TALLINNA POLÜTEHNIKUM

IT ja telekommunikatsioon erialaosakond

Keijo Kapp

Kõrge terviklusega andmeid talletava andmebaasilahenduse prototüüp

Lõputöö

ARVUTID JA ARVUTIVÕ RGUD

PA-12A

Juhendajad: Deivis Treier, Tarmo Teder

Tallinn 2015

AUTORIDEKLARATSIOON

Deklareerin, et käesolev lõputöö, mis on minu iseseisva töö tulemus, on esitatud Tallinna Polütehnikumi lõputunnistuse taotlemiseks Arvutid ja arvutivõrgud erialal. Lõputöö alusel ei ole varem eriala lõputunnistust taotletud.

Autor Keijo Kapp ….....

(allkiri ja kuupäev)

Töö vastab kehtivatele nõuetele.

Juhendaja Deivis Treier .....

(allkiri ja kuupäev)

Juhendaja Tarmo Teder .....

(allkiri ja kuupäev)

Sisukord

Lühendid ja terminid

JSON – *JavaScript Object Notation* – standard andmete hoidmiseks ja edastamiseks tekstikujul

JSONB – *JSON Binary* – binaarne struktueeritud JSON andmetüüp

ISKE – Peamiselt Eesti avaliku sektori infovaradel kasutatav kolmeastmeline etalonturbe süsteem

MVC – *Model-View-Controller* – konventsioon, milles eristaatkse andmeid (*model*), vaadet/kasutajaliidest (*view*), kasutajaliidese loogikat (*controller*) ja enamasti ka äriloogikat (*service*)

HSM – *Hardware Security Module* – riistvaraline krüptoseade, sarnane ID-kaardile

OCSP – *Online Certificate Status Protocol* – allkirja kehtivukinnituse protokoll

TSA – *Time Stamp Authority* – digiallkirjastamisel kasutatav protokolli

TSL – *Trust Status List* – usaldatavate osapoolte nimekiri

PKCS – *Public Key Cryptography Standards* – avaliku võtmega krüpteerimise standardite kogum, mille hulka kuuluvad näiteks PKCS11 (krüptoseadmega suhtlemine) ja PKCS12 (võtmete konteinerfail)

Sissejuhatus

Lõputöö eesmärk on luua prototüüplahendus kahe ISKE kõrge tervikluse tagamise turvameetme rakendamiseks. Lõputöö tulemus on suunatud Siseministeeriumi Infotehnoloogia- ja Arenduskeskuse (SMIT) poolt arendatavatele infosüsteemidele.

ISKE on Eestis kasutatav infosüstemide turvameetmete süsteem. Seda kasutatakse peamiselt riigi ka kohalike omavalitsuste infosüsteemide turvalisuse tagamiseks. Selliste infosüsteemide alla lähevad ka kõik SMIT-i loodavad infosüsteemid.

ISKE järgi tähendab andmete terviklus andmete õigsuse/täielikkuse/ajakohasuse tagatust, autentsust ning volitamatute muutuste puudumist. (RIA, 2014) Sellist olukorda on formaalselt võimalik tagada ka ilma meetmeid rakendamata, näiteks käsitledes infosüsteemiga kokkupuutuvaid osapooli “usaldatavatena”, kuid tehniline võimekus selle situatsiooni tagamiseks on kindlam.

Prototüüp (ingl.k. *Proof Of Concept*) eesmärk on näidata kontseptsiooni reaalset toimimist. Kogu lahenduse kontseptsioon koosneb 4 osast:

- Andmebaasi kirjete krüptoaheldamine
- Andmebaasis olevate andmete digiallkirjastamine
- Andmebaasi kirjete revisioonide hoidmine
- JSONB andmetüübi kasutamine andmete hoidmiseks

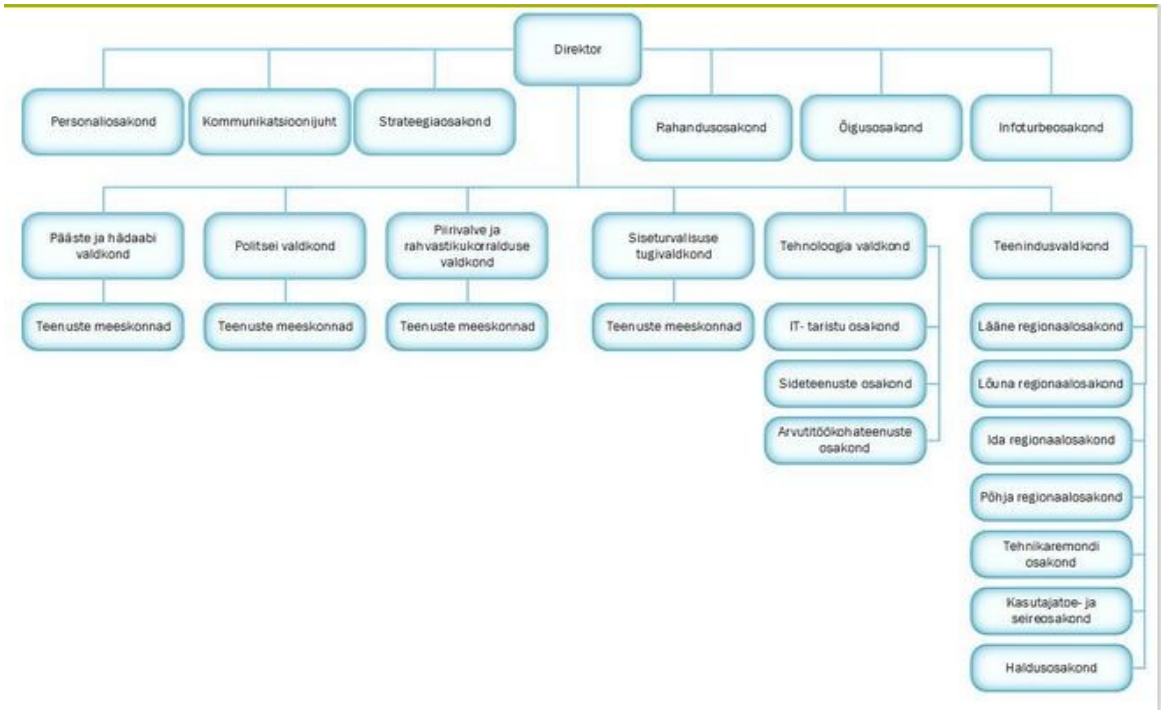
Lõputöö käsitleb ainult kolme esimest kontseptsiooni osa. Neljas on siiski kontseptsiooni töötamiseks vajalik, sest see aitab kergemini käsitleda kõrge terviklusega andmeid.

1. Ettevõttest

Siseministeeriumi Infotehnoloogia- ja Arenduskeskus on Eesti Siseministeeriumi haldusalas IT teenuseid pakkuv ettevõte. Ettevõte arendab ja haldab muuhulgas Siseministeeriumi haldusalas olevaid infosüsteeme ning asutuste töökohti.

Ettevõtte kontorid asuvad Tallinnas, Tartus, Pärnus ja Jõhvis. Töötajaid on ettevõttel umbes 250. Peamised kliendid on Politsei- ja Piirivalveamet, Päästeamet, Häirekeskus ja Siseministeerium. Lisaks haldab ettevõte ka mitmete riigiasutuste poolt kasutatavat operatiivraadiosidevõrku. Tehniline partnerlus ja tootetugi on Maanteeametiga.

Ettevõtte struktuur on kirjeldatud järgneval skeemil. Lõputöö autor osales piirivalve ja rahvastikukorralduse valdkonna töös.



Joonis 1: Ettevõtte struktuur

2. Lahenduse eesmärk

Ettevõttel on vaja luua tehniline võimekus kõrge terviklusega andmete hoidmiseks rakendades vastavaid tarkvaralisi turvameetmeid. Lahendus peaks olema võimalikult abstraktne, et seda saaks kasutada erinevate infosüsteemide loomiseks.

