Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Jan Starc

Simetrična enkripcija kliničnih podatkov zaradi zaščite zasebnosti pacienta

diplomsko delo

univerzitetni študijski program
prve stopnje
računalništvo in informatika

Mentor: prof. dr. Marko Bajec

Ljubljana, 2018

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Simetrična enkripcija kliničnih podatkov zaradi zaščite zasebnosti pacienta

Tematika naloge:

Podatki o zdravstvenem stanju pacienta spadajo med ene izmed najbolj občutljivih osebnih podatkov. V diplomski nalogi preučite, kako bi lahko z enkripcijo na učinkovit zaščitili zdravstvene podatke brez da bi s tem bistveno vplivali na učinkovitost poizvedovanja po podatkih. Razvijte prototip rešitve, ki bo demonstriral delovanje glavnih komponent.

Zahvaljujem se mentorju prof. dr. Marku Bajcu za pomoč pri izdelavi diplomske naloge. Zahvaljujem se Nenadu Živkoviću iz podjetja Parsek za predlog zanimive teme te diplomske naloge in vse nasvete glede postavitve arhitekture.

Prav tako se zahvaljujem mami in očetu za vso pomoč in podporo v času študija. Zahvala pa gre tudi vsem prijateljem, s katerimi sem preživel zelo prijetna študijska leta.

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
AES	advanced encryption standard	napredni standard za enkripcijo
API	application programming interface	aplikacijski programski vmesnik
ECB	electronic codebook	elektronska kodna knjiga
FHIR	fast healthcare interoperability resources	hitri zdravstveni interoperabilni viri
JSON	JavaScript object notation	notacija JavaScript objektov
REST	representational state transfer	arhitektura za izmenjavo podatkov med spletnimi storitvami
URL	uniform resource locator	enolični krajevnik vira

Povzetek

Naslov: Simetrična enkripcija kliničnih podatkov zaradi zaščite zasebnosti pacienta

Avtor: Jan Starc

Na področju zdravstva se za potrebe elektronskih kartotek, napotnic, receptov, izvidov,... shranjujejo velike količine kliničnih podatkov. Podatki, ki se shranjujejo, morajo biti pooblaščenim osebam na različnih lokacijah v vsakem trenutku hitro dostopni. Ker shranjeni podatki vsebujejo tudi zasebne podatke pacientov, predstavlja njihova hramba varnostno tveganje. Podatke je torej potrebno čim bolje zaščititi, da maksimalno zmanjšamo tveganje razkritja podatkov, hkrati pa to ne sme opazno vplivati na funkcionalnost same rešitve s stališča končnega uporabnika.

Cilj diplomske naloge je raziskati možne pristope k reševanju tega problema in opisati koncept shranjevanja občutljivih podatkov v podatkovni bazi s pomočjo simetrične enkripcije. Prav tako je potrebno opisani koncept tudi preizkusiti v praksi in za ta namen implementirati prototip.

Ključne besede: simetrična enkripcija, klinični podatki, e-kartoteka, mikrostoritve, REST API, FHIR, HL7, Java EE.

Abstract

Title: Symmetric encryption of clinical data to ensure privacy of the patient

Author: Jan Starc

In the medical field, huge amount of clinical data is stored for the needs of e-records, referrals, recipes, test results,... It is crucial that stored data is accessible to authorized persons at any time, from any location. The problem is that stored data contains sensitive personal information of patients, and consequently storing clinical data poses a security risk. Clinical data needs to be protected as good as possible, but at the same time, this must not affect the functionality of the software solution from the user's point of view.

The goal of this thesis is to find the possible approaches to solution of this problem and to describe a concept of storing sensitive clinical data in the database, based on symmetric encryption. To prove the adequacy of the concept to solve the above described problem, a prototype must be developed.

Keywords: symmetric encryption, clinical data, electronic health record, microservices, REST API, FHIR, HL7, Java EE.

Poglavje 1

Uvod

1.1 Opis problema

Zdravstvo je eno izmed redkih področij, ki je globalno gledano s stališča informatizacije precej neurejeno. Zaradi vse večje informatizacije tudi na tem področju se posledično v zadnjem času shranjuje vse več kliničnih podatkov. Pri hrambi teh podatkov se je potrebno zavedati, da hranimo zelo pomembne in hkrati izredno občutljive podatke. Rezultati meritev, izvidi, kartoteke in ostali zdravstveni podatki morajo biti v vsakem trenutku, na vsaki lokaciji pooblaščenim osebam takoj na voljo. Hkrati pa ti podatki vsebujejo tudi občutljive zasebne podatke pacientov, ki jih je potrebno dobro zaščititi. Hramba teh podatkov torej predstavlja veliko varnostno tveganje, kar je zanimiv izziv, ko želimo te podatke zaščititi.

Cilj te diplomske naloge je raziskati pristope pri reševanju tega problema in najti način, kako najbolj učinkovito, varno in za končnega uporabnika nemoteče hraniti občutljive klinične podatke.

1.2 Motiv

Motivacija za izbrano diplomsko temo je zanimanje za razvoj zalednih sistemov in želja po delu na čim bolj praktičnem primeru. Moja diplomska

naloga rešuje konkreten problem na področju enkripcije podatkov pacientov, ki morajo biti po eni strani kar se da dobro zaščiteni, po drugi strani pa hkrati zelo hitro in v vsakem trenutku dostopni.

K raziskovanju tega področja me je motiviralo tudi to, da v zadnjem času vsakih nekaj mesecev pride do večje izdaje zaupnih podatkov, ko napadalcem uspe vdreti v podatkovne baze podjetij in pridobiti zaupne podatke uporabnikov. V teh primerih ne gre za nepomembna podjetja z nekaj tisoč uporabniškimi računi, ampak za internetne velikane, banke, podjetja v zabavni industriji,...

Med nekaj najbolj odmevnih podjetij, kjer je prišlo do večjih vdorov v podatkovne baze, spadajo Yahoo (leta 2013 je napadalcem uspelo pridobiti uporabniške račune 3 miljard uporabnikov), eBay (leta 2014 so napadalci dobili podatke 145 milijonov uporabnikov), banka JP Morgan (leta 2014 so napadalci dobili finančne podatke 76 milijonov gospodinjstev in 7 milijonov podjetij) in Uber (leta 2016 so napadalci dobili podatke 57 milijonov uporabnikov in 600 tisoč voznikov) [11].

Problem zlorabe podatkov nedvomno obstaja, poleg tega pa gre v primeru zdravstevnih podatkov za zelo zaupne podatke, ki jih je treba za vsako ceno zaščititi.

Diplomska naloga demonstrira reševanje tega problema z uporabo sodobnih arhitekturnih pristopov. Arhitekturni pristop pri implementaciji prototipa vključuje mikrostoritve, kot programski jezik je izbrana Java EE [27]. Cilj diplomske naloge je, da na zanimivem primeru čim bolj raziščem možnosti, ki mi jih izbrane tehnologije ponujajo, hkrati pa demonstriram, da se lahko z izbranim arhitekturnim pristopom postavi hiter, varen, razširljiv in v praksi uporaben sistem za enkripcijo velikih količin kliničnih podatkov.

Poglavje 2

Pregled področja

2.1 Pregled arhitekturnih pristopov

2.1.1 Mikrostoritve

Arhitektura mikrostoritev je način, kako načrtovati in razviti aplikacije kot množico ohlapno povezanih storitev, kjer vsaka izmed njih omogoča določeno funkcionalnost aplikacije. Ta arhitektura predstavlja odmik od klasične, monolitne arhitekture. Pri tem je potrebno omeniti, da ena mikrostoritev ni nujno povezana le z eno samo aplikacijo, ampak si lahko različne aplikacije, ki za delovanje potrebujejo isto ali dovolj podobno funkcionalnost, eno mikrostoritev delijo.

To nam pri razvoju aplikacij prinaša mnogo prednosti, tako s stališča razvoja, kot s stališča robustnosti, odpornosti proti napakam in skalabilnosti. Če se razvoja lotimo pravilno, ni potrebno za vsako aplikacijo razviti vsega od začetka ali kopirati delov kode iz že obstoječih aplikacij v novo, ampak lahko iz že obstoječega nabora mikrostoritev preprosto uporabimo tiste, ki jih potrebujemo. Dodatne funkcionalnosti, ki jih potrebujemo, pa dodamo tako, da razvijemo nove mikrostoritve. Ta pristop nas spodbuja k temu, da razvijamo čim bolj samostojne enote, ki so lahko ponovno uporabne v sklopu drugih aplikacij. Vsaka komponenta aplikacije je pri tem pristopu

torej razvita ločeno, funkcionalnost končne aplikacije pa nato predstavlja vsoto funkcionalnosti mikrostoritev, ki jih uporablja.

Glede skalabilnosti, robustnosti in fleksibilnosti nam ta pristop prinaša mnogo prednosti, hkrati pa tudi mnogo izzivov. Ena od glavnih prednosti je fleksibilnost oziroma odpornost na napake, saj odpoved ali napaka na eni mikrostoritvi ne vpliva na ostale mikrostoritve. Prav tako nam ta pristop prinaša prednosti v primeru visoke obremenitve. Najbolj obremenjene mikrostoritve lahko skaliramo bistveno preprosteje kot eno obsežno monolitno aplikacijo.

Ta pristop s seboj prinaša tudi mnogo izzivov. Namesto ene aplikacije, ki se izvaja na enem strežniku ali z izravnalnikom obremenitve (*angl.* load balancer) hkrati na večih strežnikih, imamo lahko pri pristopu z mikrostoritvami rešitev, ki ima lahko posamezne komponente (mikrostoritve) spisane v različnih programskih jezikih, ki se izvajajo na množici strežnikov, na različni strojni opremi in različnih virtualnih okoljih. Takšno kompleksno konfiguracijo pa je zelo težko obvladovati. Ko pride do povečanja obremenitve, morajo biti vse komponente, ki sestavljajo aplikacijo, koordinirane na način, da lahko identificiramo ozka grla, ki jim moramo dodati vire, kar ponovno zagotovi tekoče delovanje aplikacije [30, 15].

2.1.2 Aplikacijski programski vmesniki (API)

Zelo pomemben koncept pri razvoju mikrostoritev so tudi aplikacijski programski vmesniki API (*angl.* Application Programming Interface). API-ji mikrostoritvam omogočajo povezavo z zunanjim svetom, torej z aplikacijami, ki jih uporabljajo, zalednimi sistemi, ostalimi mikrostoritvami,... [30]

API-ji so množica rutin, orodij in protokolov, ki pomagajo pri razvoju programske opreme. Splošno gledano gre v primeru API-ja za množico jasno definiranih metod komunikacije med različnimi komponentami. Dobro definiran API olajša razvoj programske opreme, saj ponuja gradnike, ki jih mora programer le še povezati in s tem izkoristiti funkcionalnosti, ki jih ponujajo. Ena od bistvenih lastnosti dobrih API-jev je tudi, da skrivajo morebitno

kompleksnost in podrobnosti same implementacije znotraj modulov, saj programerju za učinkovito uporabo API-ja ni potrebno razumeti podrobnosti same implementacije. Zato je pri načrtovanju API-jev ključno, da ustvarimo programsko orodje v takšni obliki, da ga lahko programer hitro razume in uporabi v sklopu neke rešitve [7, 17].

REST API

Trenutno najbolj uporabljan pristop gradnje API-jev v okviru mikrostoritev je REST, vendar so mnoge mikrostoritve zaradi svojih specifičnih potreb osnovni REST API standard že precej nadgradile. [30]

REST je arhitekturni pristop, ki definira množico omejitev, ki naj bi se upoštevale pri načrtovanju spletnih storitev. Predstavlja arhitekturo, kjer je posamezen vir enoznačno določen z naslovom URL. REST spletne storitve dovoljujejo sistemom, ki jim pošiljajo zahtevke, da dostopajo in spreminjajo spletne vire (*angl.* resources). Do virov dostopajo preko predhodno definirane množice operacij, ki jih REST API določene spletne storitve podpira [42].

Pri REST spletnih storitvah zahtevki po določenem viru povzročijo odziv, ki vsebuje podatke, ki so najpogosteje formatirani v JSON, XML ali HTML formatu. Za pošiljanje zahtevkov se večinoma uporablja protokol HTTP, najpomembnejše operacije, ki jih podpirajo REST API-ji so standardizirane HTTP metode: GET, POST, PUT in DELETE [16].

REST deluje na podoben način kot spletni brskalniki. Spletni brskalniki z uporabo naslova URL pošljejo zahtevek na nek strežnik, ki jim odgovori v obliki datoteke HTML in v primeru sodobnih spletnih strani še slogovne datoteke CSS, slik in ostalih datotek. Spletni brskalnik nato iz prejetih datotek prikaže zahtevano spletno stran. V primeru REST API-ja pa aplikacija pošlje zahtevek v dogovorjeni obliki, REST API pa na zahtevek odgovori v vnaprej določenem formatu.

2.2 Pregled področja računalniške varnosti

2.2.1 Enkripcijski algoritmi

V kriptografiji je enkripcija proces kodiranja sporočila ali informacije na način, da lahko le avtorizirane osebe dostopajo do vsebine. Enkripcija ne preprečuje, da bi tretja oseba prestregla sporočilo na poti od pošiljatelja do prejemnika, vendar pa onemogoča ali vsaj bistveno otežuje, da bi tretja oseba razbrala vsebino sporočila. V enkripcijski shemi se originalno sporočilo imenuje navadno besedilo (*angl.* plaintext), ki je šifrirano z enkripcijskim algoritmom v šifrirano besedilo, katerega pomen lahko ugotovimo le, če sporočilo dešifriramo. Enkripcijski algoritmi ponavadi uporabljajo psevdonaključne enkripcijske ključe, ki jih generira nek algoritem. V principu je možno dešifrirati sporočilo, ne da bi imeli enkripcijski ključ, vendar bi za dobro načrtovane sisteme potrebovali znatne računske zmogljivosti in specifično znanje. Avtorizirana oseba (prejemnik) pa lahko sporočilo preprosto dekriptira s ključem [18].

Simetrična enkripcija

Pri simetrični enkripciji sta enkripcijski in dekripcijski ključ enaka. Tako pošiljatelj kot prejemnik morata imeti isti ključ, da lahko dosežeta varno izmenjavo sporočil. Ključ predstavlja porazdeljeno skrivnost med dvema ali več stranmi, ki so vključene v šifrirano komunikacijo. Predpogoj je, da imajo vsi, ki so vključeni v komunikacijo, dostop do istega ključa, kar je tudi ena od glavnih slabosti simetrične enkripcije, saj to predstavlja varnostno tveganje.

Poznamo dva tipa simetričnih algoritmov – tokovne šifrirne algoritme (*angl.* stream cypher) in bločne šifrirne algoritme (*angl.* block cypher). Tokovni šifrirni algoritmi ponavadi šifrirajo posamezne bajte. Bločni algoritmi vzamejo vnaprej določeno število bitov in jih šifrirajo kot eno enoto. V primeru sporočila, ki je krajši od velikosti bloka, pa prilagodijo dolžino sporočila na večkratnik velikosti bloka [13].

Asimetrična enkripcija

Asimetrična enkripcija je kriptografski sistem, ki uporablja par ključev. Javni ključi so lahko javno znani, privatni ključi pa morajo biti znani le prejemniku sporočila. To služi dvema funkcijama – avtentikaciji, saj lahko preko javnega ključa ugotovimo, da je imetnik privatnega ključa res poslal sporočilo in enkripciji, saj lahko le tisti, ki ima zasebni ključ dešifrira sporočilo, ki je bilo šifrirano z njegovim javnim ključem. Torej lahko vsak šifrira sporočilo z uporabo prejemnikovega javnega ključa. Šifrirano sporočilo pa je lahko dešifrirano le s prejemnikovim zasebnim ključem.

Moč asimetrične enkripcije je pogojena s količino računskega napora (faktorjem dela), ki ga je potrebno vložiti v to, da najdemo zasebni ključ, ki ustreza javnemu ključu. Glavni pogoj pri asimetrični enkripciji je, da tretja oseba ne pridobi zasebnega ključa. Na drugi strani lahko javni ključ javno objavimo, brez da bi s tem ogrozili varnost kriptiranih podatkov [13].

2.2.2 Pregled podobnih rešitev

Ker gre v primeru hrambe medicinskih podatkov za varnostno zelo kritične podatke, vsaka podrobnost o implementaciji pa za rešitve iz področja računalniške varnosti predstavlja varnostno tveganje, je razmeroma težko pridobiti podatke o rešitvah, ki rešujejo podobne probleme kot moja diplomska naloga. V nadaljevanju tega poglavja je opisanih nekaj rešitev iz področja hrambe medicinskih podatkov ter bolj splošnih rešitev iz področja varne hrambe podatkov, ki so služili kot usmeritev pri pripravi te diplomske naloge.

