoditega $\text{stack}_h.initialize(startnode, h)$ ja $\text{stack}_h.update(t)$. Meetodit $initialize(startnode, h)$ kasutatakse pinu andmestruktuuri algväärtustamiseks, kus *startnode* on Merkle'i puu lehe järjekorranumber ning *h* maksimaalne puu kõrgus. Meetodit *update(t)* kasutatakse alampuu uuendamiseks, kus parameeter *t* defineerib mitu korda käivitatakse joonisel 14 välja toodud *consolidate* funktsiooni või joonisel 13 kirjeldatud uue lehe arvutamist. Merkle'i puu väljaarvutamisel algväärtustatakse kõik stack_h andmestruktuurid nii, et stack_h väärtusteks on esimese lehe tee iga tipu vasakud alluvad. Lisaks arvutatakse välja esimene autentimistee $auth = auth_0, \ldots, auth_{h-1}$, milleks on esimese lehe tee parempoolsed alluvad. Joonis 15

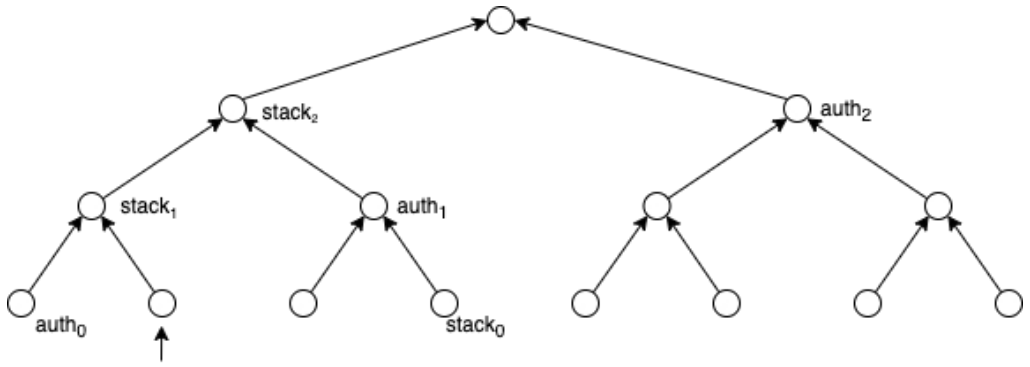
sisaldab andmestruktuuride algväärtusi peale Merkle'i puu juurtipu arvutamist. Nool puu lehe juures näitab puu lehte, mille kohta autentimisteed arvutatakse [53].



Joonis 15. Merkle'i puu algväärtused peale juurtipu väljaarvutamist.

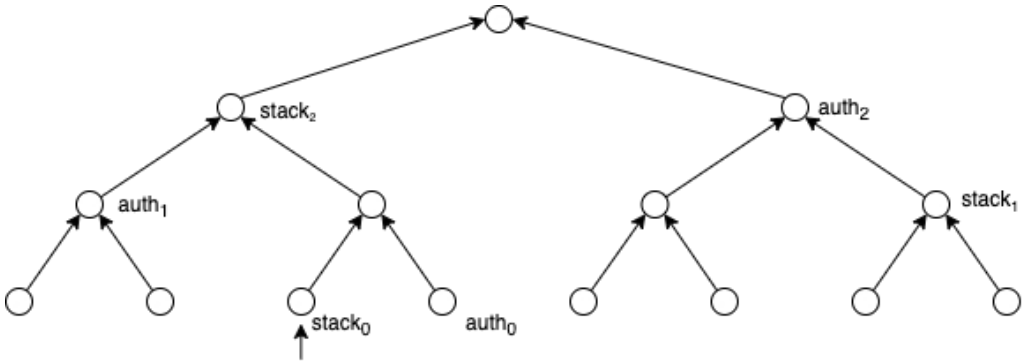
Esimese lehe autentimistee väljaarvutamiseks tuleb kõigepealt arvutada lehe BLT-OT avalik võti kasutades joonisel 13 kirjeldatud algoritmi. Ülejäänud autentimistee tipud on juba arvutatud ning asuvad andmestruktuuris *auth* . Selleks, et arvutada järgmise lehe autentimisteed tuleb kõigepealt uuendada *auth* ning seejärel stack_h andmestruktuure. Kui *leaf* on hetkel aktiivse lehe järjekorranumber, siis autentimistee tippu $auth_h$ tuleb järgmise lehe jaoks uuendada kui $leaf+1$ jagub arvuga 2^h . Vajalik autentimistee tipp on juba arvutatud ning asub andmestruktuuris stack_h . Peale autentimistee uuendamist on stack_h andmestruktuur tühi. Selleks, et stack_h andmestruktuuri saaks kasutada ka teiste autentimisteede arvutamisel tuleb see uuesti initsialiseerida. Selleks kasutatakse $\text{stack}_h.initilize(startnode, h)$ meetodit, kus *startnode* arvutatakse valemi $startnode = leaf + 1 + 2^h \oplus 2^h$ järgi. Selleks, et vältida stack_h juurräsi väljaarvutamist ühekorraga, kasutatakse $\text{stack}_h.update(n)$ funktsiooni. Kuna $auth_h$ autentimistee tippu tuleb järgmisena muuta 2^h sammu pärast, siis me võime igat stack_h struktuuri uuendada ühe autentimistee arvutamisel kaks korda. See tagab, et iga autentimistee arvutamisel tehakse maksimaalselt *(H-1)*2* operatsiooni ning stack_h andmestruktuur sisaldab alampuu tipuräsi parajasti siis, kui seda on vaja autentimistee elemendil $auth_h$. Kui Merkle'i puu koosneb 4096 lehest (puu kõrgusega 12), siis tehakse autentimistee arvutamisel maksimaalselt (12-1)*2=22 operatsiooni [53].

Joonisel 16 on välja toodud pinu ja autentimistee elementide väärtused peale esimese autentimistee arvutamist. Arvutuse käigus kopeeriti stack_0 pinust alampuu juurtipp autentimistee $auth_0$ andmestruktuuri ning initsialiseeriti stack_0 andmestruktuur uuesti, kus *startnode=4* ($1+1+2^1$) ja *h=0*. Ülejäänud stack_h ja $auth_h$ struktuure ei uuendatud, kuna stack_h struktuurides asuvad juba vastavad alampuu juurtipud.



Joonis 16. Pinu ja autentimistee väärtused peale esimest sammu.

Sarnaselt on joonisel 17 välja toodud pinu ja autentimistee elementide väärtused peale teise autentimistee arvutamist.

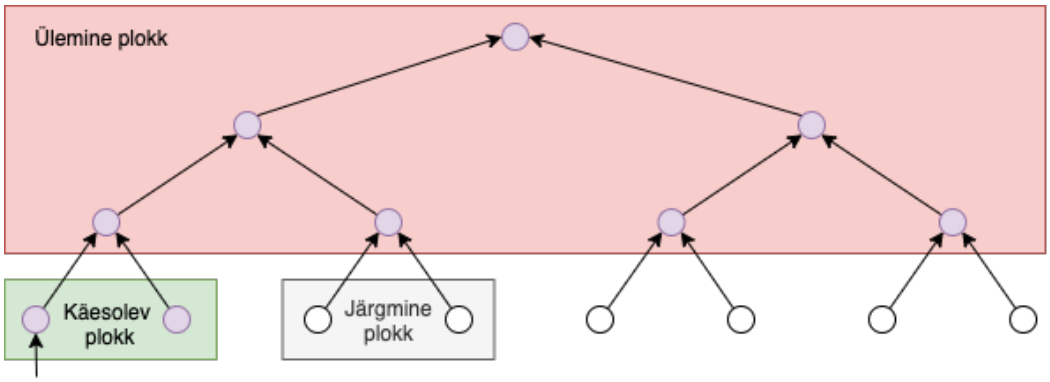


Joonis 17. Pinu ja autentimistee väärtused peale teist sammu.