Lahenduse loomise hõlmab erinevate võimaluste uurimist ja olulisimate realiseerimist. Kuna lahendus ei saa olla suunatud kindlale infosüsteemile, ei saa ka rääkida konkreetsetest nõuetest vaid ainult kontseptsioonist.

2.1. Andmebaasi kirjete krüptoaheldamine

Eesmärk on vastavalt ISKE meetmele **HT.10 Andmebaasi kannete krüptoaheldamine** (RIA, 2014) luua funktsionaalsus, mis seob andmebaasitabeli piires kirjed selliselt, et igasugused, nii tahtlikud kui rikke tõttu tekkinud, manipulatitsioonid andmetega oleksid leitavad. Andmed seotakse ridade väärtusest tuletatava räsiväärtuse kaudu nii, et see räsi oleks seotud ka eelmise rea räsi väärtusega.

Kuigi ISKE meede seda välja ei too, ei ole lihtsa aheldamisega võimalik tagada andmebaasis oleva viimase kirje muutumatust, sest sellest ei sõltu mõne teise kirje räsi. Samuti on võimalik märkamatult muuta viimaseid kirjeid, kui muudetavast kirjest alates ahel üle arvutada. Selle vältimiseks peab aheldamist teostav funktsionaalsus võimaldama viimase kirje räsi turvalist salvestamist. Kontseptsiooni järgi salvestatakse räsi mõnes teises süsteemis, kuhu infosüsteemi meeskonnaliikmetel ligipääsu ei ole.

Meetme eesmärgi saavutamisel ei piisa ainult krüptoaheldamisest - seda ahelat peab ka perioodiliselt kontrollima. Kontrollmehhanismi loomine ja rakendamine ei ole lõputöö skoobis.

2.2. Andmete digiallkirjastamine

Digiallkirjastamine toimub vastavalt ISKE **HT.34 Digiallkirja kasutamine** meetmele. Meede lubab allkirjastamiseks kasutada ainult mehhanisme, mis vastavad Eesti digitaalallkirja seadusele. (RIA, 2014) Lisaks terviklikkuse tagamisele annab see andmetele vähemalt Eesti piires ka juriidilise tõendusväärtuse.

Digiallkirjastamisele kuuluvad kõik kõrge terviklikkuse nõudega andmed, mida potensiaalselt väljastatakse infosüsteemist. See tagab ka vähem turvalistes keskkondades andmete terviklikkuse ja autentsuse. Infosüsteemi sees on andmete terviklikkus tagatud krüptoaheldamise meetmega ning kõigi andmete allkirjastamine on nii rahalistel kui tehnilistel põhjustel ebaotstarbekas.

Kontseptsiooni järgi on nendeks andmeteks dokumendid (nt. avaldused, taotlused) koos nende juurde käivate lisadega (nt. pilt, allkiri). Masintöödeldavad dokumendid on JSON vormingus ning vajadusel käib nende juurde lisana ka tavainimesele loetavas HTML vormingus dokument.

2.2.1. JSON vorming

ECMA-404 ehk *Javascript Object Notation* (JSON) on viimastel aastatel väga populaarseks saanud vorming masintöödeldavate andmete hoidmiseks ja edastamiseks tekstikujul. Seda kasutatakse laialdaselt veebis, andmebaasides, konfiguratisioonides ja paljudel teistel eesmärkidel.

Vorming baseerub väikesel osal *ECMA-262 3rd Edition* standardist, mida tuntakse programmeerimiskeelena *Javascript*. (ECMA International, 2013) JSON-i käsitletakse paljudel juhtudel XML-i kaasaegse anternatiivina, olles palju mahuefektiivsem ja enamikel juhtudel ka inimesele loetavam. Lisaks on JSON-il erinevalt XML-ist tugi erinevate andmetüüpidele (massiiv, number, string, objekt jne.) analoogselt *Javascript*'iga.

JSON andmekomplekte nimetatakse ka JSON dokumentideks. Andmed on dokumendis hierarhilise puuna, kusjuures originaalstandardi (RFC4627) järgi võib puu juurelement olla ainult objekt või massiiv. (Crockford, 2006) ECMA-404 standardile vastavad implementatsioonid võimaldavad juurelemendina ka teisi andmetüüpe kasutada, kuid mitte kõik implementatsioonid ei toeta seda, mistõttu on mõistlik sellest hoiduda.

Juurelemendina objekti kasutamine välistab ka teatud XSS (ingl.k. c*ross-site scripting*) turvaprobleemi, sest selliselt vormindatud dokument ei ole terviklik *Javascript*'i programm, kuid see-eest kõigi parserite poolt aksepteeritav JSON dokument.

Oluliseks puuduseks autori arvates on see, et JSON ei toeta kommentaare. Kuigi algselt oli *Javascript*'iga analoogselt kommentaaride tugi olemas, eemaldatati see, sest väidetavalt hakkasid paljud arendajad kasutama kommentaare semantilise informatsiooni hoidmiseks JSON dokumendis, mis üldiselt halb muster.

JSON dokument võib välja näha näiteks selline:

```
{
      "Image": {
            "Width":  800,
            "Height": 600,
            "Title":  "View from 15th Floor",
            "Thumbnail": {
                  "Url": "http://www.example.com/image/481989943",
                  "Height": 125,
                  "Width":  "100"
            },
            "IDs": [116, 943, 234, 38793]
      }
}
```

(Crockford, 2006)

Reavahed, tühikud jm. mittenähtavad tähemärgid ei ole väljaspool stringe olulised.

2.2.2. JSONB andmetüüp

PostgreSQL 9.4 tõi uuendusena kaasa JSONB andmetüübi, millega saab hoida JSON dokumenti andmebaasis strukteeritult binaarsel kujul. See võimaldab käsitleda üksikuid JSON dokumendi välju tabeli väljadena. Näiteks saab JSONB andmetüübiga üksikuid JSON välju lugeda, kirjutada ja indekseerida. Selline dünaamilisus võib oluliselt muuta andmete hoidmise põhimõtteid. Näiteks kaob vajadus paljude relatsioonide (võõrvõtmete) kasutamise järeldusele, sest andmed saab kirjutada otse JSON dokumendi sisse.

2.3. Kirjete muudatuste ajaloo hoidmine

Kõrge terviklus eeldab ka seda, et andmeid reaalselt ei kustutata ega muudeta, vaid märgitakse mitteaktiivseteks. Kustutamise või muutmise teeb võimatuks ka krüptoaheldamise meede, mis sisuliselt ongi selleks, et selliseid muutmisi või kustutamisi tuvastada. Need manipulatsioonid tuleb teostada rakenduses loogiliselt. Näiteks peab kirje kustutamise asemel lisama uue kirje, mis märgib eelmise (loogiliselt kustutatud) kirje mittekehtivaks.

Vaatamata probleemi esinemisele paljudes kaasaegsetes infosüsteemides ei ole üldist olemasolevat lahendust sellele. Üks enam-kasutatavamaid lahendusi on mitteaktiivsete kirjete kandmine eraldi tabelisse, kuid krüptoaheldamise meetme rakendamisel on see keeruline, sest ahel peab olema katkematu ning ahela lülide hoidmine erinevates tabelites teeb ahela kontrollimise keerukaks.

Kontseptsiooni järgi peab terve tabel olema konsistentne tervik ning sisaldama kogu kirjete ajalugu. Lisaks peab olema võimalik kiiresti leida iga üksiku kirje muutused ajas – nii enne kui peale vastavat kirjet.

On oluline märkida, et ebaaktiivsed kirjed ei pruugi muutuda ebaolulisteks või vähemolulisteks. Ka konkreetsele kirje revisioonile võib olla viiteid. Näiteks kui isikule vastab kirje ning isikuandmeid muutes luuakse isikust uus kirje, siis varasemate isikuandmetega loodud dokumendid viitavad ikkagi nendele eelmistele kirjetele/revisioonidele.