CryptDB

Ena od najbolj zanimivih rešitev, po kateri sem se zgledoval pri pripravi te diplomske naloge, je CryptDB [1, 10]. CryptDB je rešitev s področja enkripcije podatkovnih baz, ki brez bistvene upočasnitve delovanja omogoča izvajanje večine SQL poizvedb na kriptiranih podatkih. S tem morebitnemu napadalcu tudi ob vdoru v podatkovno bazo ne razkriva nobenih uporab-

nih podatkov, zato lahko tudi v primeru varnostno zelo kritičnih podatkov uporabljamo varnostno bolj tvegane storitve hrambe podatkov v oblaku.

Tudi sama arhitektura in način izvajanja poizvedb pri rešitvi CryptDB sta služila kot zgled pri reševanju problema te diplomske naloge. Ideja v primeru te rešitve je, da se na strani aplikacije uporablja hiter namestnik (*angl.* proxy). Namestnik hrani shemo podatkovne baze in enkripcijski ključ, vendar ne shranjuje nobenih podatkov in direktno ne izvaja poizvedb. Ko aplikacija izvede poizvedbo, jo namestnik na poti prestreže. Nekriptirane atribute, ki so bili del poizvedbe, s pravim ključem kriptira in spremenjeno poizvedbo pošlje na podatkovno bazo. Poizvedba se nato izvede na kriptirani podatkovni bazi s kriptiranimi atributi v poizvedbi, rezultati poizvedbe se vrnejo namestniku, ki jih dekriptira in v dekriptirani obliki vrne aplikaciji. Bistvo tega pristopa je, da podatkovna baza le hrani kriptirane podatke in nad njimi izvaja poizvedbe, vendar nikoli ne pridobi ključa za dekripcijo teh podatkov.

Ker spreminjanje poizvedb in obdelavo kriptiranih podatkov v celoti izvaja namestnik, poizvedbe pa se, enako kot v primeru brez enkripcije, v celoti izvajajo na podatkovni bazi, CryptDB ne zahteva nobenih sprememb na obstoječih sistemih za upravljanje podatkovnih baz (SUPB). Kljub temu, da CryptDB precej prispeva k varnosti podatkov, je upočasnitev ob njegovi uporabi le 26% [10] v primerjavi z običajnimi SQL poizvedbami na nekriptiranih podatkih.

Enkripcija na strani uporabnika pri varnostno tveganih aplikacijah v oblaku

Ker se danes vse več podatkov hrani v podatkovnih bazah v oblakih, ki jih ponujajo tretje osebe, to posledično pomeni, da tisti, ki najame oblačne storitve nekega podjetja, nima direktnega vpliva na varnost hrambe teh podatkov. To je še posebej problematično v primeru hrambe občutljivih podatkov. Zaradi majhnega vpliva na konfiguracijo okolja v oblaku in posledično nivoja varnosti, lahko najemnik teh storitev v večini primerov nadzira le, katere

podatke oziroma v kakšni obliki bo podatke shranjeval v oblak [32].

Zato je v zadnjem času enkripcija podatkov na strani uporabnika (*angl.* client-side encryption) pred shranjevanjem na podatkovno bazo, vse bolj uporabljan pristop, saj želimo čim bolj zmanjšati tveganje, ki se pojavi kot posledica hrambe podatkov v oblaku. Če so podatki kriptirani pred shranjevanjem v podatkovno bazo na način, da jih potencialni napadalec niti z zelo zmogljivo strojno opremo ni zmožen dekriptirati, potem varnost same oblačne storitve za hrambo podatkov nima bistvenega vpliva na varnost samih podatkov.

Vse trenutne rešitve, ki vključujejo enkripcijo na strani uporabnika, bazirajo na kompromisu med hitrostjo iskanja in varnostjo shranjenih podatkov. Neke splošne rešitve oziroma pristopa, ki bi ustrezal večini scenarijev hrambe občutljivih podatkov v oblaku, trenutno še ni na trgu, zato lahko, predvsem zaradi vse večje rabe podatkovnih baz v oblaku, na tem področju v naslednjih nekaj letih pričakujemo veliko raziskav in razvoja novih rešitev [32].

Varnostni načrt za shranjevanje e-kartotek v oblaku v primeru indijskega zdravstvenega sistema

Raziskav primernih pristopov za celovito informatizacijo zdravstva se lotevajo tudi v Indiji. Indija je s stališča reševanja tega problema nedvomno zanimiva, saj ima ogromno populacijo, hkrati pa zelo razvejan zdravstveni sistem s številnimi javnimi in zasebnimi bolnišnicami.

Predlagan pristop [14] temelji na hrambi medicinskih podatkov v oblaku in velik poudarek namenja varnosti komunikacije in varnosti hrambe podatkov v bazi. Problem varnosti je, tako s stališča zagotavljanja varne povezave, kot kriptiranja podatkov na sami bazi, rešen s pomočjo asimetrične enkripcije. Velik poudarek je namenjen tudi avtorizaciji medicinskega osebja za dostop do medicinskih podatkov s strani pacientov. Osnovna ideja tega sistema je, da bi imela vsak pacient in zdravnik za namen avtorizacije svoj par asimetričnih ključev.

V primeru, da želi zdravnik pisati v kartoteko nekega pacienta, morata pacient in s strani pacienta avtorizirani zdravnik izvesti proces rokovanja s

sistemom (*angl.* handshake). Po uspešnem procesu rokovanja sistem generira par ključev in zdravniku pošlje javni ključ. Zdravnik s tem javnim ključem kriptira in pošlje podatke, ki jih želi dodati v kartoteko. Takoj po prejemu podatkov sistem z zasebnim ključem, ki je del prej generiranega para ključev, podatke dekriptira. Nato generira nov par ključev in pravkar prejeti vnos s strani zdravnika kriptira z novim ključem in ga shrani v podatkovno bazo.

Tudi v primeru branja je proces podoben. Pacient in zdravnik s sistemom najprej izvedeta proces rokovanja, prav tako se preverja, če ima zdravnik pravico za dostop do pacientovih podatkov. Nato sistem poišče podatke v bazi in jih dekriptira. Po dekripciji sistem podatke zaradi varnega pošiljanja kriptira z zdravnikovim javnim ključem in jih pošlje zdravniku. Zdravnik po prejemu dekriptira podatke s svojim zasebnim ključem in dostopa do zahtevane kartoteke.

Sistem s tem pristopom tako v primeru branja, kot v primeru pisanja, zagotavlja izolacijo med enkripcijsko shemo podatkov, ki se pošiljajo in podatkov, ki se shranjujejo.

Opisani koncept torej na celovit in precej kompleksen način skrbi za varno hrambo in izmenjavo kliničnih podatkov v kontekstu zdravstvenega sistema. Prav tako, sploh glede na kompleksnost sistema, dosega relativno dobre čase poizvedb, zato lahko služi kot dober zgled pri reševanju problemov na področju informatizacije zdravstva.

2.2.3 Tehnologija veriženja podatkovnih blokov v zdravstvu

Zanimivo področje, kjer se na področju e-zdravstva odpirajo še številne možnosti za razvoj, je povezano s tehnologijo veriženja podatkovnih blokov (*angl.* blockchain). Porazdeljena glavna knjiga (*angl.* distributed ledger) in tehnologija veriženja podatkovnih blokov je trenutno zelo aktualna na mnogih področjih, kot sta na primer finance in pravo. Zelo poenostavljeno je veriga podatkovnih blokov enotna zgodovina transakcij, ki je replicirana in distribuirana na več decentraliziranih lokacijah. Ker so podatki distribuirani

na več decentraliziranih lokacijah, je praktično nemogoče, da bi jih kdorkoli lahko spreminjal za nazaj. Ta tehnologija ponuja varen in celovit mehanizem za vzdrževanje pregledne evidence, kateri podatek je kje shranjen, kako se je spreminjal skozi čas in kdo ga je spreminjal.

V zdravstvu je cilj, da bi s pomočjo te tehnologije še bolj varno in celostno sledili posameznim kartotekam. V večini implementacij je kartoteka nekega pacienta porazdeljena na različnih lokacijah, pri različnih organizacijah in ponudnikih zdravstvenih storitev. Del kartoteke je shranjen pri zdravstveni ustanovi, kjer ima pacient osebnega zdravnika, nekaj delov kartoteke je porazdeljenih v podatkovnih bazah zdravstevnih ustanov, kjer je ta pacient obiskoval različne specialiste. Del pacientove kartoteke so lahko tudi podatki iz prenosnih naprav (*angl.* wearables), ki beležijo pacientove vitalne znake in shranjujejo podatke v popolnoma ločeno podatkovno bazo.

Tehnologija veriženja podatkovnih blokov lahko pomaga pri tem, da iz posameznih delov kartoteke lažje sestavimo celotno kartoteko, za katero lahko z gotovostjo trdimo, da je popolna in posodobljena. Če bi zdravstevni standard baziral na osnovi te tehnologije, bi bila vsaka sprememba pacientove kartoteke preverjena in skupaj z ostalimi transakcijami dodana kot blok v daljšo verigo podatkovnih blokov. Posledično bi lahko bili v vsakem trenutku prepričani, da smo pridobili popolno sliko pacientove zdravstvene zgodovine, z jasno informacijo o tem, kako se je spreminjala skozi čas in kdo je izvajal te spremembe.

Podatkovni bloki so zaradi teh razlogov zelo učinkoviti proti napadom na integriteto podatkov (*angl.* integrity-based attacks). Pri teh napadih poskuša napadalec dodati ali brisati podatke iz neke kartoteke (na primer dodajanje ali brisanje podatkov o neki alergiji) na način, da sprememba ni sledljiva. Originalni podatek je le zamenjan z novim, vendar ni nikjer v zapisniku (*angl.* log) zabeleženo, da je prišlo do spremembe originalnega podatka. To pa je skrb vzbujajoče tako s stališča varnosti pacienta, kot zaupanja v samo zdravstevno institucijo. S podatkovnimi bloki bi lahko zagotovili, da imamo v vsakem trenutku celotno zgodovino sprememb kartoteke, v kateri lahko

natančno vidimo, kaj je bilo spremenjeno, kdaj je bilo spremenjeno in kdo je to spreminjal, s praktično ničelnim tveganjem, da so v bazi tudi lažne informacije, ki jih je nekdo spreminjal za nazaj.

Potrebno pa je izpostaviti tudi področja, kjer ta tehnologija, vsaj kratkoročno, ne obeta bistvenih izboljšah. Tehnologija veriženja podatkovnih blokov ne bo veliko prispevala k varovanju podatkov v podatkovnih bazah, preprečevanju, da bi zdravstveno osebje izkoriščalo svoj dostop za pregled podatkov pacientov, ki niso v njihovi negi ali preprečevanju vdora v naprave, kjer lahko zdravstevno osebje dostopa do podatkov v nekriptirani obliki.

Tehnologija veriženja podatkovnih blokov torej zelo učinkovito rešuje določene probleme, ki se trenutno pojavljajo pri razvoju rešitev iz področja e-zdravstva. Vendar se je potrebno zavedati tudi omejitev te tehnologije in jo učinkovito združiti z ostalimi varnostnimi rešitvami, da lahko razvijemo varen sistem za informacijsko podporo zdravstvu [5].

Poglavje 3

Uvod v praktični del

3.1 Podroben opis problema

Problem, ki sem ga reševal s tem diplomskim delom, je definirati in implementirati prototip rešitve, ki bi kot sestavni del neke aplikacije s področja e-zdravstva omogočala shranjevanje in dostop do kliničnih podatkov na čim bolj varen način. Osredotočil sem se predvsem na problem varnega shranjevanja podatkov v bazi in na izbiro primerne, sodobne arhitekture za rešitev mojega problema.

Celoten sistem mora torej omogočati varno shranjevanje in dostop do podatkov, hkrati pa mora biti tako glede hitrosti, kot tudi same uporabe zasnovan na način, da se končni uporabnik modula, ki v ozadju skrbi za vidik varnosti, ne zaveda. V primeru, da je uporabnik izvor podatkov (npr. dodajanje novega izvida), ta podatke pošlje na strežnik v nekriptirani obliki. Na poti od uporabnika do podatkovne baze se morajo kritični atributi ustrezno kriptirati in v podatkovno bazo zapisati v kriptirani obliki. V primeru, da želi uporabnik pridobiti podatke iz podatkovne baze, je treba zagotoviti učinkovito iskanje podatkov neglede na to, ali uporabnik išče po kriptiranih ali nekriptiranih atributih. Ko podatke v bazi najdemo, pa je potrebno na poti od podatkovne baze do avtoriziranega uporabnika podatke še dekriptirati, da jih lahko uporabniku prikažemo v osnovni obliki, saj imajo le v taki

obliki zanj uporabno vrednost.

V podatkih, ki se shranjujejo, je zato potrebno identificirati atribute, ki predstavljajo večje varnostno tveganje in v primeru vdora v podatkovno bazo nepooblaščeni osebi omogočajo identifikacijo pacienta oziroma povezavo pacienta z njegovimi kliničnimi podatki. Najvišji nivo varnosti bi sicer dosegli, če bi, seveda poleg zagotavljanja varne komunikacije, v bazi kriptirali vse podatke pod čim več različnimi ključi ali enkripcijskimi algoritmi. Vendar bi tak nivo zaščite bistveno otežil iskanje podatkov in posledično tudi povečal čas dostopa do podatkov, kar za praktično rabo ni sprejemljivo, saj morajo biti podatki hitro dostopni v vsakem trenutku. Zato je eden izmed ciljev te naloge najti čim boljši kompromis med zaščito in hitrostjo dostopa do podatkov.

Iskanje podatkov po kriptiranih atributih predstavlja dodaten sklop problemov. V splošnem velja, da bolje kot so podatki kriptirani, oziroma manj informacij kot kriptiran podatek razkriva o samemu sebi ali relacijah z drugimi podatki[1], bolj računsko zahtevne in časovno kompleksne so operacije iskanja po teh podatkih [1]. Zato je tudi v tem primeru potrebno poiskati enkripcijski algoritem, ki nudi čim boljši kompromis med varnostjo kriptiranih podatkov v podatkovni bazi in hitrostjo iskanja po njih.

Čeprav je tveganje za razkritje kriptiranih podatkov ob uporabi dobrega enkripcijskega algoritma in primerno velikega ključa izredno majhno, se vseeno povečuje sorazmerno s časom, ki ga ima napadalec na voljo, da ugotovi (oziroma s poskušanjem ugane) enkripcijski ključ. Ker gre v primeru te diplomske naloge za hrambo zelo občutljivih podatkov, je kljub majhnemu tveganju priporočljivo ključe na določeno časovno obdobje menjavati. Zaradi tega je potrebno rešiti tudi problem periodične menjave ključa. Menjava ključa in ponovna enkripcija podatkov, ki so bili prej kriptirani s starim ključem, je relativno časovno kompleksna operacija, zato je potrebno najti način, ki ključe menjava postopoma, brez da bi to vplivalo na hitrost do-

[1]Recimo, če sta kriptirani vrednosti istega podatka pod istim ključem enaki ali ne, ali če enkripcija ohranja relacijo večje-manjše pri številih in podobno

stopa do podatkov ali celo povzročalo morebitne izpade delovanja celotne aplikacije.

Zadnji od večjih problemov, ki jih rešuje to diplomsko delo, je izbira primerne, sodobne arhitekture, ki že s svojo zasnovo čim bolj porazdeli tveganje razkritja podatkov. Arhitekturo je potrebno zasnovati čim bolj porazdeljeno in modularno, da napad na eno komponento aplikacije (npr. podatkovno bazo) še ne povzroči razkritja uporabnih kliničnih podatkov. Napadalec bi moral torej istočasno vdreti v vse dele aplikacije, da bi lahko pridobil uporabne podatke in uspešno povezal klinične podatke s pacienti. Poleg tega pa mora sama arhitektura v primeru, da se ugotovi, da je ena od komponent varnostno kritična, z minimalnimi prilagoditvami celotne aplikacije omogočati preprosto menjavo oziroma popravke kritične komponente.

3.2 Prvi koraki do rešitve problema

Namen tega poglavja je opisati problem enkripcije, možne rešitve in najti čim boljši kompromis med varnostjo podatkov in hitrostjo delovanja varnostnega modula.

Glede hrambe podatkov v podatkovni bazi obstajata dve skrajnosti, ki se bistveno razlikujeta glede praktičnosti in varnosti, nekriptirana podatkovna baza in polna homorfična enkripcija.

Relacijske podatkovne baze inženirji razvijajo in optimizirajo že več kot štirideset let. Za namen iskanja so bile razvite posebne podatkovne strukture (indeksi), ki omogočajo hitrejše iskanje podatkov. Posledično je nekriptirana podatkovna baza zelo hitra. Vendar pa v primeru vdora v nekriptirano podatkovno bazo napadalec takoj dobi dostop do vseh podatkov, zato je ta pristop nedvomno varnostno kritičen.