Kirjeldatud algoritm on realiseeritud BLT-kiipkaardirakenduses ja asub git-i koodihoidlas „*treehash*" harus *blt-jc-applets/blt-javacard/src/main/java/com/github/mihkelsaar/blt* kataloogis *MerkleTree.java* klassis. TREEHASH autentimistee algoritmi kasutades peab arvestama, et see võib olla kaitstud patendiga ning algoritmis kulub palju aega puu lehtede uuesti arvutamiseks.

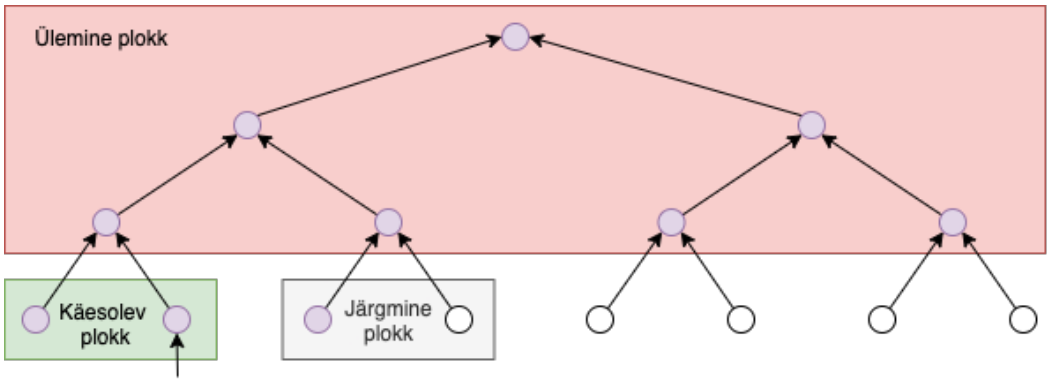
5.4.2 Plokkidel põhinev autentimistee arvutamine kiipkaardil

Plokkide puhul on Merkle'i puu jagatud *ülemiseks, käesolevaks* ja *järgmiseks* plokiks. Kui meil on Merkle'i puu kõrgusega N, siis *ülemises* plokis hoitakse kiipkaardi mälus lehti N/2 kihist. *Käesolevas* plokis hoitakse lehti, mis on vajalikud hetkel aktiivse lehe autentimistee väljaarvutamiseks *ülemise* ploki leheni. *Järgmises* plokis hoitakse lehti, mida on tulevikus vaja autentimistee väljaarvutamiseks. Joonisel 18 on kujutatud *ülemist*, *käesolevat* ja *järgmist* plokki, kui puu kõrgus on 4 (kaheksa lehte). Nool puu lehe juures näitab puu lehte, mille kohta autentimisteed arvutatakse.



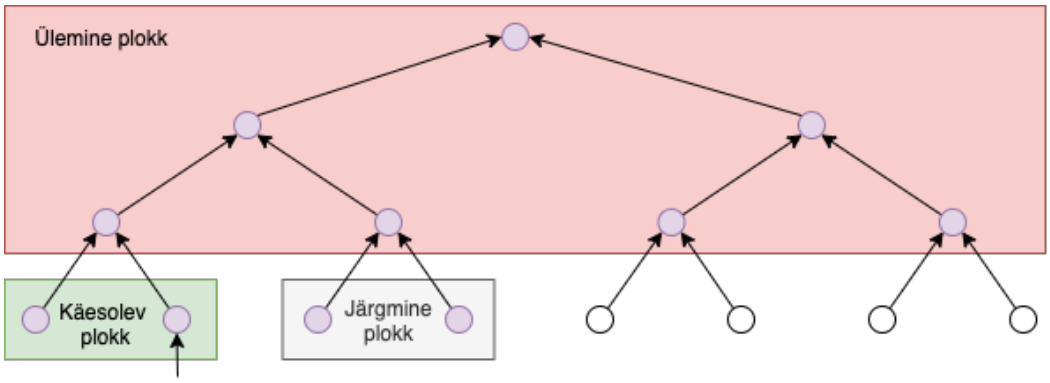
Joonis 18. Plokkide algolek.

Joonisel 18 on punase kastiga eraldatud Merkle'i puu *ülemine* plokk, rohelise kastiga *käesolev* plokk ning halliga *järgmine* plokk. Plokis olevad välja arvutatud lehed on lillat värvi. Esimese lehe autentimistee on kergesti arvutatav, sest *käesolevas* plokis on olemas kõik alampuu lehed, et koostada *autentimistee* ülemisse plokki ning ülemises plokis kõik lehed, et koostada tee sealt edasi Merkle'i puu juurtippu. Selleks, et vältida ploki kõikide lehtede väljaarvutamist korraga, arvutatakse iga autentimistee arvutamise järel välja *järgmise* ploki uus leht. Joonisel 19 on välja toodud plokkide olek peale esimese lehe arvutamist. Autentimistee arvutuse käigus lisati *järgmisesse* plokki uus leht (joonisel märgitud lillana) ning liigutati järgmisena arvutatava autentimistee lehe indeks ühe võrra paremale.



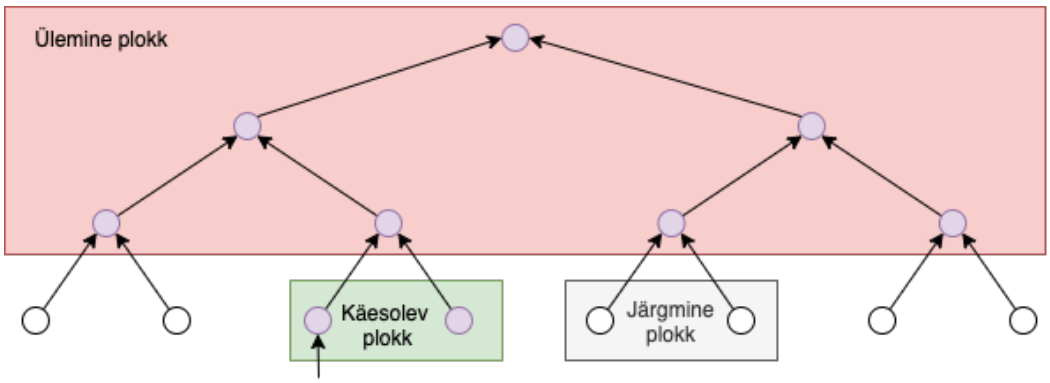
Joonis 19. Plokkide olek peale esimese autentimistee arvutamist.

Peale teise lehe autentimistee genereerimist arvutatakse kõigepealt välja uus *järgmise* ploki leht (joonis 20).



Joonis 20. Plokkide olek enne teise autentimistee arvutamise lõppu.

Kuna peale teise lehe autentimistee arvutamist on *käesolevas* plokis kõikide lehtede autentimisteed arvutatud, siis kopeeritakse *järgmise* ploki sisu *käesolevasse* plokki ning nihutatakse *järgmist* plokki edasi 2 lehe võrra. Joonisel 21 on välja toodud plokkide sisu peale teise lehe autentimistee arvutamist.