3. Lahenduse analüüs

3.1. ISKE

ISKE on Riigi Infosüsteemi Ameti poolt väljatöötatud kolmeastmeline etalonturbe süsteem. Selle aluseks on võetud Saksamaal kasutatav *IT Baseline Protection Manual*. ISKE rakendamise eesmärk on tagada infosüsteemides töödeldavate andmete piisava tasemega turvalisus. Peamiselt on süsteem loodud riigi ja kohalike omavalitsuste andmekogude turvalisuse tagamiseks. (RIA, 2014)

Infosüsteemide vastavuse nõue ISKE turbeastmele määratakse vastava infosüsteemi kohta käiva määrusega. Näiteks Isikut Tõendavate Dokumentidega Andmekogu (ITDAK) turbeklassi määrab "Isikut tõendavate dokumentide andmekogu pidamise põhimäärus".

Meetmeid võib ka mitte rakendada, kui eesmärk suudetakse teisiti tagada. Sellisel juhul peab looma vastavad dokumendid meetme mittetäitmiseks.

Lõputöö eesmärk on luua kahe konkreetse ISKE turvameetme rakendamise prototüübid, mis on eelpool kirjeldatud. Neid meetmeid pole sellisel kujul ettevõttes varem rakendatud.

3.2. Kasutatavad tehnoloogiad

3.2.1 Java

Java on multiplatvormne programmeerimiskeel ja selle käivitusmootor (virtuaalmasin). Javat kasutatakse laialdaselt serveri-, töölaua-, mobiili- jm. rakenduste loomiseks. Java peamiseks eelisteks on paljude platvormide tugi, mis võimaldab peaaegu ilma

platvormispetsiifilise koodi kirjutamist ja ülekompileerimist jooksutada programmi mitmetel platvormidel. Lisaks Java'l väga suur kausutajaskond, mis tagab kerge lahenduse leidmise enamikele probleemidele.

Probleemidena on Java'l võrreldes teiste kõrgtaseme keeltega suhteliselt keeruline sõltuvuste (ingl.k. *dependency*) haldus, mitu täiesti erinevat ehitussüsteemi (*Maven*, *Ant*/*Ivy*) ning liiga palju erinevaid koodikonventsioone, millest halvemal juhul ei peeta kinni. Olulisimaks probleemiks peab autor asünkroonse I/O teostamise keerulisust, millest tuleneb massiivne erinevate lõimede kasutamine, mis omakorda võib tekitada raskesti avastatavaid vigu piisavalt kogenematul arendajal.

3.2.2 Groovy

Groovy on Java platvormil ja süntaksil baseeruv programmeerimiskeel. Groovy loob võimaluse kasutada nn. *duck typing*'ut, mis tähendab, et andmetüüpide eraldi väljatoomine koodis ei ole vajalik. See ja teised Groovy võimalused võimaldavad kirjutada dünaamilist ning kohati ilusamat ja paremini loetavat koodi kui Java.

Teisalt on Groovy harjumatu varasemalt Java arendajale, sest süntaks on küll sarnane, aga ka oluliselt erinev. Võib olla keeruline lugeda koodi, kus ei ole muutujate väärtuste tüübid eraldi välja toodud. Lisaks on Groovy olulselt aeglasem võrreldes Java'ga, sest andmetüüpide määramine ja sellest tulenevalt käivitatava koodi valikute otsustamine toimub programmi käitamise ajal.

3.2.3 Grails

Grails on Groovy programmeerimiskeelel baseeruv MVC raamistik veebirakenduste loomisks. See baseerub Java *Spring Framework* raamistikul. Erinevalt Spring Framework raamistikust on Grails üles ehitatud vastavalt *Conventsion Over Configuration* (COC) arendusmeetodile. See tähendab, et näiteks ei pea arendaja kontrolleri (ing.k. *controller*) koodi märgistama, et tegemist on kontrolleriga ning määrama, millisele tegevusele (*action* või *route*) see vastab. Piisab, kui kontroller on vastavalt konventsioonile nimetatud ja selle kood asub vastavas failis. See võib oluliselt kiirendada produktiivsust, sest arendaja peab füüsiliselt vähem koodi kirjutama. Samas eeldab see kohati väga mahukate konventsioonide teadmist.

Kohati võib Grails'i käsitleda ka omaetteplatvormiga, sest see sisaldab spetsiifilist funktsionaalsust (näiteks teadete logimine, testimine ja dokumenteerimine), ehitus- ja sõltuvuste haldamise süsteemi ja konventsioone teekide arendamiseks ja rakenduse konfigureerimiseks.

3.2.4 PostgreSQL

PostgreSQL on laialdaselt kasutatav relatsiooniline andmebaasisüsteem. Erinevalt konkureerivatest andmebaasisüstemidest (MySQL, MSSQL) on PostgreSQL'il oluline eesmärk järgida täpselt SQL standardit. PostgreSQL on üks lihtsamini liidestatavaid andmebaasisüsteeme, omades samal ajal spetsiifiilisi võimalusi nagu näiteks JSON ja JSONB andmetüübid.

3.2.5 PKCS11 ja HSM

PKCS11 on *Public Key Cryptography Standards* (*PKCS*) perekonna standard. See käsitleb krüptoseadmega suhtlemist tarkvaratasandil. Sellised seadmed on näiteks ID-kaarid, pangakaardid, HSM'id või ka virtuaalsed seadmed (näiteks PKCS11SPY või SoftHSM). Üldiselt väljastab iga krüptoseadme tootja oma krüptoseadmele PKCS11 standardile vastava teegi, mis laetaks dünaamiliselt kasutaja rakendusse.

PKCS11 defineerib muuhulgas, kuidas kasutada ja manipuleerida krüptoseadmes olevaid võtmeid ja sertifikaate. PKCS11 teekide konfigureerimine toimub tavaliselt rakendusväliselt näiteks eraldi konfiguratsioonifaili või keskkonnamuutujate kaudu.

4. Töö teostus

4.1. Krüptoaheldamine

Krüptoahealdamise praktilise osa eesmärk on luua olukord, kus ükski osapool ei saaks ainuisikuliselt märkamatult muuta ega kustutada andmebaasis olevaid andmeid. Osapoolteks on kontseptsiooni järgi vähemalt kaks administratiivset tsooni: ühes tsoonis asub rakendus ja andmebaas koos vastavate administraatoritega ja teiste meeskonnaliikmetega ning teiseks tsooniks on mõni teine osakond, kontseptsiooni järgi näiteks infoturbe osakond. Lõputöö ei käsitle olukorda, kus andmete muutmine peab olema välistatud, välja arvatud vastutava isiku, näiteks organisatsiooni juhi, nõusolekul.

Autor on käsitlenud mitut erinevat lähenemist. Mõlemal juhul peab vastav funktsionaalsus laias laastus tegema igal kirje sisestamisel 4 sammu. Need sammud tuleb teostada vahetult enne kirje sisestamist ning atomaarselt.

1. Viimase kirje räsi laadimine vahemälust, andmebaasist või muust allikast
2. Sellest räsist ja sisestatava rea väärtusest uue räsi arvutamine
3. Uue räsi toimetamine teise administratiivsesse tsooni koos toetavate andmetega (nt. aeg, rea identifikaator, eelmine räsi)
4. Sisestatava rea muutmine kirjutades saadud räsi vastavale väljale.

4.1.1. Syslog

Räsi saatmiseks serverisse otsustati kasutada *syslog* nimelist protokolli. Tegemist on protokolliga, mida kasutatakse keskseks ja standardseks logimiseks. Syslog'i kasutamine

on tavaliselt väga lihtne ning on dokumenteeritud “The GNU C Library” dokumentatsioonis. (Free Software Foundation, 2015) Tihti käsitletaksee *syslog* protokolli ebaturvalise ning ebastabiilsena. Võib esineda logiteadete kadumist ning halvimal juhul on võimalik teateid manipuleerida. Siiski on võimalik neid probleeme vältida kasutades vastavaid meetmeid, näiteks IP/TCP ja *Transport Level Security* (TSL).