Polna homorfična enkripcija (PHE) pa izvede kriptiranje vseh podatkov v bazi z dobrim enkripcijskim pristopom. PHE omogoča katerokoli splošno operacijo nad kriptiranimi podatki, ne da bi jih bilo potrebno pri tem odkriptirati. Poleg tega je semantično zelo varna, saj ne izdaja praktično nobenih informacij o kriptiranih podatkih oziroma relacij med kriptiranimi podatki. Vendar pa je PHE, kljub vsem dobrim lastnostim, trenutno še bistveno prepočasna za praktično uporabo šifriranja celotne podatkovne baze, saj je tudi v najboljših implementacijah vsaj nekajkrat počasnejša od poizvedb nad nekriptiranimi podatki [1].

Poleg tega moramo v trenutnih praktičnih implementacijah za vsako poizvedbo preiskati celotno bazo, saj PHE za razliko od nekriptirane baze, ki s pomočjo indeksa zelo hitro najde iskani podatek, ne omogoča učinkovitega iskanja z indeksi.

Na tej točki lahko zaključimo, da bo nedvomno potrebno glede enkripcije poiskati čim boljši kompromis med varnostjo in hitrostjo. V nadaljevanju tega poglavja bom podal odgovore na spodnja vprašanja in s tem utemeljil mojo odločitev glede pristopa k enkripciji:

1. Kateri so ključni atributi, ki jih je sploh potrebno kriptirati?
2. Kakšen tip enkripcije (simetrična, asimetrična) je najbolj primeren za rešitev tega problema?
3. Kateri enkripcijski algoritem izbrati?

3.2.1 Ključni atributi, ki jih je potrebno kriptirati

Celotna diplomska naloga je zasnovana na odprtokodnem HL7 FHIR standardu, ki je trenutno eden izmed najbolj pogosto uporabljanih standardov na področju e-zdravstva. Vsi klinični podatki se po tem standardu shranjujejo v obliki virov (*angl.* resources), ki so, gledano z vidika podatkovnega modela, generalizacija vseh relacij, ki jih shranjujemo (npr. pacient, opažanje, stanje, zdravilo,...). Vir je entiteta, ki mora ustrezati naslednjim pogojem:

- Ima enolični identifikator (URL), po katerem ga lahko najdemo.
- Se identificira kot eden izmed tipov vira, ki jih definira HL7 FHIR specifikacija (npr. pacient, opažanje,...).
- Vsebuje množico strukturiranih obveznih podatkov, ki opisujejo ta tip.
- Hrani oznako različice, ki se spremeni v primeru, da pride do spremembe vira.

Viri so med seboj povezani z referencami. Torej, kartoteka nekega pacienta je sestavljena iz množice virov, ki hranijo referenco na vir tipa pacient ali pa so z virom tipa pacient v tranzitivni odvisnosti [21].

Operacije nad kriptiranimi podatki so v primerjavi z operacijami nad nekriptiranimi podatki časovno bistveno bolj kompleksne. Glede na to, da se v primeru aplikacij s področja e-zdravstva shranjujejo velike količine podatkov, do katerih želimo hitro dostopati, vseh podatkov nima smisla kriptirati, saj bi s tem bistveno upočasnili vse poizvedbe in ogrozili uporabnost celotne aplikacije.

Ko se odločamo o tem, katere atribute se splača kriptirati, da dosežemo optimalno razmerje med varnostjo podatkov in hitrostjo aplikacije, je potrebno razumeti probleme, ki jih rešuje to diplomsko delo, v kontekstu celotne aplikacije. V primeru dejanske aplikacije s področja e-zdravstva je pričakovano, da bo imela dobro poskrbljeno za vidik varnosti. To pomeni, da bo s pravimi protokoli in zaščito poskrbljeno za varno komunikacijo med vsemi vpletenimi, poleg tega pa bodo posamezne komponente sistema primerno zaščitene pred morebitnimi napadi.

Varnostni modul, ki je rezultat te diplomske naloge, predstavlja v kontekstu celotne aplikacije zadnji korak zaščite podatkov. Naloga varnostnega modula je, da v primeru, ko bi potencialnemu napadalcu uspelo zaobiti vse ostale varnostne mehanizme in pridobiti dostop do podatkov, ki se hranijo v podatkovni bazi, onemogoči interpretacijo podatkov. Glede na način hrambe podatkov po tem standardu, bi do te situacije prišlo, če bi napadalcu uspelo povezati vire tipa pacient z ostalimi viri povezanimi s tem pacientom, saj bi lahko na ta način napadalec dobil vpogled v pacientovo kartoteko.

Glede na to lahko zaključimo, da je varnostno najbolj kritičen atribut referenca na vir tipa pacient, saj le ta atribut omogoča, da medsebojno povežemo vire, ki shranjujejo medicinske podatke in pacienta. Poleg tega so varnostno kritični še demografski podatki pacientov, ki jih hranimo v podatkovni bazi, saj bi morebitni napadalec v primeru, da bi pridobil vire tipa pacient, pridobil tudi dostop do vseh njihovih demografskih podatkov.

Podatki, ki jih hranijo vsi ostali viri pa so, če zanemarimo, da hranijo referenco na vir tipa pacient, le množica vsebinsko nepovezanih medicinskih podatkov. Ti viri hranijo zelo malo ozko specifičnih podatkov o objektu, ki ga opisujejo (npr. podatki o neki alergiji, informacije o nekem zdravilu,...), tako da jih je brez reference na pacienta praktično nemogoče povezati s pacientom. Glede teh virov lahko zaključimo, da bi jih bilo precej nesmiselno kriptirati, saj so brez reference na pacienta popolnoma vzeti iz konteksta in tudi v nekriptirani obliki ne razkrivajo praktično nobenih uporabnih podatkov.

3.2.2 Primerna vrsta enkripcije

Enkripcija podatkov v podatkovni bazi bistveno vpliva na vse poizvedbe nad kriptiranimi podatki. Pri odločitvi za izbiro tipa enkripcije (simetrična ali asimetrična) se moramo zato osredotočiti predvsem na problem iskanja po podatkovni bazi.

Poizvedbe nad kriptiranimi podatki želimo izvajati na popolnoma enak način kot nad nekriptiranimi podatki, saj zaradi enkripcije ne želimo upočasniti poizvedb. Uporabnik bi neglede na to, ali išče podatek, ki je na bazi kriptiran ali ne, na strežnik poslal poizvedbo z nekriptiranimi podatki. Atributi, ki so na bazi kriptirani, bi se na poti od uporabnika do baze kriptirali, na bazi bi se izvedlo iskanje, na poti nazaj od baze do uporabnika pa bi se podatki dekriptirali, tako da bi jih uporabnik videl v osnovni obliki.

Ker standard HL7 FHIR temelji na relacijskem podatkovnem modelu, bom v nadaljevanju tega poglavja uporabljal terminologijo poizvedovalnega jezika SQL. Najprej je potrebno pregledati tipe SQL poizvedb, ki jih želimo podpreti tudi nad kriptiranimi podatki in si ogledati minimalne pogoje, ki jim moramo zadostiti, da bodo te poizvedbe delovale.

Preverjanje enakosti v poizvedbah

Glede varnosti kriptiranih podatkov je potrebno omeniti, da so v primeru, ko hočemo zagotoviti najvišji nivo varnosti, problematične tudi informacije, ki jih nek podatek razkriva v relaciji z drugimi podatki (na primer enakost).

V primeru, da uporabljamo deterministični enkripcijski algoritem, se enolični identifikator pacienta vedno preslika v isto kriptirano sliko. Ta vrsta enkripcije nam omogoča enostavno iskanje po kriptiranih podatkih. Edina razlika v primerjavi z nekriptiranimi podatki je, da poizvedbo izvedemo s kriptirano vrednostjo iskanega atributa, še vedno pa gre za enostavno preverjanje enakosti. Slabost tega pristopa je, da bi v primeru, če bi napadalcu na podlagi vsebine (ne reference) uspelo ugotoviti, da je nek medicinski podatek povezan z nekim pacientom, lahko ta preprosto pridobil še preostanek kartoteke tega pacienta, saj se referenca na pacienta vedno kriptira v isto

kriptirano vrednost.

Največji nivo varnosti bi dosegli, če bi uporabljali naključno (nedeterministično) enkripcijo. Bistvena lastnost naključne enkripcije je, da lahko en original preslika v več različnih slik. V kontekstu zgornjega primera, bi se enolični identifikator pacienta v vsakem viru, ki vsebuje medicinski podatek tega pacienta, kriptiral v različne vrednosti. Vendar naključna enkripcija za naš primer rabe ni primerna, saj je naše glavno vodilo, da poizvedbe nad kriptiranimi podatki izvajamo na popolnoma enak način kot nad nekriptiranimi podatki. Nakjučna enkripcija bi to onemogočala, saj spremeni osnovno predpostavko, na kateri je bazirano preverjanje enakosti – da je vrednost iskanega atributa enolično določena in točno enaka vrednosti tega atributa v bazi.

Da ne spreminjamo načina izvajanja poizvedb nad kriptiranimi podatki, moramo nekriptiran original vedno preslikati v isto kriptirano sliko. Podoben problem kot pri naključni enkripciji bi nastal tudi, če bi bila referenca na pacienta v virih enega pacienta v istem trenutku kriptirana z različnimi enkripcijskimi ključi. Ker je kriptirana slika odvisna tako od vhodnega podatka (enoličnega identifikatorja pacienta) kot ključa, bi bila posledično zaradi različnih ključev različna tudi slika (kriptirana vrednost reference). Zato moramo zagotoviti tudi, da bo referenca enega pacienta v vseh virih, ki shranjujejo medicinske podatke tega pacienta, v istem časovnem trenutku kriptirana z istim ključem, saj bomo le tako zadostili zgornjemu pogoju.

Simetrična ali asimetrična enkripcija?

Simetrična in asimetrična enkripcija se poleg razlik, opisanih v poglavju Pregled področja, med seboj bistveno razlikujeta v hitrosti. Algoritmi za simetrično enkripcijo so povprečno 100-krat hitrejši kot algoritmi za asimetrično enkripcijo [36, 24, 12], vendar imajo slabosti, kot so pomanjkanje skalabilnosti in zahtevna varna izmenjava ključa. Asimetrična enkripcija pa, sicer za ceno hitrosti, ponuja prednosti javnih ključev in preprosto izmenjavo ključa.

Ker asimetrična enkripcija za kriptiranje in dektriptiranje podatkov po-

trebuje bistveno več časa in računske moči kot simetrična enkripcija, mora biti, da se odločimo za asimetrično enkripcijo, njena raba nujna za zagotavljanje varnosti. Prav zaradi bistvene razlike v hitrosti se v praksi v večini sistemov uporabljajo tako imenovani hibridni kriptosistemi [22, 9].

Glavna ideja hibridnih kriptosistemov je, da združi prednosti obeh pristopov. Varnostno najbolj kritični koraki, kot je na primer izmenjava ključa za simetrično enkripcijo med dvema vpletenima stranema, se izvedejo z asimetrično enkripcijo. Največja slabost simetrične enkripcije je v tem, da si obe strani, vpleteni v komunikacijo, ne moreta na varen način izmenjati enkripcijskega ključa, če predhodno še nista komunicirali, saj se preko nevarnega kanala ne moreta na varen način medsebojno dogovoriti, kakšen bo ključ. Asimetrična enkripcija pa je za najbolj varno kritičen korak izbrana zato, ker omogoča, da oba vpletena, brez skupnih podatkov, ki bi si jih morala predhodno izmenjati, lahko varno komunicirata preko nevarnega kanala. S tem se izognemo varnostnemu tveganju, da bi na poti lahko morebitni napadalec prestregel ključ za simetrično enkripcijo. Hibridni pristop se uporablja v številnih standardih, kot so SSL (*angl.* Secure Socket Layer), SSH (*angl.* Secure Socket Hasing) in PGP (*angl.* Pretty Good Privacy). Ti standardi danes tvorijo osnovo za varno komunikacijo na spletu.

Da se lahko odločimo, kateri algoritem je primeren za primer, ki ga rešuje ta diplomska naloga, je treba razumeti še nekaj ključnih dejstev glede arhitekturnega pristopa (podrobneje opisanih v poglavju Arhitektura). Vse poizvedbe bodo potovale od uporabnika, preko vmesne mikrostoritve, ki bo izvajala enkripcijo oz. dekripcijo in prilagajala poizvedbe, do FHIR strežnika in po isti poti nazaj. V primeru ene instance take mikrostoritve ne bi bilo potrebe po asimetrični enkripciji, saj mora le ta mikrostoritev poznati enkripcijske ključe. Ključev v tem primeru ni potrebno nikomur pošiljati ali jih kamorkoli prenašati, saj tako enkripcijo kot dekripcijo izvaja ena mikrostoritev, ki ima ključe varno shranjene. V primeru porazdeljenega sistema z več instancami iste mikrostoritve za enkripcijo in dekripcijo, pa bi bilo potrebno le ključe med posameznimi instancami izmenjati z varnejšo, asimetrično en-

kripcijo, za vse operacije enkripcije in dekripcije pa bi lahko, enako kot v zgornjem primeru, uporabljali hitrejšo, simetrično enkripcijo.

Glede na to lahko zaključimo, da za vse operacije enkripcije in dekripcije, ki jih izvaja vmesna mikrostoritev, razen izmenjave enkripcijskih ključev med posameznimi instancami, ni nobene potrebe za uporabo asimetrične enkripcije. Zato se nam kot očitna rešitev ponuja simetrična enkripcija, ki bo vse operacije izvedla bistveno hitreje, hkrati pa zaradi tega končna aplikacija ne bo trpela z vidika varnosti.

3.2.3 Izbira enkripcijskega algoritma

V prejšnjem podpoglavju je bila zaradi očitnih prednosti pri hitrosti za končno aplikacijo izbrana simetrična enkripcija. V nadaljevanju si bomo z vidika uporabe ogledali nekaj najbolj široko uporabljanih algoritmov za simetrično enkripcijo in izbrali najprimernejšega za rešitev problema tega diplomskega dela.

DES in 3DES

Algoritem DES (*angl.* Data Encryption Standard) je danes precej zastarel, saj je postal standard za simetrično enkripcijo že leta 1974. Bil je prvi enkripcijski standard, ki ga je priporočal NIST (*angl.* National Institute of Standards and Technology). V zadnjih 40 letih je bilo za algoritem DES odkritih že mnogo varnostnih lukenj, na podlagi katerih je osnovanih mnogo napadov na ta algoritem. V današnjem času ta algoritem velja za nevarnega in neprimernega za šifriranje zaupnih podatkov.

3DES ali trojni DES je bil razvit kot izboljšava osnovnega algoritma DES, glavna razlika med njima pa je, da je čistopis v primeru algoritma 3DES trikrat zaporedoma kriptiran z osnovnim algoritmom DES. To sicer poveča nivo varnosti, vendar bistveno upočasni enkripcijo. Posledično je 3DES počasnejši od ostalih primerljivo varnih enkripcijskih algoritmov, zato je za praktično uporabo manj primeren [29].

AES (Rijndael)

Algoritem AES (*angl.* Advanced Encryption Standard) oziroma Rijndael, kot je bil imenovan pred standardizacijo s strani NIST, je novi enkripcijski standard, ki ga NIST priporoča in je na tem mestu nadomestil zastareli DES [8]. Obstaja več verzij tega algoritma, vse verzije imajo enako dolžino bloka (128 bitov), razlikujejo pa se v dolžini ključa (128, 192 ali 256 bitov).

V zadnjih 20 letih, odkar je bil algoritem predstavljen, so že mnogi poskušali nanj izvesti učinkovit napad. Do danes je bilo uspešnih le nekaj napadov po stranskem kanalu (*angl.* side channel attack) na nekaj specifičnih implementacij algoritma AES. Napadi po stranskem kanalu ne poskušajo razbiti samega enkripcijskega algoritma, ampak se osredotočajo na napake v specifičnih implementacijah tega algoritma v programski ali strojni opremi in jih izkoriščajo za napade. Zato uspešen napad po stranskem kanalu vsekakor ne pomeni, da določen enkripcijski algoritem ni varen, izpostavlja le težave v določeni implementaciji. Napad z grobo silo je trenutno edini znani napad proti temu algoritmu. Pri algoritmu AES je treba poudariti še, da je danes tudi strojno podprt na skoraj vseh novejših procesorjih proizvajalcev Intel in AMD. Pri teh dveh proizvajalcih je AES-NI (*angl.* Advanced Encryption Standard New Instructions) nabor ukazov vključen kot razširitev standardnega x86 nabora ukazov [38]. Poleg tega pa strojno podporo algoritmu AES nudi še večina novejših procesorjev z arhitekturo ARM. Strojna podpora pričakovano prinese bistvene pohitritve, v idealnih pogojih je lahko strojno podprta enkripcija in dekripcija nekajkrat (tudi do desetkrat) hitrejša od strojno nepodprte enkripcije [26].