Joonis 21. Plokkide olek peale teise autentimistee arvutamist.

Plokkidel põhinev algoritm võtab võrreldes TREEHASH algoritmil põhineva autentimistee koostamise algoritmiga rohkem mälu, kuid on tunduvalt kiirem, kuna plokkide puhul ei arvutata lehti mitu korda välja. Kuna lehe arvutamisel tehakse 32 AES ja 33 SHA-256 räsifunktsiooni väljakutset, siis on ajaline võit plokkide kasutamisel märkimisväärne.

Kirjeldatud algoritm on realiseeritud BLT-kiipkaardirakenduses ja asub asub git-i koodihoidlas „*master*" harus *blt-jc-applets/blt-javacard/src/main/java/com/github/mihkelsaar/blt* kataloogis *MerkleTree.java* ja *MerkleTreeBlock.java* klassides. See koodi põhineb Ahto Truu poolt kirjutatud koodil.

5.4.3 Autentimistee arvutamine BLT-serveri komponendis

Kuna BLT-serveri komponendis hoitakse kõiki Merkle'i puu tippe mälus ning iga tipp sisaldab viidet tipu alluvatele ning järglasele, siis autentimistee koostamine on tee läbimine lehest juurtipuni, kus salvestatakse autentimistee jaoks maha vajalikud puu tipud. Vastav kood on realiseeritud *blt-signature/src/main/java/com/github/mihkelsaar/blt/key* kataloogis *LeafNode.java* klassis.

5.5 Piirangud

Eelmistes jaotistes tutvustatud BLT-kiipkaardi algoritmide realisatsioonidel on järgmised piirangud:

- Kiipkaardil realiseeritud Merkle'i puu algoritm eeldab, et puu lehtede arv on alati kahe aste. Piirang lihtsustab puu tippude väljaarvutamist, kuna loodav puu on alati täielik.
- Kuigi BLT-kiipkaardil realiseeritud Merkle'i puu autentimistee koostamise algoritm on efektiivne, siis käesoleva lahenduse puhul peab arvestama, et kõige kulukam operatsioon on Merkle'i puu lehe väljaarvutamine, mille käigus tehakse 33 räsimise ja 32 AES funktsiooni väljakutset.

- Peatükis 4.4 tutvustatud andmemudelit pole otseselt kasutatud BLT-OT võtmete jaoks, kuna need klassid võtaks kiipkaardil liiga palju mälu.

- BLT-kiipkaardirakenduses kasutatakse Merkle'i puu tippude väljaarvutamisel ning privaatvõtmete arvutamisel alati SHA-256 räsialgoritmi.

Eelmistes jaotistes tutvustatud BLT-serveri algoritmide realisatsioonidel on järgmised piirangud:

- BLT-server hoiab Merkle'i puu tippe alati mälus.

- BLT-server ei salvesta andmebaasi ega kõvakettale genereeritud BLT-TB ega registreeritud BLT-OT-N avalike võtmeid.

- Kui BLT-server taaskäivitada, siis genereeritakse alati uued BLT-TB võtmed.

- BLT-serveris kasutatakse puu tippude väljaarvutamisel ning privaatvõtmete arvutamisel alati SHA-256 räsialgoritmi.

BLT-serveri piirangud on tingitud sellest, et käesolev magistritöö keskendub BLT-kiipkaardirakendusele ning sellest tulenevalt on BLT-serveri komponent hoitud võimalikult minimaalsena.

5.6 Kokkuvõte

Käesolevas peatükis anti ülevaade BLT-privaatvõtmete, Merkle'i puu juurtipu ja autentimisteede koostamise algoritmidest BLT-kiipkaardi ja BLT-serveri komponentides. BLT-OT-võtmete genereerimiseks tutvustati algoritmi, mis põhines sümmeetrilistel võtmetel. BLT-avaliku võtmete väljaarvutamiseks kirjeldati Merkle'i puu koostamiseks kasutatavat TREEHASH ja plokkidel põhinevat algoritmi. Nende algoritmide edasiarendusi kasutati autentimisteede koostamiseks.

6 Analüüs

Antud peatükk keskendub realiseeritud BLT-OT-N-signatuuriskeemi analüüsile. Selleks antakse ülevaade BLT-võtmete ja -allkirja suurustest ning võrreldakse neid teise signatuuriskeemidega. Lisaks antakse ülevaade BLT-OT-N-võtmete genereerimise, BLT-OT-N-allkirja koostamise ja BLT-allkirja verifitseerimise jõudlustestide (ingl *performance test*) tulemustest.

6.1 Sissejuhatus

Selleks, et hinnata loodud BLT-signatuuriskeemi efektiivsust, võrreldakse BLT-signatuuriskeemi võtmete ja allkirjade suurust XMSS, XMSS^{MT}, SPHINCS, SPHINCS^{+}, RSA ja ECDSA signatuuriskeemidega. Teine pool peatükist keskendub BLT-OT-N-võtmete genereerimise, BLT-OT-N-allkirja koostamise ja BLT-allkirja verifitseerimise jõudlustestide tulemustele. Jõudlustestide eesmärk on kontrollida, kas eelmises peatükis realiseeritud kiipkaardikomponent vastab peatükis 3.5 kirjeldatud mittefunktsionaalsetele nõuetele. Peatükis ei vaadelda BLT-serveri komponendi jõudlust, kuna antud töö fookus on BLT-kiipkaardil.

Ülejäänud peatüki struktuur on järgmine. Jaotis 6.2 analüüsib BLT-allkirja ja võtmete suurust. Jaotises 6.3 antakse ülevaade BLT-kiipkaardi jõudlustestide keskkonnast, tööriistadest ja tulemustest. Jaotis 6.4 keskendub arutelule ning jaotis 6.5 sisaldab peatüki kokkuvõtet.

6.2 Võtmete ja allkirja suuruse analüüs

Käesolevas jaotises antakse ülevaade BLT-võtmete ja -allkirja suurustest ning võrreldakse neid RSA, ECDSA, XMSS, XMSS^{MT}, SPHINCS ja SPHINCS^{+} signatuuriskeemidega. Jaotis 6.2.1 keskendub võtmete ja jaotis 6.2.2 allkirja suuruse analüüsile.

6.2.1 Võtmete suuruse analüüs

BLT-OT-võtmed on tuletatavad ühest AES salajasest võtmest, mille suurus on 32 baiti, ning 32-baidisest seemnest. Lisaks on AES krüpteerimisfunktsiooni algväärtustamisel vaja hoida ühte algväärtusvektorit suurusega 16 baiti (AES plokkšifri ühe ploki suurus). Seega on BLT-privaatvõtme suuruseks 32*2+16=80 baiti. BLT-OT-N avalik võti on genereeritud Merkle'i puu juurräsi, mis SHA-256 räsifunktsiooni kasutades on 32 baiti.

Tabelis 12 on välja toodud BLT-OT-N, XMSS, XMSS^{MT}, SPHINCS-256, SPHINCS^{+}, RSA ja ECDSA siganatuurisekeemide võtmete suurused. Tabeli esimene veerg sisaldab signatuuriskeemi nime ning parameetreid, teine veerg privaatvõtme suurust baitides ning kolmas veerg avaliku võtme suurust baitides.