4.1.2. Andmebaasi päästikud

Andmebaasi päästik (ingl.k. *trigger*) on funktsioon, mis käivitatakse andmebaasimootoris kasutaja poolt defineeritud sündmusel või sündmustel. Andmebaaside termonoloogias kutsutakse kasutaja poolt defineeritud funktsioone terminiga *Stored Procedure*. Neid funktsioone saab kasutada ka teistel eesmärkidel, nt. kasutaja poolt defineeritud andmetüübid, matemaatilised funktsioonid, indekseerimine jne.

PostgreSQL päästikute kirjutamine ja kasutamine on suhteliselt hästi dokumenteeritud. Päästikuid saab kirjutada erinevates programmeerimiskeeltes: lihtsamatel juhtudel pgSQL, keerulistemal PL/pgSQL, PL/Python või PL/v8 (Javascript) ning kõige keerulistemal juhtudel C, C++ või muu keel, mida C-keeles kirjutatud andmebaasimootor saab linkida. (The PostgreSQL Global Development Group, 2015; Hitoshi Harada, 2015)

Kuna vajalik funktsionaalsus on suhteliselt keeruline ning PL (ingl.k. *Procedural Language*) eelliidestega keelte funktsionaalsus on piiratud (peamiselt turvalisuse ja stabiilsuse tagamiseks), otsustas autor kasutada C keelt. Autoril on varasemalt suhteliselt pikk kogemus C ja C++ programmeerimiskeelte kasutamisel.

4.1.3. Päästiku realiseerimine C-keeles

Krüptoaheldamise päästiku lähtekood koosneb viiest lähtekoodi failist, millele lisandub *Makefile* ja *README.md*. *Makefile* on *GNU Make* nimelise ehitustööriista poolt kasutatav fail, *README.md* on *Markdown* formaadis väike dokumentatsioonifail.

Päästiku kompileerimiseks saab kasutada *GNU Make* tööriist. Kompileerimiseks peab lähtekoodi juurkaustas käivitama käsu

```
make
```

Programm *make* loeb vastavas kaustas olevat faili *Makefile* ning käivitab selles failis

oleva kompileerimise käsu õi käsud. Kompileerimise väljundiks objektifail *hashchain-trigger.so*, mis ekspordib andmebaasiserverile kaks funktsiooni:

- *hashchainTrigger* – funktsioon, mis räsiaheldab sisestatavad kirjed
- *logHashTrigger* – funktsioon, mis saada räsiväärtuse *syslog*'i serverile

Funktsioonide ja päästikute registeerimiseks peab vastavas andmebaasis käivitama SQL käsud. Päästikute loomise käsud käivitatakse iga krüptoaheldatava tabeli jaoks eraldi.

```
-- Funktsioonide registeerimine

-- krüptoaheldamise funktsioon
CREATE OR REPLACE FUNCTION "hashchainTrigger"()-- funktsiooni nimeks on 'hashchainTrigger'
        RETURNS trigger                        -- funktsiooni kasutatakse päästiku jaoks
        AS '/path/to/hashchain-trigger.so'     -- kompileerimise väljundfail
        LANGUAGE C;                            -- funktsioon on kirjutatud C keeles

-- Räsi logisse saatmise funktsioon
CREATE OR REPLACE FUNCTION "loghashTrigger"()   -- funktsiooni nimeks on 'logHashTrigger'
        RETURNS trigger                         -- funktsiooni kasutatakse päästiku jaoks
        AS '/path/to/hashchain-trigger.so'      -- kompileerimise väljundfail
        LANGUAGE C;                             -- funktsioon on kirjutatud C keeles

-- Päästikute registreerimine

-- Päästik, mis krüptoaheldab kirjed
CREATE TRIGGER hashchainTrigger                    -- päästiku nimi
        BEFORE INSERT                              -- enne sisestamist
        ON tabeli_nimi                             -- tabeli nimi
        FOR EACH ROW                               -- iga rea jaoks eraldi
        EXECUTE PROCEDURE "hashchainTrigger"();    -- funktsiooni nimi

-- Päästik, mis kirjed logisse saadab
CREATE TRIGGER logHashTrigger                      -- päästiku nimi
        BEFORE INSERT                              -- enne sisestamist
        ON tabeli_nimi                             -- tabeli nimi
        FOR EACH ROW                               -- iga rea jaoks eraldi
        EXECUTE PROCEDURE "logHashTrigger"();      -- funktsiooni nimi
```

Olulisimad päästiku lähtekoodi osad on toodud lisas. (vt. *Lisa 1*, *Lisa 2*, *Lisa 3*)

4.1.4. PL/v8 ja PL/pgSQL kasutamine päästikutes

Arusaadavatel põhjustel väldib ettevõte madalataseme keelte nagu C kasutamist. Põhjuseks tuuakse näiteks see, et selliselt kirjutatud kood ei ole tihtipeale platvormist sõltumatu (nt. erinevad andmetüüpide suurused).

Osa päästiku funktsionaalsuse implementeerimiseks on võimalik kasutada PL/pgSQL või PL/v8 keelt. Selle osa funktsionaalsus piirdub räside arvutamise ja kirje muutmisega. Räsi saatmiseks välisele osapoolele kasutatatakse siiski C keeles loodud funktsionaalsusest.

PL/pgSQL puhul näeks krüptoaheldamise funktsiooni ja vastava päästiku loomise kood välja selline:

```
CREATE or REPLACE FUNCTION tabeli_nimiHashchainTrigger() RETURNS trigger AS $row_stamp$
        DECLARE
                hash char(128);
                previousHash char(128);
        BEGIN
                -- Eelmise räsi lugemine andmebaasist muutujasse prefiousHash
                SELECT hashchain INTO previousHash FROM hashchain_test ORDER BY id DESC
                        LIMIT 1;
                -- Uue räsi arvutamine muutujasse hash
                SELECT encode(digest(CAST((NEW.*, previousHash) AS text),'sha256'), 'hex')
                        INTO hash;
                -- Rea räsiväärtuse välja muutmine vastavalt
                NEW.hashchain := hash;
                -- Tagastatakse uus (muudetud) rida
                RETURN NEW;
        END;
        $row_stamp$ LANGUAGE plpgsql;

CREATE TRIGGER hashchain                                        -- päästiku nimi on “hashchain”
        BEFORE INSERT                                           -- enne sisestamist
        ON tabeli_nimi                                          -- tabeli nimi
        FOR EACH ROW                                            -- iga rea jaoks eraldi
        EXECUTE PROCEDURE tabeli_nimiHashchainTrigger();        -- käivitatav funktsioon
```

Kogu see kood tuleb käivitada iga tabeli jaoks eraldi, seega tekib iga tabeli jaoks eraldi funktsioon. PL/v8-t kasutades saaks sama tulemuse ainult ühe funktsiooniga:

```
CREATE OR REPLACE FUNCTION hashchainTrigger() RETURNS trigger AS $row_stamp$
        const tableName = '"' + TG_TABLE_SCHEMA + '"."' + TG_TABLE_NAME + '"';
        var q = plv8.execute('SELECT hashchain FROM ' + tableName
              + ' ORDER BY id DESC LIMIT 1');
        NEW.hashchain = q[0].hashchain; // eelmise rea hashchain väärtus
        var row = JSON.stringify(NEW);
        NEW.hashchain = plv8.find_function("sha256")(row);
        return NEW
$row_stamp$ LANGUAGE "plv8";

-- . . . päästiku loomine iga tabeli jaoks eraldi
```

Kuid kuna see vajab eraldi PostgreSQL laiendust (*plpgv8*), siis selle kasutamisest loobuti.