Blowfish in Twofish

Blowfish je relativno hiter enkripcijski algoritem, ki je bil prvič predstavljen leta 1993. Algoritem lahko uporablja ključe, dolge od 32 do 448 bitov, velikost bloka pri tem algoritmu pa je 64-bitov, kar je manj varno kot 128-bitni bloki, ki jih uporablja AES. Manjša velikost bloka naredi algoritem Blowfish v nekaterih primerih bolj izpostavljen tako imenovanemu rojstnodnevnemu

napadu (*angl.* birthday attack) [4]. Zaradi nekaj znanih slabosti v določenih primerih uporabe tudi Bruch Schneier, avtor algoritma Blowfish, predlaga uporabo algoritma Twofish.

Twofish je algoritem, ki je bil ustvarjen z namenom, da nadomesti algoritem Blowfish. Prvič je bil objavljen leta 1998 in je bil eden izmed algoritmov na razpisu Advanced Encryption Standard process, ki ga je razpisala NIST, da bi postavili nov enkripcijski standard, po tem, ko se je algoritem DES izkazal za neprimernega. Do sedaj na algoritem Twofish ni nobenega znanega uspešnega napada, zato sam algoritem velja za varnega. Twofish je, glede na nekatere lastnosti, podoben algoritmu Rijndael, saj prav tako uporablja 128-bitne bloke in ključe velikosti 128, 192 in 256 bitov [31]. Na prej omenjenem razpisu za novi enkripcijski standard je bil skupaj z algoritmom Rijndael eden izmed petih finalistov, vendar se je izkazal za malenkost počasnejšega v primeru 128-bitnih ključev [37, 8].

Po tem, ko je bil Rijndael izbran za novi enkripcijski standard, je Rijndael postal mnogo hitrejši od algoritma Twofish na procesorjih, ki podpirajo AES nabor ukazov, kar danes velja za večino sodobnih procesorjev.

Zaključek in izbira algoritma

V tem poglavju je bilo opisanih le nekaj najbolj znanih algoritmov, ki se uporabljajo za simetrično enkripcijo. Glede na to lahko zaključimo, da obstaja množica bolj ali manj primernih algoritmov, v primeru te diplomske naloge bi bila primerna algoritma AES in Twofish.

Za praktični del te diplomske naloge je bil izbran algoritem AES oziroma Rijndael, ker v primerjavi z ostalimi algoritmi dosega primerljive (če že ne v vseh primerih najbolj optimalne) čase in ima v zadnjem času še zavidljivo strojno podporo. Poleg tega pa ni zanemarljivo tudi dejstvo, da je v zelo široki uporabi že precej časa, kar ga v praksi naredi bolj varnega, saj so bile mnoge napake v implementacijah že odkrite in popravljene. AES je za reševanje problema te diplomske naloge zato najbolj primeren, vsekakor pa ni edini primeren algoritem za reševanje tega problema.

Poglavje 4

Arhitektura

V prvem delu tega poglavja se bom navezal na opis problema in navedel osnovne funkcionalosti, ki so potrebne za njegovo rešitev. V nadaljevanju tega poglavja pa bom opisal izbrane arhitekturne pristope, ki bodo v naslednjem poglavju vodili tudi do praktične rešitve problema.

Z arhitekturo te rešitve želimo doseči, da hranimo medicinske podatke na varen način, hkrati pa lahko s stališča končne aplikacije izvajamo poizvedbe nad kriptiranimi podatki na enak način kot nad nekriptiranimi podatki.

4.1 Osnovne funkcionalnosti

Če strnemo probleme poglavja Opis problema, lahko zaključimo, da mora prototip za demonstracijo delovanja imeti implementirane naslednje sklope funkcionalnosti:

1. Kriptiranje kritičnih atributov in shranjevanje le-teh v podatkovno bazo.
2. Iskanje po kriptiranih atributih v podatkovni bazi.
3. Dekriptiranje podatkov na poti od podatkovne baze do aplikacije in prikaz uporabniku v nekriptirani obliki.
4. Periodična menjava ključa.

4.2 Cilji

Cilj je postaviti arhitekturo na način, da lahko celotna aplikacija deluje na varen način preko sistema, razvitega za to diplomsko nalogo, ne da bi se končni uporabnik ali razvijalec drugih delov aplikacije tega zavedal, saj bo podsistem navzven deloval kot HL7 FHIR API vmesnik. Edina dovoljena sprememba na strani aplikacije je lahko ta, da se za dostopno točko do FHIR strežnika, namesto direktnega dostopa do strežnika, uporablja v ta namen razvita mikrostoritev, ki je del varnostnega modula. Prav tako je cilj, da ne bodo potrebne spremembe na strani podatkovne baze, saj želimo, da podatkovna baza preko API vmesnika normalno shranjuje podatke, neglede na to, da so nekateri podatki kriptirani.

Strežnik s podatkovno bazo nikoli ne pridobi dekripcijskega ključa, ampak le hrani kriptirane podatke. Torej tudi v primeru, da napadalec pridobi poln dostop do podatkovne baze, ta vseeno ne more pridobiti praktično ničesar uporabnega iz podatkov v podatkovni bazi, saj so zaradi kriptirane reference popolnoma neuporabni za interpretacijo (podrobnejša razlaga v poglavju „Ključni atributi, ki jih je potrebno kriptirati“).

Na strani aplikacije ne želimo delati večjih sprememb, saj je namen, da to rešitev čim lažje vključimo v že obstoječo aplikacijo, zato želimo, da bodo potrebne spremembe zaradi tega varnostnega modula minimalne.

Prav tako želimo ohraniti prednosti na strani FHIR strežnika oziroma podatkovne baze. Kriptiranje podatkov ne sme poslabšati hitrosti poizvedb v podatkovni bazi. To prednost ohranimo s tem, da poizvedbe predhodno obdelamo in po vseh podatkih poizvedujemo na enak način, neglede na to, ali so kriptirani ali ne.

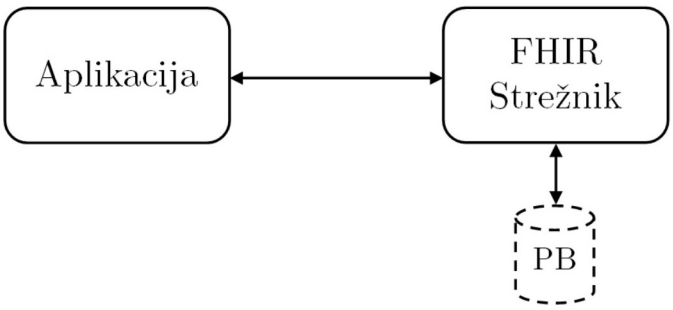
Pristop, da strežnik s podatkovo bazo nikoli ne pridobi ključev pa je potreben zato, ker želimo čimbolj porazdeliti tveganje razkritja podatkov. Da bi napadalec lahko pridobil kartoteke vseh pacientov, bi moral v primeru porazdeljenega pristopa istočasno vdreti v strežnik s podatkovno bazo in mikrostoritev, ki izvaja enkripcijo in dekripcijo ter shrambo ključev.

4.3 Postavitev sistema

V prvem delu tega poglavja se bom posvetil načrtovanju arhitekture in utemeljitvi, zakaj je taka arhitektura primerna za rešitev problema. V nadaljevanju pa bom na nivoju komunikacije med posameznimi komponentami opisal, kako naj bi glavne komponente med seboj komunicirale v primeru glavnih funkcionalnosti.

4.3.1 Osnovna postavitev sistema

V osnovi ima sistem po FHIR specifikaciji le dve glavni komponenti – aplikacijo in FHIR strežnik. Aplikacija je lahko kakršnakoli aplikacija s področja e-zdravstva, ki uporablja HL7 FHIR standard in rabi za delovanje dostop do medicinskih podatkov. FHIR strežnik navzven izpostavlja standardiziran FHIR RESTful API vmesnik in ima dostop do podatkovne baze, ki hrani medicinske podatke. Podatkovna baza (na spodnji sliki označena s PB) navzen ni vidna, torej aplikacija ne more direktno dostopati do nje. Do baze lahko direktno dostopa le FHIR strežnik, vsa komunikacija med aplikacijo in strežnikom pa poteka preko standardiziranega API vmesnika [20].



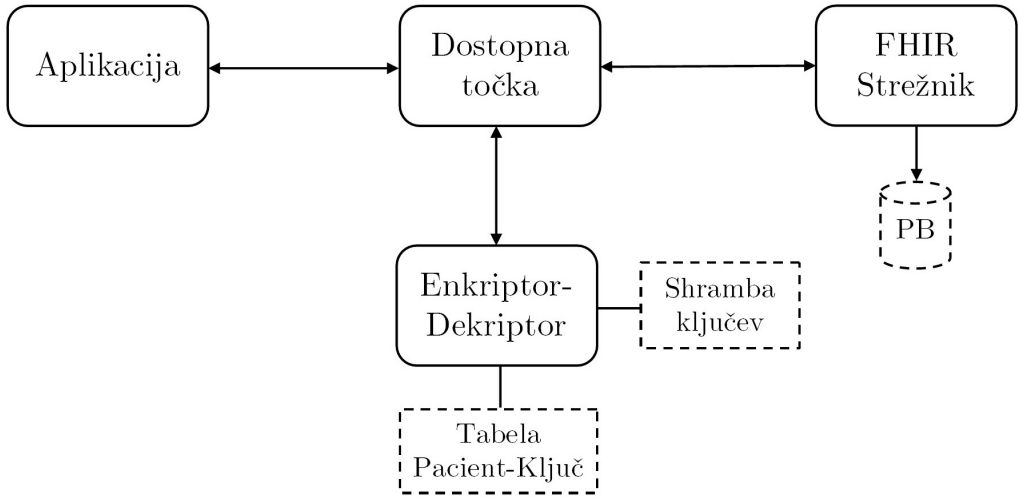
Slika 4.1: Osnovna postavitev sistema brez varnostnega modula

Zgornja slika opisuje osnovno postavitev sistema, na kateri se gradijo rešitve s področja e-zdravstva. Taka postavitev ni namenjena uporabi v praksi, saj za varnostni vidik ni poskrbljeno.

4.3.2 Ideja rešitve

Glavni cilji arhitekture so, da v primerjavi z osnovno postavitvijo (Slika 3.1) ni potrebnih bistvenih sprememb na strani aplikacije. Poleg tega želimo čim manjše spremembe v hitrosti, zato moramo na enak način poizvedovati po podatkih, neglede na to ali so kriptirani ali ne. Zadnji glavni cilj pa je porazdelitev tveganja, zato želimo, da so kriptirani podatki shranjeni ločeno od ključev, s katerimi so ti podatki kriptirani.

Ideja rešitve je, da se osnovni postavitvi sistema dodajo še dve glavni komponenti – mikrostoritev, ki opravlja funkcijo **dostopne točke** in mikrostoritev, ki izvaja enkripcijo oziroma dekripcijo, **enkriptor-dekriptor (ED)**.



Slika 4.2: Glavne komponente sistema z varnostnim modulom

S tako arhitekturo je edina sprememba na strani aplikacije ta, da namesto, da dostopa direktno do strežnika, dostopa do dostopne točke, ki je dostopna preko standardiziranega API vmesnika.

Naloga dostopne točke je, da preusmerja zahtevke med aplikacijo, ED in FHIR strežnikom ter tako omogoča normalno delovanje aplikacije. Dostopna točka je tako edina komponenta sistema, ki je vidna navzven oziroma edina komponenta, do katere lahko aplikacija direktno dostopa.

Druga mikrostoritev, predvidena v tem sistemu pa je ED, ki omogoča, da poizvedovanje po podatkih poteka na enak način, neglede na to, če so kriptirani ali ne. Do te mikrostoritve lahko dostopa le dostopna točka. ED je tudi edina komponenta tega sistema, ki lahko dostopa do ključev, ki so varno shranjeni (podrobnejša razlaga v poglavju Implementacija). Prav tako ne shranjuje nobenih podatkov in ne izvaja poizvedb oziroma ne komunicira direktno s FHIR strežnikom ali podatkovno bazo. Naloga ED je le, da „prevaja“ med kriptiranimi in nekriptiranimi podatki, ter tako omogoča nespremenjeno delovanje z vidika aplikacije.

Na strani FHIR strežnika glede na osnovno postavitev ni bistvenih sprememb, edina razlika je, da namesto, da bi direktno komuniciral z aplikacijo, komunikacija poteka preko dostopne točke, ki preusmerja poizvedbe. FHIR strežnik neglede na to, če so podatki kriptirani ali ne, procesira zahtevke in komunicira s podatkovno bazo na popolnoma enak način.

V **tabeli pacient-ključ** ni shranjen dejanski enkripcijski ključ, ampak le ime ključa, preko katerega lahko dostopamo do enkripcijskega ključa, ki je varno shranjen v **shrambi ključev**. Tabela pacient-ključ je bistvena komponenta pri inkrementalni reenkripciji, ki je podrobneje razložena v naslednjem podpoglavju.

4.3.3 Inkrementalna reenkripcija

Inkrementalna reenkripcija je obsežnejši problem, ki zahteva nekaj razmisleka. Ker gre v primeru hrambe medicinskih podatkov za varnostno zelo kritične podatke, smo v poglavju Opis problema zaključili, da bi bilo dobro enkripcijske ključe na določeno časovno obdobje menjavati, kljub temu, da je možnost, da napadalec ugotovi ključ in s tem pridobi dostop do podatkov, izredno majhna.

V primeru uporabe varnostnega modula v praksi v sklopu neke zdravstvene aplikacije, bi manjše zdravstvene ustanove hranile kartoteke nekaj sto pacientov, v primeru večjih bolnišnic ali enotnih zdravstvenih sistemov na ravni regij ali držav, pa bi bila ta številka bistveno večja. Poleg tega je

treba upoštevati, da je kartoteka po FHIR specifikaciji sestavljena iz manjših virov, ki vsebujejo zdravstvene podatke in hranijo referenco na pacienta. Ker posamezen vir hrani zelo malo podatkov (npr. opis simptoma, meritev tlaka, podatki o neki alergiji, informacije o terapiji,...), se z vsakim obiskom pri zdravniku ustvari precej dodatnih virov, zato število virov, ki jih hranimo v bazi hitro narašča. Ker je referenca edini atribut, ki ga bomo kriptirali v vseh virih (razen virov tipa pacient) in jo vsebujejo vsi viri, število referenc za reenkripcijo narašča sorazmerno s številom virov, ki jih hranimo v bazi. Neglede na to, kakšno je število virov, pa je v praktično vseh primerih preveliko, da bi celoten proces menjave ključa in reenkripcije lahko izvedli na vseh podatkih naenkrat, saj si ne smemo privoščiti izpadov ali zelo počasnega delovanja zaradi vzdrževanja podatkovne baze.

Posledično se je potrebno problema lotiti postopoma. Na kakršen koli način bi se lotili postopne menjave ključa, bi bili istočasno v bazi atributi, ki so kriptirani z različnimi ključi. Vendar, če je v vsakem trenutku jasno, kateri atribut je kriptiran s katerim ključem, hkratna raba večih ključev ne povzroča dodatne zmede, ampak je celo prednost, saj poveča nivo varnosti. V primeru, da je hkrati v rabi več ključev in bi napadalcu uspelo vdreti v podatkovno bazo in pridobiti ali ugotoviti enega izmed kjučev, ki so v tistem trenutku v rabi, s tem ne bi pridobil nekriptirane vsebine celotne baze, ampak le vsebino dela podatkov, ki je kriptirana s tem ključem.

Ker moramo v vsakem trenutku vedeti, kateri atribut v podatkovni bazi je kriptiran s katerim ključem, je treba posamezne vire, ki jih hranimo v bazi, razbiti na logične, dovolj majhne podmnožice podatkov, ki so kriptirani z istim ključem in jih lahko naenkrat reenkriptiramo. Relativno majhna, logična podmnožica podatkov, ki bi lahko služila kot osnova za inkrementalno reenkripcijo, je kartoteka enega pacienta. V tem primeru moramo za uspešno reenkripcijo kartoteke posameznega pacienta hraniti le podatek, s katerim ključem je njegova kartoteka trenutno kriptirana. Poleg tega je kartoteka enega pacienta dovolj majhna podenota, da njena reenkripcija ne predstavlja znatne obrementive celotnega sistema. Zato je idealna tudi za reenkripcijo,

saj lahko število kartotek, ki jih v nekem trenutku reenkriptiramo, enostavno sproti prilagajamo prostim resursom, ki so na voljo, ne da bi s tem opazno upočasnili delovanje celotnega sistema.

Vse spremembe, ki jih izvajamo na izbrani podmnožici podatkov (kartoteki enega pacienta) pa morajo biti izvedene v sklopu transakcije, da se izognemo tveganju, da bi ob delno uspešni reenkripciji prišlo do neskladja med shranjeno relacijo pacient-ključ in dejanskim stanjem kartoteke tega pacienta v podatkovni bazi, saj bi s tem tvegali izgubo podatkov.