Tabel 12. Signatuuriskeemide võtmete suurused.

Signatuuriskeem	**Privaatvõti**	**Avalik võti**
BLT-OT-N	80 baiti	32 baiti
XMSS (n=32, w=16, len=67, h=10 [54])	132 baiti	64 baiti
XMSS^{MT} (n=32, w=16, len=67, h=20, d=4 [54])	132 baiti	64 baiti
SPHINCS-256 [48]	1 088 baiti	1 056 baiti
SPHINCS^{+} (n=192, h=51, d=17, b=7, k=45, w=16 [48])	96 baiti	48 baiti
RSA (3072-bitine võti [3])	1 728 baiti	384 baiti
ECDSA (P-256 [55])	96 baiti	64 baiti

Tabelist 12 on näha, et BLT-OT-N-signatuuriskeemi võtmed on teiste signatuuriskeemidega võrreldes väiksemad. RSA 3072-bitisest privaatvõtmest on BLT-OT-N-privaatvõti peaaegu viis korda väiksem ning avalik võti umbes 12 korda väiksem. ECDSA-signatuuriskeemiga võrreldes on vahe aga väiksem – BLT-OT-N-privaatvõti on umbes 1,2 ning avalik võti kaks korda väiksem. Räsifunktsioonidel põhinevate signatuuriskeemidega võrreldes on samuti BLT-OT-N-võtmed väiksemad. Näiteks XMSS ja XMSS^{MT} signatuuriskeemidega võrreldes on BLT-OT-N-privaatvõti 1,65 ja avalik võti üle kahe korra väiksem.

Kuigi BLT-OT-N-võtmed on teiste signatuuriskeemidega võrreldes väiksemad, siis tuleb arvestada, et BLT-OT-N-võtmete kasutamiseks tuleb ehitada Merkle'i puu vastavalt eelmises peatükis kirjeldatud TREEHASH algoritmile. Merkle'i puu ehitamine ning autentimistee koostamiseks vajalikke tippude hoidmine nõuab aga lisaressurssi. Kuna Merkle'i puu tippude räsiväärtustest ei saa allkirjastamisvõtit tuletada, siis võib neid hoida kiipkaardil EEPROM mälus.

6.2.2 Allkirja suuruse analüüs

Eelmises peatükis realiseeritud BLT-signatuuriskeem koosneb kolmest osast: BLT-OT-N-allkirjast, BLT-TB-allkirjast ning KSI ajatemplist. Kui BLT-OT-N-allkirja koostamisel kasutati puud kõrgusega *H*, siis BLT-OT-N-allkiri sisaldab *H-1* autentimistee elementi Lisaks sisaldab BLT-OT-N-allkiri 32 32-baidist BLT-OT-privaatvõtmest tulenevat väärtust. Seega on BLT-OT-N komponendi suuruseks 4096 BLT-OT-võtme korral 32*32+12*32=1408 baiti. BLT-TB allkirja suurus oleneb samuti Merkle'i puu kõrgusest. Lisaks sisaldab BLT-TB allkiri allkirjastamise aega (4 baiti) ning allkirjastamiseks kasutatud võtit (32 baiti).

Tabel 13 sisaldab BLT, XMSS, XMSS^{MT}, SPHINCS, SPHINCS^{+}, RSA ja ECDSA siganatuurisekeemide allkirja suurusi, kus esimeses veerus on signatuuriskeemi nimi koos parameetritega ning teises veerus allkirja suurus baitides.

Tabel 13. Signatuuriskeemide allkirja suurused.

Signatuuriskeem	**Allkirja suurus**
BLT-OT-N (4096 võtit) + BLT-TB (2 592 000 võtit) + KSI ajatempel	4 575 baiti
XMSS (n=32, w=16, len=67, h=10 [54])	2 500 baiti
XMSS^{MT} (n=32, w=16, len=67, h=20, d=4 [54])	8 930 baiti
SPHINCS-256 [48]	41 000 baiti
SPHINCS^{+} (n=192, h=51, d=17, b=7, k=45, w=16 [48])	30 696 baiti
RSA (3072-bitine võti [3])	384 baiti
ECDSA (P-256 [55])	64 baiti

BLT-signatuuriskeemi puhul kasutatakse allkirja salvestamiseks TLV

(*Type-Length-Value*) vormingut [56], mis suurendab BLT allkirja suurust. Allkirja suuruse testides on kasutatud BLT-TB-allkirja juures võtit, mis koosneb 2592000 allkirjastamisvõtmest (iga sekundi kohta üks allkirjastamisvõti 30 päeva jooksul). Tabelist on näha, et BLT-allkiri on koos ajatempliga tunduvalt suurem kui RSA ja ECDSA allkirjad. Võrreldes räsifunktsioonidel põhinevate allkirjadega on XMSS allkiri umbes 1.8 korda väiksem. Seevastu on $XMSS^{MT}$, SPHINCS, ja $SPHINCS^{+}$ allkirjad BLT-allkirjast suuremad. Siinjuures peab arvestama, et BLT-allkiri sisaldab ajatemplit, mille suuruseks on umbes 1 KB.

6.3 Jõudlustestid

Antud jaotis annab ülevaate realiseeritud BLT-OT-N-signatuuriskeemi võtmete genereerimise, allkirja koostamise ning allkirja verifitseerimise jõudlustestide tulemustest. Jaotises 6.3.1 antakse ülevaade jõudlustestide keskkonnast ja tööriistadest. Jaotis keskendub 6.3.2 AES ja SHA-256 funktsioonide, jaotis 6.3.3 BLT-OT-võtmete, jaotis 6.3.4 BLT-OT-N-võtmete genereerimise, jaotis 6.3.5 allkirjastamise ning jaotis 6.3.6 verifitseerimise jõudlustestidele.

6.3.1 Jõudlustestide keskkond ja tööriistad

BLT-kiipkaardirakenduse testimiseks kasutatakse *Javacos A22* kiipkaarti (*ATR=3B9F958131FE9F00664653051000FF71DF0000000000EC*), millel on 150KB EEPROM ja 2,7 KB RAM mälu. Antud kiipkaart toetab JavaCard 2.2.2 platvormi jaoks kirjutatud rakendusi. Kiipkaardile on paigaldatud kaks *Java Card* rakendust. Nendest *blt-jc-applet.cap* rakendus sisaldab BLT-OT-N-signatuuriskeemi realisatsiooni ning rakendust kasutatakse BLT-OT-N-signatuuriskeemi jõudlustestimiseks. Selleks, et mõõta BLT-OT-võtmete ja BLT-OT-võtmete genereerimisel kasutatavate AES ning SHA-256 räsifunktsiooni kiirust, kasutatakse kiipkaardi rakendust nimega *primitive-operations.cap*.

Magistritöö jaoks loodi kiipkaardi jõudlustestide käivitamiseks uus raamistik. Loodud raamistik põhineb *JUnit 4.2* raamistikul [57] ja koosneb kiipkaardirakenduse paigaldajast, mis põhineb *GlobalPlatformPro* teegil [58], ja mõõdab kiipkaardil ühe käsu täitmiseks kulunud aega.

Kuna BLT-allkirja verifitseerimine toimub BLT-kliendirakenduses, siis kasutatakse allkirja verifitseerimise jõudlustestide jaoks *JMH* nimelist tööriista [59]. *JMH* tööriist on mõeldud Java programmeerimiskeeles kirjutatud programmide jõudluse testimiseks. Antud testide käivitamiseks kasutatakse MacbookPro arvutit, kus on 32GB DDR4 mälu ja 2,6 GHz kuuetuumaline Intel Core protsessor.