On oluline, et kõik 3 meetodit (C, PL/pgSQL ja PL/v8) ei anna täpselt sama tulemuse, näiteks PL/v8 võtab räsiväärtuse reast loodud JSON tekstist, PL/pgSQL ühendab väljade väärtused komaga ning C keeles kirjtutatud päästik lisab lihtsalt kõikide väljade väärtused üksteise järgi. Seetõttu tuleb tabeli piires teha kohe nende meetodite vahel õige valik.

Peab veenduma, et räsi saatmise päästik käivituks igal juhul peale krüptoaheldamise defineeritud päästikut. Vastasel juhul saadetakse logisse vale väärtus.

4.1.5. Testimine

Lahenduste testimiseks tegi autor oma tööjaama andmebaasiserverisse test-andmebaasi ning sinna tabeli *hashchain_test*. Tabeli loomise käsk võis olla näiteks selline:

```
CREATE TABLE hashchain_test (
        id BIGSERIAL,
        data TEXT,
        hashchain VARCHAR(255)
)
```

Välja *hashchain* andmetüüp peaks olema *VARCHAR*, *CHAR* või muu tekstitüüp ning pikkusega vähemalt niipalju, et mahutada räsialgoritmi väljundit tekstikujul 16nd-koodi teisendatuna. *SHA256* väljund on 256 bitti ehk 32 baiti, mis 16nd-koodi teisendatuna on 64 tähemärki.

Autor lisas tabelisse paarkümmend suvaliste väärtustega rida ning pisteliselt käsitsi kontrollis päästiku arvutatud väärtuste õigsust.

Näiteks võib võtta sellise lihtsa tabeli (*Joonis 2*)

id bigint	data text	hashchain character varying(255)
1	asdasd	130fec4fef848a9b74631d367a187b7bb6807bbb65a1e287218e455
2	asdasd	60dfa4a3862882155ac7c931c2a96a54a92fb050ca1a53eba9e4658
3	asdasd	85bce30db9dd2ca694d0ffdd8acd7cf5e409ef314e864d81bc959fc
4	asdasd	edeaf3b588b0a243ca95823c41873bb6466bbf322ac1197670d596b
5	asdasd	53f9b3bec73cd3ebd738deb3acef735fb8c136db70c933b94a6b5ba
6	asdasd	2154a4432131dacf77b0dcb839c4fc047207cbf081fd349cc8efd5e
7	foo	c56061ec02d231cf2b08764e1cdac2e36bad58102a284012a51b883
8	bar	c8e65ab5bd727469a891dfe16520455d9e387dffae5e494fbeed7ca
9	baz	b8192958e0a30c885cf1b84328d44cca844a0dd106b2598f9675d8a
10	Lorem ipsum dolor sit amet, consectetur adipiscing elit	7e98d542bd95cee738e59ad3ea28fa0d2e4579575e46ea2c76efe9f

Joonis 2: Krüptoaheldamise test-tabel

Kaheksanda rea puhul võtab C keeles kirjutatud päästik 7. rea räsi ja liidab selle 8. rea teiste väärtustega. Tulemuseks on string

“c56061ec02d231cf2b08764e1cdac2e36bad58102a284012a51b88380a24a98b8bar”.
Sellest SHA256 räsiväärtus on

“c8e65ab5bd727469a891dfe16520455d9e387dffae5e494fbeed7ca15bc665e7”.

Seega päästik töötab õigesti. Niimoodi testis autor mitmeid ridu erinevate meetodidega loodud räsidega.

Räside logimist testis autor süsteemi logi jälgimisega. Kuna kõik räsid kanti testimise käigus vastavalt ootustele süsteemi logifaili, siis saab ka seda töö osa töötava lahendusena käsitleda. Edasine logimise konfigureerimine, näiteks kauglogimine, ei ole lõputöö skoobis.

4.2. Andmete digiallkirjastamine

Tegemist on lõputöö ajaliselt kõige mahukama osaga. Mahukaks tegi selle osa see, et digiallkirjastamiseks on mitu tehnilist võimalust ning nende kõigi kasutamisel oli vähemalt lõputöö tegemise ajal omajagu probleeme.Näiteks Java teek *digidoc4j* algselt ei töötanud vastupidiselt ootustele kasutataval Groovy/Grails platvormil ning sellel puudus kogu vajaliku funktsionaalsuse (nt. PKCS11) tugi. Sellel oli ka muid puudusi: *beta* arendusstaatus, fataalse vea esinemine vähemalt ühe testitud virtuaalse krüptoseadmega (SoftHSM) ning autori arvates keeruline konfigureerimine.

Nende probleemide vältimiseks kaaluti kasutada eraldi C/C++ adapterprogrammi, mis suudaks suhelda krüptoseadmega läbi *libdigidocpp* teegi, kuid eesmärgiks võeti siiski *digidoc4j* kasutamine.

Ka testimine oli keeruline, sest teek suhtleb erinevate väliste osapooltega, nt. OCSP server, *Trust Status List* (TSL) repositoorium, *Time Stamp Authority* (TSA) server jne. Kogu protses tuleb konfigureerida vastavalt sellele, milliseid võtmeid kasutatakse.

4.2.1. Konteinerid, signatuurid ja signeerimismeetodid

Tehniliselt ei allkirjastata üksikuid dokumente/faile vaid konteinereid. Failide allkirjastamine tähendab nende lisamist konteinerisse ja konteineri allkirjastamist. Eestis kasutatakse kahte tüüpi konteinereid: DDOC ja BDOC, millest viimane jaguneb veel kaheks - *BDOC-TM/ASIC-E LT-TM* ja *BDOC-TS/ASIC-E LT*. (Sertifitseerimiskeskus AS, 2015) Kuna DDOC konteinerite kasutamine pole soovitatav 2015. aastast ning need peaksid kasutuselt kaduma, siis ei tegelenud autor selle formaadi uurimisega. Sertifitseerimiskeskus AS, mis tegeleb digiallkirjastamise infrastruktuuri haldamise ja arendamisega, soovitab siseriiklikke dokumente allkirjastada BDOC-TM (*Time Mark*) allkirjaga, mistõttu ei ole ka BDOC-TS (*Time Stamp*) konteinerid skoobis.

BDOC konteinerid on ZIP-formaadis failid, mis sisaldavad kindla ülesehitusega failide

struktuuri. See võimaldab konteinerit avada põhimõtteliselt iga implementatsiooniga, mis suudab ZIP faile lugada. Kõik allkirjastatavad failid asuvad konteineri “juurkaustas”. Kõik signatuurid asuvad konteineri META-INF kaustas failinimega *signatureX.xml*, kus *X* on signatuuri indeks (järjekorranumber alates 0-st). Lisaks on juurkaustas fail *mimetype*, kus on kirjas konteineri tüüp MIME tüübina, näiteks *application/vnd.etsi.asic-e+zip*. Kaustas *META-INF* on ka *manifest.xml*, mis sisaldab infot muuhulgas kõikide failide ja nende tüüpide kohta.

Signatuurid on *XML Advanced Electronic Signature* (*XAdES*) formaadis. Need sisaldavad andmeid allkirjastaja, tema sertifikaatide, allkirjastamise aja ja allkirja kehtivuskinnitusserveri (OCSP serveri) kohta.

Allkirjastamise protsess on laias laastus selline:

1. Konteineri loomine ja failide lisamine konteinerisse
2. Konteineris olevatest andmetest räsi arvutamine
3. PIN koodiga krüptoseadme sessiooni avamine (PKCS11 kasutamisel) või salajase võtme faili dekrüpteerimine (PKCS12 puhul).
4. Räsi saatmine krüptoseadmesse või krüpteerimine failist loetud võtmega
5. Krüptoseadmega allkirjastamisel “allkirja” lugemine seadmest
6. Allkirjale kehtivuskinnituse võtmine (OCSP päring või muu sarnane tehnoloogia)
7. Allkirja vormindamine ja lisamine konteinerile.