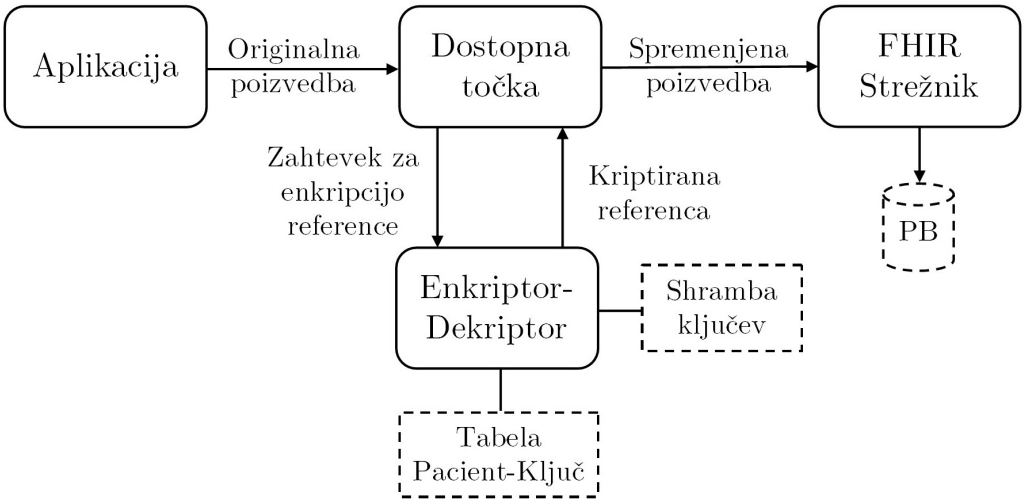
Hkratna raba večih ključev torej poveča nivo varnosti in je nujen pogoj za inkrementalno reenkripcijo atributov. Ker želimo imeti čim večjo kontrolo nad postopno reenkripcijo in s tem postopkom čim manj obrementiti sistem, se splača imeti dobro strategijo upravljanja ključev. Zato je poleg relacije pacient-ključ koristno hraniti še podatek, kdaj je bil ključ pri določenem pacientu nazadnje zamenjan. Ta podatek nam pomaga pri učinkoviti reenkripciji. Ko ugotovimo, da je pri določeni podmnožici pacientov poteklo preveč časa od zadnje menjave ključa, lahko generiramo nov ključ, kartoteke pacientov iz te podmnožice postopoma reenkriptiramo z novim ključem in za vsako uspešno reenkriptirano kartoteko spremembo ključa in čas zadnje spremembe zapišemo v tabelo pacient-ključ.

4.4 Pregled izvajanja zahtevkov na načrtu arhitekture

V tem poglavju si bomo na shemi arhitekture celotne rešitve ogledali nekaj osnovnih funkcionalnosti aplikacije, ki so služile kot opora pri programiranju prototipa. Poleg tega je potrebno razložiti še nekaj podrobnosti glede same arhitekture, ki so bile v prejšnjem poglavju namenoma izpuščene, saj jih je najlažje razložiti na praktičnih primerih.

4.4.1 GET zahtevki

Spodnjo shemo je najlaže razložiti na praktičnem primeru, zato je opisano delovanje sistema na primeru GET zahtevka, ki želi pridobiti vse vire tipa Observation, ki pripadajo nekemu pacientu. Edini kriptiran atribut, ki ga vsebuje vir tipa Observation, je referenca na pacienta. V primeru GET zahtevka bi enaka shema kot za vir Observation veljala tudi za vse ostale tipe virov, ki vsebujejo kriptirano referenco na pacienta.

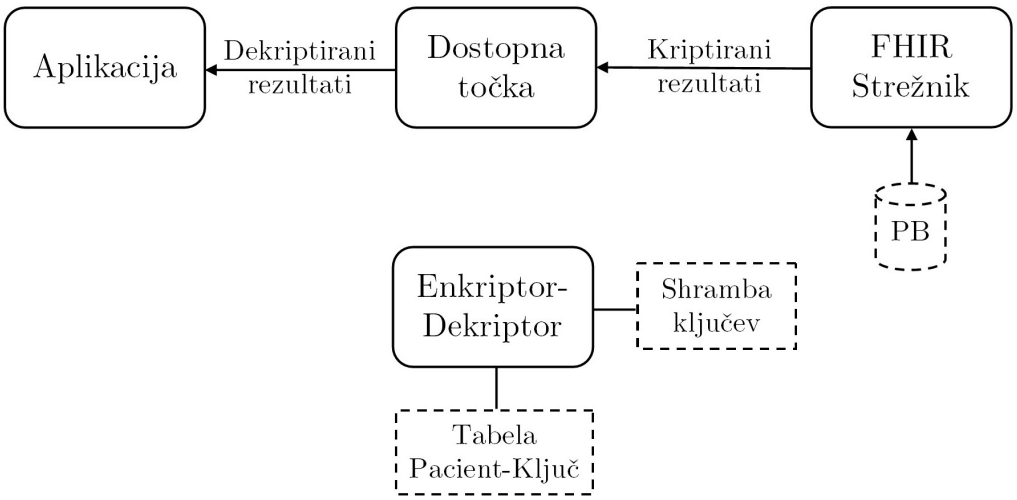


Slika 4.3: GET zahtevek - Poizvedba

Ko aplikacija izvede poizvedbo, jo dostopna točka prestreže. Dostopna točka zazna, da gre za GET poizvedbo po viru Observation, ki vsebuje nekriptiran identifikator pacienta, zato ED pošlje zahtevek za enkripcijo reference na pacienta, saj je referenca v bazi shranjena v kriptirani obliki.

ED najprej v tabeli pacient-ključ preveri, s katerim ključem je trenutno kriptiran pacient. Nato dostopa do shrambe ključev in pridobi ustrezen ključ, z njim kriptira referenco na pacienta in kriptirano referenco pošlje dostopni točki.

Dostopna točka po prejemu odgovora s strani ED spremeni poizvedbo, tako da zamenja nekriptirano referenco s kriptirano in spremenjeno poizvedbo pošlje FHIR strežniku. FHIR strežnik dostopa do podatkovne baze in izvede poizvedbo na kriptiranih podatkih.



Slika 4.4: GET zahtevek - Rezultat poizvedbe

Ko FHIR strežnik dobi rezultate poizvedbe, jih posreduje dostopni točki. Dostopna točka začasno hrani nekriptirano referenco na pacienta, saj v vmesnem času čaka na rezultat GET zahtevka. Ko dostopna točka dobi rezultate, jih obdela tako, da v vseh virih zamenja kriptirano referenco z nekriptirano in jih v taki obliki pošlje aplikaciji.

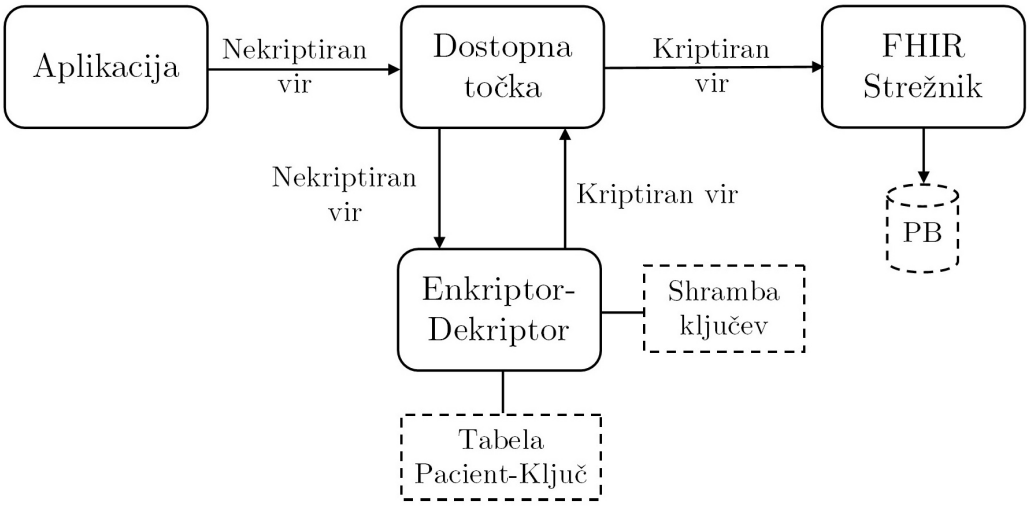
4.4.2 POST zahtevki

Pošiljanje POST zahtevkov poteka na podoben način kot v primeru GET zahtevkov. Ponovno bo postopek opisan na primeru vira tipa Observation. Tudi v primeru POST zahtevkov velja, da je postopek enak za vse tipe virov, ki vsebujejo referenco na pacienta.

Aplikacija generira vir tipa Observation, ki vsebuje nekriptirano referenco na nekega pacienta. Ta vir pošlje dostopni točki, ki ga v nespremenjeni obliki posreduje enkriptorju-dekriptorju (v nadaljevanju ED).

ED zazna, da gre za vir tipa Observation, ki vsebuje nekriptirano referenco na pacienta. Najprej v tabeli pacient-ključ preveri, pod katerim ključem je kriptirana kartoteka pacient. Nato dostopa do shrambe ključev, pridobi ustrezen ključ in z njim kriptira referenco. Po uspešni enkripciji reference obdela vir tako, da zamenja nekriptirano referenco s kriptirano in vir v tej obliki pošlje dostopni točki.

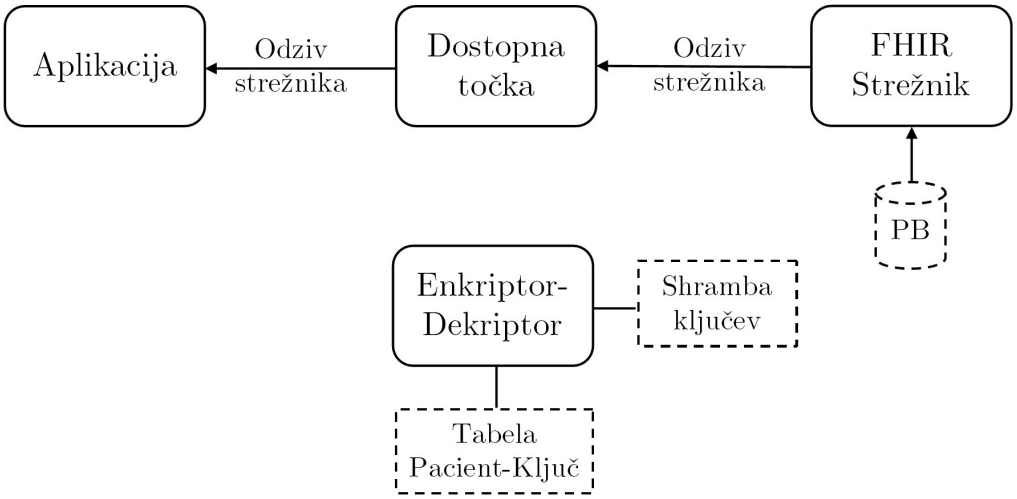
Dostopna točka kriptirani vir v obliki POST zahtevka posreduje FHIR strežniku, ki dostopa do podatkovne baze in prejeti vir shrani.



Slika 4.5: POST zahtevek - Zahteva

FHIR strežnik nato pošlje rezultat operacije. Če je bilo shranjevanje vira uspešno, strežnik dostopni točki pošlje enolični identifikator tega vira, ki se je avtomatsko generiral ob ustvarjanju tega vira. V primeru, da shranjevanje ni bilo uspešno, strežnik javi napako.

Dostopna točka prejet odziv s strani strežnika v nespremenjeni obliki posreduje aplikaciji. S tem korakom se POST poizvedba zaključi.

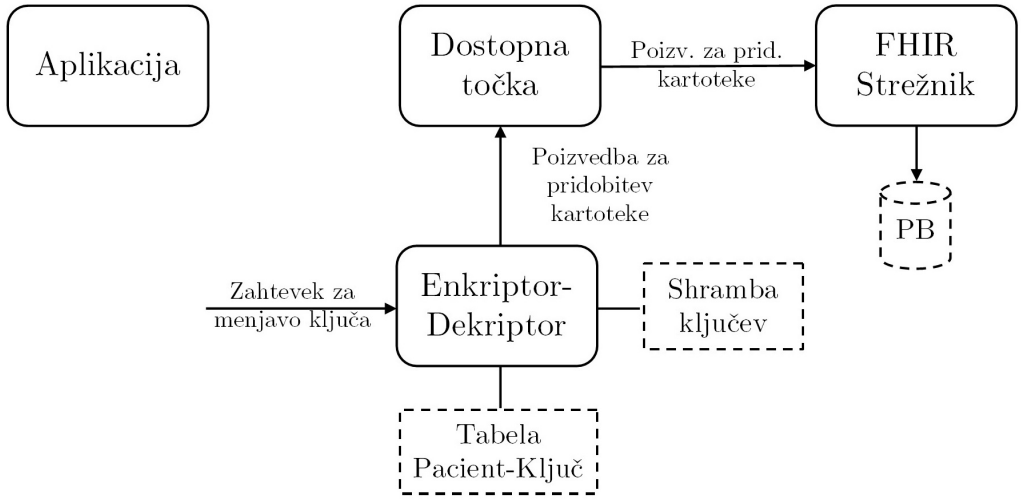


Slika 4.6: POST zahtevek - Rezultat

4.4.3 Menjava ključa

Menjava ključa je postopek, ki ga izvaja administrator sistema in ostalim uporabnikom ni dostopna. Dobro je, da menjavo ključa in reenkripcijo izvaja nekdo, ki pozna celoten sistem in se zaveda varnostnih tveganj.

Menjava ključa poteka v dveh korakih. Na začetku postopka administrator generira nov ključ, ki se varno shrani v shrambo ključev. Nato na ED pošlje zahtevek za menjavo ključa nekega pacienta. ED s ključem, pod katerim je pacient trenutno shranjen, kriptira njegov identifikator, nato pa generira poizvedbo za pridobitev celotne kartoteke tega pacienta. Poizvedbo pošlje dostopni točki, ta jo posreduje FHIR strežniku. FHIR strežnik za pridobitev kartoteke izvede iskanje v podatkovni bazi.



Slika 4.7: Menjava ključa, korak 1 - Poizvedba za pridobitev kartoteke

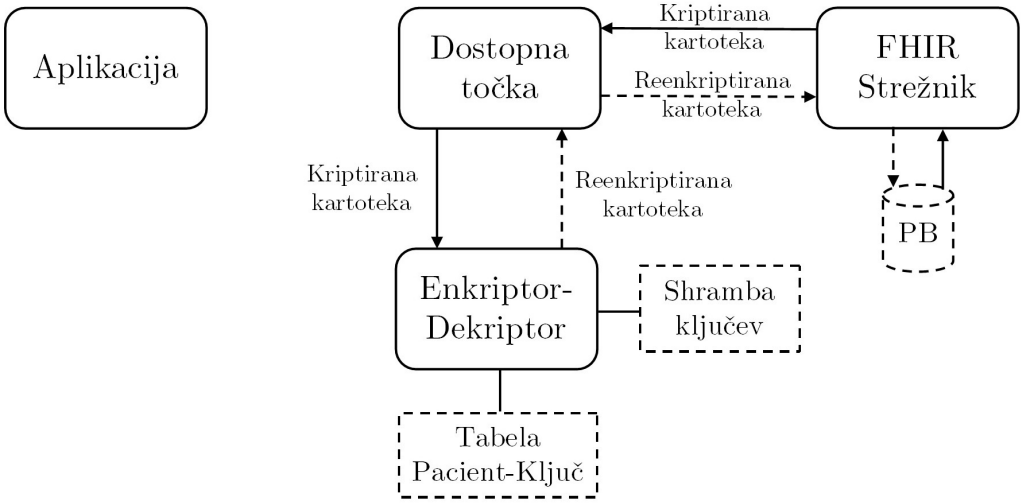
V drugem koraku FHIR strežnik kriptirano kartoteko posreduje dostopni točki, ta pa jo v nespremenjeni obliki pošlje naprej ED.

ED v vseh virih v kartoteki zamenja referenco, kriptirano s starim ključem z referenco, ki je kriptirana z novim. Ko je postopek uspešno zaključen, ED generira zahtevek za posodobitev podatkov, ki vsebuje celotno reenkriptirano kartoteko pacienta. Ta poizvedba se pošlje do dostopne točke in naprej do FHIR strežnika, ki dostopa do baze, kjer se izvede posodobitev.

Celotna poizvedba se izvede v obliki transakcije, saj moramo zagotoviti, da ne pride do delno uspešnih posodobitev. Delno uspešna posodobitev bi povzročila neskladje med stanjem v tabeli pacient-ključ in dejanskim stanjem na bazi, posledica tega pa bi bila izguba podatkov.