6.3.2 AES ja SHA-256 jõudlustestid

Kuna BLT-OT-võtmete genereerimine põhineb AES krüpteerimisel ja SHA-256 räsifunktsioonil, siis esmalt mõõdame kiipkaardil AES krüpteerimise ja SHA-256 räsifunktsioonide arvutamise kiirust. Selleks kasutatakse eraldi kiipkaardirakendust, kus on realiseeritud need funktsioonid. Testide sisendiks on naturaalarv, mis kirjeldab, mitu korda funktsiooni (AES või SHA-256) käivitatakse. Kuna SHA-256 räsifunktsiooni kiirus sõltub räsitavate andmete suurusest, siis mõõdavad jõudlustestid räsifunktsiooni kiirust 32- ja 1024-baidise sisendi puhul. 32 baidist sisendit kasutatakse ühe BLT-OT-salajase võtme ning 1024 baidist (32 salajast võtit) sisendit BLT-OT-avaliku võtme väljaarvutamiseks. Kirjeldatud jõudlustestide tulemused on välja toodud tabelis 14. Tabelis on välja toodud ühe funktsiooni (AES või SHA-256) keskmine arvutuse aeg. Esimeses reas on välja toodud käivitatud operatsioonide arv, teises reas AES tulemused 32-baidise sisendi (kaks AES plokki) korral, kolmandas SHA-256 räsifunktsiooni tulemused 32 baidise sisendi korral ning viimases reas SHA-256 räsifunktsiooni tulemused 1024 baidise sisendi puhul.

Tabel 14. AES ja SHA-256 räsifunktsiooni jõudlustestide tulemused.

	128	**256**	**512**	**1024**	**2048**	**4096**
AES (2 plokki)	3,99 ms	3,14 ms	2,71 ms	2,49 ms	2,38 ms	2,38 ms
SHA-256 (32 baiti)	7,58 ms	6,76 ms	6,32 ms	6,11 ms	6,10 ms	6,11 ms
SHA-256 (1024 baiti)	107,61 ms	107,65 ms	107,66 ms	107,67 ms	107,58 ms	107,57 ms

Tulemustest on näha, et väiksemate funktsiooni käivitamiste arvu puhul on krüpteerimiseks ja räsimiseks kuluv aeg pikem kui suuremate käivitamiste puhul. See on tingitud sellest, et jõudlustestide aja sisse arvutatakse ka kiipkaardi APDU käsu

saatmise ja vastuse saamise aeg. Lisaks on näha, et AES funktsiooni väljakutse on võrreldes SHA-256 räsifunktsiooniga, mis kasutab 32 baidist sisendit, umbes kolm korda kiirem.

6.3.3 BLT-OT-võtmete genereerimise jõudlustestid

BLT-OT-võtmete genereerimise käigus arvutatakse 32 salajast võtit ning nende võtmete põhjal arvutatakse BLT-OT-avalik võti. Ühe BLT-OT võtme arvutamise käigus käivitataks 32 AES funktsiooni, 32 SHA-256 räsifunktsiooni 32-baidise sisendiga ning üks SHA-256 funktsioon 1024-baidise sisendiga. Tabelis 15 on välja toodud BLT-OT-võtmete jõudlustestide tulemused, kus esimene rida sisaldab genereeritavate võtmete arvu, teine rida nende võtmete genereerimiseks kulunud aega millisekundites ning kolmas rida sisaldab aega millisekundites, mis kulus keskmiselt ühe võtme genereerimiseks.

Tabel 15. BLT-OT-võtmete genereerimise jõudlustestide tulemused.

	128 võtit	**256 võtit**	**512 võtit**	**1024 võtit**	**2048 võtit**	**4096 võtit**
Kokku	47 703 ms	95 461 ms	190 749 ms	381 322 ms	762 557 ms	1 525 142 ms
Keskmine	372,68 ms	372,89 ms	372,56 ms	372,38 ms	372,34 ms	372,35 ms

Tabelist on näha, et 4096 BLT-OT-võtme genereerimiseks kulub umbes 25,4 minutit ning keskmiselt kulub ühe BLT-OT-võtme genereerimiseks 372 millisekundit. Võtmete arvu kahekordistumisel kahekordistub ka võtmete genereerimiseks kuluv aeg.

6.3.4 BLT-OT-N-võtmete genereerimise jõudlustestid

BLT-OT-N-võtme väljaarvutamise käigus genereeritakse BLT-OT-võtmed ning koostatakse antud võtmete põhjal Merkle'i puu kasutades TREEHASH või plokkidel põhinevat Merkle'i puu koostamise algoritmi. BLT-OT-N-võtmete genereerimise tulemused on välja toodud tabelis 16. Tabeli esimeses reas on BLT-OT-võtmete (Merkle'i puu lehtede) arv, teises reas TREEHASH ning kolmandas reas plokkidel põhineva algoritmi testide tulemused.

Tabel 16. BLT-OT-N-võtme genereerimise jõudlustestide tulemused.

	128 lehte	**256 lehte**	**512 lehte**	**1024 lehte**	**2048 lehte**	**4096 lehte**
TREEHASH	52 030 ms	105 068 ms	205 311 ms	420 350 ms	833 070 ms	1 655 859 ms
Plokk	51 754 ms	102 878 ms	204 724 ms	407 390 ms	815 851 ms	1 631 995 ms

Tabelist on näha, et BLT-OT-võtmete kahekordistumise korral suureneb ka BLT-OT-N-võtme väljaarvutamine umbes kaks korda. Võrreldes tulemusi eelmise jaotises välja toodud BLT-OT-võtmete genereerimisega on näha, et Merkle'i puu koostamisele kulub TREEHASH algoritmi kasutades 4096 lehe puhul 1655859-1525142=130717 millisekundit (umbes 2 minutit). Kokku kulub 4096 lehe puhul BLT-OT-N-võtmete väljaarvutamiseks umbes 27,6 minutit. TREEHASH ja plokkidel põhineva algoritmi kiiruste erinevused on väga väikesed.

6.3.5 BLT-OT-N-allkirja koostamise jõudlustestid

BLT-OT-N-allkirja koostamine koosneb kolmest operatsioonis: allkirjastamispäringu koostamine BLT-serverile, BLT-serveri allkirja verifitseerimine ning BLT-OT-võtme autentimistee arvutamine. BLT-OT-N-allkirja koostamise jõudlustestide tulemused TREEHASH algoritmi kasutades on välja toodud tabelis 17 ning plokkidel põhinevat algoritmi kasutades tabelis 18, tabelite esimestes ridades on BLT-OT-võtmete arv Merkle'i puus. Iga allkirjastamise operatsiooni kohta on välja toodud selleks kulunud minimaalne, maksimaalne ning keskmine aeg millisekundites. Testi käigus kasutati ära kõik allkirjastamiseks genereeritud võtmed.

Tabelist 17 on näha, et TREEHASH algoritmi kasutades on allkirja koostamisel kõige kulukam operatsioon autentimistee arvutamine. Autentimistee arvutamine sõltub BLT-OT-võtmete arvust. BLT-OT võtmete arvu kahekordistumise korral kasvab autentimistee arvutamine keskmiselt 0,4 sekundit. Autentimistee arvutamisel on kõige kulukam operatsioon BLT-võtme koostamine, mis koosneb 32-st AES ja 33 SHA-256 räsioperatsioonist.