4.2.2 *libdigidocpp*

C-keeles kirjtuatud *libdigidoc*'i tugi lõppeb 2015. aasta detsempris. Selle asemele on loodud C++ keeles teek *libdigidocpp*. (Sertifitseerimiskeskus AS, 2015) See on täielik DigiDoc'i implementatsioon, pakkudes toetust kõikidele kasutatavatele DigiDoc'i signatuuride- ja konteineriformaatidele ning erinevatele signeerimismehhanismidele (PKCS11, PKCS12, OpenSSL).

4.2.3. *digidoc4j*

Java teegi *jdigidoc* arendamine lõpetati 2015 aasta juunis ning tugi on samuti lõppemas. (Sertifitseerimiskeskus AS, 2015) Selle asemel on uuem Java teek *digidoc4j*, mis on küll *beta* arendusjärgus, kuid aktiivselt arenduses.

Kuna rakenduste arendamiseks kasutatav Groovy baseerub Java programmeerimiskeelel, on teoreetiliselt võimalik kasutada Java teeke Groovy rakendustes. Siiski ei olnud algselt võimalik teeki muutmata kujul kasutada Groovy/Grails platvormil. Probleemiks olid teatud klassimuutujate läbivalt vigased deklaratsioonid teegis. Teek kasutas mõnede objektide kloonimiseks serialiseerimist, kuid Groovy loodud baitkood ei suutnud oma dünaamilise ehituse tõttu deserialiseerimisel leida serialiseeritud objektide klasse. Isegi, kui see probleem ei oleks osutunud fataalseks, oli tegemist jõudluse piirajaga, sest neid objekte ei oleks üldse pidanud serialiseerima ning kogu see protsess ei ole ühekordne tegevus. Autor tegi vastava paranduse *digidoc4j* ametlikku koodibaasi. (Kapp, 2015)

Veel osutus oluliseks probleemiks ametliku PKSC11 toe puudumine, mis oli eelduseks, et rakendus saaks turvaliselt allkirjastamist läbi viia. Teegil on ainult PKSC12 tugi, mis aga ei ole nii turvaline, sest sel juhul hoitakse allkirjastaja salajast võtit failis, millest on seda võimalik PIN koodi teades lugeda. PKCS11 puhul hoitakse võtit krüptoseadmes, kust kontseptsiooni järgi ei ole seda kuidagi võimalik lugeda. Teegil on siiski mitteametlik ning kohati ebastabiilne PKCS11 tugi. Nimelt kasutab *digidoc4j* alusena modifitseeritud *sd-dss* teeki, mis toetab PKCS11 standardit analoogselt PKCS12-ga. Kuid teadmata põhjusel ei tööta see lahendus, kui arendaja kasutab räsialgoritmi SHA256 (mis on vaikimisi kasutatav algoritm), kusjuures teiste algoritmidega lahendus töötab. Hoolimata vea põhjuse leidmisele kulutatud pikast ajast ei suutnud autor tuvastada vea põhjust ja olemust. Autori arvates on viga parandatud *sd-dss* originaalkoodibaasis, kuid kuna *digidoc4j* ei toeta ametlikult PKCS11 protokolli, ei ole seda veaparandust modifitseeritud versioonis tehtud.

On ka väiksemaid probleeme *digidoc4j* teegiga. Need puudutavad peamiselt koodi kvaliteeti. Kohati ei peeta kinni üldtunnustatud muutujate, meetodite ja klasside nimekonventsioonist. Näiteks klass *PKCS11SignatureToken* peaks konventsiooni järgi olema nimega *Pkcs11SignatureToken* ning meetod *addTSLSource* nimega *addTslSource*, samas mitte igal poole ei eksita selle reegli suhtes. Lisaks on osa äriloogikat kirjutatud

Configuration klassi, mis teeb rakendusespetsiifilise konfiguratsioonimehhanismi loomise väga keeruliseks. Grails'il on äriklassi rakenduste jaoks hea konfiguratsioonimehhanism, kuid selle probleemi tõttu ei saa seda kasutada *digidoc4j* teegi konfigureerimiseks.

Ka seda probleemi püüdis autor parandada, luues (ingl.k. *fork*) ametlikust repositooriumist isikliku repositooriumi, kus konfiguratsiooniklassist on eraldi välja toodud abstraktne (ingl.k. *abstract*) klass mis sisaldab konfiguratsiooniklassi meetodeid, mille implementatsioon ei ole ülejäänud teegi seisukohalt oluline. Selle "täiustuse" raames tuli välja veel üks viga teegis, mille tingis Java *Reflection* tehnoloogia kasutamine, mille tõttu ei leidnud teek klassi *CustomContainer* implementatsioonidest konstruktorit, mis aksepteeriks parameetrina klassi *Configuration* (autori repositooriumis *AbstractConfiguration*) alamklassi instantsi. Ka selle vea aitas autor parandada, tehes vastava muudatuse *digidoc4j* repositooriumisse. (Kapp, 2015)

4.2.4. Digiallkirjastamise testimine

Testimine toimus ettevõtte poolt antud Groovy/Grails projektiga. Kuna *digidoc4j* kasutab pakihaldussüsteemi *Ivy*, pidi kõigepealt autor teisendama *digidoc4j* repositooriumis oleva *ivy.xml* failis defineeritud andmed *Maven*'i formaati *digidoc4j-versioon.pom* faili. Seejärel pidi autor looma projektisisese *Maven* repositooriumi ning käsitsi installima *digidoc4j* failid sinna. Lisaks pidi projekti lisama *sd-dss* teegi failid. Need lisati projekti juurkaustas oleva *lib* kausta alla. Grails käsitleb selles kaustas olevaid faile projektis kasutatavate teekidena.

PKCS11-ga allkirjastamisel kasutas autor krüptoseadmena oma isiklikikku ID-kaarti ning testimiseks virtuaalse seadmena SoftHSM/SoftHSMv2 teeke. SoftHSM-iga sai testida ainult test-võitmeid nagu näiteks neid, mis olid *digidoc4j* repositooriumis PKCS12 formaadis.

Kuna *digidoc4j* ei sisaldanud endas PKCS11-ga allkirjastamise klassi, pidi autor looma selle. Klass sisaldab kahte meetodit, ühega küsib *digidoc4j* sertifikaati, millega allkirjastada (näiteks võib ühes krüptoseadmes olla mitme isiku võtmed ja sertifikaadid) ning teisega toimub allkirjastamine, st. krüptoseadmesse räsi saatmine ja sealt allkirja lugemine. Selle klassi kood on välja toodud lisas. (vt. *Lisa 4*)

Groovy skriptiga allkirjastamise näide, kasutades seda klassi, on toodud lisas. (vt. *Lisa 5*)

4.3. Kirjete revisioonide hoidmine

See meede ei ole otseselt ISKE meetmete hulgas välja toodud, kuid on eelduseks krüptoaheldamise meetme rakendamisele ja kõrge tervikluse tagamisele. Võimalusi selle eesmärgi saavutamiseks on mitmeid ning neil kõigil on oma nõrkused.

Levinuim lahendus, mida ka ettevõte ja autor on kasutanud, on mitteaktiivsete kirjete kandmine teise tabelisse, nn. ajalootabelisse. Selle lahenduse saab aga välistada, sest kontseptsiooni järgi peab tabel tervikuna sisaldama infot kogu kirjete ajaloo kohta.

4.3.1. Eelmisele kirjele viitamine

Teine variant oleks lisada uued kirjed tabeli lõppu ning viidata nendest eelmistele (loogiliselt muudetud või kustutatud) kirjetele. Kõik kirjed, mida on niimoodi viidatud, loetakse mitteaktiivseteks, seega aktiivseks kirjeteks loetakse kirjed, millele nii ei viidata.

Näiteks võib tuua sellise tabeli joonise (*Joonis 3*)

id bigint	previous bigint
14	3
16	4
17	5
18	17
20	18

Joonis 3:Kirjete revisioonide salvestamise test-tabel

Nii id kui previous väli peavad olema indekseeritud ning märgitud unikaalsetena. Väli *previous* viitab eelmisele kirje revisioonile. Antud juhul kirje identifikaatoriga *18* eelmine versioon on kirje identifikaatoriga *17* ning järgmine versioon identifikaatoriga 20.