V primeru, da je bila posodobitev uspešna, FHIR strežnik to preko dostopne točke sporoči ED, ki dostopa do tabele pacient-ključ in vanjo zabeleži ime novega ključa ter čas menjave. V primeru neuspešne transakcije se vse spremembe na bazi zavržejo in se ohrani staro stanje, v tabeli pacient-ključ ne zabeležimo nobene spremembe.

Postopek je s tem zaključen, celotna kartoteka pacienta pa je uspešno reenkriptirana.



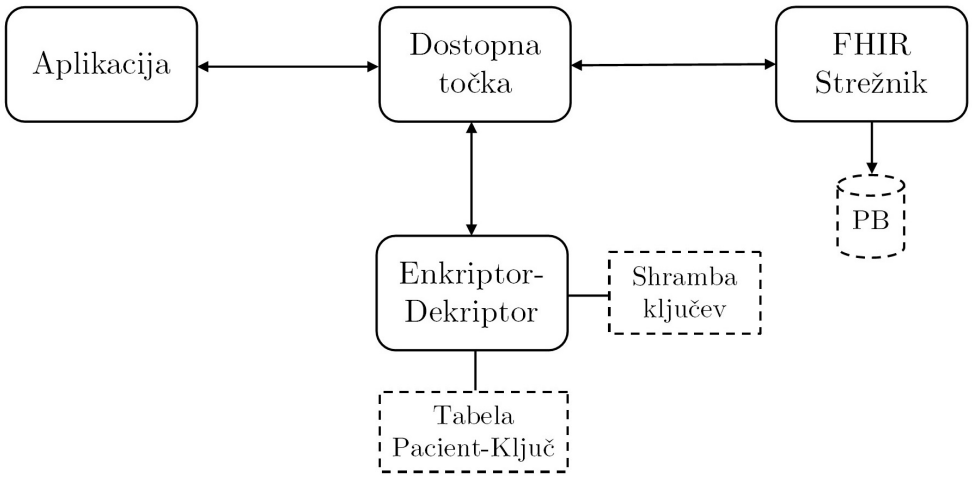
Slika 4.8: Menjava ključa, korak 2 - Reenkripcija kartoteke

Poglavje 5

Implementacija

Za praktično rešitev problemov in demonstracijo delovanja rešitve, opisane v poglavju Arhitektura, je bil razvit prototip. Implementacija prototipa podpira vse glavne funkcionalnosti, ki so bile opisane v prejšnjem poglavju.

Če si še enkrat ogledamo arhitekturo celotnega sistema, vidimo, da ga sestavljajo štiri glavne komponente: aplikacija, dostopna točka, ED in FHIR strežnik. Te komponente dostopajo do ostalih komponent, ki podpirajo njihovo delovanje. Glavni del implementacije prototipa sta dve komponenti: dostopna točka in ED.



Slika 5.1: Osnovna postavitev sistema

5.1 Uporabljena orodja

Obe glavni komponenti (dostopna točka in ED) sta bili razviti po arhitekturnem pristopu mikrostoritev v programskem jeziku Java EE [27]. V sklopu celotne rešitve se je uporablja knjižnica Hapi-FHIR, ki je odprtokodna implementacija FHIR specifikacije v Javi [19].

Ker projekt zaradi številnih zunanjih knjižnic vsebuje mnoge odvisnosti, se je pojavila potreba po bolj strukturirani gradnji projekta, zato je uporabljen Apache Maven [2].

Kot strežnik za testiranje se je uporabljal Apache Tomcat 9 [3]. Pri testiranju diplomske naloge sem uporabljal lokalno gostovanje (*angl.* localhost) zato je naslov uporabljenega testnega strežnika `http://localhost:7050/`.

5.2 Dostopna točka

Dostopna točka deluje kot standardiziran HL7 FHIR API vmesnik in skrbi za medsebojno komunikacijo med vsemi komponentami. Za demonstracijo delovanja so bile podprte funkcionalnosti na treh tipih virov: Patient, Observation in Condition [35, 34, 33].

5.2.1 Naslovi in standard

Po FHIR specifikaciji [20] mora biti osnovni URL naslov storitve v obliki[1] `http(s)://server{/path}`. Na tem naslovu morajo biti dostopni vsi viri, ki jih opisuje FHIR standard.

Ker mora dostopna točka navzven izpostavljati standardiziran FHIR vmesnik, je v primeru moje rešitve dostopna točka, skladno s standardom, dostopna na `http://localhost/hapi.do`.

[1] Parameter „/path“ je opcijski

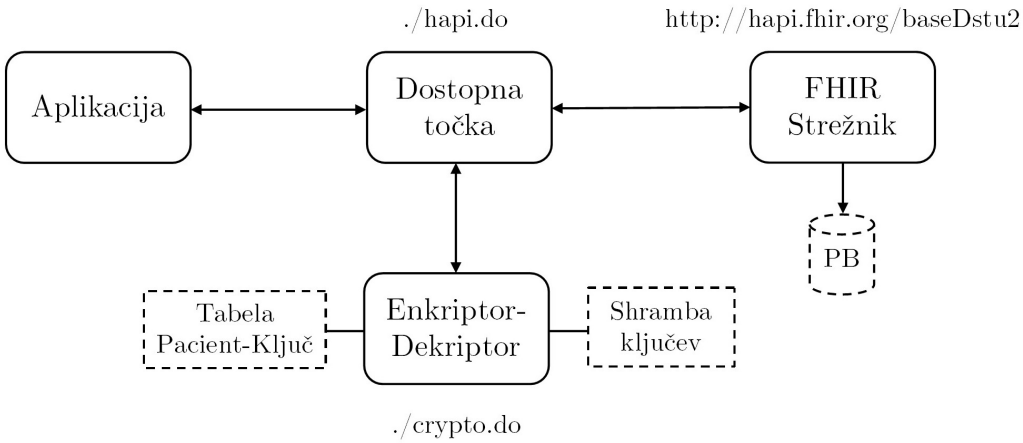
Vse interkacije s storitvijo so po standardu definirane kot[2]
`VERB [base]/[type]/[id] {?_format=[mime-type]}`

- VERB: Glagol, ki opisuje tip zahtevka - GET, POST, PUT, DELETE,...
- base: Naslov strežnika
- type: Ime tipa vira (npr. Patient)
- id: Logični identifikator vira
- mime-type: Specificira format rezultata - JSON, XML ali RDF

V implementaciji prototipa se lahko do virov, ki so bili testno implementirani, skladno s standardom dostopa preko zahtevkov na naslovih[3]:

`http://localhost/hapi.do/Patient/id`
`http://localhost/hapi.do/Observation/id`
`http://localhost/hapi.do/Condition/id`

ED je dostopen na naslovu `http://localhost/crypto.do/`.



Slika 5.2: Naslovi glavnih komponent aplikacije

[2]Vsebina v oglatih oklepajih je obvezna, vsebina v zavitih oklepajih je opcijska
[3]„id“ mora biti v dejanski poizvedbi zamenjan z logičnim identifikatorjem vira

Za FHIR strežnik je uporabljena prosto dostopna implementacija HapiFHIR strežnika, ki je namenjena testiranju in je dosegljiva na naslovu `http://hapi.fhir.org/baseDstu2`. Prototip je testiran na verziji 3.5.0.

Prototip, razvit za diplomsko nalogo, podpira GET in POST zahtevke. Atribut, ki se v prototipu kriptira na vseh virih, razen virih tipa Patient, je referenca na pacienta.

Tudi implementacijo je najlaže razložiti na praktičnih primerih, zato je opisana na primeru dveh tipov virov, Patient in Observation. Implementacija za vir tipa Condition ni ločeno opisana, saj je popolnoma enaka kot za vir tipa Observation.

Za čim bolj nazoren opis implementacije prototipa je v vseh primerih uporabljen „1“ kot identifikator pacienta, kriptirana vrednost te reference je „kriptirano1“, ključ, pod katerim je kriptirana kartoteka tega pacienta, pa je „ključ1“.

5.2.2 GET zahtevki

Prototip se odziva na GET zahtevke, ki zahtevajo vire tipa Patient, Observation in Condition. V ta namen so implementirani trije razredi: `hapiPatient`, `hapiObservation` in `hapiCondition`. Ti trije razredi so razširitev razreda `HttpServlet`.

V primeru, da aplikacija pošlje GET[4] zahtevek po viru tipa Patient[5] `./hapi.do/Patient/1` ali `./Patient?given=Testni&family=Pacient`, dostopna točka zazna, da v tem primeru kriptiranje ni potrebno, saj je pacient v bazi shranjen v nekriptirani obliki. Zato zahtevek v nespremenjeni obliki posreduje strežniku.

Strežnik prejme zahtevek, izvede poizvedbo na podatkovni bazi in dosto-

[4]Prototip podpira zahtevke le za podmnožico parametrov, ki jih opisuje specifikacija (iskanje po identifikatorju ter imenu in priimku)

[5]Zaradi lažje razumljivosti so vsi zahtevki opisani kot „./pot/tipVira/identifikator“, kjer je „./“ okrajšava za naslov strežnika „http://localhost“

pni točki pošlje rezultat poizvedbe. Dostopna točka zahtevek nato posreduje aplikaciji.

V primeru, da aplikacija pošlje GET zahtevek po viru tipa Observation `./hapi.do/Observation?patient=1`, kjer je potrebno kriptiranje, je postopek drugačen. Dostopna točka zazna, da je v primeru te reference potrebno kriptiranje, saj je v bazi shranjena v kriptirani obliki. Zato enkriptorju-dekriptorju (ED) pošlje GET zahtevek za kriptiranje identifikatorja `./crypto.do/Observation?encrypt=true&_id=1`.

ED izvede poizvedbo in v tabeli Pacient-Ključ preveri, s katerim ključem je kriptirana kartoteka pacienta 1, recimo da poizvedba vrne „ključ1“. Nato ED dostopa do shrambe ključev in s tem ključem kriptira vrednost 1, ki se kriptira v „kriptirano1“. ED kriptirano vrednost formatira v JSON format in pošlje odgovor dostopni točki.

Dostopna točka s tem dobi kriptirano vrednost atributa in spremeni začetno poizvedbo, tako da nekriptiran identifikator pacienta zamenja s kriptiranim identifikatorjem in generira spremenjen GET zahtevek[6]: `./baseDstu2/Observation?_content=Patient/kriptirano1`.

Dostopna točka si dekriptirano verzijo reference med čakanjem na odziv strežnika začasno shrani. Nato dostopna točka spremenjen zahtevek pošlje strežniku.

Strežnik izvede iskanje in vrne odgovor v JSON obliki. V primeru, da so viri s to referenco najdeni, dostopna točka v vseh virih zamenja kriptirano referenco z (začasno shranjeno) nekriptirano referenco in tako obdelan rezultat poizvedbe vrne aplikaciji.

[6]Podrobnejša razlaga oblike zahtevka v podpoglavju "Implementacija kriptirane reference"

5.2.3 Implementacija kriptirane reference

Javno dostopni HAPI strežnik pričakuje integriteto referenc. Posledično ni mogoče shraniti vira, ki vsebuje kriptirano referenco na pacienta, saj vrednost tujega ključa ne ustreza primarnemu ključu vira tipa Patient. Tudi če na strežniku izklopimo pogoj za integriteto referenc ob shranjevanju (vir s kriptirano referenco lahko tako uspešno shranimo), iskanje ne deluje, saj strežnik še vedno pričakuje, da je kriptiran identifikator pacienta tuji ključ, ki je povezan z identifikatorjem pacienta. Zato je implementirana razširitev, ki vsebuje kriptirani identifikator pacienta, ki jo strežnik ne zazna kot tuji ključ. Iskanje virov, ki pripadajo določenemu pacientu, zato deluje z delno spremenjeno poizvedbo.

Namesto poizvedbe `./baseDstu2/Observation?patient=kriptirano1`, ki predpostavlja, da v bazi obstaja pacient, katerega nekriptirani identifikator je „kriptirano1“, je poizvedba oblike `./baseDstu2/Observation?_content=Patient/kriptirano1`, saj iščemo po vsebini prej opisane razširitve.

5.2.4 POST Zahtevki

Prototip podpira POST zahtevke, ki zahtevajo vire tipa Patient, Observation in Condition. Z njimi operirajo isti trije razredi, kot z zahtevki GET: `hapiPatient`, `hapiObservation` in `hapiCondition`.

V primeru, da želi uporabnik ustvariti novega pacienta, aplikacija pošlje POST z dodanim JSON objektom, ki vsebuje vir tipa Patient. Dostopna točka zahtevek prestreže in zazna, da gre za vir tipa Patient, kjer kriptiranje ni potrebno, zato zahtevek v nespremenjeni obliki posreduje naprej strežniku.

V primeru, da je bilo shranjevanje uspešno, od strežnika prejme odziv, ki vsebuje identifikator pacienta. Nato se na ED pošlje zahtevek, naj se pacientu dodeli ključ. Ko se pacientu dodeli eden izmed ključev, se identifikator pacienta, ime dodeljenega ključa in čas dodelitve ključa zapišejo v tabelo pacient-ključ. V primeru neuspešnega shranjevanja pa FHIR strežnik zavrže poslani zahtevek in dostopni točki sporoči napako. Dostopna točka v tem

primeru javi napako in ne pošlje zahtevka za dodelitev ključa pacientu.

V primeru, da želi uporabnik dodati nov vir, aplikacija pošlje POST z dodanim JSON objektom, ki vsebuje vir tipa Observation[7]. Ta vir vsebuje referenco na pacienta v nekriptirani obliki.

Dostopna točka prejme poslani zahtevek in zazna, da gre za tip vira, kjer je potrebna enkripcija reference na pacienta. Zahtevek zato v nespremenjeni obliki posreduje ED.

ED nato nad prejetim JSON objektom, ki vsebuje vir tipa Observation, izvede kriptiranje reference na pacienta in jo shrani v obliki prej opisane razširitve, nato pa spremenjeni vir vrne dostopni točki.

Dostopna točka prejeti JSON objekt, ki vsebuje vir s kriptiranimi atributi, pošlje strežniku, ki ga shrani v podatkovno bazo in v obliki JSON objekta pošlje odziv, ki vsebuje informacijo o uspešnosti operacije in identifikator pravkar dodanega vira. Dostopna točka odziv posreduje aplikaciji, s tem pa se proces dodajanja novega vira zaključi.

5.3 Enkriptor-dekriptor

Enkriptor-dekriptor (ED) je poleg dostopne točke druga bistvena komponenta prototipa. Pri razlagi implementacije ED bom na kratko opisal razreda, ki sprejemata GET in POST zahtevke, nato pa se bom posvetil samem procesu enkripcije, hrambi ključev in nastavitvam algoritma AES.

Za obdelavo in kriptiranje poizvedb sta bila ustvarjena dva razreda `cryptoObservation` in `cryptoCondition`. Ko prejmeta zahtevek po enkripciji ali dekripciji v obliki GET ali POST zahtevka, se najprej generira instanca razreda `cryptoService` (podrobnejša razlaga sledi v naslednjem podpoglavju). Nato sledi inicializacija in klic funkcije `encrypt()` ali `decrypt()`, ki izvede proces enkripcije ali dekripcije. Po zaključeni enkripciji ali dekripciji se rezultat v obliki JSON objekta pošlje pošiljatelju zahtevka.

[7]Pošiljanje POST zahtevka je razloženo za vir tipa Observation, vendar enak postopek velja za vse tipe virov, ki vsebujejo kriptirano referenco na pacienta

5.3.1 Razred cryptoService

V namen enkripcije in dekripcije je ustvarjen razred `cryptoService`. Ta razred uporablja implementacijo algoritma AES iz knjižnice Bouncy Castle, ki je ena izmed najbolj pogosto uporabljanih knjižnic za kriptografijo v Javi [6].

V nadaljevanju bom opisal javne funkcije razreda `cryptoService`, ki jih kot del vmesnika, ki ga ponuja ta razred, uporabljajo funkcije ostalih razredov: `init()`, `encrpyt()`, `encryptWithNewKey()`, `addNewKeyToKeyStore()`.

Funkcija `init()` služi incializaciji servisa za enkripcijo. Najprej definira Bouncy Castle kot knjižnico za enkripcijo, nastavi enkripcijski algoritem AES s pravimi nastavitvami (podrobnejša razlaga v poglavju Nastavitve algoritma AES) in pridobi shrambo ključev (*angl.* KeyStore). S tem je inicializacija zaključena, vmesnik razreda `cryptoService` pa pripravljen za uporabo.

Funkcija `encrypt()` kot argument prejme identifikator pacienta. Najprej v tabeli pacient-ključ izvede poizvedbo, da pridobi ime ključa, s katerim je trenutno kriptirana kartoteka tega pacienta. Nato dostopa do shrambe ključev (*angl.* KeyStore) in pridobi ključ s tem imenom. Po uspešni pridobitvi ključa se identifikator kriptira in se ga v kriptirani obliki vrne klicatelju te funkcije.