Tabel 17. BLT-OT-N-allkirja koostamise jõudlustestide tulemused kasutades TREEHASH autentimistee algoritmi.

		128 lehte	**256 lehte**	**512 lehte**	**1024 lehte**	**2048 lehte**	**4096 lehte**
Allkirjastamis-päringu koostamine	min.	675 ms	663 ms	668 ms	666 ms	658 ms	658 ms
	max.	698 ms	684 ms	695 ms	707 ms	697 ms	707 ms
	kesk.	680 ms	680 ms	680 ms	680 ms	680 ms	680 ms
BLT-serveri allkirja verifitseerimine	min.	2 749 ms	2 737 ms	2 743 ms	2 729 ms	2 732 ms	2 719 ms
	max.	2 791 ms	2 790 ms	2 791 ms	2 797 ms	2 802 ms	2 805 ms
	kesk.	2 772 ms	2 768 ms	2 766 ms	2 762 ms	2 770 ms	2 758 ms
Autentimistee arvutamine	min.	898 ms	1 348 ms	1 360 ms	1 375 ms	1 386 ms	1 399 ms
	max.	4 993 ms	6 188 ms	6 960 ms	7 740 ms	8 703 ms	9 476 ms
	kesk.	2 821 ms	3 661 ms	4 079 ms	4 482 ms	4 875 ms	5 283 ms
Kokku	min.	4 322 ms	4 748 ms	4 771 ms	4 770 ms	4 776 ms	4 776 ms
	max.	8 482 ms	9 662 ms	10 446 ms	11 244 ms	12 202 ms	12 988 ms
	kesk.	6 273 ms	7 109 ms	7 525 ms	7 924 ms	8 325 ms	8 721 ms

Tabelist 18 on näha, et plokkidel põhinevat autentimistee algoritmi kasutades kulub autentimistee arvutamiseks alla kahe sekundi. See on tingitud sellest, et plokkidel põhinevas algoritmis genereeritakse iga autentimistee arvutamisel üks Merkle'i puu leht ning enamus autentimistee arvutamiseks kasutavatest puu tippudest hoitakse kiipkaardi mälus. Samuti peab arvestama, et plokkidel põhinev algoritm võtab TREEHASH algoritmist tunduvalt rohkem mälu.

Tabel 18. BLT-OT-N-allkirja koostamise jõudlustestide tulemused kasutades plokkidel põhinevat Merkle'i puu ja autentimistee koostamise algoritmi.

		128 lehte	**256 lehte**	**512 lehte**	**1024 lehte**	**2048 lehte**	**4096 lehte**
Allkirjastamis-päringu koostamine	min.	679 ms	680 ms	675 ms	678 ms	679 ms	678 ms
	max.	682 ms	682 ms	683 ms	682 ms	683 ms	684 ms
	kesk.	680 ms	680 ms	679 ms	679 ms	680 ms	680 ms
BLT-serveri allkirja verifitseerimine	min.	2 738 ms	2 727 ms	2 662 ms	2 731 ms	2 720 ms	2 721 ms
	max.	2 775 ms	2 780 ms	2 791 ms	2 796 ms	2 803 ms	2 792 ms
	kesk.	2 755 ms	2 757 ms	2 764 ms	2 760 ms	2 755 ms	2 759 ms
Autentimistee arvutamine	min.	838 ms	1 279 ms	1 280 ms	1 283 ms	1 287 ms	1 291 ms
	max.	1 345 ms	1 928 ms	1 926 ms	2 167 ms	2 182 ms	2 716 ms
	kesk.	1 201 ms	1 644 ms	1 659 ms	1 662 ms	1 673 ms	1 677 ms
Kokku	min.	4 255 ms	4 686 ms	4 617 ms	4 692 ms	4 686 ms	4 690 ms
	max.	4 802 ms	5 390 ms	5 400 ms	5 645 ms	5 668 ms	6 192 ms
	kesk.	4 636 ms	5 081 ms	5 102 ms	5 101 ms	5 108 ms	5 116 ms

BLT-serveri allkirja verifitseerimine sõltub BLT-TB-allkirja autentimistee elementide arvust. Käesolevates jõudlustestides sisaldab BLT-TB-allkiri 16 autentimistee elementi.

6.3.6 BLT-allkirja verifitseerimise jõudlustestid

BLT, XMSS, XMSS^{MT}, SPHINCS, RSA ja ECDSA-allkirja verifitseerimise jõudlustestide tulemused on välja toodud tabelis 19. Tabeli esimeses veerus on signatuuriskeemi koos parameetritega, ning teises veerus on allkirja verifitseerimiseks kulunud keskmine aeg millisekundites. XMSS, XMSS^{MT}, SPHINCS, RSA ja ECDSA allkirjade verifitseerimise jõudlustestides kasutatid *Bouncy Castle* [60] teeki.

Tabel 19. Allkirja verifitseerimise jõudlustestide tulemused.

Signatuuriskeem	**Keskmine allkirja verifitseerimie aeg**
BLT-OT-N (4096 BLT-OT võtit) + BLT-TB (2592000 võtit) + KSI ajatempel	0,306 ms
XMSS (n=32, w=16, len=67, h=10 [54])	1,793 ms
$XMSS^{MT}$ (n=32, w=16, len=67, h=20, d=4 [54])	7,358 ms
SPHINCS-256 [48]	1,498 ms
RSA (3072-bitine võti [3])	0,275 ms
ECDSA (P-256 [55])	0,123 ms

Tabelis 19 puudub $SPHINCS^{+}$ sigantuuriskeemi tulemused, kuna see algoritm pole *Bouncy Castle* teegis realisteeritud. Verifitseerimise testide juures peab arvestama, et allkirja verifitseerimine sisaldab ka aega, mis kulus allkirja lugemiseks kõvakettalt. Samuti sisaldab BLT-allkiri ajatemplit, mida peaks ülejäänud signatuuriskeemide juures eraldi hoidma ning verifitseerima.

6.4 Arutelu

Kuigi BLT-kiipkaardil realiseeritud BLT-OT-N-signatuuriskeem on suurte võtmete arvu juures aeglane ning seetõttu reaalsete rakenduste puhul raskesti kasutatav, saab lahendust optimeerida.

- BLT võtmete ja allkirja koostamise algoritmi saab muuta efektiivsemaks kasutades BLT-OT-võtme väljaarvutamisel *Winternitzi* signatuuriskeemile [44] sarnast lahendust. Kui me kasutaks *Winternitzi* signatuuriskeemi *w* parameetrina väärtust 2, siis oleks iga BLT-OT-võtme väljaarvutamisel vaja teha 16 AES operatsiooni, 32 SHA-256 räsifunktsiooni (32 baidise sisendiga) ning ühte SHA-256 räsifunktsiooni 512 baidise sisendiga. See lähenemine vähendaks BLT-OT-võtme genereerimiseks kuluvat aega umbes 372,35-16*2,38-32*6,11-107,61/2=84,94 millisekundi võrra. Samuti väheneks BLT-OT-allkirja suurus 16*32=512 baidi võrra [9].

- Kiipkaardil saab realiseeritud TREEHASH algoritmi asemel kasutada optimeeritud autentimistee koostamise algoritme [61], [62].