Sellisest tabelist saab kirje ajaloo kätte ühe päringuga:

```
WITH RECURSIVE history(id, previous) AS (
        SELECT id, previous FROM tabeli_nimi WHERE id=vaatlusalune_kirje
        UNION DISTINCT
        SELECT tabeli_nimi.id, tabeli_nimi.previous
        FROM history, tabeli_nimi
        WHERE tabeli_nimi.id = history.previous      -- eelmne kirje
           OR tabeli_nimi.previous = history.id      -- järgmine kirje
)
SELECT * FROM history;
```

WITH RECURSIVE SQL lausend võimaldab teha rekursiivseid päringuid, kus kirje viitab mõnele kirjele ning see omakorda mõnele muule kirjele ja ei ole teada, kui pikk see ahel

võib olla. *WITH* lausend ja selle kasutamine koos näidetega on dokumenteeritud PostgreSQL'i dokumentatsioonis. (The PostgreSQL Global Development Group, 2015)

Testimiseks kasutas autor enam, kui 40 miljoni kirjega tabelit, kus oli ka muid (ebaolulisi) välju. Tabel oli loodud skriptiga, mis loos seoseid juhuslike kirjete vahel. Test-päringute kiiruse välja toomine ei ole siinkohal oluline, sest see on väga sõltuv serveri keskkonnast. Oluline on aga keeruliste päringute puhul vaadata päringuplaani (ingl.k. q*uery execution plan*). Selleks tuleb päringu ette lisada sõna *EXPLAIN*. Antud päringu plaan on järgmine:

```
CTE Scan on history
  CTE history
    ->  Recursive Union
          ->  Index Scan using _pkey on tabeli_nimi
          ->  Nested Loop
```

Kõige tähtsam on, et päringuplaanis ei esine "*Seq scan*" algusega ridu ridu. "*Seq scan*" tähendab, et andmebaasi mootor ei kasuta indekseid, mistõttu tuleb ridade filtreerimiseks iga rida eraldi läbi vaadata, mis omakorda piirab oluliselt päringu jõudlust. Selles päringuplaanis ei ole üldis

4.3.2. Kirjete sidumine ühise

Lihtsama päringuga saaks kogu kirje ajaloo, kui seoks kirjed mingisuguse ühise unikaalse indentifikaatoriga. Sel juhul viitavad kõik kirjed revisioonid sellele indentifikaatorile. Identifikaatoritena kasutatakse ettevõtte infosüsteemides *universally unique identifier* (UUID) tüüpi stringe. UUID formaat on kirjeldatud standardiga RFC4122. (Network Working Group, 2005)

Aktiivseks kirjeks loetakse kirje, millel on kõige suurem järjekorranumber. See tekitab vajaduse igal päringul kirjed sorteerida järjekorranumbri järgi. Sõltuvalt revisioonide arvust võib see olla ajakulukas. Autor on arvestanud kuni kümne revisiooniga kirje kohta ning eeldab, et kogu kirjete ajaloo päringut ei tehta märkimisväärselt tihti.

Testimine toimus eelmise testiga samas suurusjärgus oleva tabeliga. Kui kõikidele asjasse puutuvatele väljadele on indeksid seadistatud, ei paistnud teist-päringute tegemisel välja mingeid kitsaskohti. On oluline, et järjekorranumbri väljale oleks paigaldatud *btree* tüüpi indeks, sest "räsi" tüüpi indekseid ei saa andmebaasiserver sorteerimiseks kasutada.

Kokkuvõtteks on infosüsteemi arendaja või arendajate otsustada, millist meetodit kasutada. Autori eesmärk oli leida vaid võimalikke kitsaskohti.

Kokkuvõte

Lõputöö käigus uuriti erinevaid tehnilisi lahendusi, mida saaks kasutada andmete kõrge tervikluse tagamiseks. Lahenduste eesmärk oli rakendada ISKE HT.10 ja HT.34 meetmed ning uurida andmete revisioonide hoidmise võimalusi. Lõputöö väljundiks olid prototüüplahendused nende eesmärkide saavutamiseks.

ISKE meede HT.10 käsitleb andmete räsiaheldamist. Antud lõputöö kontekstis tähendab see kõrge terviklusega andmeid hoidvate andmebaasitabelite piires kirjete krüptograafilist sidumist märkamatute muudatuste ärahoidmiseks. Muudatuste tuvastamise mehhanismi loomine ei ole antud lõputöö skoobis. Võimalusi meetme rakendamiseks leiti mitmeid ja osad nendest ka realiseeriti.

HT.34 teeb kohustuslikuks kõrge terviklikkuse nõuetega dokumentide digiallkirjastamise vastavalt Eesti Vabariigi digiallkirja seadusele. See esmapilgul triviaalne vajadus osutus tegelikult palju keerulisemaks. Ka siin oli potensiaalseid lahendusi mitu, kuid nendel kõigil olid omad puudused. Näiteks ei olnud Java teegil ametlikku PKCS11 tuge ning eraldi teises keeles kirjutatud programmi kirjutamine ja kasutamine oleks olnud palju keerulisem ning võimalik, et ebastabliisem ning seega ebasoovitav. Lõpuks suudeti minna seda teed, et kasutada Java teeki mitteametliku PKCS11 toega.

Autor tegi kaks parandust DigiDoc'i Java teeki nimega *digidoc4j*, mis oli lõputöö alguses *beta* arengujärgus. Esimene muudatus parandas koodis ebaühtsuse, mis üllatavalt põhjustas üsna raskesti tuvastatava probleemi. See probleem ei võimaldanud kasutada teeki ettevõtte poolt kasutatava Grails raamistikuga.

Teine parandus oli veaparandus, mis ei olnud küll ettevõtte poolt arendatava tarkvara jaoks oluline, kuid kuna see viga tuli teegi arenduse käigus välja, siis oli see vaja lihtsuse

huvides ära parandada. Siiski ei olnud see veaparandus perfektne, sest autor ei pidanud vajalikuks kontrollida ühte kasutusjuhtu, kuid just sellel kasutusjuhul parandus ei töötanud. See oli esimene kord autori jaoks, kui *test-driven* arendusmeetod (ing.k. *Test-Driven Development* – TDD) näitas oma reaalset vajalikkust.

Uurimuse tulemust kasutatakse vähemalt ühe uue infosüsteemi loomisel, millel on kõrge tervikluse ja käideldavuse nõue. Laiem eesmärk on aga lõputöös käsitletud kontseptsiooni kasutada mitmetel kõrge turvaastmega infosüsteemide loomisel.

Kasutatud kirjandus

ISKE

Riigi Infosüsteemi Amet (2015), “Infosüsteemide turvameetmete süsteem ISKE” [https://www.ria.ee/ee/iske.html]

Riigi Infosüsteemi Amet (2014), “ISKE kataloogid” [https://www.ria.ee/public/ISKE/ISKE_kataloogid/ISKE_kataloogid_7.pdf]

JSON

Phil Haack (2009), “JSON Highjacking”[http://haacked.com/archive/2009/06/25/json-hijacking.aspx]

Ecma International (2013), “The JSON Data Interchange Format” (ECMA-404) [http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf]

Digiallkirjastamine

Sertifitseerimiskeskus AS (2015), “What's the difference between the digital signature formats .ddoc, .bdoc and .asice?” [http://www.id.ee/index.php?id=37370]

Sertifitseerimiskeskus AS (2015), “DigiDoc teegid - üldinfo ja ülevaade” [http://www.id.ee/index.php?id=30290]

PostgreSQL

The PostgreSQL Global Development Group (2015), “PostgreSQL 9.4.5 Documentation” [http://www.postgresql.org/docs/9.4/static/index.html]