Funkcija `encryptWithNewKey()` je funkcija, ki se kliče pri reenkripciji. Za razliko od funkcije `encrypt()`, ki kot argument sprejme le identifikator pacienta, ta funkcija sprejme še ime ključa. Potem poskuša pridobiti ključ iz shrambe ključev, nato identifikator pacienta kriptira s tem ključem in kriptirano vrednost reference z novim ključem vrne funkciji, ki izvaja reenkripcijo.

Zadnja izmed javnih funkcij tega razreda je `addNewKeyToKeyStore()`, ki generira nov ključ. Funkcija na začetku s kriptografsko varnim psevdonaključnim algoritmom generira nov ključ dolžine 128, 192 ali 256 bitov. Po generiranju se novi ključ zapiše v shrambo ključev pod imenom, ki ga ta funkcija ob klicu prejme kot argument, nato pa se posodobljena shramba ključev shrani.

5.3.2 Shramba ključev – KeyStore

Shramba ključev, Java KeyStore [28] je del paketa java.security, ki omogoča varno upravljanje in hrambo kriptografskih ključev in certifikatov. Vnosi v KeyStoru so zaščiteni z geslom KeyStora, posamezen vnos v KeyStoru pa enolično določa alias. Vsak vnos v KeyStoru je še dodatno zaščiten z geslom.

Tudi v prototipu te diplomske naloge je uporabljen KeyStore, ki varno hrani ključe pod njihovimi aliasi.

5.3.3 Nastavitve algoritma AES

Algoritem AES ob nastavljanju poleg dolžine ključa zahteva še dva parametra: način delovanja (*angl.* mode of operation) in način zapolnjevanja (*angl.* padding).

AES je bločni algoritem, kar pomeni, da je primeren le za varno kriptografsko transformacijo (enkripcijo ali dekripcijo) fiksne dolžine bitov, ki ji pravimo blok. V primeru algoritma AES je dolžina bloka 128 bitov.

Način delovanja opisuje, kako večkrat zaporedoma uporabiti operacijo nad posameznim blokom, da lahko varno kriptografsko transformiramo tudi podatke, ki so daljši od enega bloka.

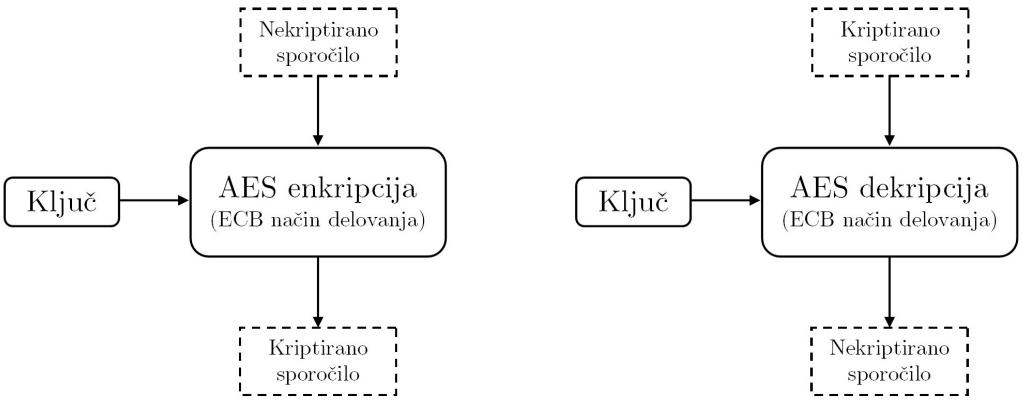
Za algoritem AES v večini implementacij obstaja 5 glavnih načinov delovanja: ECB (*angl.* Electronic Code Book), CBC (*angl.* Cipher Block Chaining), CFB (*angl.* Cipher FeedBack), OFB (*angl.* Output FeedBack) in CTR (*angl.* Counter) [29].

Podrobnosti posameznih načinov delovanja ne bom razlagal, utemeljil bom le, zakaj je bil v primeru mojega prototipa izbran način ECB in navedel glavne razlike tega načina v primerjavi z ostalimi.

Vsi zgoraj opisani načini delovanja, razen ECB, za vsako operacijo enkripcije poleg podatka, ki ga kriptirajo, zahtevajo še binarno zaporedje predpisane dolžine, ki mu pravimo inicializacijski vektor (v nadaljevanju IV). IV mora biti neponavljajoč in za večino načinov delovanja tudi naključen. Namen IV je, da zagotovimo, da dobimo različne kriptirane vrednosti, tudi v

primeru, da večkrat zaporedoma neodvisno kriptiramo isti podatek z istim ključem. Pri načinih delovanja, ki uporabljajo IV, kriptirana vrednost torej ni odvisna le od podatka in ključa, ampak tudi od IV [23].

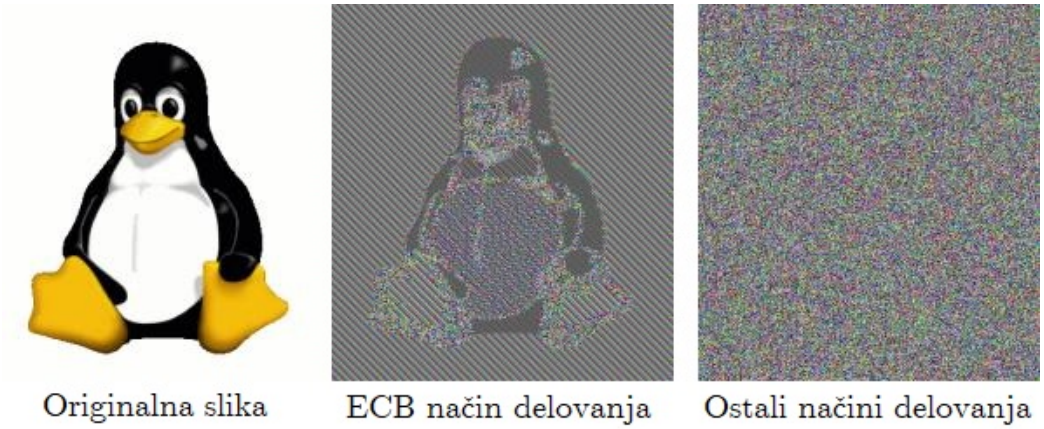
Če postavimo vsebino tega odstavka v kontekst te diplomske naloge vidimo, da so vsi načini delovanja, ki uporabljajo IV, za uporabo v tej diplomski nalogi neprimerni. V prejšnjih poglavjih smo postavili zahtevo, da iskanje po kriptiranih podatkih poteka na enak način kot iskanje po nekriptiranih podatkih. Odločili smo se tudi, da celotno kartoteko enega pacienta kriptiramo z istim ključem. Predpogoj, da lahko izvedemo iskanje na kriptiranih podatkih na enak način kot na nekriptiranih podatkih je, da se isti nekriptiran original, ki je kriptiran z istim ključem, vedno preslika v enako kriptirano sliko. V primeru algoritmov, ki uporabljajo IV, to ne drži, saj isti original zaradi različnih (naključnih) IV preslikajo v več različnih kriptiranih slik.



Slika 5.3: Enkripcija in dekripcija (Algoritem AES v ECB načinu delovanja)

V splošnem sicer velja, da ECB zaradi pomanjkanja razpršenosti ni najboljši način za kriptiranje podatkov, saj zaradi dejstva, da se ob istem ključu isti nekriptiran original vedno slika v enako nekriptirano sliko, slabo skriva vzorce.

Vendar pa je to eden od kompromisov med varnostjo in hitrostjo, ki ga moramo v implementaciji sprejeti zaradi predpogoja, da ne želimo spreminjati načina poizvedb po podatkih.



Slika 5.4: Primerjava načina ECB z ostalimi načini na primeru bitmap slike, ki ima velika področja enake barve [40, 39, 41]

V primeru prototipa način delovanja ECB ni tako kritičen, kot se morda zdi iz zgornje slike. Edino tveganje, ki obstaja zaradi tega načina delovanja je primer, ko bi napadalcu na podlagi vsebine nekega medicinskega podatka uspelo povezati medicinski podatek s pacientom. S tem bi ugotovil preslikavo med nekriptirano in kriptirano vrednostjo identifikatorja pacienta in posledično pridobil tudi kartoteko tega pacienta. Kartoteke ostalih pacientov s tem ne bi bile ogrožene, saj nekriptiran original (nekriptiran identifikator pacienta) in kriptirana slika (kriptiran identifikator pacienta) ne razkrivata ničesar o samem ključu.

5.4 Testna aplikacija

Ker je bistvo te diplomske naloge razviti sistem, ki skrbi za varnost podatkov neke že obstoječe aplikacije s področja e-zdravja, ki uporablja HL7 FHIR standard, je bila za namen te diplomske naloge razvita preprosta testna aplikacija, ki z generiranjem GET in POST zahtevkov simulira nekaj najpogostejših API klicev, ki bi jih izvajala dejanska aplikacija. Testna aplikacija generira in pošlje zahtevek, po prejemu odgovora pa prikaže odziv.

Iste funkcionalnosti, kot jih podpira testna aplikacija, lahko testiramo

tudi direktno s pošiljanjem veljavnih GET ali POST zahtevkov dostopni točki.

5.5 FHIR Strežnik

Za namen testiranja rešitve je bil uporabljen prosto dostopen Hapi-FHIR strežnik, ki je namenjen testiranju in uporablja standardiziran HL7 FHIR API. Strežnik je bil v času testiranja dosegljiv na naslovu `http://hapi.fhir.org/baseDstu2`.

Ta implementacija je izbrana, ker za razliko od večine ostalih testnih strežnikov ponuja zelo dober grafični vmesnik, preko tega vmesnika pa tudi direkten dostop do virov, ki se hranijo v podatkovni bazi, kar bistveno olajša testiranje rešitve.

Poglavje 6

Testiranje rešitve in demonstracija delovanja

V tem poglavju je opisano, kako lahko s testno aplikacijo pošiljamo POST in GET zahtevke. Sledi opis funkcionalnosti menjave ključa, na koncu poglavja pa je podana še primerjava časov med rešitvijo z varnostnim modulom in rešitvijo brez varnostnega modula.

6.1 POST zahtevki

Testna aplikacija podpira GET in POST zahtevke za vire tipov Patient, Observation in Condition. V nadaljevanju tega poglavja bo najprej prikazano dodajanje vira tipa Patient, nato dodajanje vira tipa Observation prej ustvarjenemu pacientu. Enak postopek bi bil tudi v primeru dodajanja vira tipa Condition.

6.1.1 Dodajanje vira tipa Patient

Nov vir tipa Patient lahko ustvarimo preko testne aplikacije, s klicem funkcije `addPatient()` ali s pošiljanjem POST zahtevka, ki ima v telesu dodan veljavni FHIR JSON objekt na naslov dostopne točke `./hapi.do/Patient`.

Če kličemo funkcijo `addPatient()`, ki ustvari JSON objekt in ga pošlje na dostopno točko, v testni aplikaciji dobimo odgovor:


```
{
  "resourceType": "Bundle",
  "id": "b2f5f087-ef18-4433-8e7a-66825ae23dcc",
  "type": "transaction-response",
  "link": [
    {
      "relation": "self",
      "url": "http://hapi.fhir.org/baseDstu2"
    }
  ],
  "entry": [
    {
      "response": {
        "status": "201 Created",
        "location": "Patient/12515/_history/1",
        "etag": "1",
        "lastModified": "2018-08-11T12:15:44.332+00:00"
      }
    }
  ]
}
```


Odgovor z izhodnim statusom „201 Created“ potrdi uspešno dodajanje novega vira. Enolični identifikator pacienta je „Patient/12515“.

6.1.2 Dodajanje vira tipa Observation

V tem koraku bomo testnemu pacientu, ki smo ga ustvarili v prejšnjem koraku in ima identifikator „Patient/12515“, dodali vir tipa Observation.

Vir tipa Observation lahko pacientu dodamo preko testne aplikacije s klicem funkcije `getObservationToPatient()`. Kot argument funkciji podamo nekriptirani identifikator pacienta. Testna aplikacija ustvari JSON objekt z nekriptirano referenco na pacienta in ga pošlje dostopni točki na naslov `./hapi.do/Observation`.

Aplikacija podobno kot v prejšnjem primeru prikaže odziv strežnika v obliki JSON objekta:


```
{
  "resourceType": "Bundle",
  "id": "6371097e-5fd7-4cd0-9722-14a36e4cd5e3",
  "type": "transaction-response",
  "link": [
    {
      "relation": "self",
      "url": "http://hapi.fhir.org/baseDstu2"
    }
  ],
  "entry": [
    {
      "response": {
        "status": "201 Created",
        "location": "Observation/12517/_history/1",
        "etag": "1",
        "lastModified": "2018-08-11T12:29:23.766+00:00"
      }
    }
  ]
}
```


Izhodni status je bil „201 Created“, torej je bilo dodajanje vira uspešno. Enolični identifikator vira tipa Observation je „Observation/12517“.

Če direktno na FHIR strežniku preverimo, kako je ta vir shranjen vidimo, da se v podatkovno bazo shrani kot:


```
...
"entry": [
{
"fullUrl": "http://hapi.fhir.org/baseDstu2/Observation/12517",
  "resource": {
    "resourceType": "Observation",
    "id": "12517",
    "meta": {
      "versionId": "1",
      "lastUpdated": "2018-08-11T12:29:23.766+00:00"
    },
    "extension": [
      {
        "id": "encryptedReference",
        "url": null,
        "valueString": "Patient/9MwcCeb9By4CgF7nPIGzKQ=="
      }
    ],
 ...
}
```


V delu „extension“ vidimo, da je bila razširitev s kriptirano referenco uspešno ustvarjena in da je pacient „Patient/12515“ v podatkovni bazi dejansko shranjen kot „Patient/9MwcCeb9By4CgF7nPIGzKQ==“, kar je kriptirana vrednost te reference.

S tem lahko potrdimo, da shranjevanje virov s kriptiranimi referencami na prototipu deluje na način, kot je bilo opisano v poglavjih Arhitektura in Implementacija.

6.2 GET zahtevki

V prejšnjem podpoglavju smo ustvarili vir tipa Patient, z identifikatorjem „Patient/12515“ in mu dodali vir tipa Observation. V viru tipa Observation se je referenca na pacienta shranila v kriptirani obliki.

6.2.1 Iskanje vira tipa Observation

V tem razdelku bomo iskali vir tipa Observation, ki pripada pacientu z identifikatorjem „Patient/12515“. Iskanje lahko izvedemo s klicem funkcije `getAllObservationForPatient()`, ki ji kot argument podamo nekriptiran identifikator pacienta ali preko GET zahtevka, ki ga pošljemo dostopni točki na naslov `./hapi.do/Observation?patient=12515/`.

Od dostopne točke prejmemo naslednji odgovor:


```
"entry": [
{
"fullUrl": "http://hapi.fhir.org/baseDstu2/Observation/12517",
"resource": {
  "resourceType": "Observation",
  "id": "12517",
  "meta": {
    "versionId": "1",
    "lastUpdated": "2018-08-11T12:29:23.766+00:00"
  },
  "extension": [
    {
      "id": "encryptedReference",
      "url": null,
      "valueString": "Patient/12515"
    }
  ], ...
```

Vir z identifikatorjem „Observation/12517“, ki smo ga ustvarili v prejšnih korakih, je bil uspešno najden. Prav tako je vrednost reference v aplikaciji prikazana kot „Patient/12515“, čeprav je v bazi dejansko shranjena kot „Patient/9MwcCeb9By4CgF7nPIGzKQ==“.

S tem lahko potrdimo, da tudi iskanje virov, ki hranijo kriptirano referenco na pacienta deluje na način, opisan v poglavjih Arhitektura in Implementacija.

6.3 Menjava ključev

Menjava ključev je administratorska funkcija, zato ni dostopna preko testne aplikacije.

V prototipu je menjava ključa dostopna kot API zahtevek direktno na ED. Nov ključ lahko generiramo z GET zahtevkom direktno na ED, ki je oblike `crypto.do/GenerateNewKey?keyAlias=imeKljuca`. Recimo, da pošljemo zahtevek z vrednostjo parametra „keyAlias=key2“.

ED generira nov ključ in ga shrani v shrambo ključev. Nato želimo z novim ključem reenkriptirati kartoteko pacienta, ki smo ga ustvarili v prvem koraku z zahtevkom `./crypto.do/ChangeKey?_id=12515&keyAlias=key2`.