- Kui kiipkaarditootjad realiseeriks BLT-signatuuriskeemist riistvaralise versiooni, siis suure tõenäosusega oleks allkirja koostamise kiirus võrreldav RSA, ECDSA ja DSA kiirusega.

Samuti peab arvestama, et erinevalt RSA, ECDSA ja SPHINCS algoritmidest saab BLT-OT-N-võtit allkirjastamiseks kasutada piiratud arv kordi. Lisaks peab BLT-signatuuriskeemis hoidma olekut kasutatud võtmetest.

6.5 Kokkuvõte

Käesolev peatükk võrdles BLT-võtmete ja -allkirja suurust teiste enamlevinud signatuuriskeemidega. BLT-OT-N-võtmed on võrreldes teiste signatuuriskeemidega väiksemad, kuid võrreldes RSA ja ECDSA skeemidega on BLT-allkirjad tunduvalt suuremad. Teiste räsifunktsioonidel põhinevate skeemidega võrreldes on BLT-allkirja suurus enamasti väiksem. Ainult teatud XMSS skeemi parameetrite korral on XMSS allkiri väiksem.

Teine osa peakükist keskendus BLT-kiipkaardi jõudlustestidele. Jaotises 6.3 välja toodud tulemuste põhjal on BLT-kiipkaardil realiseeritud BLT-OT-N-signatuuriskeem suurte võtmete arvu juures aeglane ning seetõttu reaalsete rakenduste puhul raskesti kasutatav. Jõudlustestide tulemuste põhjal välja pakutud optimeeringud aga võivad võtmete genereerimiseks ja allkirjastamiseks kuluvat aega tunduvalt vähendada, mis võimaldaks BLT-kiipkaarti tulevikus reaalsetes rakendustes kasutada.

7 Kokkuvõte

Magistritöö käigus realiseeriti kiipkaardi jaoks räsifunktsioonil põhinev BLT-signatuuriskeemi prototüüp, mis on immuunne teada olevatele kvantarvutirünnetele. Loodava süsteemi funktsionaalsete ja mittefunktsionaalsete nõuete kirjeldamiseks kasutati agentorienteeritud modelleerimisest tuttavaid rolli- ja eesmärgimudeleid. Kirjeldatud eesmärkide põhjal loodi UML-skeemid, mis kirjeldasid loodud prototüübi arhitektuuri. Komponendiskeemi abil kirjeldati loodava süsteemi komponente ja nende vahelisi seoseid ning liideseid. Klassiskeemide abil kirjeldati BLT-võtmete ja -allkirja andmestruktuure ning järgnevusskeemide abil anti detailne ülevaade komponentide vahel liikuvatest andmetest ning päringutest. Kiipkaardil kasutatavate võtmete genereerimiseks tutvustati algoritmi, mis põhineb sümmeetrilisel krüptograafial. BLT-avaliku võtmete väljaarvutamiseks kirjeldati Merkle'i puu koostamiseks kasutatavaid TREEHASH ja plokkidel põhinevaid algoritme. Nende algoritmide edasiarendusi kasutati autentimisteede koostamiseks.

Võrreldes teiste räsipõhiste signatuuriskeemidega on loodud BLT-signatuuriskeemi võtmed tunduvalt väiksemad. Erinevalt teistest skeemidest sisaldab BLT-allkiri ka allkirjastamishetke aega. Kuigi ajatempel suurendab allkirja suurust, siis on allkiri teiste räsipõhiste signatuuriskeemidega võrreldes enamasti väiksem. Samuti on allkirja verifitseerimine võrreldes teiste räsifunktsioonidel põhinevate allkirjaskeemidega võrreldes vähemalt viis korda kiirem.

Kuigi loodud prototüübis võtab allkirjastamine tunduvalt rohkem aega ning allkirjad on tunduvalt suuremad kui RSA, ECDSA ja DSA algoritmide korral, siis saab realiseeritud skeemis vähendada allkirja suurust ning võtmete genereerimiseks kuluvat aega kasutades võtmete koostamisel Winternitzi sigantuuriskeemist tuttavat lahendust. Samuti saaks kiirendada allkirjastamist, kasutades autentimisteede koostamisel kiiremaid algoritme. Kui kiipkaarditootjad realiseeriks BLT-signatuuriskeemist riistvaralise versiooni, siis suure tõenäosusega oleks see arvestatav alternatiiv RSA, ECDSA ja DSA sigantuuriskeemidele.

Kasutatud kirjandus

[1] European Commission, „Regulation no 910/2014 of the European Parliament and of the Council of 23 July 2014 on electronic identification and trust services for electronic transactions in the internal market and repealing directive 1999/93/EC (eIDAS regulation)“, *Off. J. Eur. Union*, nr L 257:73-114, 2014.

[2] European Commission, „Study on the use of Electronic Identification (eID) for the European Citizens' Initiative“, 2017.

[3] R. L. Rivest, A. Shamir ja L. Adleman, „A method for obtaining digital signatures and public-key cryptosystems“, *Commun. ACM*, kd 21, nr 2, lk 120–126, 1978.

[4] T. El Gamal, „A public key cryptosystem and a signature scheme based on discrete logarithms“, *IEEE Trans. Inf. Theory*, kd 31, nr 4, lk 469–472, 1985.

[5] D. Johnson, A. Menezes ja S. Vanstone, „The elliptic curve digital signature algorithm (ECDSA)“, *Int. J. Inf. Secur.*, kd 1, nr 1, lk 36–63, 2001.

[6] P. W. Shor, „Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer“, *SIAM Rev.*, kd 41, nr 2, lk 303–332, 1999.

[7] J. Buchmann, E. Dahmen ja A. Hülsing, „XMSS - A practical forward secure signature scheme based on minimal security assumptions“, *Int. Work. Post-Quantum Cryptogr.*, lk 117–129, 2011.

[8] D. J. Bernstein *et al.*, „SPHINCS: practical stateless hash-based signatures“, *Eurocrypt*, lk 368–397, 2015.

[9] A. Buldas, D. Firsov, R. Laanoja, H. Lakk ja A. Truu, „A new approach to constructing digital signature schemes.“, *Advances in Information and Computer Security*, 2019, lk 363–373.

[10] A. Buldas, R. Laanoja ja A. Truu, „A server-assisted hash-based signature scheme“, *NordSec 2017*, 2017, kd 10674, lk 3–17.

[11] C. Adams, P. Cain, D. Pinkas ja R. Zuccherato, „Internet X. 509 public key infrastructure time-stamp protocol (TSP)“, *RFC*, kd 3161, 2001.

[12] D. Cooper *et al.*, „Internet X. 509 public key infrastructure certificate and certificate revocation list (CRL) profile.“, *RFC*, kd 5280, lk 1–151, 2008.

[13] S. Santesson, M. Myers, R. Ankney, A. Malpani, S. Galperin, ja C. Adams, „X. 509 Internet public key infrastructure online certificate status protocol - OCSP“, *RFC*, kd 6960, 2013.

[14] L. Sterling ja K. Taveter, *The art of agent-oriented modeling*. MIT Press, 2009.

[15] M. Fowler, *UML distilled: a brief guide to the standard object modeling language*. Addison-Wesley Professional, 2004.

[16] Z. Chen, *Java card technology for smart cards: architecture and programmer's guide*. Addison-Wesley Professional, 2000.

[17] K. Beck ja C. Andres, *Extreme programming explained: embrace change*. Addison-Wesley, 2004.