Hitoshi Harada (2015), “PL v8” [http://pgxn.org/dist/plv8/doc/plv8.html]

GitHub Pull Requests

Kapp (2015), “Make logger declarations consistent (final static)” [https://github.com/open-eid/digidoc4j/pull/6]

Kapp (2015), “Fix creation of custom container with custom configuration” [https://github.com/open-eid/digidoc4j/pull/8]

UUID

Network Development Group (2005), “A Universally Unique IDentifier (UUID) URN Namespace” [https://tools.ietf.org/html/rfc4122]

Lisad

Lisa 1 Krüptoaheldamise päästik: README.md

```
PostgreSQL trigger, mis räsiaheldab sisestatavaid kirjeid ja/või saadab tulemuse syslog
serverile kujul "tabeli_nimi rea_id hash"

Repos olev *hashchain-trigger.so* on kompileeritud PostgreSQL 9.4 jaoks Ubuntu 14.04 x64
masinal.

#### Kompileerimine

    make

#### Triggeri seadistamine

Stored procedure loomine

    CREATE FUNCTION "hashchainTrigger"() RETURNS trigger
       AS '/path/to/hashchain-trigger.so'
       LANGUAGE C;

    CREATE FUNCTION "logHashTrigger"() RETURNS trigger
       AS '/path/to/hashchain-trigger.so'
       LANGUAGE C;

Räsiaheldamise trigger

    -- tabel peab sisaldama välja hashchain, mis on piisava suurusega
    -- et räsi salvestada
    CREATE TRIGGER hashchainTrigger BEFORE INSERT ON tabeli_nimi
       FOR EACH ROW EXECUTE PROCEDURE "hashchainTrigger"();


Räside logimise trigger

    -- tabel peab sisaldama välja hashchain, mis on piisava suurusega
    -- et räsi salvestada
    CREATE TRIGGER hashchainTrigger BEFORE INSERT ON tabeli_nimi
       FOR EACH ROW EXECUTE PROCEDURE "logHashTrigger"();
```

Lisa 2: Krütpoaheldamise päästik: Makefile

```
  default:
         gcc trigger.c hashchain.c logHash.c -I/usr/include/postgresql/9.4/server -lpq
  -lcrypto -shared -fPIC -o hashchain-trigger.so
         chmod 755 hashchain-trigger.so
```

```
#include <syslog.h>

#include <postgres.h>
#include <executor/spi.h>
#include <commands/trigger.h>
#include <utils/rel.h>
#include <fmgr.h>
PG_MODULE_MAGIC;

#include "hashchain.h"
#include "logHash.h"

#ifdef __cplusplus
#define DECLARE_TRIGGER(name) PG_FUNCTION_INFO_V1(name); \
       extern "C" Datum name(PG_FUNCTION_ARGS)
#else
#define DECLARE_TRIGGER(name) PG_FUNCTION_INFO_V1(name); \
       extern Datum name(PG_FUNCTION_ARGS)
#endif

void __attribute__((constructor)) init() {
       openlog("hashchainTrigger", LOG_NDELAY, LOG_LOCAL1);
       syslog(LOG_INFO, "hashchain module loaded");
}

DECLARE_TRIGGER(hashchainTrigger) {
       if(!CALLED_AS_TRIGGER(fcinfo)) elog(ERROR, "hashchainTrigger:”
              “ not called as trigger");

       TriggerData* trigdata = (TriggerData*) fcinfo->context;

       if(!TRIGGER_FIRED_FOR_ROW(trigdata->tg_event) ||
          !TRIGGER_FIRED_BY_INSERT(trigdata->tg_event)) {
              elog(ERROR, "hashchainTrigger: only for row-level INSERT events");
              return PointerGetDatum(TRIGGER_FIRED_BY_UPDATE(trigdata->tg_event)
                     ? trigdata->tg_newtuple : trigdata->tg_trigtuple);
       }

       HeapTuple inputRow = trigdata->tg_trigtuple;
       Relation rel = trigdata->tg_relation;

       HeapTuple modifiedRow = hashchain(rel, inputRow);
       return PointerGetDatum(modifiedRow);
}

DECLARE_TRIGGER(logHashTrigger) {
       if(!CALLED_AS_TRIGGER(fcinfo)) elog(ERROR, "hashchainTrigger:”
              “ not called as trigger");

       TriggerData* trigdata = (TriggerData*) fcinfo->context;

       if(!TRIGGER_FIRED_FOR_ROW(trigdata->tg_event) ||
          !TRIGGER_FIRED_BY_INSERT(trigdata->tg_event)) {
              elog(ERROR, "hashchainTrigger: only for row-level INSERT events");
              return PointerGetDatum(TRIGGER_FIRED_BY_UPDATE(trigdata->tg_event)
                     ? trigdata->tg_newtuple : trigdata->tg_trigtuple);
       }

       HeapTuple inputRow = trigdata->tg_trigtuple;
       Relation rel = trigdata->tg_relation;

       logHash(rel, inputRow);

       return PointerGetDatum(inputRow);
}
```

```
package ee.smit.keijokapp;

import java.security.cert.X509Certificate;

import org.digidoc4j.SignatureToken;

import eu.europa.esig.dss.DigestAlgorithm;
import eu.europa.esig.dss.SignatureValue;
import eu.europa.esig.dss.ToBeSigned;
import eu.europa.esig.dss.token.AbstractSignatureTokenConnection;
import eu.europa.esig.dss.token.DSSPrivateKeyEntry;
import eu.europa.esig.dss.token.Pkcs11SignatureToken;


public class PKCS11SignatureToken implements SignatureToken {
	protected AbstractSignatureTokenConnection signatureTokenConnection = null;
	protected DSSPrivateKeyEntry keyEntry = null;

	/**
	 *
	 * @param pkcs11Path path to PKCS11 library
	 * @param pin PIN code
	 */
	public PKCS11SignatureToken(String pkcs11Path, char[] pin) {
		signatureTokenConnection = new Pkcs11SignatureToken(pkcs11Path, pin);
		keyEntry = signatureTokenConnection.getKeys().get(0);
	}

	/**
	 *
	 * @param pin PIN code
	 */
	public PKCS11SignatureToken(char[] pin) {
		this("/usr/lib/x86_64-linux-gnu/opensc-pkcs11.so", pin)
	}

	@Override
	public X509Certificate getCertificate() {
		return keyEntry.getCertificate().getCertificate();
	}

	@Override
	public byte[] sign(org.digidoc4j.DigestAlgorithm digestAlgorithm,
			byte[] dataToSign) {
		ToBeSigned toBeSigned = new ToBeSigned(dataToSign);
		DigestAlgorithm dssDigestAlgorithm =
			DigestAlgorithm.forXML(digestAlgorithm.toString());
		SignatureValue signature =
			signatureTokenConnection.sign(toBeSigned,
						dssDigestAlgorithm,
						keyEntry);
		return signature.getValue();
	}
}
```

Lisa 5: Allkirjastamine Groovy skriptiiga

```
import org.digidoc4j.Container;
import org.digidoc4j.ContainerBuilder;
import org.digidoc4j.Signature;
import org.digidoc4j.SignatureBuilder;
import ee.smit.keijokapp.PKCS11SignatureToken; // autori loodud prototüüpklass

InputStream file = new ByteArrayInputStream(“test”.getBytes());

// konteineri loomine
Container container = ContainerBuilder
    .aContainer()
    .withDataFile(file, “testfile”, “text/plain”)
    .build();

PKCS11SignatureToken signatureToken =
       new PKCS11SignatureToken(privateKeyPath, “1234”.toCharArray());

// signeerimine
Signature signature = SignatureBuilder
       .aSignature(container)
       .withSignatureDigestAlgorithm(DigestAlgorithm.SHA512).
       .withSignatureToken(signatureToken).
       .invokeSigning();

// signatuuri lisamine konteinerile
container.addSignature(signature);

// konteineri salvestamine
container.saveAsFile("test-container.bdoc");
```