Če direktno na testnem strežniku preverimo, kako je v podatkovni bazi shranjen vir „Observation/12517“:

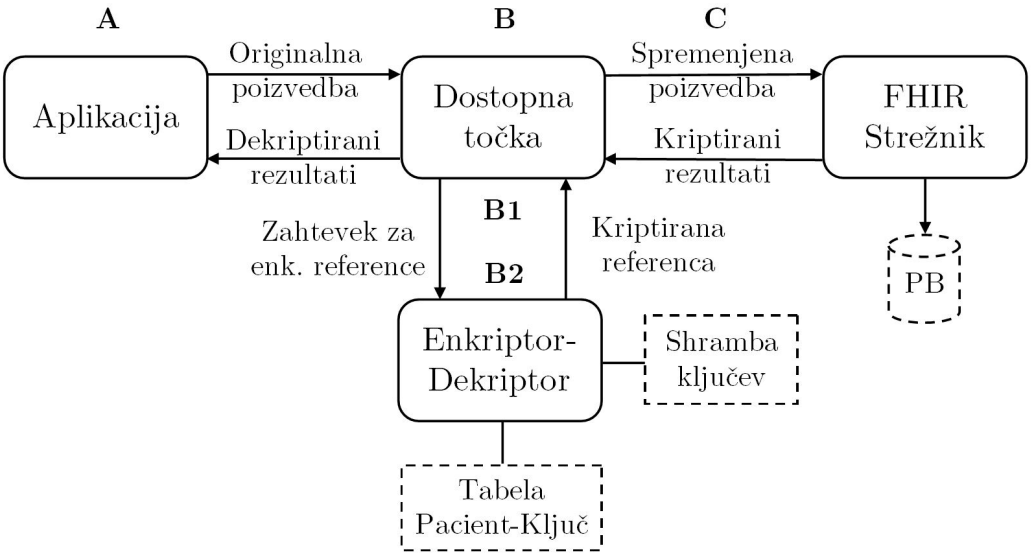
```
{
...
"entry": [
{
"fullUrl": "http://hapi.fhir.org/baseDstu2/Observation/12517",
"resource": {
  "resourceType": "Observation",
  "id": "12517",
  "meta": {
    "versionId": "2",
```

```
    "lastUpdated": "2018-08-11T13:48:05.557+00:00"
  },
  "extension": [
    {
      "id": "encryptedReference",
      "url": null,
      "valueString": "Patient/ACfmEkq7vtgohowkyvntVw=="
    }
  ],
...
}
```

Referenca na pacienta je po uspešni reenkripciji shranjena kot „Patient/ACfmEkq7vtgohowkyvntVw==“ (vrednost kriptirane reference pred reenkripcijo je bila „Patient/9MwcCeb9By4CgF7nPIGzKQ==“), zato lahko zaključimo, da je tudi reenkripcija implementirana na način, ki je bil opisan v prejšnjih poglavjih in da vsi sklopi funkcionalnosti, ki smo jih predvideli v poglavjih Arhitektura in Implementacija, delujejo.

6.4 Primerjava časov

V tem podpoglavju se bomo posvetili primerjavi časov izvajanja poizvedb med aplikacijo z varnostnim modulom, opisanim v tej diplomski nalogi in aplikacijo brez tega modula in analizirali razliko med njima.



Slika 6.1: Shema GET zahtevka z mesti meritve časa

V primeru GET zahtevka, ki je opisan v nadaljevanju, smo poizvedovali po viru tipa Observation, ki ima kriptirano referenco na pacienta, ki je bil za potrebe testiranja ustvarjen na začetku tega poglavja.

Aplikacija pošlje poizvedbo z nekriptiranim identifikatorjem pacienta dostopni točki, dostopna točka pošlje zahtevek za enkripcijo ED. Ko dostopna točka prejme odgovor, spremeni poizvedbo in zamenja nekriptirani identifikator s kriptiranim. Nato pošlje zahtevek FHIR strežniku, ki vrne rezultat s kriptirano referenco, dostopna točka ga obdela in vrne rezultat aplikaciji v dekriptirani obliki.

Za meritev časa je bilo izbranih 5 merilnih mest:

- A: Čas izvajanja celotnega zahtevka – čas, ki preteče od trenutka, ko aplikacija pošlje dostopni točki GET zahtevek, do trenutka, ko prejme odgovor.

- B: Čas čakanja na kriptirano referenco – čas, ki preteče od trenutka, ko dostopna točka pošlje ED zahtevek za enkripcijo reference, do trenutka, ko prejme rezultat. B je vsota merilnih mest B1 in B2.
- B1: Čas obdelave zahtevka – čas, ki ga dostopna točka in ED potrebujeta za obdelavo zahtevkov z izključenim časom enkripcije (B2).
- B2: Čas enkripcije – čas, ki ga ED potrebuje za poizvedbo za ime ključa, dostop do shrambe ključev in proces enkripcije.
- C: Čas izvajanja poizvedbe na HAPI strežniku – čas, ki preteče od trenutka, ko dostopna točka pošlje poizvedbo s kriptirano referenco, do trenutka, ko prejme odgovor.

#\Mesto meritve	**A**	**B**	**B1**	**B2**	**C**
1	249	85	51	34	162
2	250	85	52	33	163
3	248	85	51	34	160
4	248	84	51	33	161
5	248	84	50	34	161
6	249	85	51	34	162
7	273	88	51	37	182
8	250	86	51	35	161
9	251	86	52	34	162
10	254	89	54	35	162
Povprečje	**252**	**85.7**	**51.4**	**34.3**	**163.6**

Tabela 6.1: Meritev časa izvajanja poizvedb, 10 meritev (v ms)

V stolpcu A je meritev časa izvajanja poizvedbe ob uporabi varnostnega modula, v stolpcu C pa čas izvajanja poizvedbe na HAPI strežniku. Časi v stolpcu C so enaki, kot bi bili v primeru aplikacije brez varnostnega modula, saj so odvisni le od časa, ki preteče med tem, ko aplikacija pošlje zahtevek na FHIR strežnik do trenutka, ko prejme rezultat.

Če primerjamo povprečen rezultat ob uporabi varnostnega modula (stolpec A) s povprečnim rezultatom brez uporabe varnostnega modula (stolpec C) ugotovimo, da je čas izvajanja zahtevka ob uporabi varnostnega modula 58,6% daljši. Od tega večji del (60,0%) odpade na sam proces enkripcije (stolpec B1), manjši del (40,0%) pa na obdelavo zahtevkov, spremembo poizvedbe in obdelavo rezultatov (stolpec B2).

Če na razliko gledamo s stališča uporabnika v kontekstu celotne aplikacije, vidimo, da upočasnitev ob uporabi varnostnega modula ni velika. Absolutno gledano varnostni modul poveča celoten čas izvajanja zahtevka za povprečno 85 ms (stolpec B), kar za končnega uporabnika, ki aplikacijo uporablja interaktivno, ni opazno.

Poleg tega bi lahko skupni rezultat z uporabo varnostnega modula (stolpec A) še izboljšali. Večji del skupnega časa odpade na čakanje odziva s strani Hapi-FHIR strežnika (stolpec C). V primeru zgornjih meritev se je uporabljal javno dostopen testni strežnik, ki je geografsko precej oddaljen (v času testiranja je bil strežnik lociran v kraju Mountain View, Kalifornija, ZDA). Posledica geografske oddaljenosti je precej velika zakasnitev, saj je strežnik na ping zahtevek odgovarjal relativno dolgih 140 ms[1]. Zakasnitev posledično vpliva tudi na čas izvajanja zahtevkov, ki bi se ga dalo v primeru uporabe geografsko bližjega strežnika bistveno zmanjšati.

Poleg tega bi se dalo nekaj časa prihraniti še na samem času enkripcije. Čeprav je knjižnica Bouncy Castle [6], ki je bila za enkripcijo uporabljena v primeru te diplomske naloge v praksi precej priljubljena, ne podpira AES-NI nabora ukazov, posledično pa ne izkorišča strojnega pospeševanja enkripcije. Ob uporabi knjižnice, ki podpira strojno pospeševanje enkripcije, bi bili časi enkripcije nedvomno krajši [25].

V diplomski nalogi so bili uporabljeni 256-bitni ključi. Z uporabo krajših ključev bi, sicer za ceno manjše varnosti, lahko pridobili še nekaj dodatnih milisekund.

[1]V primeru ping zahtevka na strežnik www.arnes.si, ki je lociran v Ljubljani, je bil povprečen čas ping zahtevka 10 ms

Poglavje 7

Sklepne ugotovitve

Izbrani arhitekturni pristop se je izkazal za primernega, kar je bilo prikazano s prototipom. Na osnovi arhitekture mikrostoritev je razvita hitra in prilagodljiva rešitev problema. Prototip omogoča shranjevanje kliničnih podatkov, simetrično enkripcijo občutljivih atributov in dostop do podatkov. Celotna rešitev je razvita skladno s standardom HL7 FHIR, ki je trenutno eden od najbolj pogosto uporabljanih standardov na področju e-zdravstva. Prototip dobro deluje, pri razvoju pa so bile upoštevane vse zahteve, zastavljene v okviru te diplomske naloge. Primernost arhitekturnega pristopa lahko potrdimo tudi na podlagi rezultatov meritev.

Zasnova varnostnega modula omogoča enostavno vključitev v rešitev s področja e-zdravstva, saj ponuja standardiziran HL7 FHIR API vmesnik, ki skriva kompleksnost rešitve v ozadju in s stališča aplikacije ne zahteva praktično nobenih sprememb.

Tudi rezultati meritev časa so vzpodbudni. Z vidika končnega uporabnika varnostni modul ne prinaša bistvene upočasnitve delovanja sistema. Poleg tega bi lahko v primeru praktične uporabe tega modula brez večjih posegov v implementacijo izboljšali komponente, ki trenutno predstavljajo največja ozka grla.

Glede na rezultate ima izdelani varnostni modul potencial za nadaljnjo uporabo in vključitev v aplikacijo s področja e-zdravstva.

Literatura

[1] Raluca Ada Popa, Catherine M. S. Redfield, Nikolai Zeldovich, and Hari Balakrishnan. Cryptdb: Protecting confidentiality with encrypted query processing. *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, pages 1–16, 2011.

[2] Apache Maven. Dosegljivo: `https://maven.apache.org/`. [Dostopano: 7. 8. 2018].

[3] Apache Tomcat. Dosegljivo: `http://tomcat.apache.org/`. [Dostopano: 7. 8. 2018].

[4] Karthikeyan Bhargavan and Gaëtan Leurent. On the practical (in)security of 64-bit block ciphers. *ACM CCS*, 2016. [Dostopano 6. 8. 2018].

[5] Blockchain in health care: The good, the bad and the ugly. Dosegljivo: `https://www.forbes.com/sites/forbestechcouncil/2018/04/13/blockchain-in-health-care-the-good-the-bad-and-the-ugly/#3c23b48c6278`. [Dostopano: 20. 8. 2018].

[6] Bouncy Castle. Dosegljivo: `https://www.bouncycastle.org/java.html`. [Dostopano: 10. 8. 2018].

[7] Steven Clarke. Measuring API usability. Dosegljivo: `http://www.drdobbs.com/windows/measuring-api-usability/184405654`. [Dostopano: 7. 8. 2018].

[8] Commerce department announces winner of global information security competition. Dosegljivo: https://www.nist.gov/news-events/news/2000/10/commerce-department-announces-winner-global-information-security. [Dostopano: 6. 8. 2018].

[9] Ronald Cramer and Victor Shoup. Design and analysis of practical public-key encryption schemes secure against adaptive chosen ciphertext attack. *SIAM Journal on Computing*, 33(1):167–226, 2004.

[10] Cryptdb. Dosegljivo: https://css.csail.mit.edu/cryptdb/. [Dostopano: 13. 8. 2018].

[11] The 17 biggest data breaches of the 21st century. Dosegljivo: https://www.csoonline.com/article/2130877/data-breach/the-biggest-data-breaches-of-the-21st-century.html. [Dostopano: 12. 8. 2018].

[12] Hans Delfs and Helmut Knebl. *Applied Cyptography Protocols, Algorithms and Source code in C*. Wiley, 2007.

[13] Hans Delfs and Helmut Knebl. *Introduction to Cryptography: Principles and Applications, Second Edition*. Springer, 2007.

[14] Pradeep Deshmukh. Design of cloud security in the electronic health record for indian healthcare services. *Journal of King Saud University – Computer and Information Sciences*, (29):281–287, 2017.

[15] Nicola Dragoni, Saverio Giallorenzo, and Alberto Lluch Lafuente. Secure hybrid encryption from weakened key encapsulation. *Microservices: yesterday, today, and tomorrow*, page 16, 2017. [Dostopano 12. 8. 2018].

[16] Charles Fan. CRAP and CRUD: From Database to Datacloud. Dosegljivo: https://blog.dellemc.com/en-us/crap-and-crud-from-database-to-datacloud/. [Dostopano: 7. 8. 2018].

[17] David Garlan and Mary Shaw. An introduction to software architecture. *Advances in Software Engineering and Knowledge Engineering*, 1, 1993. [Dostopano 6. 8. 2018].

[18] Oded Goldreich. Foundations of cryptography. *Basic Applications*, 2, 2004.

[19] Hapi FHIR. Dosegljivo: `http://hapifhir.io/`. [Dostopano: 7. 8. 2018].

[20] HL7 FHIR - RESTful API. Dosegljivo: `https://www.hl7.org/fhir/http.html`. [Dostopano: 7. 8. 2018].

[21] HL7 FHIR - Base Resource Definitions. Dosegljivo: `https://www.hl7.org/fhir/resource.html`. [Dostopano: 3. 8. 2018].

[22] Dennis Hofheinz and Eike Kiltz. Secure hybrid encryption from weakened key encapsulation. *Advances in Cryptology – CRYPTO 2007*, pages 553–571, 2007.

[23] Kuo-Tsang Huang, Jung-Hui Chiu, and Sung-Shiou Shen. A novel structure with dynamic operation mode for symmetric-key block ciphers. *International Journal of Network Security and Its Applications (IJNSA)*, 5(1):19, 01 2013.

[24] Shoukat Ijaz Ali, Abu Bakar Kamalrulnizam, and Ibrahim Subariah. A generic hybrid encryption system (hes). *Research Journal of Applied Sciences, Engineering and Technology*, 5(9):2692–2700, 03 2013.

[25] GenoSpace Boosts Population Analytics* Application Performance. Dosegljivo: `https://www.intel.com/content/www/us/en/healthcare-it/solutions/documents/aes-ni-boosts-security-and-performance-genospace-study.html`. [Dostopano: 12. 8. 2018].

[26] Intel® Advanced Encryption Standard Instructions (AES-NI). Dosegljivo: `https://software.intel.com/en-us/articles/intel-`

advanced-encryption-standard-instructions-aes-ni/. [Dostopano: 12. 8. 2018].

[27] Java EE. Dosegljivo: https://www.oracle.com/technetwork/java/javaee/overview/index.html. [Dostopano: 7. 8. 2018].

[28] Class Key Store. Dosegljivo: https://docs.oracle.com/javase/7/docs/api/java/security/KeyStore.html. [Dostopano: 10. 8. 2018].

[29] Alfred Menezes J., Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

[30] Pattern: Microservice Architecture. Dosegljivo: http://microservices.io/patterns/microservices.html. [Dostopano: 12. 8. 2018].

[31] Shiho Moriai and Lisu Yin Yiqun. Cryptanalysis of twofish (ii). *Cr.yp.to*, 2000. [Dostopano 6. 8. 2018].

[32] Stefano M.P.C. Souza and Ricardo S. Puttini. Client-side encryption for privacy-sensitive applications on the cloud. *Procedia Computer Science*, (97):126–130, 2016.

[33] Resource condition - content. Dosegljivo: https://www.hl7.org/fhir/condition.html. [Dostopano: 6. 8. 2018].

[34] Resource observation - content. Dosegljivo: https://www.hl7.org/fhir/observation.html. [Dostopano: 6. 8. 2018].

[35] Resource patient - content. Dosegljivo: https://www.hl7.org/fhir/patient.html. [Dostopano: 6. 8. 2018].

[36] Prakash Sankalp. An analytical study of hybrid cryptography and implementation of rsa with hash functions. Doktorska dizertacija, Jagannath University, 2017.

[37] Bruce Schneier and D. Whiting. A performance comparison of the five aes finalists. *Proceedings of the Third AES Candidate Conference*, 2000. [Dostopano 6. 8. 2018].

[38] Gueron Shay. Intel advanced encryption standard (aes) instruction set white paper. Dosegljivo: `https://www.intel.com/content/dam/doc/white-paper/advanced-encryption-standard-new-instructions-set-paper.pdf`, 2010. [Dostopano: 6. 8. 2018].

[39] Tux_ecb.jpg. Dosegljivo: `http://en.wikipedia.org/wiki/Image:Tux_ecb.jpg`. [Dostopano: 11. 8. 2018].

[40] Tux.jpg. Dosegljivo: `https://commons.wikimedia.org/wiki/File:Tux.jpg`. [Dostopano: 11. 8. 2018].

[41] Tux_secure.jpg. Dosegljivo: `http://en.wikipedia.org/wiki/Image:Tux_secure.jpg`. [Dostopano: 11. 8. 2018].

[42] Web Services Architectrue. Dosegljivo: `https://www.w3.org/TR/2004/NOTE-ws-arch-20040211/#relwwwrest`, 2004. [Dostopano: 7. 8. 2018].