[18] R. Agarwa ja D. Umphress, „Extreme programming for a single person team“, *Proceedings of the 46th Annual Southeast Regional Conference on XX*, 2008, lk 82–87.

[19] K. Beck, *Test driven development: by example*. Addison-Wesley Professional, 2003.

[20] Github, „GitHub“. https://github.com (Viimati külastatud 04.05.2020).

[21] GitHub, „GitHub Actions“. https://github.com/features/actions (Viimati külastatud 04.05.2020).

[22] D. J. Anderson, *Kanban: successful evolutionary change for your technology business*. Blue Hole Press, 2010.

[23] GitHub, „Project management“. https://github.com/features/project-management/ (Viimati külastatud 04.05.2020).

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

[25] P. Gallagher ja A. Director, „Secure hash standard (shs)“, *FIPS PUB*, kd 180, lk 183, 1995.

[26] M. J. Dworkin, „SHA-3 standard: permutation-based hash and extendable-output functions“, *NIST FIPS*, kd 202, 2015.

[27] R. C. Merkle, „Secrecy, authentication, and public key systems“, *Ph. D. Thesis, Stanford Univ.*, 1979.

[28] R. C. Merkle, „Protocols for public key cryptosystems“, *1980 IEEE Symposium on Security and Privacy*, 1980, lk 122–134.

[29] M. Bosamia ja D. Patel, „Current Trends and Future Implementation Possibilities of the Merkel Tree“, *Int. J. Comput. Sci. Eng.*, kd 6, nr 8, lk 294–301, 2018, doi: 10.26438/ijcse/v6i8.294301.

[30] ISO/IEC, „Time-stamping services — Part 1: Framework“, *ISO/IEC 18014-1*, 2008.

[31] ANSI ASC, „Trusted time stamp management and security“. ANS X9.95, 2016.

[32] S. Haber ja W. S. Stornetta, *How to time-stamp a digital document*. Springer, 1991.

[33] J. Benaloh ja M. de Mare, „Efficient broadcast time-stamping“, 1991.

[34] D. Bayer, S. Haber ja W. S. Stornetta, „Improving the efficiency and reliability of digital time-stamping“, *Sequences II: Methods in Communication, Security and Computer Science*, Springer, 1992, lk 329–334.

[35] A. Buldas ja M. Saarepera, „On provably secure time-stamping schemes“, *Advances in Cryptology - ASIACRYPT 2004*, 2004, lk 500–514.

[36] A. Buldas, R. Laanoja, P. Laud ja A. Truu, „Bounded pre-image awareness and the security of hash-tree keyless signatures“, *ProvSec 2014, Proceedings*, 2014, kd 8782 LNCS, lk 130–145.

[37] A. Buldas ja R. Laanoja, „Security proofs for hash tree time-stamping using hash functions with small output size“, *ACISP 2013, Proceedings*, 2013, kd 7959 LNCS, lk 235–250.

[38] S. Haber ja W. S. Stornetta, „Secure names for bit-strings“, *Proceedings of the 4th ACM Conference on Computer and Communications Security*, 1997, lk 28–35.

[39] A. Buldas, A. Kroonmaa ja R. Laanoja, „Keyless signatures' infrastructure: How to build global distributed hash-trees“, *Nord. 2013, Proc.*, kd 8208 LNCS, lk 313–320, 2013.

[40] L. Lamport, „Constructing digital signatures from a one-way function“, 1979.

[41] Cybernetica AS, „Krüptograafiliste algoritmide elutsükli uuring“, 2015.

[42] C. Dods, N. P. Smart ja M. Stam, „Hash based digital signature schemes“, *Cryptography and Coding, Proceedings*, 2005, kd 3796 LNCS, lk 96–115.

[43] R. C. Merkle, „A certified digital signature“, *CRYPTO '89, Proceedings*, 1989, kd 435 LNCS, lk 218–238.

[44] J. A. Buchmann, E. Dahmen, S. Ereth, A. Hülsing ja M. Rückert, „On the security of the Winternitz one-time signature scheme“, *IJACT*, kd 3, nr 1, lk 84–96, 2013.

[45] A. Hülsing, „W-OTS+ - Shorter signatures for hash-based signature schemes“, *AFRICACRYPT 2013, Proceedings*, 2013, kd 7918 LNCS, lk 173–188.

[46] A. Hülsing, L. Rausch ja J. A. Buchmann, „Optimal parameters for XMSS MT“, *CD-ARES 2013, Proceedings*, 2013, kd 8128 LCNS, lk 194–208.

[47] O. Goldreich, *Foundations of cryptography: volume 2, basic applications*. Cambridge university press, 2009.

[48] D. J. Bernstein, R. Niederhagen, A. Hülsing, J. Rijneveld, S. Kölbl ja P. Schwabe, „The SPHINCS+ signature framework“, *ACM CCS*, 2019.

[49] Oracle, „Java Card 3 Platform Development Kit User Guide, Classic Edition“.

[50] ISO/IEC, „Identification cards - integrated circuit(s), card with contact“, kd ISO/IEC 78, 1998.

[51] Sun Microsystem, „Java Card Platform 3.0.1 Connected Edition“, *Desember 2018*, lk 5, 2009.

[52] H. Lakk, „BLT signature formats and protocol“, 2019.

[53] G. Becker, „Merkle signature schemes, merkle trees and their cryptanalysis“, *Ruhr-University Bochum, Tech. Rep*, 2008.

[54] A. Hülsing, D. Butin, S.-L. Gazdag, J. Rijneveld ja A. Mohaisen, „XMSS: eXtended Merkle Signature Scheme“, *RFC*, kd 8391, lk 1–74, 2018.

[55] M. Adalier ja others, „Efficient and secure elliptic curve cryptography implementation of Curve P-256“, *Workshop on Elliptic Curve Cryptography Standards*, 2015, kd 66.

[56] Guardtime OÜ, „TLV Element“. https://guardtime.github.io/ksi-java-sdk/com/guardtime/ksi/tlv/TLVElement.html (Viimati külastatud 04.05.2020).

[57] P. Tahchiev, F. Leme, V. Massol ja G. Gregory, *JUnit in action*. Manning Publications Co., 2010.

[58] M. Paljak, „Global Platform Pro“. https://github.com/martinpaljak/GlobalPlatformPro (vaadatud mai 01, 2020).

[59] A. Shipilev, S. Kuksenko, A. Astrand, S. Friberg ja H. Loef, „OpenJDK: jmh“, *http//openjdk.java.net/projects/code-tools/jmh (Viimati külastatud 04.05.2020).*

[60] Legion of the Bouncy Castle Inc, „Bouncy Castle“. https://www.bouncycastle.org/java.html (Viimati külastatud 04.05.2020).

[61] M. Szydlo, „Merkle tree traversal in log space and time“, *EUROCRYPT 2004, Proceedings*, 2004, kd 3027 LNCS, lk 541–554.

[62] B. Schoenmakers, „Explicit optimal binary pebbling for one-way hash chain reversal“, *FC 2016, Revised Selected Papers*, 2017, kd 9603 LNCS, lk 299–320.

Lisa 1

Lähtekood: https://github.com/mihkelsaar/blt/

TREEHASH algoritmi lähtekood asub „*treehash*” harus.

Plokkidel põhinev autentimistee koostamise algoritmi lähtekood asub „*master*” harus.